Investigations into the Model Driven Design of Distribution Patterns for Web Service Compositions

Ronan Barrett

Bachelor of Science in Computer Science

A Dissertation submitted in fulfilment of the

requirements for the award of

Doctor of Philosophy (Ph.D.)

to the

DCU

Dublin City University

Faculty of Engineering and Computing, School of Computing

Supervisor: Dr. Claus Pahl

January, 2008

Declaration

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of Doctor of Philosophy is entirely my own work, that I have exercised reasonable care to ensure that the work is original, and does not to the best of my knowledge breach any law of copyright, and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed: [illegible]

Student ID: 53138406

Date: January 25, 2008

Contents

Abstract

Increasingly, distributed systems are being used to provide enterprise level solutions with high scalability and fault tolerance. These solutions are often built using Web services that are composed to perform useful business functions. Acceptance of these composed systems is often constrained by a number of non-functional properties of the system such as availability, scalability and performance. There are a number of distribution patterns that each exhibit different non-functional characteristics. These patterns are re-occurring distribution schemes that express how a system is to be assembled and subsequently deployed.

Traditional approaches to development of Web service compositions exhibit a number of issues. Firstly, Web service composition development is often ad-hoc and requires considerable low level coding effort for realisation. Such systems often exhibit fixed architectures, making maintenance difficult and error prone. Additionally, a number of the non-functional requirements cannot be easily assessed by examining low level code.

In this thesis we explicitly model the compositional aspects of Web service compositions using UML Activity diagrams. This approach uses a modeling and transformation framework, based on Model Driven Software Development (MDSD), going from high level models to an executable system. The framework is guided by a methodological framework whose primary artifact is a distribution pattern model, chosen from the supplied catalog.

Our modeling and transformation framework improves the development process of Web service compositions, with respect to a number of criteria, when compared to the traditional handcrafted approach. Specifically, we negate the coding effort traditionally associated with Web service composition development. Maintenance overheads of the solution are also significantly reduced, while improved mutability is achieved through a flexible architecture when compared with existing tools. We also improve the product output from the development process by exposing the non-functional runtime properties of Web service compositions using distribution patterns.

Acknowledgements

Firstly, I would like to thank Dr. Claus Pahl for his time and encouragement over the last four years. Claus you have been a pleasure to work with. Your attention to detail and ability to maintain focus have been invaluable.

I am indebted to you Mum and Dad for all your encouragement during my whole academic life. You have always been so supportive, I hope you know how much I appreciate it. All those registered letters to get me into Computer Science have definitely paid off Mum! Thanks of course to Bren, Conor, Alan, Niamh and Daragh (and Adam of course) for all your help. Especially to Bren for all the excellent advise and help you gave me before and during my PhD.

A big thanks to all my friends in CA who made the time fly in. I believe in naming people so here we go, in original bay order. Thanks to Cara, JJ, Declan, Mark, Bernard, Claire K, Hego, Noel, Claire W, Neil, Sara, Riona, Gav, Caroline, George, Tommy, Adel, Niall, Karl, James, Noreen and Grainne. The biggest thanks go to Mark and Claire K. First of all thanks Mark for being a great friend and colleague. Our endless hours of talking rubbish made sure there was always plenty of craic in CA. We even had some intelligent conversations, the odd time! Secondly huge thanks to the best bay buddy ever, Claire. It was always great having you around. For all your help on this thesis I owe you about a million bottles of Erdinger!

I do of course have to thank all my olde skool friends for the usual weekend diversions. Thanks to Keogher, Hego, Beachy, Cara, Paco, Raymo, Deery and Lockie. I should also add in Matt here, as your FT228 olde skool. Matt your excuses for missing Friday pints always amuse us!

Finally, I'd like to dedicate this thesis to Dr. Cormac who never got the opportunity to submit his masterpiece.

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Problem Context

Increasingly, distributed systems are being used to provide solutions with high scalability and fault tolerance. These distributed systems also enable and encourage the reuse of components, a long held goal of software engineering. There are many conceptual architectures for distributing components. One such architecture is Service Oriented Architectures (SOA) [61], which exposes application logic through an interface that can be interacted with using a common or standard communications protocol. SOA has superseded previous distributed system architectures where many fine grained distributed functions and objects were executed to realise a certain business goal.

Web services are an example of an SOA. Web services themselves are pieces of software functionality, available at a certain location, that can be invoked using Web based technologies [9]. Our motivation for considering Web services in this thesis is the numerous perceived benefits that can be gained from using this technology. These benefits include the simplicity of Web services and their use of standardised XML. Of course other technologies that may function as an SOA, such as CORBA [129], could have been considered.

For Web services to be really useful in an enterprise environment, where many disparate systems must work together, they must be combined with other Web services. This practice is termed composition. Enterprise systems are frequently built from many existing discrete applications. These applications are often legacy applications. These legacy applications can be exposed using Web service and can be composed to perform some useful business function using Web service compositions.

Acceptance of these composed systems is often constrained by the non-functional properties exhibited by the composition. Non-functional properties may be partitioned into design time and runtime categories. The runtime non-functional properties, also know as Quality of Service (QoS) attributes, include reliability and efficiency. Design time non-functional properties include maintainability and portability. We align our classification of non-functional properties with the ISO 9126 software product evaluation standard [88].

There are a number of architectural configurations or distribution patterns that express how a composed system is to be deployed. Each distribution pattern exhibits different QoS attributes, appropriate for a given context. Patterns help document a system's architecture by clearly exposing the non-functional trade-offs accepted by the software architect at design time. We examine these attributes and contexts in detail. However, the amount of code required to realise these distribution patterns is considerable. In this thesis, we propose a novel Model Driven Software Development (MDSD) based approach using UML 2.0, which takes existing Web service interfaces as its input and generates an executable Web service composition, based on a distribution pattern chosen by the software architect. This executable Web service composition can meet any number of non-functional requirements, depending on the distribution pattern chosen by the software architect.

1.2 Problem Statement

Traditional approaches to development of Web service compositions raise a number of issues. The problems identified in this thesis are non-functional requirement issues. Firstly, Web service composition development is often ad-hoc and requires considerable low level coding effort for realisation [9]. This effort is increased in proportion to the number of Web services in a composition or by a requirement for the composition participants to be flexible [29]. Such systems often exhibit fixed architectures thus making maintenance difficult and error prone. Additionally, a number of the non-functional Quality of Service (QoS) requirements [72] such as efficiency and reliability of a composition cannot be assessed easily by examining low level code.

1.3 Research Objective

With our problem statement in mind we propose that the following research objectives should be met within the context of this thesis. These objectives are split into two categories, as follows.

- Development Process
- Product Output

Firstly, we will improve the development process for Web service compositions. The goal of this process improvement is to increase the quality of the software created during the development process, with respect to design time non-functional properties like maintainability. We consider that the development effort for creating Web service compositions should be reduced. As a byproduct of reducing the development effort we intend to reduce the maintenance overheads traditionally associated with mutating compositions. More specifically, we will obviate the coding effort usually associated with Web service composition development thereby significantly reducing the development effort. Maintenance overheads of the solution will be significantly lowered. In addition, improved maintainability will be achieved through a flexible architecture. Comprehensibility will also be improved by using standards based models throughout the development process.

Secondly, we will improve the control developers have over the product output from the development process. The objective of improving the product output is to provide control over the QoS of Web service compositions through distribution patterns. This approach to managing QoS is to expose properties such as efficiency to ensure the resultant composition is of the required quality.

1.4 Solution

Our solution for realising the objectives outlined is to explicitly model the compositional aspects of Web service compositions. This approach uses a modeling and transformation framework, going from high level models to an executable system. We base our development approach on the Model Driven Software Development (MDSD) approach [182]. MDSD considers models as formal specifications of the structure or function of a system, where the modeling language is in fact the programming language. Having rich, well specified, high level models allows for the auto-generation of a fully executable system based entirely on the model.

Our approach is expressed using a modeling and transformation framework consisting of five components, as follows.

- A Catalog of Distribution Patterns - Enumeration of the possible distribution schemes for Web service compositions.
- Modeling Notations - Definition of required modeling notations.
- Model Relations - Definition of the web of dependencies between modeling languages.
- Model Transformations - Rules for transforming between modeling notations.
- Methodological Framework - Outline of how high level distribution models are used for code generation.

We assume that Web service compositions have three modeling aspects. Two aspects, service modeling and workflow modeling, are considered by [152]. Service modeling expresses interfaces and operations, while workflow modeling expresses the control and data flow from one service to another. We introduce an additional aspect, distribution pattern modeling [193]. These distribution pattern models are re-occurring distribution schemes that express how a composed system is to be assembled and subsequently deployed. Patterns express proven techniques, which make it easier to reuse successful designs and architectures [73]. Having the ability to model, and thus alter the distribution pattern, allows an enterprise to configure its systems as they evolve, reducing maintenance overheads. Different distribution patterns realise different non-functional requirements, which are documented in our catalog of distribution patterns. We use UML 2.0 Activity diagrams to model the distribution patterns. The UML provides for improved comprehensibility of solutions as it is a standards based notation.

Our framework is based upon models that drive code generation. These models are generated based on existing Web service interfaces, requiring only limited intervention from a software architect, who identifies the distribution pattern, to complete the model. This approach obviates the need for low level composition glue coding. All the models used with the context of the framework are formally defined and placed within an MDSD context. The models themselves are defined using the Eclipse Modeling Framework (EMF) [39]. Relations and transformations are defined between these models to enable code generation. We use the QVT (Query/View/Transformation) language [133] to define the relations, before expressing the transformations using the Atlas Transformation Language (ATL) [96]. The framework is guided by a methodological framework whose primary artifact is a distribution pattern model and whose output is an executable system. This methodological framework is accompanied by a tool that implements the approach.

To assess the usefulness of our solution we have evaluated our approach by comparing it to a traditional handcrafted approach. Central to this evaluation is our consideration of the ALMA (Architecture Level Modifiability Analysis) method [90, 142]. We also provide a critical comparison of our approach to existing tools and methodologies.

1.5 Outline of this Document

The structure of this thesis is as follows. Chapter 2 presents a review of the technologies, approaches and terminology central to Web services and specifically our model driven approach. Quality attributes, which enable the technologies and approaches to be assessed, are also discussed. In Chapter 3 we present the state of the art in software patterns, distributed compositions and model driven development approaches. These related works provide context to our research and expose gaps in the existing literature. Our research contribution begins in Chapter 4, where we present our modeling and transformation framework. This framework, based upon Model Driven Software Development (MDSD), enables the generation of Web service compositions, based upon a set of models, which describe the distribution scheme of a composition. The five components of the framework are outlined in detail in the following five chapters. In Chapter 5 we introduce the first component, a catalog of distribution patterns, which is the basis for our modeling approach. These patterns describe re-occurring distribution schemes, which express how a composed system is to be assembled and subsequently deployed. Chapter 6 discusses the modeling infrastructure required to support our modeling and transformation framework. This component consists of eight languages, or notations, describing the constructs of the various models built within the context of the framework. In Chapter 7 model relations are presented. These model relations provide the semantic mappings between the modeling notations, defining the web of dependencies that must hold between source and target modeling notation. Chapter 8 introduces model transforms. The model transforms define how a source model is converted into a target model, whilst respecting the relations previously defined. The final component of the modeling and transformation framework is outlined in Chapter 9. This methodological framework component ties together the four previous components to ensure that non-functional attribute quality control is no longer an afterthought of the Web service composition generation process. This is achieved by using distribution pattern models as the driver for the executable system generation effort. In Chapter 10 we evaluate how well our approach has met its objectives and compare our efforts to existing approaches and tools. Finally, in Chapter 11 we present our conclusions and consider future work.

Chapter 2

Background

2.1 Introduction

Software engineering is a rapidly evolving discipline featuring a myriad of technologies and approaches. These technologies and approaches attempt to manage the complexity inherent in today's software systems. In this chapter, we review the technologies, approaches and terminology central to Web services and specifically our model driven approach. Our discussions present the technologies incrementally, motivating each as it is encountered. Along with discussing the relevant technologies we also consider quality attributes, which enable the technologies and approaches to be assessed by profiling different approaches.

Initially in Section 2.2 we review relevant software architectural approaches, including design patterns and Service Oriented Architectures (SOA). In Section 2.3 we discuss Web service technologies, before presenting semantically enhanced Web service technologies in Section 2.4. Section 2.5 presents the state of the art in software modeling approaches. Finally, in Section 2.6 we review the quality attributes of software systems.

2.2 Software Architecture

Software architecture considers the structure of a system, and how each of its discrete parts work together, in a given environment. The IEEE define the concept succinctly, as follows;

"Architecture is the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution." [167]

Software architectures provide many benefits to the software development life cycle. The benefits include early verification of systems, by examining the system for correctness and completeness, as well as abstraction of complexity using views, increasing the comprehensibility of a system. Early consideration of the software architecture enables the estimation of a system's quality measures. The architecture can then be tailored to improve the desired quality measures [26, 47, 74].

Architectures often feature abstract components to manage the complexity of describing systems. This technique can also be used to expose pieces of software for reuse and subsequent composition, as in Component Based Software Development [173]. The internal structure of such components can become quite complex. This complexity can be reduced through the use of design patterns or architectural styles. Architectural styles may be considered to be small pattern languages [6]. These patterns or styles are akin to a sub-architecture, that encourages the reuse of well thought out and validated architectural approaches.

Distribution of developed components is essential so that reuse is facilitated. These distributed components expose their functionality using interfaces, possibly in combination with a registry, along with well known or standardised protocols and messaging formats [173]. There are many approaches to component distribution. One such emerging approach, that encourages the use of distributed components is Service Oriented Architectures (SOA).

Each of these architectural approaches is considered in more detail in the following sections.

2.2.1 Component Based Software Development

Component Based Software Development (CBSD) is concerned with the architectural design of reusable software. These pieces of software, accessed using well defined interfaces, are built with reuse, in different contexts, in mind. Components reduce redundancy by promoting reuse, and promote productivity and quality, whilst reducing production costs [173]. Examples of CBSD technologies used in industry include, Microsoft's Component Object Model (COM) [173] and Sun's Enterprise Java Beans [122].

2.2.2 Design Patterns

Design patterns provide software solutions to re-occurring problems that present themselves in software development, with the objective of making them more maintainable, self-documenting and reusable [73]. Patterns are often documented, along with related patterns, as catalogs. These catalogs describe the problem context to which the patterns can be applied, how the pattern solves the design problem and document the consequences of using a given pattern.

There are two forms of patterns, architectural patterns and core patterns. An architectural pattern focuses on the components that make up a system, and how these components work together. One such pattern, sometimes called a paradigm is the Model-View-Controller (MVC) [21]. The MVC provides for clear separation of concerns within a systems architecture.

Architectural patterns are a form of architectural style. Architectural styles are cataloged by Shaw et al. in [162]. The catalog considers patterns which regularly occur in system designs. Each pattern features a set of element types, such as software components, a topological layout of the elements indicating their relationships, a set of semantic constraints outlining how the component will function, and finally a set of interaction mechanisms, which specify how the elements coordinate through the specified topology [26]. An example of an architectural style is the layered pattern, which groups together tasks at a particular level of abstraction. We consider architectural patterns in more detail in Section 3.2.1.

The second form of patterns is core patterns. There are three subsets of such patterns; creational, structural and behavioural.

Creational patterns abstract the instantiation process, whereby the process of creating an object is delegated to another object. This process reduces the coupling between the object creator and the object that it wishes to use. An example of a creational pattern is the abstract factory pattern, which delegates the instantiation of an object.

Structural patterns consider different mechanisms to enable class and object composition. These patterns enable independently developed libraries to work with each other. An example of a structural pattern is the adapter pattern, which adapts an object's interface so it is compatible with another object's requirements.

Behavioural patterns consider generic algorithms and communications mechanisms between objects. The patterns all abstract away the complexity of run-time control flow. An example of a behavioural pattern is the iterator pattern, which provides sequential access to a collection of related elements.

2.2.3 Distributed Systems

A distributed system consists of a number of physically independent computers working together as a single coherent system. The distribution of resources should be transparent to the end user, providing for improved scalability and fault tolerance because there is no longer a single point of failure within a system where redundancy exists. Improved re-use of heterogeneous resources, such as components, and ease of evolution are also provided for by the distributed system paradigm [174].

A system consisting of a number of discrete components can utilise the benefits of a distributed system by partitioning its components onto a number of physically independent computers. Communication between these components is achieved over a network, using well defined protocols, assuming appropriate pluming code is written, or generated by a tool. These components are referred to as distributed components.

2.2.4 Service Oriented Architecture

There are many conceptual architectures for distributing components, one such architecture is Service Oriented Architectures (SOA) [61]. An SOA encapsulates and subsequently exposes application logic through an interface that can be interacted with using a common or standard communications protocol. SOAs maximise loose coupling and reuse by extending the benefits of Object Oriented Programming (OOP) and CBSD without actually relying on these mechanisms [173]. In fact, unlike OOP and CBSD based integration approaches, SOA does not rely on proprietary middleware platforms such as Microsoft's .NET [147] or Sun's Enterprise Java Bean (EJB) [122] technologies. SOAs instead communicate based on an agreed interface and communication contract that is language and platform independent. Web services, discussed in the following section, are an example of an SOA implementation.

2.3 Web Service Technologies

Web services are pieces of software functionality, available at a certain location, that can be invoked using Web based technologies [9]. The basic technology stack for first generation Web services is WSDL, SOAP and UDDI. Web Service Description Language (WSDL) [184] is an XML based language for defining the interface of a given Web service along with its input and output message types. Simple Object Access Protocol (SOAP) [186] is an XML based language used for structured and typed message exchange between service providers and clients. Finally, Universal Description, Discovery and Integration (UDDI) [127] is a Web service based registry used to publish and subsequently discover Web services. Figure 2.1 outlines the relationship between the technologies with respect to the W3C's Web service architecture framework [185].

Web service toolkits such as Apache Axis [13] facilitate the generation and execution of Web services. The following sections outline some of the additional technologies or specifications necessary for Web services to be useful in the context of an enterprise system.

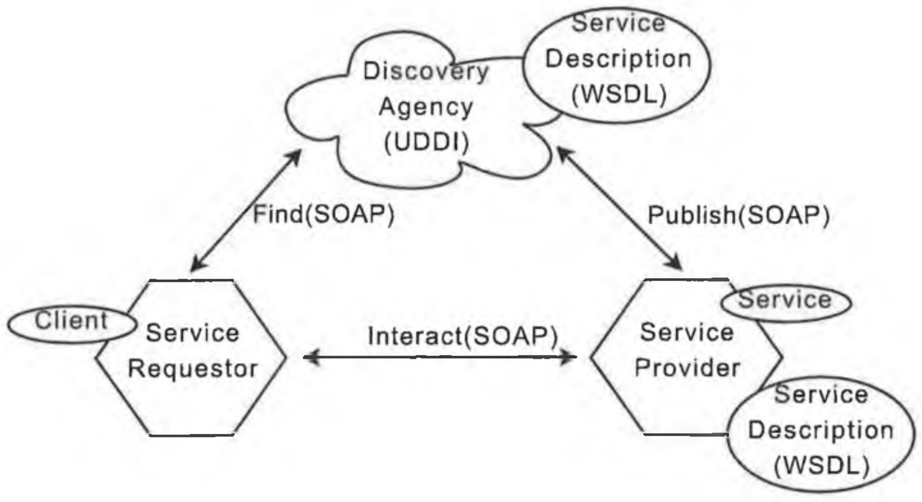


Figure 2.1: Logical representation of a Web service architecture [185].

2.3.1 Web Service Composition

Enterprise systems are often built by wiring a number of Web services together to realise some novel functionality. This practice of gluing Web services together is termed composition. Web service composition is often ad-hoc, where no architectural models are drawn, and considerable low level coding effort is required for realisation [9]. The emphasis of such compositional developments is to gain integration between services, rather than to achieve certain non-functional requirements, such as Quality of Service (QoS). There are many different ways in which these novel applications can be assembled to express varying QoS values, which are often not considered.

We consider a collaboration to be the high level cooperation of components to achieve some compound novel task. For example, a number of Web Services working together to achieve a goal, different to their discrete goals, may be considered a collaboration. A collaboration is the coming together of a number of elementary Web Services to form a complex Web Service.

When considering Web Service compositions, services comprised of two or more elementary services, the area of collaboration, or more specifically, orchestration and choreography are of great importance. Figure 2.2 illustrates an orchestration based model, while

Figure 2.3 illustrates a choreography based model. Although these terms are often used interchangeably there are subtle differences between the two mechanisms.

Chris Peltz [144] defines both terms well;

> "Orchestration refers to an executable business process that may interact with both internal and external Web Services"

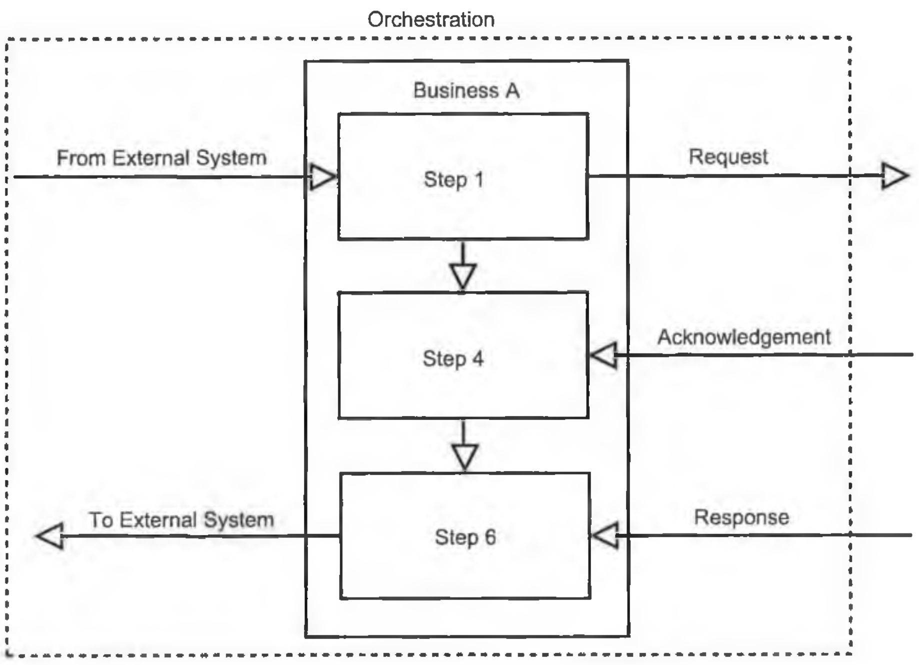


Figure 2.2: Orchestration models the internals of a private process.

while choreography is defined as;

> "More collaborative in nature, where each party involved in the process describes the part they play in the interaction."

The main difference between the two mechanisms is that orchestration takes the perspective of one of the businesses partners and how Web Services messages interact from its point of view, while choreography has no central partner where messages are exchanged between mutually significant partners. Additionally it may be said that a choreography is a user specific execution plan, while an orchestration is fixed based upon some set of business rules [42].

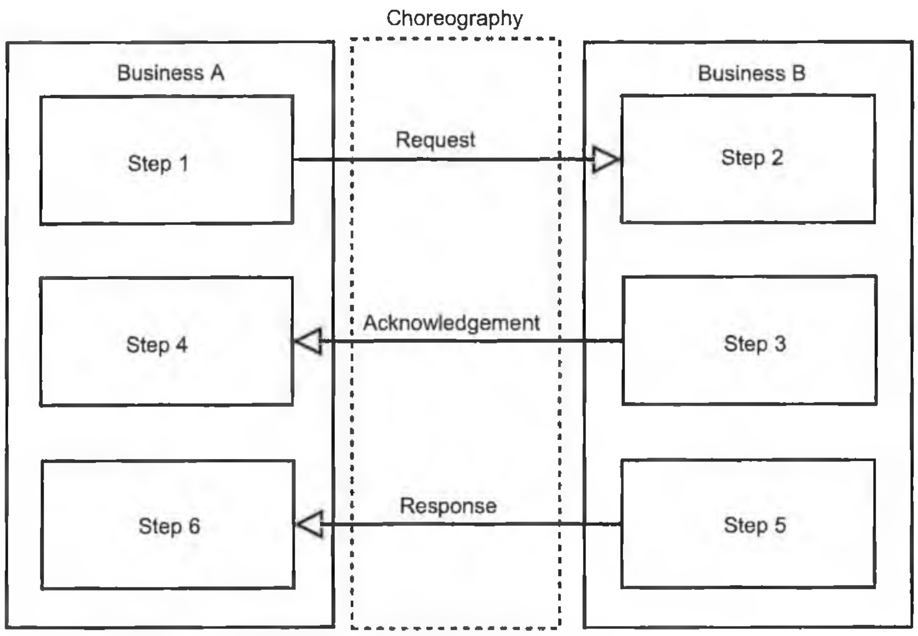


Figure 2.3: Choreography models only public message exchange.

2.3.2 Collaboration Standards

Second generation Web service technologies address the deficiencies of first generation Web services, such as transaction support, composition support, reliability and security, by proposing a number of specifications to address these concerns [175]. Collaboration specifications are considered second generation Web service technologies, which enable the composition of a number of discrete Web services. The two main collaboration languages are Web Services Business Process Execution Language (WS-BPEL) and Web Service Choreography Description Language (WS-CDL) [145]. WS-BPEL [12, 128] provides XML constructs for describing the interaction logic for participants in a complex process flow. This is an orchestration language because the internal and external flow of messages between services in a composition is modeled. WS-CDL [187] on the other hand, is a choreography language describing only the external messages exchanged between collaborators in a composition. Critical evaluation of collaboration standards is investigated by van der Aalst et al. in [179]. Within the Web service community considerably more support is provided for the WS-BPEL language. The supremacy of WS-BPEL is probably due to the fact it was standardised before WS-CDL, with support from large industrial companies such as IBM and Microsoft. These corporations also built early prototype tools to encourage early adoption of their standard.

2.4 Semantic Web Services

The Semantic Web consider how the currently human centric Web can be re-orientated towards automatic consumption and comprehension by computers. By marking up Web pages using commonly defined concepts that are comprehensible by computers, computers can process information without human intervention. This idea can be extended to Web services, where Web services are described using commonly defined concepts that are comprehensible by computers. These services can then be consumed and composed automatically on our behalf by computers. The technologies required for the realisation of this goal are discussed in the following sections.

2.4.1 Semantic Web Ontologies

The Web Ontology Language (OWL), is a language for capturing the conceptual data of a domain and their inter-relationships, for use in the description of resources [113]. OWL extends both XML and the Resource Description Framework (RDF), providing for the creation of ontologies relevant to any given domain [150]. These ontologies describe the vocabulary for a given domain. This technology enables the conceptualisation of a domain, or its semantic description, to describe any form of resource, such as Web pages and Web services. Such approaches can be used to assist in the automatic composition of discrete Web services. Semantic descriptions enable unambiguous, computer interpretable documentation of resources. Authoring tools, such as Protégé, assist in the creation of ontologies by providing an editor based GUI, validation facilities and querying support [169].

2.4.2 Semantic Web Service Description

OWL-S is an ontology based on OWL, which is used for defining the properties and capabilities of Web services in a more descriptive and machine comprehensible manner than WSDL, thus enabling automatic Web service compositions[109, 110]. The ontology is separated into three distinct subontologies or parts.

- Profile
- Process Model
- Grounding

The profile describes the capabilities of the service and is more descriptive than its WSDL counterpart. Its main function is to advertise what the service does for the purpose of discovery by clients, which wish to utilise its capabilities. The second subontology is the process model, which describes how a service works. It is used to enable Web service integration and composition. Finally, the last part is the service grounding, which provides a description of how a service can be accessed. The grounding provides a link between the semantically marked up process model and the service's WSDL interface description document. In fact, both technologies are complimentary in that OWL-S provides an unambiguous, ontology based description of the abstract types used in a service, whilst WSDL provides an XML Schema based description of the abstract types necessary for invocation. Both the profile and the process model refer to a domain ontology to allow for semantic mappings between the discrete service interfaces.

Some OWL tools useful for assisting in the creation and use of OWL-S documents are the OWL-S Editor [59] and OWLSM [112]. A European based project with the same aim as OWL-S is the Web Service Modeling Language (WSML) [195].

2.5 Modeling Technologies

Models are formal or informal representations of a system, often described using a graphical notation. Informal models are used to guide the development of systems by describing the artifacts within a system and how they interact. These informal models are often derived by software architects and then may be passed to software developers for realisation in code. Formal models, however, directly contribute to code generation in that they describe the system in a machine comprehensible way, enabling code generation [71]. These models may also be used for analysis and reasoning of the quality measures of the proposed solution.

Models that can be used to generate code are part of the Generative Programming (GP) software engineering methodology [50]. Generative Programming is useful when families of similar software systems are to be built, albeit with a different set of configurations. This approach splits the software development cycle in two. The first cycle is developing a system to enable reuse. This system consists of all the infrastructure required to model and generate a family of similar software systems. The second cycle is development with reuse, where the specific model of the system to be built is used with the previously defined infrastructure to generate a concrete system.

The following sections detail some of the modeling technologies which assist in model based generative programming efforts.

2.5.1 Meta Object Facility

The Meta Object Facility (MOF) is a standards based, universal mechanism for describing different kinds of modeling constructs [71]. MOF is standardised by the Object Management Group (OMG) [137]. This technology enables the formal description of any number of modeling languages, such as UML, discussed in Section 2.5.2. MOF enables the use of different modeling constructs for different modeling domains. This approach obviate the need for an all encompassing single modeling language. In fact, MOF itself is modeled using a restricted subset of one of the languages which it defines, UML [161].

2.5.2 Unified Modeling Language

The Unified Modeling Language (UML) is a standards based graphical language for the modeling of software systems [60]. The UML is standardised by the Object Management Group (OMG) [140], whose meta-model is defined using MOF constructs. This modeling mechanism enables the precise and unambiguous description of software systems. Ambiguities in the description of software systems can result in invalid system realisations [161].

UML specifies, constructs and documents a system using two categories of diagrams, structural and behavioural. Structural diagrams describe the static structures of a system along with the inter-relations between components of the system. Behavioural diagrams describe the dynamic behaviour of a system. Structural diagrams such as UML Class diagrams are appropriate for service modeling, while behavioural diagrams like UML Activity diagrams are appropriate for modeling both workflows and distribution patterns.

Tool support for UML is considerable. Commercial tools include Rational Software Architect [86], Poseidon [76] and MagicDraw [87]. Open source tools are also available, such as Eclipse UML2 [65], ArgoUML [178] and Dia [77].

2.5.3 Object Constraint Language

The Object Constraint Language (OCL) is a standards based formal language, based on mathematical set theory and predicate logic, for the description of invariant conditions upon a model [138, 190]. OCL is standardised by the Object Management Group (OMG). OCL expressions may be used to query the current state of a model. The language may also be used to assert constraints and/or query any MOF based model. Constraints which cannot be expressed diagrammatically can be expressed using OCL. The combination of UML and OCL can provide for semantically rich conceptual models, similar in expressiveness to ontologies expressed in semantic languages such as OWL [54].

2.5.4 Behaviour Modeling

The UML Activity diagram is a member of the UML behavioural diagram category. Activity diagrams illustrate the sequential flow of actions within a system, capturing actions and their results [11, 60]. These diagrams consist of actions, which are the basic unit of behaviour within an activity, and control flows, which illustrate the transitions through the system. Activity diagrams have a number of constructs, however we will only discuss a subset of these constructs, that have relevance to our pattern catalog.

The start point of an Activity diagram is identified by solid filled circle, termed the initial node. Actions are used to represent something that is performed to produce a result. These actions appear as rectangles with rounded corners. Edges are used to represent transitions between actions and are illustrated by arrows. These edges are triggered by the completion of actions. Activity partitions, also called swim lanes, are used to group actions together. The partitions may be used to explicitly illustrate where an action is performed. Finally the end point of an Activity diagram is illustrated by a circle surrounding a smaller solid circle, termed the activity final point. Figure 2.4, illustrates a UML Activity diagram and labels each of the features discussed.

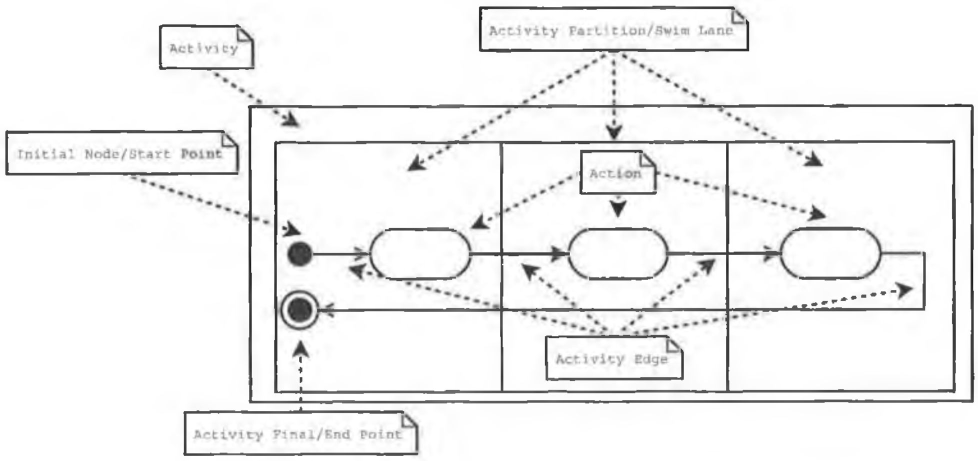


Figure 2.4: A UML Activity diagram.

2.5.5 Extending UML

The UML provides a number of mechanisms for extending the existing language. The UML can be extended using MOF, to create additional modeling constructs. However, the simplest method to extend the UML is the use of UML profiles [71]. Profiles allow, through the use of stereotypes and tag definitions and constraints, the extension of existing UML constructs so they may be utilised in previously unimagined contexts. Although profiles extend UML they must respect the original semantics of the extended constructs. Additional semantics can be applied to the extended constructs using constraints.

A stereotype defines a new modeling construct based upon a previously defined construct, similar to the object oriented extends mechanism [60]. The stereotype has all the features of its parent construct, in addition to some context specific semantics. Stereotypes are identified by the use of guillemets around the stereotype name. The use of stereotypes ensures the UML does not become over complex, enabling the reuse of existing modeling constructs in many different contexts.

Tag definitions enable the assignment of name/value pairs upon UML constructs [60]. The UML comes with predefined tag definitions. However, user defined tag definitions can be created and assigned to stereotypes.

A profile defines a number of stereotypes and associated tag definitions, which when applied by the software architect to UML constructs, allow for the assignment of context specific data to the tag definitions. The OMG maintains a number of profiles, such as the UML Profile for CORBA [130] and the UML Profile for EDOC [132].

An alternative to extending the UML is to define an entirely new modeling language, or Domain Specific Language (DSL), using MOF. This approach obviates the need for the end user to understand UML. Instead novel, potentially more intuitive notational constructs can be used. However, the end user will have to understand the new modeling notation, and prior knowledge of the UML may be wasted. The use of many diverse DSLs can cause fragmentation issues, which the UML was designed to alleviate [161].

2.5.6 Model Driven Development

Model Driven Development (MDD), or Model Driven Software Development (MDSD), is an emerging approach for building software [182, 30]. MDD considers models, at different levels of abstraction, as the primary artifact to reason about a given domain and devise a solution. Relationships are defined between these models to describe the web of dependencies between the models. These relationships are used to assist in the generation and reasoning of the final solution.

One specific MDD based approach is the Model Driven Architecture (MDA) [71], proposed by the Object Management Group (OMG) in November 2000. The approach stipulates models as the primary software artifact, instead of traditional procedural based code [160]. These models are used to abstract away the complexity of a solution. Previously models were used merely to guide the development process. MDA, however, considers models as formal specifications of the structure or function of a system, where the modeling language is in fact the programming language, and is used to generate the program code via conversion rules. Transformation between models can be performed automatically using predefined transformation rules.

Rich, well specified, high level models, often defined in the Unified Modeling Language (UML), allow for the auto-generation of a fully executable system based entirely on models [60]. The combination of raised abstraction and increased automation should increase the quality of software, by exposing the important attributes of the models early in the development process, and result in increased productivity in the development lifecycle [160].

The MDA consists of a four layer modeling stack, as illustrated in Figure 2.5 [31, 32]. From the bottom layer upwards, the initial layer is the real system, which is represented by a model. The model is an abstract representation of the system from a particular perspective. This model conforms to a meta-model, which itself conforms to a meta-meta model. UML, Business Process Modeling Notation (BPMN) and Common Warehouse Metamodel (CWM), are all examples of meta-models in MDA terminology. Meta-models define the vocabulary that may be used in specific models. To avoid the definition of incompatible meta-models a further model layer is defined, the meta-meta model. The meta-meta model is defined using another OMG standard, the Meta Object Facility (MOF), discussed in Section 2.5.1. MOF is a universal way of specifying meta-modeling languages such as UML and BPMN using common MOF constructs, providing interoperability between different meta-modeling vocabularies.

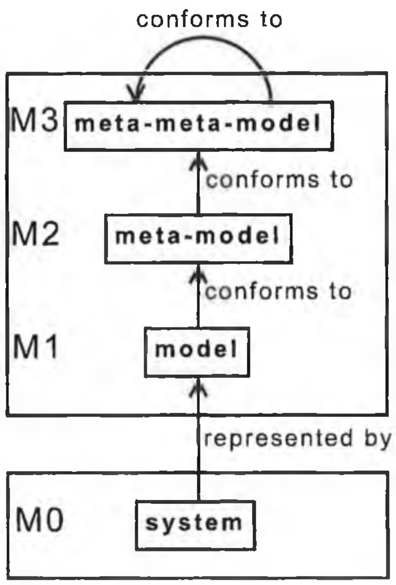


Figure 2.5: OMG four layered modeling stack [32].

In addition to the modeling stack, MDA models describe a system from a number of levels of abstraction, or views. Firstly a Computation Independent Model (CIM) describes what the system is supposed to do, rather than describe the means with which the system achieves its goals i.e. CIM represent's the systems business model. A Platform Independent Model (PIM) describes the system's specification without making reference to any particular platform dependent technology. Finally, a Platform Specific Model (PSM) describes how a system is to be realised using a particular technology [101].

There is considerable tool support for the MDA approach. The Eclipse EMF project [39], a plugin to the Eclipse development environment [64] discussed in detail in Section 2.5.7.2, supports the creation of meta-models, and also the editing of models. The Eclipse based UML2 project [65] is an EMF based representation of the UML 2 meta-model, enabling the use of UML and EMF together. Traditional CASE based tools (Computer Aided Software Engineering), such as IBM's Rational Software Architect [86] are being upgraded to support the MDA approach. Additionally a European based project, Modelware [91], has helped provide a number of MDA based tools, such as the model transformation tools ATL [96] and SmartQVT [176], along with a tool integration suite called ModelBus [168].

2.5.7 Model Transformations

Model transformations enable the transformation of one representation of a system to another. For example, a system may be defined using one form of notation known to business modeling specialists, while the software engineers require the use of a different notation. These different models are however related, and so it should be possible to transform at least some of the model artifacts from one model to another. Transformations are based upon mathematical relations. These relations are sets of elements, where each element in a set is mapped to another element of the other set. Relations are specified at system design time and ensure a mapping between candidate models is possible. A formal definition of relations, including a case study, can be found in the work of Akehurst [5]. There are a number of transformation mechanisms available, some of which are outlined below. A transformation from a candidate source model, Ma, to a candidate target model, Mb, is illustrated in Figure 2.6. Further classification of model transformation approaches is provided by Czarnecki et al. in [51], and more recently in [52]. In the following subsections we consider a number of transformation platforms as well as supporting tools and technologies. Frameworks that implement model transformations are presented in Section 3.10 and compared in Section 3.11. We do not consider the theoretical foundations of the transformations themselves.

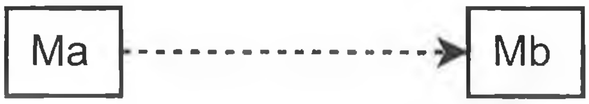


Figure 2.6: Generic transformation of source model to target model.

2.5.7.1 XML Metadata Interchange

XMI or the XML Metadata Interchange Format is a serialisation format for models [80]. The XMI specification standardised by the OMG enables models of any level to be represented using XML [134]. This format enables the persistence of models in a tool independent format. It also enables the transformation of a model from one format to another, using languages such as the Extensible Stylesheet Language for Transformations (XSLT) [183]. XSLT is a declarative, template based language, for transforming one form of XML into another form of XML. Although this approach to model transformation is well established it is not particularly effective, due to the number of incompatible XMI versions utilised in tools, resulting in versioning problems. The XSLT code used to generate such transforms is also very difficult to maintain as it is at a low level.

2.5.7.2 Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF) is an open source modeling framework that integrates with the Eclipse development environment [64]. EMF relates user drawn models directly to their implementations enabling Java code generation [39]. EMF supports XML Schema, UML (via XMI) and Java representations of models, and interchange between these formats. Meta-models in EMF are defined in ECore, which is closely related to the OMG's EMOF, or Essential MOF, which is a lightweight subset of the MOF standard [39, 137]. Like EMOF, the meta-model for ECore defined meta-models, is the meta-model language itself, it this case ECore. A number of ECore meta-models are available for use in EMF, including UML2 [65] and XML Schema [68].

EMF provides wizards out of the box, which enable the transformations from and to XML Schema, UML, Java and ECore representations of a model. This is made possible as EMF comes complete with ECore representations of XML Schema, UML and Java meta-models. This facility is illustrated in Figure 2.7.

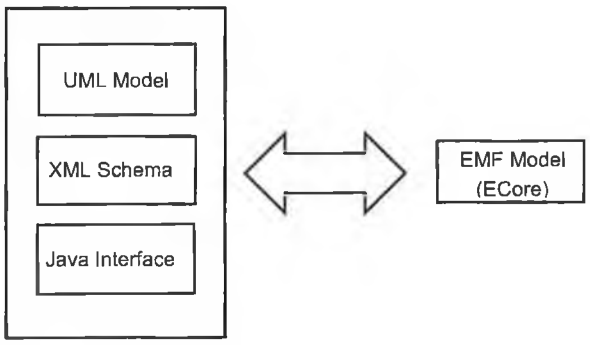


Figure 2.7: Interchange between different modeling formats in EMF.

2.5.7.3 Java Emitter Template

The Java Emitter Template (JET) is EMF's transformation language, capable of generating Java code from ECore based models [149]. JET is a declarative template language similar to JSP (Java Server Pages) [172] that is capable of emitting code. Like JSP, JET may also incorporate procedural code blocks. The operational context of EMF and JET is illustrated in Figure 2.8. The JET language has no formal relationship with ECore.

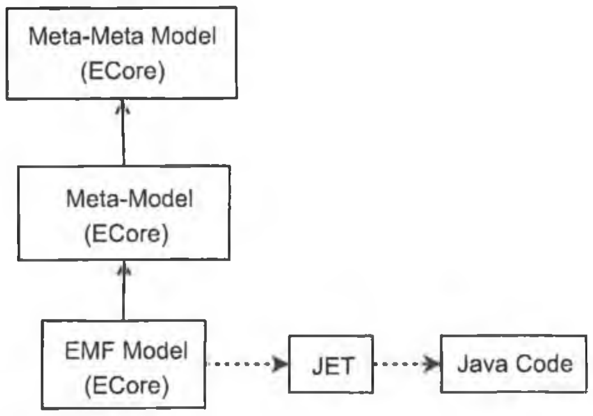


Figure 2.8: EMF/JET operational context.

2.5.7.4 Model Transformation Framework

The Model Transformation Framework (MTF) is IBM's prototype model transformation language, developed as part of their participation in the QVT standardisation process [56]. The language uses simple relations to define the mappings between models. Relations are defined in MTF using a declarative language, called Relations Definition Language (RDL). A closed source transformation engine, supporting the MTF language is available. The tool integrates with the Eclipse development environment [64]. The operational context of MTF is Illustrated in Figure 2.9.

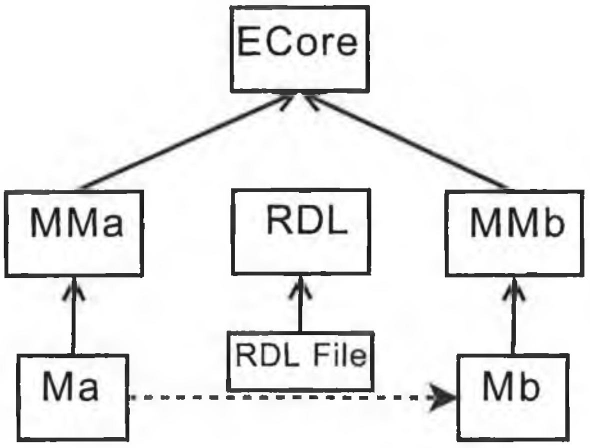


Figure 2.9: MTF operational context.

2.5.7.5 Model Transformation Language

MTL (Model Transformation Language) is a declarative model transformation language, devised by the QVT-P (Query/View/Transformation Partners), featuring two distinct sub-types of transformations, relations and mappings [15]. Relations, although not executable, enable consistency checking between two or more candidate models, ensuring a mapping is possible between the models. Mappings define the transformation between models, by refining any number of relations. These mappings are executable, in that they are capable of creating and altering a model. MTL defines a syntax for both relations and mappings, in both textual and graphical formats. Pattern matching, like that used in XLST, is utilised for searching through candidate models. No tool support for executing MTL implementations is available.

2.5.7.6 ATLAS Transformation Language

ATL (ATLAS Transformation Language) is a model transformation language featuring both declarative and imperative constructs, which conforms to the MOF meta-meta-model [96]. The language and related open source tool, ATL Development Tools (ADT), have no reliance on XMI. Instead, matched rules are used to define source and target patterns in the models to be transformed, resulting in easy to read and maintain transformations. Called rules and actions blocks may be used in addition to matched rules where necessary. The operational context of ATL is outlined in Figure 2.10.

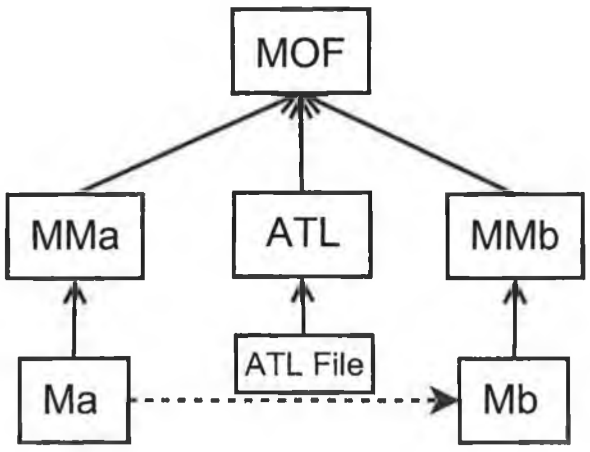


Figure 2.10: ATL operational context [96].

2.5.7.7 Query/View/Transformation Language

QVT (Query/View/Transformation) is a recently standardised OMG language utilised in the MDA approach, for expressing model transformations. QVT, like ATL, is defined using a MOF based meta-model, as illustrated in Figure 2.11.

The specification describes a hybrid language featuring both declarative and imperative parts, as outlined in Figure 2.12. Relations and Core declaratively specify the relationships between MOF models, expressing the same semantics, albeit on different abstraction levels. Operational mappings imperatively describe more complex relations which cannot be described declaratively. The black box represents a plugin, which defines relations coded in a

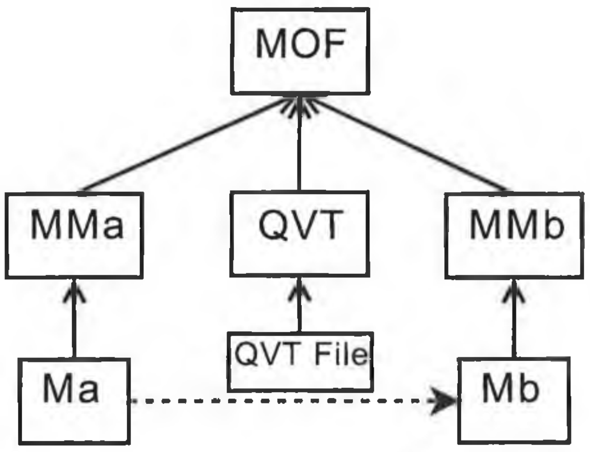


Figure 2.11: QVT operational context.

language with a MOF compliant binding. Unlike ATL, at the time of writing QVT does not have tools that support all the meta-model representations. However, one tool SmartQVT has just been released to support the transformation of operational relations [176]. A useful comparison of the architecture of both ATL and QVT is discussed in [95].

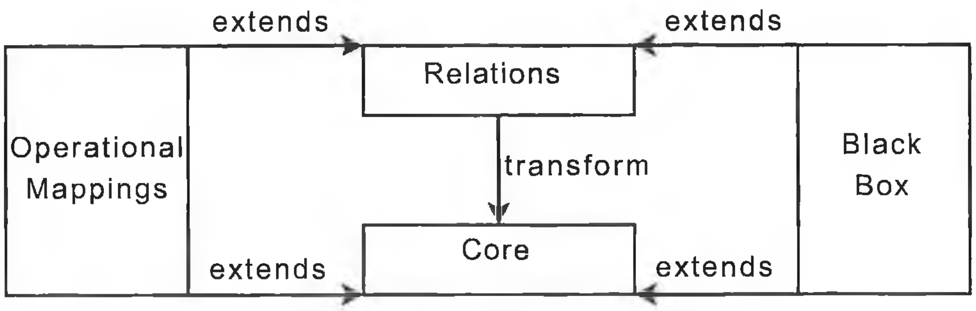


Figure 2.12: The relationship between QVT meta-models.

2.5.7.8 UML Model Transformation Tool

The UML Model Transformation Tool (UMT) is an open source model transformation, and code generation tool [153]. Unlike the other transformation languages discussed here, UMT performs transformation on XML Metadata Interchange (XMI), using an existing language, XSLT. An intermediate language, termed XML_Lite, is introduced to reduce the complexity of the transformation process. XSLT transformations can be both declarative and procedural. The transformations in UMT have no meta-model basis, nor any specific transformation language. The operational context of UMT is outlined in Figure 2.13.

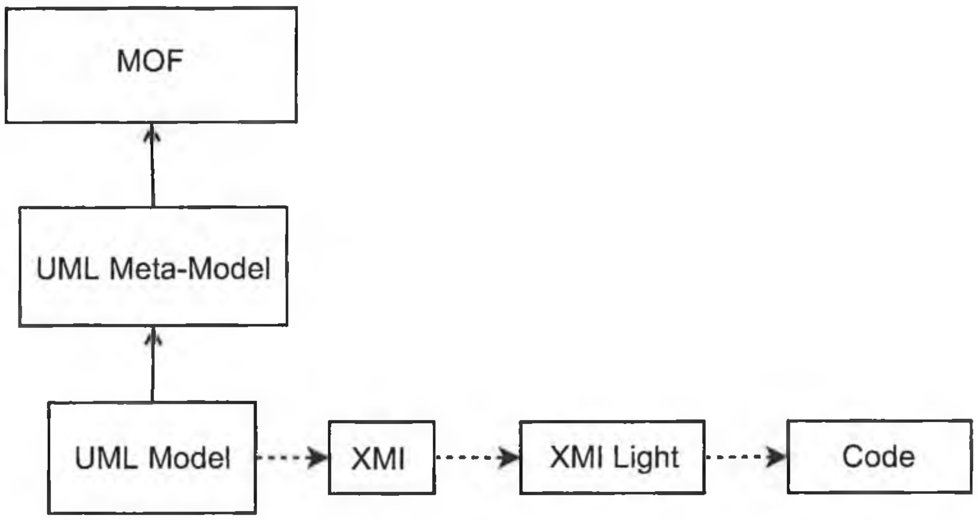


Figure 2.13: UMT operational context.

2.6 Quality Measures

Different system architectures realise different quality measures. In fact, it is often considered that the system's architecture determines the quality of a system [47]. This is particularly the case for distributed systems, where resources must be shared and competed for. Quality measures of software systems are documented using a number of attributes, which are expressed as either quantitative or qualitative values. These values represent the systems ability to satisfy the attribute's requirements as specified by the project stakeholders. Ideally, software systems should express their non-functional attributes with the least amount of ambiguity possible. Quality measures consider both non-functional design-time and run-time measures of software systems.

A number of barriers to achieving acceptable quality values for non-functional system attributes are noted by Albin in [6]. These common barriers are:

- Quality control as an afterthought

- Underestimation or misunderstanding of importance of quality measures
- Inadequate modeling methods and languages for expressing quality measures
- Lack of documented design and architecture patterns for addressing quality measures
- Difficulty in designing for quality measures
- Inadequate languages for expressing and specifying quality requirements

Bass et al. [26] do not distinguish between functional and non-functional requirements. Instead, they consider the qualities of a system as observable via execution and non-observable via execution. An example of an observable attribute would be system efficiency, while non-observable qualities would include the mutability of the system. We consider both forms in the following Sections 2.6.1 and 2.6.2. These criteria will be used later in the evaluation of our framework.

2.6.1 Observable Quality Measures

Observable quality measures may be observed during the execution of a system and are often grouped under the term, Quality of Service (QoS) [72]. In [26] Bass et al. five quality attribute categories which affect software systems at run-time are outlined; performance, security, availability, functionality and usability. The first two categories are of particular relevance to distributed systems. These important attributes are outlined below and detailed in [116, 135, 196].

- Performance - The timeliness with which a system can react to requests.
- Availability - A measure of the time the system is up and running.

These run-time non-functional properties can be aligned to the ISO 9126 software product evaluation standard as follows [88].

- Performance - ISO 9126 efficiency with regards to time and resource utilisation.
- Availability - ISO 9126 reliability with regards to fault tolerance.

2.6.2 Non-Observable Quality Measures

Non-observable quality measures cannot be observed during the execution of a system. These attributes are design-time quality measures. Five quality measure categories which affect software systems at design-time are outlined by Bass et al. in [26]; modifiability, portability, reusability, interoperability and testability. The first four categories are of particular relevance to distributed systems, and are outlined below.

- Modifiability/Mutability - Ease of modification of the system in terms of cost and effort.
- Portability - Ease of altering the system's context in terms of cost and effort.
- Reusability - Ease of reuse of a system's components.
- Interoperability - Ability of system to integrate with others.

These design-time non-functional properties can be aligned to the ISO 9126 software product evaluation standard as follows [88].

- Modifiability/Mutability - ISO 9126 maintainability with regards to analysability, changeability, stability and testability.
- Portability - ISO 9126 portability with regards to adaptability, installability, conformance and replaceability.
- Reusability - ISO 9126 portability as above.
- Interoperability - ISO 9126 portability as above.

2.7 Summary

In this chapter we have introduced all the technologies used throughout the thesis. We presented architectural approaches to software engineering, from Component Based Software

Development (CBSD) to Service Oriented Architectures (SOA). The basic Web service technology stack, along with Web service composition terminology and technology, were discussed before considering technologies, such as OWL-S, which enable automatic Web service consumption and comprehension. We motivated the use of formal models for the generation of code and introduced some of the important MDD related technologies and reviewed a number of approaches of model transformations. Finally, we presented some of the important quality attributes of software systems, which will be considered throughout the thesis.

Chapter 3

Related Work

3.1 Introduction

This chapter surveys the body of work related to software patterns, distributed compositions and model driven development approaches. These related works provide context to our research and expose gaps in the literature, which we will address. Each piece of related research will be considered with respect to its relevance and significance. We will also outline the important contributions and limitations of each presented approach.

The history of software design patterns is presented in Section 3.2, before looking in detail at one specific form of pattern, architectural design patterns in Section 3.2.1. In Section 3.3 we introduce early attempts to model distributed systems and consider two different notations for modeling such systems, before considering some modern Web application based modeling solutions in Section 3.4 and Web service modeling approaches in Section 3.5.

Traditional distributed system based compositions are investigated in Section 3.6 before examining the field of Web service compositions in Section 3.7, as well as attempts to automate the composition process in Section 3.8. Modeling of the non-functional attributes of systems is investigated in Section 3.9. Section 3.10 discusses a number of different approaches to model transformations. Finally, in Section 3.11 we summarise the features provided by all the solutions considered in this chapter.

3.2 Design Patterns

In the late nineteen seventies Alexander et al. proposed a pattern language for the design of buildings and cities [7]. These patterns represented problems and successful solutions to re-occurring architectural situations, such as city planning, they repeatedly encountered. The solutions they found and documented can be reused in different contexts. This theory is the basis of the work of Gamma et al. who apply patterns to object-oriented systems [73]. Instead of documenting patterns such as road based T junctions, Gamma et al. document patterns such as adapters and factories, which are often used in object-oriented systems. Classification of patterns is discussed in Section 2.2.2.

Many authors have proposed pattern libraries for other, more specific, software engineering contexts. Vasko et al. [181] document Web service workflow patterns in the Collaxa workflow engine, while van der Aalst et al. [180] document advanced workflow patterns. Buschmann et al. consider design patterns in a number of contexts in their Pattern-Oriented Software Architecture series [41].

3.2.1 Architectural Design Patterns

Design patterns, which express the architecture of a system, are often visible in system models. These patterns are examples of architectural design patterns or architectural styles [47]. Architectural design patterns are at a higher level of abstraction than core patterns, and so are not tied only to object oriented contexts. When architectural patterns are visible it is possible to predict the architectural quality of such systems, based on known attributes of systems with the same or a similar architecture style [47]. Architectural quality has a direct impact on the non-functional requirements, including both observable and non-observable attributes of a solution, as outlined in Section 2.6.

One particular architectural design pattern of interest, in the context of Web service compositions, is the topology. A number of distributed system topologies, or topological layouts, are identified by Ding et al. in [58]. Topologies, in network computing terminology, are the different interconnecting structures, or architectural schemes, by which pairs of nodes in the network are connected. The topologies presented by Ding et al. express the architecture of a distributed content sharing network, and are based on existing networks based topologies. The topologies presented are classified by Minar in [119], where a framework for comparing the characteristics of distributed system design is presented. For the remainder of this chapter we will refer to topologies as architectural schemes. The classification of the various architectural schemes is important as it enumerates the various way in which a distributed system can be organised. However, the classification performed by Minar does not consider their specific use in a Web service context.

There has been some work in decoupling the architectural scheme from the rest of an applications implementation. This approach enables the architectural scheme of a solution to be altered without affecting the remainder of the system, thus enabling different observable and non-observable attributes to be met after the completion of the system. We discuss some examples from the literature in the following paragraphs.

Architectural evolution of distributed systems is considered by Ranno et al. in [156]. Here, the authors consider two schemes; the distributed scheme and the centralised scheme, and note how the scheme chosen affects the architectural qualities of the system. Two particular features of architectural quality, fault tolerance and ease of administration of the system, are specifically noted. A related paper [164] notes how the scheme can affect an additional architectural quality measure, the dependability of a distributed system, as a single point of failure exists on centralised schemes. Performance trade-offs in relation to message exchange overhead, for different schemes, are also considered. The solution is based on a CORBA based composition and does not consider a Web service compositional context. Additionally the solution is restricted to two distribution schemes.

Distribution schemes for Web based contexts are considered by Web-ML [43, 37], and discussed in more detail in Section 3.5. Here, Web based business processes that span multiple computational nodes are modeled. The solution supports both centralised and decentralised process distribution, enabling the software architect to realise different architectural quality measures, depending on the customers requirements. The solution is based on a Web service compositional context, albeit restricted to only two distribution schemes.

Further motivation for alternative distribution schemes, specifically in a Web service compositional context, are provided by Sheng et al. and Chen et al. in [163, 46]. Here, scalability, availability and security problems of centralised schemes are noted as serious issues. Specific performance data for centralised vs decentralised execution are provided by Benatallah et al. in [29]. Here, the physical message count is reduced when executing in the decentralised mode. The execution time for the decentralised execution was also found to be more efficient, regardless of message size. Similar results were found by Caituiro-Monge et al. in [42] and Liu et al. in [108]. The authors conclude the decentralised scheme is generally superior in performance, with respect to response time and aggregated cost.

Woodman et al. [193, 194] note that different distribution schemes provide different levels of autonomy, further motivating the consideration of distribution schemes. Autonomy is an important non-functional consideration for businesses with sensitive information they do not wish to share with their operational partners, for security or other reasons. The authors coin the term, distribution patterns, to describe the architectural scheme of a composition. We use this term later in this thesis when discussing distribution schemes.

The advantages of asynchronous messaging, used in decentralised distribution schemes, is further explored by the IBM Symphony project [44, 126]. Here the authors look at composition of web services and propose a decentralised scheme rather than the more traditional centralised scheme. As already noted different schemes realise different non-functional requirements. A centralised scheme is where one coordinator node exists and it is responsible for coordinating all the data and control flow between the composite services. To provide for decentralisation the authors partition the program into a number of smaller components distributed at different locations. Here the composition engines communicate directly with each other reducing bottlenecks, network traffic and improving transfer time, concurrency and throughput. These gains do however come at the cost of increased complexity, with regards to error recovery and fault handling, not to mention potential deadlocking if incorrectly designed.

3.3 Distributed System Modeling

Distributed systems, discussed in Section 2.2.3, are a collection of independent computing resources that work together transparently to achieve a given goal [174]. The aggregation of a number of computing resources is termed a composition. Modeling distributed systems is an important activity as it provides a mechanism that can be used to assist in the visualisation of the system from a number of aspects. These models help manage the complexity inherent in distributed systems, and to make good design decisions before the system is implemented. Distributed system modeling is considered by Arief et al. [17]. The authors note that plain text descriptions of systems are often unintuitive and don't express complexity well. Architects usually imagine the architecture of a system from a graphical viewpoint. Textual descriptions are not conducive for this process. Instead, Architecture Description Languages (ADLs) such as Darwin and Rapide are considered [114] along with the Unified Modeling Language (UML) [140], as solutions for managing the complexity of such systems. Arief et al. do not consider a Web service context specifically, but their approach to complexity management by modeling is appropriate to Web service compositions.

In ADL distributed systems are modeled from a high level in terms of components and how these components are connected. UML, is a general purpose software engineering notation, which describes systems from many different aspects and levels of abstraction. ADLs are designed to only express software architectures, while UML may be used in many software engineering contexts such as networking and workflow representations. It may be argued that architectures expressed as ADL are more easily comprehended, by software architects, than UML, as the abstractions used are closer to the architects mental image of the system architecture [114]. However, UML has far more elaborate tools than ADLs, and its modeling notation is better supported in the community. UML can also be easily extended to represent many forms of system architectures. Arief et al. note how UML based diagrams could be used in conjunction with modeling and analysis tools to enable analysis of the modeled distributed system. A related paper [16] considers how UML models of system simulations can be used to generate an executable simulation. We will use the UML notation throughout this thesis as our modeling notation of choice.

3.4 Web Application Modeling

Modeling Web applications is considered by Meliá et al. in [115]. Here a number of view-points of a Web application are modeled with a UML based tool, ArgoUWE, based upon a UML Web engineering meta-model, called UWE. The approach is also extended to model business processes in [100]. UWE, unlike the approach taken by Arief et al. in Section 3.3. The authors utilise a standards based Model Driven Development (MDD) approach, as outlined in Section 2.5.6. Transformations between meta-models using a similar approach are outlined, this time using QVT, by Koch et al. in [125]. The approach does not consider a Web service context or modeling of distribution schemes specifically, but the authors use of the MDD modeling process in general is appropriate to Web service compositions.

3.5 Web Service Modeling

Web services, discussed in Section 2.3 are a specific form of distributed system, that can be invoked using Web based technologies [9]. As with distributed systems, modeling of Web services is an important activity, assisting in the management of complexity and also the auto-generation of simple or complex Web Services.

Modeling of Web service interfaces is considered by Provost in [151]. Here UML is used to represent the Web service interfaces usually described in verbose WSDL documents. The work is motivated by the difficulty in modeling service interfaces using an XML based language. Grønmo et al. expand upon this work by investigating the possibility of modeling service interfaces from a platform independent perspective. An XSLT based tool called UMT [153], capable of generating WSDL language definitions, is provided to support their technique. The UML models created, to represent the Web service interfaces, may then be used as a basis for defining composition models. These solutions consider a Web service context specifically, using UML and an MDD based approach. However, they do not consider compositions and subsequently do not consider the modeling of distribution schemes.

3.6 Composition Modeling

Often the compositional model of a distributed system's business rules is represented as a workflow model. This model can be used to abstract away the complexity of how the discrete components, in a composition, connect to each other. Often the compositional model is a mix of how the services should connect to each other, as well as a representation of the business rules, as a workflow. As with distributed systems in general, these workflows model the flow of information and tasks through the system, as well as the connections between the compositional participants. All the related work presented here models compositions from an orchestration perspective, or business rule workflow point of view, using different orchestration distribution schemes. None of the literature examined consider model compositions from only a choreography perspective, using different choreography distribution schemes. One framework presented does however motivate the need for the two modeling perspectives, and presents a high level approach to achieving this objective without considering a formal modeling or code generation approach. The difference between these two perspectives is outlined in detail in 2.3.1. Previous to compositional modeling approaches, distributed systems were rigid in architectural structure, resulting in difficult to maintain, and evolve, distributed systems.

However, solutions such as the enhanced CORBA run-time environment OPENFlow [81], based on the work in [156, 164], address these deficiencies by employing a dynamic architecture capable of assembling and connecting tasks which represent the discrete parts of a business workflow at run-time. Instead of hard coding the connections and the control flow between the various participants, a GUI based tool, not based on UML, is provided to model the system assembly. These systems empower the architect to alter the distribution scheme of the distributed system, without making low level code changes, based upon the non-functional requirements of the system. The solution supports both centralised and decentralised orchestration distribution schemes, but does not consider a Web service compositional context, nor does it utilise the MDD modeling process.

3.7 Web Service Composition Modeling

Web Service compositions, discussed in Section 2.3.1, consider the aggregation of a number of Web services to achieve some goal. Web service compositions are often rigid in architectural structure, resulting in distributed systems that are difficult to maintain and extend. A number of composition frameworks or environments, outlined below, provide solutions to these problems. One such environment is DECS [193], a Web service based workflow management system, which defines elementary services as tasks whose execution is managed by a coordinator at the same location. The solution is based on OPENFlow [81], described in Section 3.6. Like OPENFlow the system supports changing the distribution scheme at runtime, without having to manipulate low level code. The system supports both centralised and decentralised orchestration distribution schemes. However, unlike OPENFlow, no GUI based model support is provided. This means neither UML models nor an MDD based modeling approach is utilised. Instead, XML documents are used to model the process description.

Net Traveler, proposed by Caituiro-Monge et al. [42] is another Web service integration framework. The framework proposes an XML based control document that enables the system to operate in either a centralised or decentralised distribution scheme, based upon an orchestration and a choreography model. However, no code generation scheme is presented to generate code from these high level models. As with DECS, neither UML models nor an MDD based modeling approach is utilised.

Web service composition modeling is considered by Benatallah et al. using the SELF-SERV [163] tool, which proposes a declarative language for composing Web services based on UML 1.x statecharts. Statecharts are called state machines in UML 2.0 terminology [60]. After the statecharts describing the composition have been completed they are converted to an XML file. This declarative modeling approach should enable the fast and scalable definition of Web service compositions, avoiding the usual time intensive, volatile approach to composition definition [28]. SELF-SERV provides an environment for visually creating a UML statechart which can subsequently drive the generation of a proprietary XML routing table document. Pre- and post-conditions for successful service execution are generated based on the statechart inputs and outputs. Related papers [29, 27] provide performance measures to support the use of the decentralised orchestration distribution scheme instead of the centralised orchestration distribution scheme. Here, we see fewer messages are exchanged in a decentralised environment while execution time is also reduced for larger message sizes. The authors' more recent work [98] considers the conformance of services, with a given conversational specification. The approach takes statecharts, representing the workflow, as its input and is capable of outputting WS-BPEL process documents.

A similar environment to SELF-SERV, called Peer-Serv, is considered by Wang et al. [189]. Like Self-Serv, the authors utilise a decentralised distribution scheme for service execution. However, the authors also use the same decentralised orchestration distribution scheme for service publication and querying, improving availability, scalability and performance.

Also from the composition modeling perspective Grønmo et al. [152, 53] consider the modeling and building of compositions from existing Web services using MDD, based on approach outlined by Thöne [170]. It is noted that XML based representations of compositions are useful as a means of universal representation and exchange. However, XML is not easily comprehensible to non experts and so a more intuitive graphical representation, such as UML, should be used. The authors consider two modeling aspects, service (interface and operations) and workflow models (control and data flow concerns). Their modeling effort begins with the transformation of WSDL documents to UML, followed by the creation of a workflow engine-independent UML 1.4 Activity diagram (PIM), which drives the generation of an executable composition. Additional information required to aid the generation of the executable composition is applied to the model using UML profiles. Modeling of the distribution scheme is not considered by the authors.

A web application hypertext modeling notation, Web-ML [43], has been extended in [37] to include modeling of business processes. Here, Business Process Modeling Notation (BPMN) [136] modeling elements are used to model workflows in Web service based compositions. These models, combined with hypertext models, which describe the interactions within a Web application, are used to model entire Web service enabled Web applications. A commercial CASE (Computer Aided Software Engineering) tool WebRatio [191] is used for designing WebML Web applications and service compositions. The tool is based on the MDD modeling approach.

A platform specific UML 1.4 based business process model is investigated by Gardner in [75]. Here IBM's Rational Rose (now Rational Software Architect) is used to apply a UML profile to a WS-BPEL based UML Activity diagram. The model is capable of building a completely executable system based on the MDD process, albeit based only on a WS-BPEL workflow, as a platform specific model is used. In [11] Ambühler considers the same model based approach as Gardner, but uses the more recent UML 2.0 meta-model. Neither author considers modeling the distribution scheme of the composition.

The use of UML 2.0 for modeling compositional service specifications is critically assessed by Sanders et al. in [158]. The authors conclude UML 2.0 is very useful for describing collaborations and in turn supporting service composition, however a number of minor UML enhancements are suggested.

3.8 Semantically Enabled Web Service Composition Modeling

Semantics enable semi-automation of Web service compositions, as considered by Sirin et al. [165]. Semantics enrich the description of services, enabling computers to comprehend the functions and ways of interacting with a service. This approach reduces the amount of time the software architect must spend finding suitable, compatible service for a composition, and subsequently assembling them. Services marked up with semantics can be selected and composed based upon both functional and non-functional properties. Functional properties include input and output parameters, whilst non-functional requirements include service description. Two additional systems devised by Timm et al. and Grønmo et al. use MDD based techniques to assist in the creation of ontologies for semantically enriching services which are to be composed [154, 92]. These systems assist the composition effort by combining semantics and MDD approaches to assist in composition development. None of these semantically enriched systems introduced here consider the distribution scheme of the resultant composition.

3.9 Non-Functional Modeling

Non-functional attribute modeling, which assists in the management of the complexity of software solutions, are examined by Gray et al. in [78]. The authors consider how modeling can assist in the rapid evaluation of a system when exposed to changes in a configuration aspect. They consider two possible changes to existing systems, changes that crosscut the application's design and changes to the system so that it can scale up. The authors note that evolving models can be both tedious and error prone. Using this modeling technique many different design decisions, such as choice of communication protocol, can be tested at the modeling stage, and their effects observed and acted upon. The authors propose a model to model transformation language ECL (Extended Constraint Language), and a tool/execution environment C-Saw. Other solutions to automated model transformation and transformation languages, such as ATL (Atlas Transformation Language) [96] and GReAT

(Graph Rewriting and Transformation Language) [4] are also discussed by the authors. ATL is used extensively in the context of this thesis to enable our non-functional modeling efforts. The authors do not consider modeling distribution schemes as a non-functional modeling concern. They also do not consider a Web service compositional context for their modeling technique.

An MDA based approach to modeling the efficiency and reliability of software systems is considered by Cortellessa et al. in [48]. The authors use the MDA Computation Independent Model (CIM), Platform Independent Model (PIM) and Platform Specific Model (PSM) to analyse the efficiency and to assess the reliability of a proposed software system. Experts in the non-functional domain of the system assign estimates of efficiency and reliability cost to the models, which once transformed give an overall computational costs of the software system. Changes can be made to the models if the non-functional properties of the system do not meet the customers requirements.

A comparison of three well known model based efficiency prediction approaches is considered by Koziolek et al. in [102]. The authors specifically compare and contrast the Software Performance Engineering (SPE), Capacity Planning (CP) and umlPSI (Performance Simulator) prediction methods. As with Cortellessa et al. above, estimates of efficiency and reliability cost must be assigned to the SPE and umlPSI models. The CP method requires a system to already exist before it can be used to estimate the efficiency and reliability cost of changing the system. The results from the comparison show great variance in the precision achieved by each method. The CP method was the best prediction, however, it requires an already built system, which is not ideal for an model based efficiency estimation. The results from this experiment show the difficulty in accurately predicting non-functional properties at design time.

Other useful non-functional modeling approaches are the use of UML profiles to assign estimates of efficiency and reliability costs to models. Three such profiles are the OMG's UML Profile for Schedulability, Performance, and Time (SPT) [131], the OMG's UML Profile for Modeling Quality of Service and Fault Tolerance [139] and an extension to both of these profiles termed, Modeling and Analysis of Real-Time and Embedded systems (MARTE) [62]. All these profiles allow quantitative evaluation of systems based upon estimates of efficiency and reliability cost applied to the profile.

3.10 Model Transformations

Model transformations, discussed in Section 2.5.7, enable the creation of a new model based upon a previously defined model. Different models provide different views of the same system. Transformations enable the creation of these alternate views, as well as enabling code generation based upon models. Compositional glue code, for different distribution schemes, can be auto-generated based upon models.

The DECS [193], Net Traveler [42] and SELF-SERV [163] run-time frameworks, rely on XML to model their workflows. These tools then use this model, as a lookup table, for dispatching messages to the next composition participant. No explicit transformations of the XML model are performed.

However, UMT discussed in Section 2.5.7.8, and devised by Grønmo et al. [153] is a UML based transformation tool. The tool is capable of many transformations types including, UML models to text, text to UML models, as well as UML model to UML model. Examples of the transformations included with the tool include, WSDL to UML, UML to WSDL, UML to WS-BPEL and UML to ECore. UMT utilises the XSLT language to perform transformations on models, and is reliant on specific XMI version compatibility to work. This is not ideal as different vendors export different XMI versions limiting the initial tool selection and subsequently making inter-tool compatibility difficult. Creation and maintenance of numerous XSLT transformations for different XMI versions can be time intensive and debugging may be error prone.

Web-ML [43] and its associated tool WebRatio [191] utilises XSLT transformations to transform an XML based representation of WebML models to executable code wrappers [37]. These wrappers enable the execution of the business process model using traditional programming platforms such as .Net [147], J2EE [122] and Struts [14]. The tool does not suffer from XMI version problems, however it does require the exclusive use of the WebRatio modeling tool, resulting in tool lock-in. The problems of using XSLT outlined when discussing UMT are minimised by locking in the end user to only one tool.

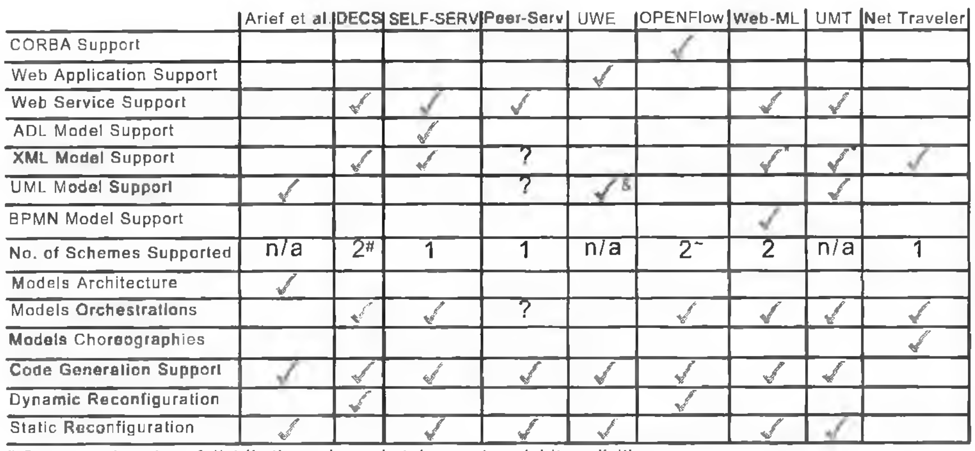
Gardner at al. transform UML Activity diagrams to WS-BPEL/WSDL models using EMF [75], discussed in Section 2.5.7.2, based APIs (Application Program Interface). Reflection is used to walk the source model, creating target models which are capable of serialising themselves as the required executable system artifacts, WS-BPEL and WSDL in this case. This code based transformation approach is not open source and so is not open to direct scrutiny. However, imperative code based approaches using large APIs may be difficult to comprehend unless the reader has an intimate knowledge of the programming environment. Often the transformational details are lost amongst lines of supporting code. Declarative approaches such as XSLT are often simpler to code and subsequently maintain. The "top down" approach used by Gardner at al. is compared and contrasted to the "bottom up" approach, used by Microsoft, in [146].

Ambühler considers a similar approach to Gardner at al., transforming UML Activity diagrams to WS-BPEL/WSDL models. However, the author uses the IBM based Model Transformation Framework, discussed in Section 2.5.7.4, to implement model transformations between source and target models. This approach is an improvement on the imperative code based approach of Gardner at al. MTF is a declarative language resulting in easier to code and subsequently maintain transformations.

Finally, Bauer and Müller [19] propose a mapping from UML 2.0 sequence diagrams to a WS-BPEL workflow. The authors use an MDD based approach to drive the software development process and provide an example of a platform independent model being transformed to a platform specific model. No executable system is generated using this approach, nor is any model transformation language considered.

3.11 Framework Comparison

A comparison of the frameworks presented throughout this chapter are illustrated in Figure 3.1. The figure provides a matrix of the features of each of the frameworks. The features were chosen to illustrate the major differences between the frameworks. The features express a number of aspects of the frameworks including: the domain of interest to the framework, the type of modeling supported, the number of distribution patterns supported, what is being modeled by the framework, whether code generation support is provided or not, and finally, whether the framework supports static or dynamic reconfiguration. This comparison will provide the basis for the evaluation of our modeling and transformation framework in Chapter 10.



	Arief et al.	DECS	SELF-SERV	Peer-Serv	UWE	OPENFlow	Web-ML	UMT	Net Traveler
CORBA Support						✓			
Web Application Support					✓				
Web Service Support		✓	✓	✓			✓	✓	
ADL Model Support			✓						
XML Model Support		✓	✓	?			✓*	✓*	✓
UML Model Support	✓			?	✓&			✓	
BPMN Model Support							✓		
No. of Schemes Supported	n/a	$2^{\#}$	1	1	n/a	$2^{\sim}$	2	n/a	1
Models Architecture	✓								
Models Orchestrations		✓	✓	?		✓	✓	✓	✓
Models Choreographies									✓
Code Generation Support	✓	✓	✓	✓	✓	✓	✓	✓	
Dynamic Reconfiguration		✓				✓			
Static Reconfiguration	✓		✓	✓	✓		✓	✓	

# Supports changing of distribution scheme but does not model it explicitly
~ Distribution scheme is defined in the orchestration
* XML support is via XMI
& UWE uses a conservative extension to UML called WebRE
n/a Feature not explicitly considered
? Not clear from paper

Figure 3.1: Comparison of frameworks.

3.12 Summary

In this chapter, we have reviewed the literature related to software patterns, distributed compositions and model driven development approaches. We identified that architectural design patterns, specifically distribution schemes, originally cataloged for a networking context can be applied to a Web service compositional context. Different distribution schemes realise different non-functional quality attributes, such as dependability, autonomy and performance, as noted in the referenced literature.

The use of modeling notations to describe distributed systems has been motivated. Models help to manage the complexity inherent in distributed systems, and to make good design decisions before the system is implemented. Two modeling notations were contrasted ADL and UML. UML was chosen as the preferred notation for this thesis due to its superior tool support and high level of intuitiveness. Models can be used to describe the distribution schemes previously investigated.

Simple approaches for modeling distributed systems have been assessed as well as more complex approaches using the MDD modeling process for modeling Web applications and simple Web service interfaces. Modeling the compositions of both non Web service and Web service based distributed systems have been investigated. We have noted that none of the modeling approaches found in the literature model compositions from a choreography perspective, with the exception of Net Traveler, which models both choreography and orchestration perspectives. However, this framework lacks a formal modeling, or code generation approach which all other frameworks feature. From the orchestration perspective, frameworks such as OPENFlow, SELF-SERV and DECS, are still relevant as a guide to modeling distributed systems. A number of MDD based modeling approaches were presented, along with a number of semantically assisted approaches to Web service compositions, which reduce the workload of the software architect or modeler.

Finally, non-functional modeling is motivated by considering the benefits of complexity management, such as ease of evolution and early design verification. Model transformation mechanisms, such as ATL, are motivated as a means to enable the creation of alternate views, which represent different non-functional models of the system. These transformations also enable code generation based upon models.

Chapter 4

Modeling and Transformation Framework

4.1 Introduction

In this chapter we introduce our modeling and transformation framework. The framework is based on the Model Driven Development (MDD), or Model Driven Software Development (MDSD) approach [182, 30]. The goal of this framework is to enable the generation of Web service compositions, based upon a set of models which describe the distribution scheme, from a choreography perspective, of a Web service composition. Models abstract complexity and enable high level reasoning about a solution from an early point in the development life-cycle. Our distribution models support the capture of non-functional quality attributes. The models consider design-time non-observable architectural quality attributes of compositions, like maintainability and portability. Additionally, run-time observable, or Quality of Service (QoS), attributes affected by the chosen distribution scheme, such as efficiency and reliability, are also documented at design time by the selection of a given distribution scheme with known QoS attributes. Together these attributes express the total quality of a composition [6].

In Section 4.2, we consider the problems related to traditional approaches of building

Web service compositions. Subsequently in Section 4.3, we outline our novel approach, including our modeling and transformation framework.

Our framework and its components are also outlined in the following conference papers, [25, 24, 22, 23].

4.2 Traditional Approach to Web Service Compositions

Web service compositions are constructed by composing a number of discrete services. Traditionally such compositions lack architectural models, up to date or otherwise, describing the components and connections between the discrete services. The development of composite Web services in this manner is often ad-hoc and requires considerable low level coding effort for realisation [9]. This effort is increased in proportion to the number of Web services in a composition or by a requirement for the composition participants to be flexible [29]. The result of these approaches is often a fixed opaque architecture.

Without formal models of non-functional quality attributes it is difficult to reason about the architectural quality of traditional compositions. For example, it is difficult to estimate the time and effort that would be required to maintain the composition by adding an extra service to the composition. Run-time attributes are also not visible in traditional Web service compositions. This lack of architectural transparency makes it difficult to assess how the composed system will perform at run-time, with respect to QoS attributes such as efficiency and reliability.

In Section 3.7 we considered a number of run-time environments, or frameworks, which provide solutions to the lack of architectural flexibility in Web service compositions. These solutions assist in the realisation of functional properties of a composition, such as workflows, by using models. However, these solutions are limited to only a fixed number of architectural schemes. Also, these solutions mix workflow modeling with modeling of the distribution scheme, resulting in complex orchestration based models which cannot be used to assess non-functional quality attributes easily. Finally, none of the frameworks considered use a formalised development approach such as MDD.

4.3 Proposed Approach to Web Service Compositions

We propose a model based development and code generation approach to address our goal of enabling the generation of Web service compositions, based upon a given distribution scheme. This approach suggests Web service compositions have three modeling aspects. Two aspects, service modeling and workflow modeling, are considered by Grønmo et al. [152] and reviewed in Sections 3.5 and 3.7. Service modeling expresses interfaces and operations, while workflow modeling expresses the control and data flow from one service to another.

Here, we consider an additional aspect, distribution scheme modeling [193], using a distribution pattern catalog that will be presented in Chapter 5. This catalog expresses a number of different distribution schemes, or patterns, which model how the composed system can be deployed. This catalog is based on network based topologies, as discussed in Section 3.2.1. These models enable the software architect to reason about the problem domain and possible solutions, with respect to non-functional quality attributes, early in the development life-cycle.

Here, we present a modeling and transformation framework, based upon model-driven service engineering, and driven by a distribution model. The UML is used throughout the framework as the conceptual modeling notation because of its widespread usage amongst software architects. The components of the framework are enumerated below, before being outlined and motivated in the following subsections and subsequently described in detail in the following Chapters. Each element of our framework addresses the barriers to achieving quality, as previously outlined in Section 2.6.

- A Catalog of Distribution Patterns - Enumeration of the possible distribution schemes for Web service compositions.

- Modeling Notations - Definition of required modeling notations.

- Model Relations - Definition of the web of dependencies between modeling languages.

- Model Transformations - Rules for transforming between modeling notations.

- Methodological Framework - Outline of how high level distribution models are used for code generation.

4.3.1 A Catalog of Distribution Patterns

The first component of our framework comprises a catalog of distribution patterns, which may be applied by software architects to Web service compositions. Distribution patterns express the distribution scheme of a composed system when it is deployed. Each of the patterns in the catalog expresses certain run-time Quality of Service (QoS) attributes, exhibited during execution of Web service compositions. The catalog enumerates the QoS attributes of each of the patterns, by referencing existing implementations in the literature, enabling the software architect to choose a pattern appropriate to the non-functional quality requirements of a given composition. Each pattern is expressed using the Unified Modeling Language (UML). This component of the framework is discussed in detail in Chapter 5.

4.3.1.1 Discussion

This component addresses a number of the quality control issues outlined in Section 4.2. The catalog provides documented solutions for addressing specific run-time QoS attributes. The catalog also utilises an expressive modeling language, UML, ensuring quality is considered early in the solution design process.

4.3.2 Modeling Notations

The modeling of distribution patterns and subsequent generation of an executable system based upon a model requires some infrastructural grounding. The modeling notation component that enables this generation consists of a number of notations, which describe the constructs of the various models, which feature in our five step methodological framework component, discussed later and illustrated in Figure 4.4. These modeling notations enable the software architect to reason about the problem and solution domain. There are eight such languages or notations, outlined below, illustrated in Figure 4.1 and described in detail in Chapter 6. In MDA terminology these notations may be considered meta-models.

- UML 2.0 Notation
- Distribution Pattern Language UML Profile (DPLProfile)
- Distribution Pattern Language Notation (DPL)
- Collaboration Notation
- Interface Notation
- Deployment Descriptor Notation
- Deployment Catalog Notation
- XML Notation

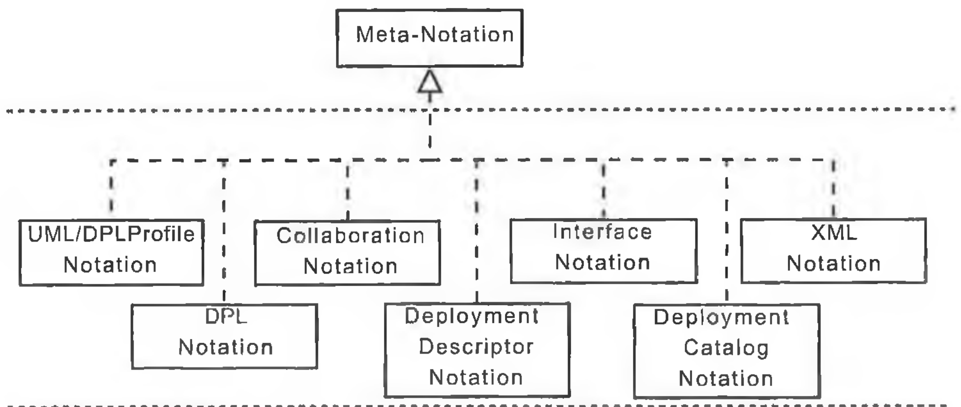


Figure 4.1: Notations used in our modeling approach.

The UML 2.0 notation utilises a subset of the standard UML 2.0 specification [140], and is used to describe distribution patterns using standard UML 2.0 Activity diagram constructs. We also utilise the UML 2.0 Class diagram constructs to represent the interfaces of the discrete Web service participants.

The Distribution Pattern Language UML Profile, or DPLProfile, is our novel extension of the UML, which allows extra distribution pattern specific information to be applied to the model. This profile is not strictly a notation it is more of an extension to an existing notation. Without this profile extension the UML would not be able to adequately describe the distribution scheme possibilities for Web service compositions. The profile extends appropriate constructs of the UML notation.

Together, the UML 2.0 notation and DPLProfile provide the infrastructure necessary for the software architect to model, at step two of the methodological framework, any of the distribution patterns outlined in our catalog, using a number of UML 2.0 based tools. The constructs used in the UML should be familiar to software architects, making this language ideal for defining distribution pattern based compositions. An instance of this notation, or model in MDA terms, is output from step one of our methodological framework, and is further refined by the software architect in step two before being used as the input to step three of the methodological framework.

The Distribution Pattern Language notation, or DPL, provides the constructs for the internal representation of a distribution pattern. DPL instances, from a distribution pattern description point of view, are equivalent to UML 2.0 notation/DPLProfile instances. The goal of DPL is to provide for ease of analysis, verification and transformation of distribution patterns. The notation has no reliance on UML, thus allowing alternatives to the UML modeling notation approach, such as π calculus [118] and Architecture Description Languages (ADL) [114], to be used instead.

In the context of this thesis, the UML is used as the modeling notation due to its wide spread acceptance in the software engineering community, however alternative notations are discussed in Section 3.3. The DPL notation is important because it allows us to reuse steps four and five of our modeling methodological framework, regardless of how the distribution pattern is modeled by the software architect. An instance of this notation is output from step three of our methodological framework.

The Collaboration notation provides the constructs necessary to define a distribution pattern based composition, to be enacted on a composition engine. An instance(s) of this notation could be a representation of a WS-BPEL or WS-CDL document(s), discussed in Section 2.3.1. Instances of the Collaboration notation are an intermediate output from step five of our methodological framework, enabling the realisation of a distribution pattern on a compositional engine. The notation could map to the structures outlined in either the WS-BPEL or WS-CDL specification.

The Interface notation provides the constructs to define a distribution pattern based compositional interface, which is to be exposed as a Web service. An instance(s) of this notation is a representation of a WSDL interface document, discussed in Section 2.3. Instances of the Interface notation are an intermediate output from step five of our methodological framework, and represent an entry point to the execution of the Collaboration notation instance previously generated. The notation maps to the structures in the WSDL specification [184].

The Deployment Descriptor notation provides the constructs to define a distribution pattern based deployment, to be enacted on a composition engine. An instance(s) of this notation is a representation of a deployment descriptor document for a particular compositional engine. Instances of the Deployment Descriptor notation are an intermediate output from step five of our methodological framework. This notation provides the link between Collaboration notation instance(s) and the Interface notation instance(s) previously generated. An example of a specification that the Deployment Descriptor notation could map to, is the PDD deployment descriptor, which is part of the Active BPEL composition engine specification [2].

The Deployment Catalog notation provides the constructs to enumerate the interfaces of a distribution pattern based deployment, to be enacted on a composition engine. An instance(s) of this notation is an enumeration of all the discrete Web service interfaces required by a composition engine. Instances of the Deployment Catalog notation are an intermediate output from step five of our methodological framework. An example of a specification, which the Deployment Catalog notation could map to is the WSDLCatalog deployment catalog, which is part of the Active BPEL composition engine specification [2].

The XML notation provides the constructs to define an XML based document. An instance(s) of this notation is a representation of an XML document. This notation is required because standards based specifications like WSDL and WS-BPEL are defined using the XML notation. The XML notation will enable these notation instances to be represented in an XML compliant notation, assuming appropriate relations are written. The notation maps to the structures in the XML specification [188].

4.3.2.1 Discussion

This component address three of the quality control issues outlined in Section 2.6. We provide adequate modeling languages for expressing quality attributes. This is achieved through the use of a novel UML extension that enables the specific expression of patterns, which realise specific quality requirements. Our approach directly addresses the problem of designing with quality attributes in mind.

4.3.3 Model Relations

The modeling notations component outlined in the previous subsection presented eight different modeling notations. Each of these models describes the composition, or a part of it, from a different aspect. The modeling relations component of our framework defines the web of dependencies that must hold between pairs of candidate notations, a source and a target notation. These candidate notations are called meta-models in MDA terminology. The goal of defining relations between notations is to record the process by which information is related between notations, thus ensuring modeling information is preserved from one model instance to the next. For example, we must show formally how the information in the UML 2.0/DPLProfile notation is related the DPL notation. These relations are considered abstract specifications or constraints, in that they are not themselves executable. We use relations to define abstract relationships rather than to define bi-directional mappings. The relations are used as a template for defining the final component of our framework, Model Transformations, in Chapter 4.3.4.

The modeling relations are described in detail in Chapter 7, and illustrated below in Figure 4.2.

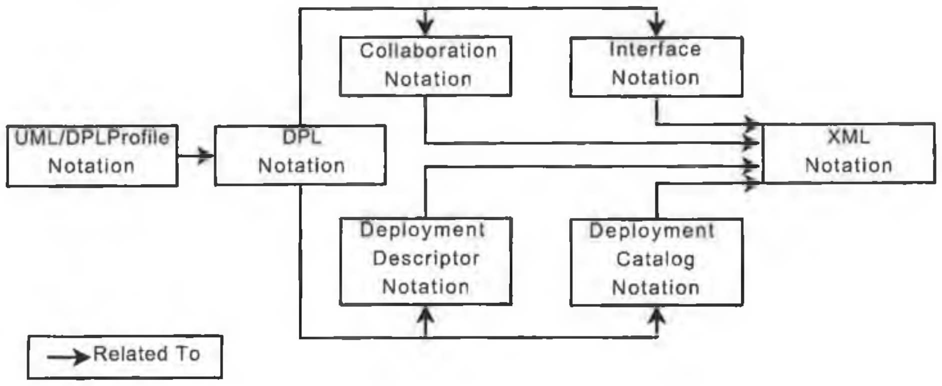


Figure 4.2: Relations defined between notations in our modeling approach.

4.3.3.1 Discussion

This component does not directly address the quality control issues outlined in Section 2.6. However, the modeling relations component provides a record of how the framework will realise the quality attributes expressed using the modeling notation, defined in the previous component. These relations are an essential modeling method to ensure the quality attributes are realised in the generated composition.

4.3.4 Model Transformations

The fourth component of our framework is modeling transformations. We have already motivated the need for eight modeling notations, and discussed the need for the definition of relations between these notations. However, we must also define how a source model, an instance of one of the notations, can be used to generate a target model, that conforms to another notation. These transformations must respect the modeling relation constraints already defined. For example, we must show how an instance of a UML 2.0/DPLProfile notation is transformed into an instance of a DPL based notation. Model transformation languages, discussed in Section 2.5.7, are used to transform a source model to a target model, using declarative statements in the transformation definition, based upon previously defined modeling relations. The modeling transformations are described in detail in Chapter 8, and illustrated below in Figure 4.3.

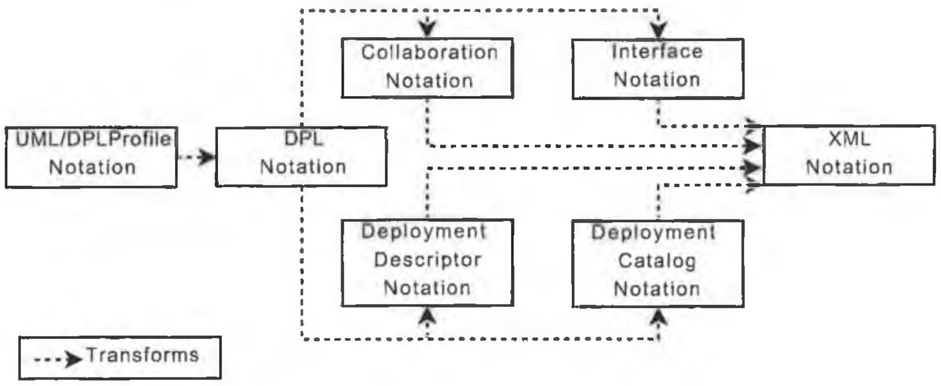


Figure 4.3: Transformations defined between notations in our modeling approach.

4.3.4.1 Discussion

As with Modeling Relations this component does not directly address the quality control issues outlined in Section 2.6. However, the modeling transformations component enables the generation of a composition, based on the quality attributes expressed using the modeling notation. These transformations, like the relations, are an essential modeling method to ensure the quality attributes are realised in the generated composition.

4.3.5 Methodological Framework

The final component of our framework is a methodological framework, which outlines our approach to distribution modeling and subsequent Web service composition generation. The approach is based on the Model Driven Software Development (MDSD) process, where models are taken as the input and code is outputted. This component consists of five steps, taking a number of existing discrete Web service interfaces as input, and outputting an executable Web service composition. The methodological framework relies on the four components, previously outlined. The steps are outlined below, illustrated in Figure 4.4, and subsequently described in detail in Chapter 9.

- Step 1 - Transform Interfaces To UML Model(s)
- Step 2 - Distribution Pattern Definition
- Step 3 - Transform UML Activity Diagram Model to DPL Model
- Step 4 - Validate DPL Model
- Step 5 - Transform DPL Model to Executable System

The methodological framework will be supported by a banking case study, which demonstrates its usage. Tool implementation details will also be provided to outline how we have implemented each step of the methodological framework.

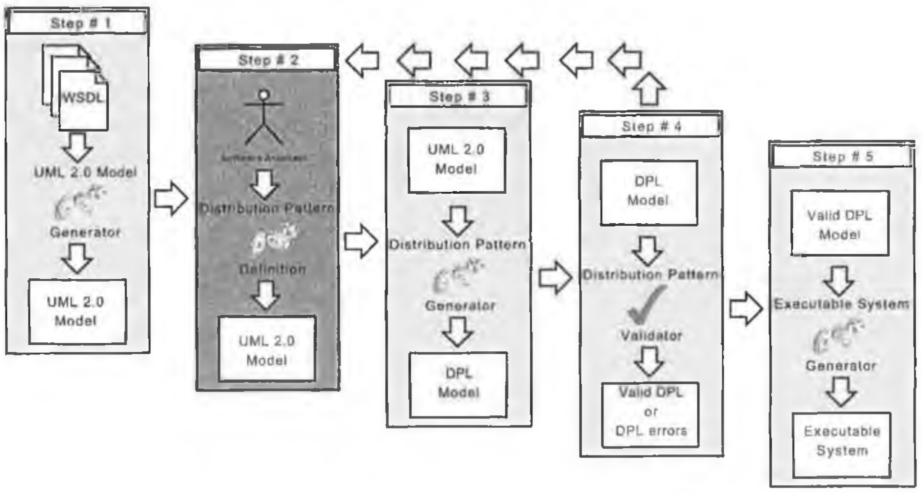


Figure 4.4: Overview of methodological framework.

4.3.5.1 Discussion

This component directly addresses two of the quality control issues outlined in Section 2.6. Our methodological framework provides a modeling method to ensure that non-functional attribute quality control is no longer an afterthought of the Web service composition generation process. This is achieved by using models within the methodological framework, which consider existing implementations that expresses given non-functional quality attributes as the driver for the executable system generation effort.

4.4 Summary

In this chapter, we have motivated the need for a new approach to Web service composition and definition. We identified that traditional Web service composition frameworks, or run-time environments, consider compositions only from an orchestration point of view. These solutions do not consider modeling of the distribution scheme of the composition from a choreography perspective, and as such do not consider non-functional quality aspects from the outset of the development process.

A modeling and transformation framework consisting of five components has been introduced. This framework supports modeling of the distribution scheme of a composition from a choreography perspective. Each of these components has been motivated with respect to non-functional quality aspects and each is essential in our MDSD approach for generating Web service compositions based on a distribution pattern model.

Our modeling and transformation framework utilises a number of concrete technologies, such as UML 2.0, Ecore, QVT and ATL. We have chosen these state-of-the-art technologies to realise our five components as they are all widely used in both the research community as well as industry. In fact, all these technologies are open source and/or freely available. Our use of concrete technologies does not unduly constrain the usefulness of our research as substitute technologies may be used where appropriate. The modeling and transformation framework itself, as well as the pattern catalog and methodological framework, are implementation technology neutral.

Chapter 5

A Catalog of Distribution Patterns

5.1 Introduction

The catalog of distribution patterns presented in this chapter represents the first component of our framework. Patterns express proven techniques, which make it easier to reuse successful designs and architectures [73]. These patterns also capture architectural decisions that assist in documenting systems [84, 18]. Additionally, patterns assist software architects in determining the non-functional attributes of systems before they are built [83].

Distribution patterns, a term coined by Woodman et al. [193], are re-occurring distribution schemes that express how a composed system is to be assembled and subsequently deployed. Each of the patterns in the catalog expresses different non-functional Quality of Service (QoS) attributes. We enumerate the attributes of each of the patterns in the catalog, enabling the software architect to choose a pattern appropriate to the non-functional requirements of a given composition. Case studies are provided for each pattern to demonstrate its usage in a real world scenario. Unlike the related work outlined in Section 3.9, we do not assign estimates of efficiency and reliability cost to the models, instead we refer the reader, where possible, to related works where the costs of a given pattern have been assessed using profiling techniques. Where explicit measure costs are not available the reader is referred to system implementations which are known to expose certain non-functional attributes. Although many of the patterns in the catalog are new to the Web service context, their properties in a networking context are know. We believe this is a pragmatic approach to cost estimation for our measures.

In Section 5.2 we discuss what distribution patterns are in detail. A classification scheme for describing our catalog is presented in Section 5.3, before presenting the catalog itself in Section 5.4. How a software architect would use the catalog is discussed in Section 5.5. Finally, in Section 5.6 we present an evaluation of the catalog itself.

5.2 Distribution Patterns

Distribution patterns are a form of architectural design pattern, discussed in Section 3.2.1. These patterns express how a composed system is to be assembled and subsequently deployed. Buschmann et al. in [40] consider a similar concept, termed a pipes and filter pattern. In our context filters are Web services, and pipes represent the connections between these Web services. The patterns may also be considered as an architectural style family, based upon the pipeline architecture of the dataflow systems style [26, 6]. The pipeline style features a linear sequence of data processors or, in this context, Web services.

There is a subtle difference between two of the modeling aspects within a Web service composition, namely workflows and distribution patterns [193]. Both aspects refer to the high level cooperation of components, termed a collaboration, to achieve some compound novel task [145]. The solutions considered in Section 3.7, do not make this distinction.

We consider workflows as compositional orchestrations, whereby the internal and external messages to and from services are modeled. In contrast, distribution patterns are considered compositional choreographies, where only the external messages flow between services is modeled. Consequently, the control flow between services are considered mutually independent. As such, a choreography can express how a system would be deployed. The internal workflows of these services are not modeled here because there are many approaches to modeling the internals of such services [53, 75].

Distribution patterns are a form of platform-independent model (PIM) [71] because the patterns are not tied to any specific implementation language or technology platform. The patterns identified are architectural patterns, in that they identify reusable architectural artifacts evident in software systems.

5.3 A Classification Scheme for Distribution Patterns

Here, we present a catalog of distribution patterns, which may be applied by software architects to Web service compositions. This catalog is described using a structured format to assist in the comprehension, comparison and utilisation of the presented patterns. The classification template is derived from the work of Gamma et al. in [73]. Like Gamma et al. we document the decisions, alternatives and compromises that occur when selecting a pattern. The pattern's objective properties are defined under the following template headings.

- Name/Structure: Each pattern is named.
- Structure: Each pattern is represented diagrammatically using UML.
- Description: The pattern's structure is described. The description details a number of the pattern's properties as outlined by Gamma et al. including "Intent, Motivation and Known Uses".
- Synonyms : Alternative names for the pattern are identified. Synonyms are denoted by "Also Known As", by Gamma et al.
- Participants: The roles of participants within the pattern are enumerated.
- Related Patterns: Any known small variations upon the pattern are noted.

Each pattern's subjective properties, with respect to non-functional quality attributes are analysed and evaluated using the following criteria.

- Advantages: Motivations for the pattern are provided.

- Disadvantages: Issues regarding the pattern are identified.
- Applicability: Contexts where the pattern should be used are investigated.
- Case Study: A real world scenario where this pattern is/could be used.
- Summary: Search-able keyword based summary of pattern attributes.

The summary uses appropriate non-functional measures as outlined in Section 2.6.2. The value low is considered undesirable, medium less so and high is considered very desirable.

5.4 Distribution Pattern Catalog

The pattern catalog was constructed by systematically researching distribution patterns, in existing network based systems. Many of the patterns discussed here are identified by Ding et al. in [58]. However, their description in a Web service composition context is novel, as is their categorisation, detailed definition and expression using a standardised modeling language, UML. As with other pattern languages the patterns here are either fundamental patterns, termed core or auxiliary patterns, or complex patterns, which are patterns constructed from other patterns in the catalog.

Our catalog is expressed using the UML notation. UML, as outlined in Section 2.5.2 is a standards based graphical language for the modeling of software systems [140]. A number of different UML behavioral diagrams were considered for modeling distribution patterns, specifically Sequence diagrams, State Machine diagrams, Communication/Collaboration diagrams and Activity diagrams [140]. Sequence diagrams and State Machine diagrams were considered too simplistic, in that they do not provide for the modeling of the data that flows through the composition or provide enough constructs to adequately represent the various features of each discrete interface involved in a composition. Communication /Collaboration diagrams are too closely tied to objects to adequately represent distribution patterns that are not tied to object oriented techniques. Our ultimate choice of Activity diagrams was based on their successful use in modeling workflows models, as seen in the frameworks outlined in Chapter 3. Although distribution models and workflow models are different they both express collaborations, albeit from a different perspective. Activity diagrams provide an ideal level of abstraction that provides enough appropriate modeling constructs to enable code generation based on the model.

Unlike other pattern catalogs our distribution patterns are documented as prototypes. This technique enables a software architect to intuitively select a pattern by browsing the generic prototypes we have provided. The architect can then use the catalog as a guide for adapting the selected pattern to their context. A shortcoming of this solution is that architects may misinterpret the distribution pattern prototypes in the catalog as being the only way to apply a given pattern. For example, an architect may assume that because a pattern has x number of nodes in the catalog pattern instance that it is not possible to use this pattern in a scenario with x+1 number of nodes. Although we believe this prototype approach to documenting the patterns is a useful mechanism for software architects, an alternative approach would be the formal representations of patterns as discussed in Section 11.3.1. This approach to pattern modeling would enhance our pattern catalog by describing the discrete parts of the patterns how the actual patterns are constructed.

In the following three subsections we enumerate the patterns we have identified and qualitatively support their use. The patterns are split into three categories: core patterns, auxiliary patterns and finally complex patterns. Core patterns represent the simplest distribution patterns most commonly observed in Web service compositions. Auxiliary patterns are patterns which can be combined with core patterns to alter a given non-functional quality attributes of a core pattern. The resultant pattern is a complex pattern. Complex patterns may also be formed by combining core patterns. We envisage that this catalog will assist software architects in choosing a distribution pattern for a given context. The catalog is outlined briefly below, and in detail in the following subsections.

- Core Patterns

- Centralised
    - Decentralised
- Auxiliary Patterns
    - Ring
- Complex Patterns
    - Hierarchical
    - Ring + Centralised
    - Centralised + Decentralised
    - Ring + Decentralised

5.4.1 Core Patterns

Core patterns are the fundamental distribution patterns most commonly encountered in Web service compositions. We identify two such patterns, Centralised and Decentralised, both of which are used as building blocks within complex patterns. QoS attributes of these patterns are documented by [29, 44, 108, 163, 193].

Pattern 1: Centralised

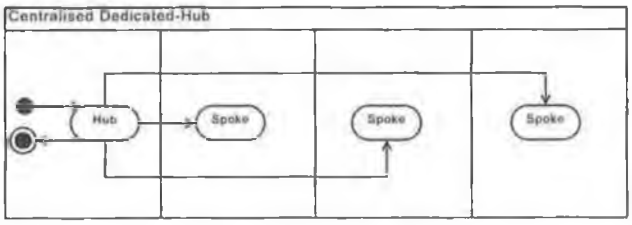


Figure 5.1: Centralised-Dedicated Hub distribution pattern.

Structure:

Description: The Centralised pattern, illustrated in Figure 5.1, manages the composition from a single location, normally the participant initiating the composition. The composition controller (the hub) is located externally from the service participants to be composed (the spokes). Two messages are exchanged between the hub and a spoke for each spoke execution i.e. synchronous communication. The composition completes after the final spoke has completed execution and has returned a response to the hub. This is the most popular, and default, distribution pattern configuration for compositions. An example of this pattern in an existing networking context is a Web browser (the spokes) and Web server (the hub) interaction.

Synonyms: Hub & Spoke, Centralised Dedicated-Hub

Participants

- Hub : Participant which controls the composition execution.
- Spoke : Participant which passively performs some service.

Related Patterns: Centralised Shared-Hub The composition controller (the hub) can be co-located with one of the service participants to be composed (the spokes), as illustrated in Figure 5.2. For this pattern to be possible the service participant to be co-located with must be under the same administrative control as the participant initiating the composition. This scenario cuts down on some of the network latency between the hub and one of the spokes, but at the cost of reduced autonomy for that spoke, along with reduced reliability.

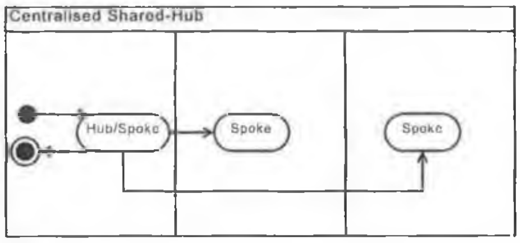


Figure 5.2: Centralised Shared-Hub distribution pattern.

Advantages

- Composition is easily maintainable, as composition logic is all contained at a single participant, the central hub.
- Low deployment overhead as only the hub manages the composition.
- Composition can consume participant services that are externally controlled. Web service technology enables the reuse of existing services.
- The spokes require no modifications to take part in the composition. Web service technology enables interoperability.
- Ease of development, as most Web service composition engines provide all the tools necessary to realise this pattern.

Disadvantages

- A single point of failure at the hub provides for poor reliability.
- The communication bottleneck at the central hub constricts scalability. SOAP messages have considerable overheads for deserialisation and serialisation of messages, emphasising this issue.
- The high number of SOAP messages between hub and spokes is sub-optimal. SOAP messages are often verbose resulting in poor efficiency in a Web service context.
- Poor autonomy in that the input and output values of each participant can be read by the central hub.

Applicability Use the Centralised pattern when

- A high number of concurrent users is not envisaged.
- The system must be built quickly using existing Web service composition engines.

- Composition participants might be changed frequently.
- Composer has no administrative control over composition participants.

Case Study A medical image management system, which uses Web services is outlined in [55]. This system features two Web services which allow neuroscience researchers access and store medical images, as well as to process, analyse and visualise the stored images. The composition of the two services is centrally controlled using WS-BPEL. The distribution pattern expressed by the composition is the centralised shared-hub pattern. From the case study in [55] it is clear that non-functional requirements were not explicitly assessed before implementing the composition, and so no distribution pattern was explicitly chosen by the architects. This situation has resulted in the use of the default distribution pattern, centralised shared-hub. However, this pattern may be ideal for a small composition like this, which is designed to be used only by a select number of clients i.e. neuroscience researchers. An additional case study using the centralised shared-hub pattern for an electronic payment application is considered by Zhang et al. in [197].

Summary: Low efficiency, low reliability, high modifiability/mutability

<u>Pattern 2:</u> Decentralised

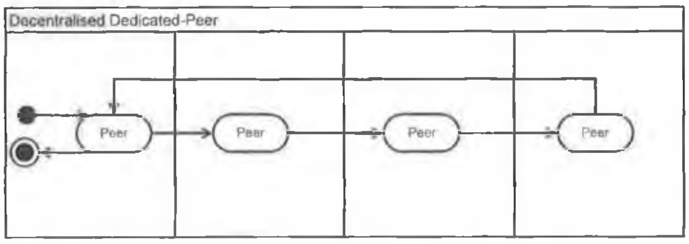


Figure 5.3: Decentralised distribution pattern.

Structure:

Description: The Decentralised pattern, illustrated in Figure 5.3, distributes the management of the composition amongst the participants. The participant which initiates the composition is located externally from the other participants. Only one messages is exchanged between the caller and the callee for each peer execution i.e. asynchronous communication. The composition completes after the final peer has completed execution and has returned execution control to the peer which commenced the composition. An example of this pattern in an existing networking context is Gnutella, a file sharing system [119, 157].

Synonyms: Peer-to-Peer, P2P

Participants

- Peer : Participant which performs some service and also participates in controlling the composition.

Related Patterns: Decentralised Shared-Peer The participant which initiates the composition can be co-located with one of the service participants to be composed, as illustrated in Figure 5.4. For this pattern to be possible the service participant to be co-located with must be under the same administrative control as the participant initiating the composition. This scenario cuts down on some of the network latency between the initiating peer and one of the peers to be composed, but at the cost of reduced autonomy for that peer. Reliability may be reduced slightly because the peer instance will block, consuming some resources, whilst it awaits the remainder of the composition to execute.

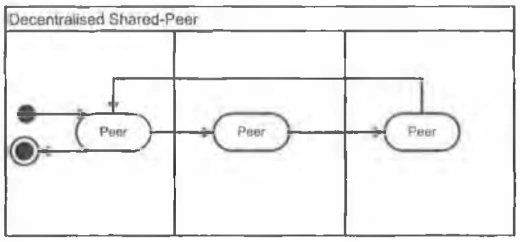


Figure 5.4: Decentralised Shared-Peer distribution pattern.

Advantages

- No single point of failure or communication bottleneck as composition management, including SOAP message deserialisation and serialisation, is distributed, resulting in improved reliability over the centralised pattern. The composition can continue even if one of the peers goes down, assuming a replacement peer is available to take its place.
- Reduced SOAP message exchange over centralised pattern, resulting in improved efficiency over centralised pattern.
- Good autonomy as each participant acts upon its private data, but only reveals what is necessary to be a compositional partner.

Disadvantages

- Increased deployment complexity as each participant must be modified to support the pattern.
- Maintaining the composition can be difficult as each participant manages different parts of the composition.
- Web service composition engines do not support this pattern out of the box.

Applicability Use the Decentralised pattern when

- A high number of concurrent users is envisaged.
- Composition participants will not be changed frequently.
- Composer has administrative control over composition participants.

Case Study A Geographic Information System (GIS), which uses Web services is outlined in [45]. An example featuring six Web services, which allows spatial problems in the city of Beijing to be analysed and resolved, is provided by the authors. The composition is distributed between the six Web services, each controlled by a WS-BPEL process. The distribution pattern expressed by the composition is the decentralised shared-peer pattern. The authors of the case study clearly state that the composition must not feature either a single point of failure or a bottleneck, thus motivating their choice of the decentralised shared-peer pattern. The pattern chosen meets their non-functional requirements.

Summary: high efficiency, high reliability, low modifiability/mutability

5.4.2 Auxiliary Patterns

Auxiliary patterns are distribution patterns which by themselves cannot facilitate Web service compositions. These patterns are often used in conjunction with core patterns to create complex patterns. The only auxiliary pattern identified here is the Ring pattern [58].

Pattern 3: Ring

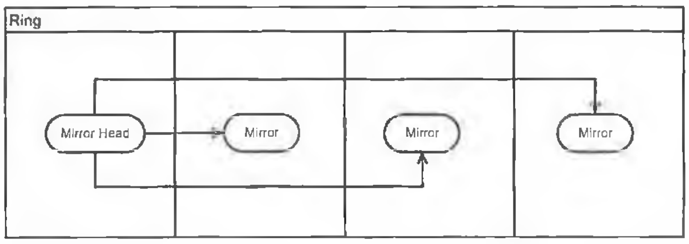


Figure 5.5: Ring distribution pattern.

Structure:

Description: The Ring pattern, illustrated in Figure 5.5, features a number of identical participants, mirrors, acting as a cluster. The pattern by itself does not facilitate composition and is normally used in association with other patterns. There is no start and end points to the ring pattern. The Ring pattern provides fault-tolerant infrastructure to a Web Service composition. The specific ring implementation defines, at the mirror head, the algorithm for determining how the load is delegated amongst the ring participants. A classification of faults, such as crashes, shutdowns and high load, which may occur in a Web service context are outlined by Juszczyk et al. in [97]. An example of this pattern in an existing networking context is a load balancing Web server.

Synonyms: Circle

Participants

- Mirror Head : A participant which delegates work to hub/spoke/peer mirrors.
- Hub Mirror : A participant mirror which controls the composition execution.
- Spoke Mirror : A participant mirror which passively performs some service.
- Peer Mirror : A participant mirror which performs some service and also participates in controlling the composition.

Advantages

- Provides improved reliability as more participants can be added to ring when required.
- No single point of failure or communication bottleneck as load can be shared amongst ring participants. Expensive deserialisation and serialisation of SOAP messages can be shared amongst participants.

Disadvantages

- Participants in the ring need to be located relatively close to each other.
- Web service composition engines do not support this pattern out of the box.
- Additional software is required to load balance/mirror the ring participants.

Applicability Use the Ring pattern when

- Increased reliability is required.

Case Study As previously stated, auxiliary patterns like the ring pattern are distribution patterns, which by themselves cannot facilitate Web service compositions. This implies that no use case will exist that directly implements the ring pattern on its own. However, a number of papers outline the scenarios in which the ring pattern could be applied to an existing distribution pattern to improve non-functional quality requirements.

In [166] Sobe notes that some classes of Web services suffer from long response times and low reliability. This poor response time is not acceptable for critical applications such as real time processing of medical images. Replication is presented as a solution to alleviate these problems by providing load balancing over a number of computational nodes. Also in [35] Birman et al. identify reliability as being essential for a Web service based hospital on-line inventory ordering system. Delays or down time in this context would make it difficult for hospitals to order urgently needed supplies. Again replication, amongst other approaches, is suggested as a solution. The ring distribution pattern, presented here, represents a replication scenario from a high level.

Summary: High efficiency, high reliability, medium modifiability/mutability

5.4.3 Complex Patterns

Complex patterns are distribution patterns which combine two or more core or auxiliary patterns. These patterns often resolve fundamental problems evident within core patterns. We identify four such patterns, Hierarchical, Ring + Centralised, Centralised + Decentralised and finally Ring + Decentralised [58].

Pattern 4: Hierarchical

Structure:

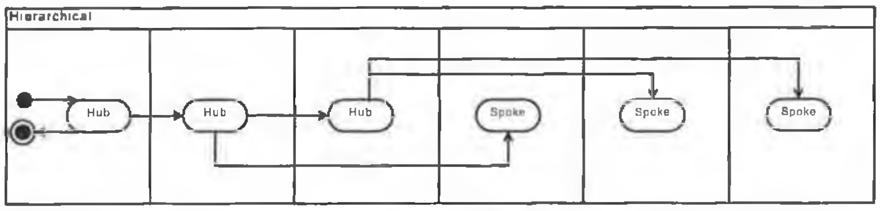


Figure 5.6: Hierarchical distribution pattern.

Description: The Hierarchical pattern, illustrated in Figure 5.6, is a tree based structure consisting of a number of levels, featuring a number of controller hubs. The pattern is related to the Centralised pattern. Two messages are exchanged between the hub and a spoke for each spoke execution i.e. synchronous communication. Two messages are also exchanged whenever hubs intercommunicate. The composition completes after the final spoke has completed execution and has returned a response to its hub, which then returns control to its owning hub, until finally the parent hub regains control of the composition and terminates. An example of this pattern in an existing network context is the Domain Name Service (DNS) [121].

Synonyms: Tree, Centralised + Centralised

Participants

- Hub : Participant which controls the composition execution.
- Spoke : Participant which passively performs some service.

Advantages

- Improved autonomy for participant spokes as they can be segregated into a number of locations or departments, where each location only reveals what is necessary to be a compositional partner. Participants which exchange verbose SOAP messages can be physically located close to each other.
- Improved reliability because of load balancing effect of hubs.

- Web service composition engines support this pattern out of the box.
- Composition is easily extensible by adding additional controlling hubs.

Disadvantages

- Secondary hubs have poor autonomy in that the input and output values of each such hub can be read by the controlling hub.
- Maintaining the composition is more difficult as there are multiple hubs managing different parts of the composition.
- Communication bottleneck at root hub.
- High number of SOAP messages between hubs and spokes is sub-optimal.

Applicability Use the Hierarchical pattern when

- Participants can be segregated into a number of locations.
- The number of participants is large.
- It is possible to delegate responsibility of certain compositions externally.

Case Study An information and communications system to assist police and the criminal justice organisations in the UK, called PITO, which uses Web services, is outlined in [63]. The system features a number of discrete services such as Motor Insurance Enquiry, Vehicle Enquiry, Finger Print Enquiry, Nominal Enquiry and Automated Number Place Recognition Enquiry. Each of these systems is provided by a local police force system. PITO then provides a central location for performing enquiries on the distributed services. Here, unlike when using a centralised pattern, the choreography is distributed amongst the participants. This hierarchical distribution pattern enables each service to invoke an authorisation service to guarantee only authorised users can gain access to sensitive information. If authorisation was left to the calling service its omission could potentially result in an insecure system.

The pattern chosen meets two important non-functional quality criteria. Firstly, improved autonomy over the centralised pattern is achieved by delegating the provision of services to local police forces. Secondly, security of the individual services is provided through local provision of access control along with the partitioning of data to locations responsible for managing such data.

Summary: Low efficiency, medium reliability, high modifiability/mutability

Pattern 5: Ring + Centralised

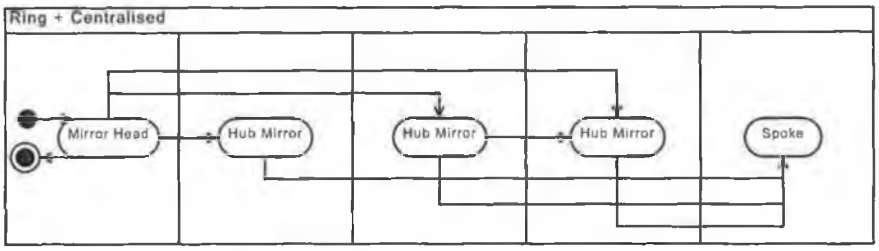


Figure 5.7: Ring + Centralised distribution pattern.

Structure:

Description: The Ring + Centralised pattern, illustrated in Figure 5.7, combines the Ring pattern with the Centralised pattern. This complex pattern eliminates the single point of failure and communication bottleneck at the central hub by providing a number of identical redundant hubs organised as a ring. As with the core centralised pattern, messages are exchanged between the hub and a spoke for each spoke execution i.e. synchronous communication. The composition completes after the final spoke has completed execution and has returned a response to the hub. The specific ring implementation defines the algorithm, at the mirror head, for determining how the load is delegated amongst the ring participants. An example of this pattern in an existing network context is a load balanced Web server (hub mirrors) serving many Web client browsers (spokes).

Synonyms: None

Participants

- Hub Mirror: A participant mirror which controls the composition execution.
- Mirror Head : A participant which delegates work to hub mirrors.
- Spoke : Participant which passively performs some service.

Advantages

- Composition can consume participant services that are externally controlled.
- Spokes require no additional modifications to take part in the composition as they use Web service based interoperability.
- No single point of failure or bottleneck at central hubs.
- Composition is easily maintainable, as composition logic is all contained on the ring participants, all of which work as central hubs.

Disadvantages

- High number of potentially verbose SOAP messages between hubs and spokes is sub-optimal.
- Poor autonomy in that input and output values from each participant can be read by the central hubs.
- Additional software is required to load balance and mirror ring participants.

Applicability Use the Ring + Centralised pattern when

- Increased reliability is required at the controlling hub.

Case Study No specific case study could be found for the ring + centralised pattern in the literature. However, a useful use-case can be extrapolated from the case study of the ring pattern [166] combined with the case study of the centralised pattern [55]. The centralised pattern case study presented a medical image management system featuring two Web services, which medical researchers use to access and store medical images. No specific non-functional requirements were explicitly stated during the case study. However, as is often the case in software systems, non-functional requirements may be stated post development. If in this scenario the system is performing poorly due to the high load of real time processing of medical images, as was noted in the case study motivating the ring pattern, the two patterns can be combined to reduce the chances of delays or down time occurring in the system.

Summary: Low efficiency, high reliability, medium modifiability/mutability

<u>Pattern 6:</u> Centralised + Decentralised

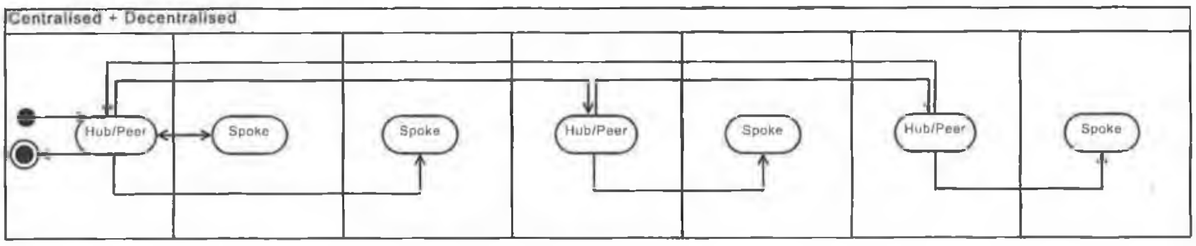


Figure 5.8: Centralised + Decentralised distribution pattern.

Structure:

Description: The Centralised + Decentralised pattern, illustrated in Figure 5.8, combines the Centralised pattern with the Decentralised pattern. This complex pattern allows a number of participants to function as hubs locally whilst functioning as peers within the larger composition. Only one message is exchanged between each hub/peer for each execution i.e. asynchronous communication. The composition completes after the final hub/peer has completed execution and has returned execution control to the hub/peer which commenced the composition. Two messages are exchanged between the hub/peer and a spoke for each spoke execution i.e. synchronous communication. An example of this pattern in an existing network context is email, whereby email clients (spokes) connect to a mail server (hub). However, the mail server also connects to other mail servers (peer-to-peer) to deliver mail.

Synonyms: None

Participants

- Hub/Peer : Participants which functions as hubs locally, controlling the local composition, whilst functioning as a peer within the larger composition.
- Spoke : Participant which passively performs some service.

Advantages

- Improved efficiency and reliability over centralised pattern.
- Good autonomy as each participant (peer or hub) acts upon its private data, but only reveals what is necessary to be a compositional partner.
- Bottlenecks reduced by having a number of participants functioning as hubs.

Disadvantages

- Increased deployment complexity as each participant, acting as a peer, must be modified to support the pattern.
- Maintaining the composition can be difficult as each participant, functioning as a hub, manages different parts of the composition.
- Web service composition engines do not support this pattern out of the box.

Applicability Use the Centralised + Decentralised pattern when

- A number of compositions are being merged to create a larger composition.

Case Study No specific case study could be found for the centralised + decentralised pattern in the literature. However, a useful use-case can be imagined in a scenario such as a mapping service. Here, a central service allows clients to search for a destination in a given city. The mapping service then delegates finding the destination and its mapping to another sub-composition, whilst also providing value added services, such as presenting ways and costs of getting to the destination, such as flying, via other sub-compositions.

Such a scenario could be realised using the centralised pattern to manage the main composition. This pattern is ideal because only a small amount of logic is required here to glue the sub-compositions together, meaning the load will be quite low. The sub-compositions, which feature the majority of the logic and processing, could utilise the decentralised pattern, necessary for efficiency and reliability reasons as noted in the case study in [45]. This situation results in the application of the centralised + decentralised distribution pattern.

Summary: Medium efficiency, medium reliability, low modifiability/mutability

Pattern 7: Ring + Decentralised

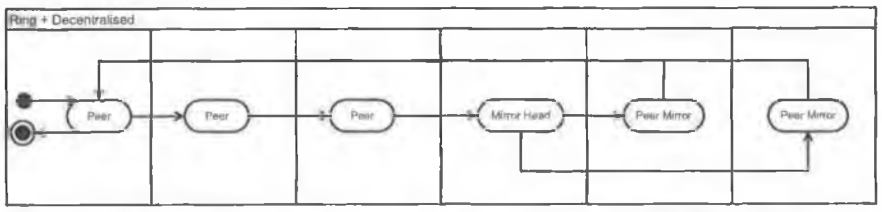


Figure 5.9: Ring + Decentralised distribution pattern.

Structure:

Description: The Ring + Decentralised pattern, illustrated in Figure 5.9, combines the Ring pattern with the Decentralised pattern. This complex pattern uses one or more rings to provide redundant copies of participants. As with the core decentralised pattern, only one messages is exchanged between the caller and the callee for each peer execution i.e. asynchronous communication. The composition completes after the final peer has completed execution and has returned execution control to the peer which commenced the composition. The specific ring implementation defines, at the mirror head, the algorithm for determining how the load is delegated amongst the ring participants. An example of this pattern in an existing network context, is a file sharing system whose peers have load balanced enabled.

Synonyms: None

Participants

- Peer : Participant which performs some service and also participates in controlling the composition.
- Mirror Head : A participant which delegates work to peer mirrors.
- Peer Mirror : A participant mirror which performs some service and also participates in controlling the composition.

Advantages

- High efficiency and reliability.
- No single point of failure or communication bottleneck as composition management is distributed.
- Ring provides improved reliability to peers as more participants can be added to ring when required.
- Good autonomy as each participant acts upon its private data, but only reveals what is necessary to be a compositional partner.

Disadvantages

- Increased deployment complexity as each participant must be modified to support the pattern.
- Maintaining the composition can be difficult as each participant manages different parts of the composition.
- Web service composition engines do not support this pattern out of the box.
- Additional software is required to load balance and mirror participants.

Applicability Use the Ring + Decentralised pattern when

- Increased reliability is required for particular participants.

Case Study No specific case study could be found for the ring + decentralised pattern in the literature. However, a useful use-case can be extrapolated from the case study of the ring pattern [166] combined with the case study of the decentralised pattern [45]. The decentralised pattern case study presented a Geographic Information System (GIS) featuring six Web services, which allows spatial problems in the city of Beijing to be analysed and resolved. The authors of the case study clearly state that the composition must not feature either a single point of failure or a bottleneck, thus motivating their choice of the decentralised shared-peer pattern. However, they do not consider the possibility that one of the six Web services might take significantly longer to process its load when compared to the other Web services. This situation would be exacerbated under high load resulting in a bottleneck. This situation could be alleviated by applying the ring pattern to any services, which may be computationally expensive, resulting in a ring + decentralised distribution pattern.

Summary: High efficiency, high reliability, low modifiability/mutability

5.5 Catalog Usage

We envisage a number of applications for our pattern catalog. Specifically, when used with our modeling framework as described in Chapter 4, the catalog can be used in conjunction with our Model Driven Development (MDD) approach, which auto-generates a fully executable Web service composition based on a distribution pattern model [25]. The software architect can browse the paper based pattern library and, based on the system's non-functional requirements, decide which pattern to apply to a composition. We also envisage in the future supporting a search facility, possibly using keywords, where an architect can select a number of non-functional quality parameters, and a software tool would suggest an appropriate pattern to use.

5.6 Catalog Evaluation

For a pattern catalog to be useful it must provide coverage of at least the most common patterns of the systems it intends to document. Ideally it should provide full coverage of all core pattern possibilities. To assess our pattern catalog we perform a small scale experiment.

Cutumisu et al. in [49] consider how to measure the effectiveness of a pattern catalog and how to objectively compare pattern catalogs. The measures considered are not pattern catalog specific and so can be applied to our pattern catalog. Four specific measures, usage, coverage, utility and precision, are outlined and formally defined. The authors note that a good pattern catalog achieves high values for each of these measures over a wide range of applications. A brief description of the four measures follows.

- Usage - Ratio of catalog pattern usage in a given scenario/application to total number of patterns in the catalog.
- Coverage - Ratio of catalog patterns usage in a given scenario/application to total number of patterns in the scenario/application.

$CaseStudy$	$Reference$
Adding High Availability and Autonomic Behavior to Web Services	[35]
Service Composition Modeling	[55]
A p2p architecture for dynamic executing GIS web service composition	[45]
Using a rigorous approach for engineering Web service compositions	[63]
Migration to web services oriented architecture	[197]

Table 5.1: Selected case studies.

- Utility - Average number of times a catalog pattern is used in a given scenario/application.
- Precision - Ratio of catalog pattern usage in a given scenario/application to the number of adaptations required to be made to make these catalog pattern useful in a given scenario/application.

Our pattern catalog contains seven distinct patterns. To evaluate its usefulness we have applied the measures and measurement methods defined by Cutumisu et al. to a number of case studies of Web service compositions outlined in the literature. Appropriate case studies, some of which were outlined earlier in this chapter, were found by systematically searching the proceedings of major conferences in the digital libraries of the ACM and the IEEE. Of the one hundred papers found only five contained non-trivial real world, fully implemented scenarios. The selected case studies all contain enough information for clear distribution pattern analysis e.g. workflow or sequence diagram of choreography. The selected case studies, some of which were outlined already in this chapter, which met our selection criteria, are outlined in Table 5.1.

Having identified some case study applications we now apply the pattern evaluation measures to them. The measures are applied to the case studies as a whole rather than to each case study individually. This is done because we are analysing large architectural patterns, rather than smaller behavioural or creational patterns. Each case study has only one architecture, although it might contain many behavioural or creational patterns. The results are outlined in Table 5.2.

The usage measure value of .571 shows that of the seven patterns in our catalog only

Measure	*Ratio*	*Value*
Usage	7:4	0.571
Coverage	5:5	1
Utility	5:4	0.8
Precision	1:1	1

Table 5.2: Pattern catalog evaluation measures.

four unique patterns were actually used in the case study applications. The centralised pattern occurs twice in the case studies. No usage of three patterns, ring + centralised, centralised + decentralised and ring + decentralised was found in the case studies.

The coverage measure value of 1 indicates that the distribution pattern of all the case study applications was present in our catalog.

The utility measure value of .8 indicates that of the four patterns used in the case study applications one is used twice, while three are used only once. This indicates, at least in our small use case sample, that the centralised distribution pattern is more prevalent than the other patterns. The highest value possible for this measure when considering architectural patterns is one.

The precision measure value of 1 indicates no adaptions are required to make the patterns in our catalog fit different scenarios. However, it should be noted that architectural patterns do not normally require adaptation as they are at a very high level of abstraction.

To summarise, we have scored good values for the majority of the measures outlined. The usage figure is the only figure which is relatively low. This is the case because no case studies were found in our sample selection that featured three of the distribution patterns. However, this is not a negative mark against the pattern catalog as it is likely that these patterns will become more prevalent as the usage of Web services in critical applications becomes more commonplace. We believe these measures indicate that our catalog is sufficiently complete to cover all the current distribution scenarios, whilst also providing adequate future proofing for future usage scenarios.

5.7 Summary

In this chapter we have presented a catalog of distribution patterns, which may be applied by software architects to Web service compositions. These patterns have a historical context in the field of network based systems, however their description in a Web service composition context is novel. Also novel is their expression using a standardised modeling language, UML. We have enumerated three categories of identified patterns. Support for their usage is provided by referenced QoS measures where possible, case study scenarios of traditional network based scenarios, along with Web service based specific scenarios.

The catalog itself has been evaluated using a number of pattern catalog measures. We have concluded that the catalog is sufficiently complete to cover all the current distribution scenarios, whilst also providing adequate future proofing for future usage scenarios. We envisage that this pattern catalog will assist software architects in choosing a distribution pattern for a wide range of different Web service composition based applications.

For completeness it should be noted that UML Interaction Overview diagrams could be used to provide frames for many interacting Activity diagrams. This scenario would occur when the distribution patterns consists of a composition of distribution patterns. These diagrams would enable the modeling of many discrete compositions onto one model. Although we do not consider this scenario here it is possible that large real world systems would require this level of modeling. Composition of architectural styles such as distribution patterns is considered by Pahl et al. in [143]. Here, an ADL like language is used to construct and subsequently combine styles. A number of operators are presented including restriction, union, intersection and refinement which define the semantics of this operator calculus, which may be applied to patterns. The approach is also compatible with UML.

Chapter 6

Modeling Notations/Languages

6.1 Introduction

In this chapter we present the modeling notations and languages necessary to enable the modeling of distribution patterns, and the subsequent generation of an executable system. The modeling infrastructure presented here represents the second component of our modeling and transformation framework. This component consists of eight languages, or notations, describing the constructs of the various models, that feature in the five step methodological framework component presented in Chapter 9. A case study illustrating the usage of the notations is provided in Chapter 9.

In Section 6.2 we contrast two different forms of languages, Domain Specific Languages (DSL) and General Purpose Languages (GPL), before discussing how a language is defined. These languages are used within a Model Driven Software Development (MDSD) framework to enable code generation from high level models. The framework used is the Model Driven Architecture framework (MDA), outlined in Section 6.3. Section 6.4 defines the eight languages, or notations, used to support our MDSD approach to code generation. Finally, in Section 6.5 we discuss the tool support available to facilitate language definition and editing.

6.2 Language Definition and Semantics

Language definition is the means by which a programming language is specified. In our context we consider languages as a form of model representing a given domain. There are two forms of languages - Domain Specific Languages (DSL) and General Purpose Languages (GPL). DSLs are task-specific languages, which trade generality for expressiveness, ease of use and intuitiveness in their narrow domain. An example of a DSL is the declarative database language SQL (Simple Query Language). GPLs, meanwhile, are flexible languages that may be applied to any number of domains, but at the cost of increased complexity. An example of a GPL is the Java programming language. A full analysis of when and where to use DSLs over GPLs is outlined by Mernik et al. in [117]. Here, we primarily consider DSLs with the exception of the XML, as each language is targeting a very narrow domain within the Web services context.

Up to now we have only discussed language definition. However, language definition and model definition are interchangeable as both have the same goal of representing a given domain. To define either a DSL or a GPL the same process must be followed. This language or model specification process consists of three parts as outlined below. The parts are discussed in more detail in the following Sections, and further detailed by Harel et al. in [82].

- Syntax
- Semantic Domain
- Semantic Mapping

An additional characteristic of languages is their pragmatics. Pragmatics relate to the usability of a language. For example some language pragmatics might be how easy it is to write using the language, how useful the language is for its specific domain and how well the language meets its stated objectives [159].

6.2.1 Syntax

There are two forms of language syntax definition - abstract and concrete. The abstract syntax for a language is developed by enumerating the abstractions, or concepts, of a specific domain and mapping their mutual relations [105]. The abstractions and concepts may also be considered to be the grammar of the language. Here, we do not differentiate between languages and models. We consider that once a language is expressed as a model it may be considered a reference model for the domain. This reference model represents an ontology of the domain modeled. An example of a definition of an abstract syntax is an object model describing a domain notation and its interrelations. Examples of languages that can be used to define the abstract syntax in a modeling context are MOF [137], KM3 [94] and Ecore [39]. In a non-modeling context, languages such as BNF [20] and EBNF [192] can be used to define the abstract syntax of a language using simple textual notations.

Languages have a concrete syntax in addition to their abstract syntax definition. A language can have more than one concrete syntax. The concrete syntax represents a user interface for the language, and is derived from its abstract syntax. Such user interfaces are often realised using tools, and are represented as XML, text, graphics etc. The concrete syntax must itself be described clearly using an abstract syntax. Once both the language and its user interface have been defined abstractly, mappings between the two can be defined. It is essential that the concrete syntax is user-friendly, simple and clear [105]. Examples of a concrete syntax are XML, a shape-based GUI in a tool and XMI [134].

6.2.2 Semantic Domain

Once a language's syntax has been formally defined, its meaning should be formally defined. Without semantics a language's meaning is ambiguous and is open to different interpretations. Each expression in a language must therefore be defined with respect to a well defined and well understood domain [82]. This semantic domain is independent of the language being defined. Semantics are useful to both language implementers and programmers as outlined by Schmidt et al. in [159] and summarised below.

- Provide precise standard for implementations.
- Useful as user documentation after language development.
- Can be used as a tool for design and analysis during language development.
- Can be used with a compiler to help automate development.

The semantic domain can be defined in two different ways, loosely using a plain natural language such as English, or more rigorously using mathematical semantics. The plain English approach simply explains what the specific language construct is and where it might be used. The axiomatic approach defines the properties of the language constructs using formal proofs. Both MOF and ECore use English language based definitions of semantics, while KM3 has its semantics precisely defined using first order logic [94]. The two approaches differ in that the axiomatic approach provides a high level of precision when compared to the potentially ambiguous plain English approach, at the cost of comprehensibility for those unfamiliar with formal mathematics.

An alternative approach to defining the semantic domain for a language is operational semantics [57, 148]. There are two approaches to defining operational semantics, explicit and implicit, both of which express the execution of a language. Explicit operational semantics are defined using mathematics to express the various state transitions a language interpreter would realise. Alternatively implicit operational semantics can be defined directly by writing an interpreter to process a language definition. The language's semantics are derived by how the interpreter interprets the language. An obvious issue with this approach is that the language semantics are tightly coupled to the interpreter implementation, making portability and comprehension of the language difficult when compared to the explicit approach.

One other approach to semantic domain definition is denotational semantics [159]. This approach is more abstract than operational semantics. Denotational semantics allow a specific part of a language definition to be defined using valuation functions. These valuation functions are mathematical objects which represent the meaning of that language construct.

6.2.3 Semantic Mapping

A language whose syntax and semantic domain have been defined requires a link, or mapping, between these two independently derived definitions. The semantic mapping provides this link. Graph transformations and mathematics are two methods for defining these links [82].

A specific mathematical approach to semantic mapping is the use of relations, or mappings, to relate a language with no semantic domain to a language with a semantic domain. The undefined language then takes on the semantics of the related language. This pragmatic solution is suggested by Kurtev et al. in [105].

6.3 Framework

Model Driven Development (MDD), or Model Driven Software Development (MDSD), described in Section 2.5.6, is an emerging approach for building software [182, 30]. MDD considers models, at different levels of abstraction, as the primary artifact to reason about a given domain and devise a solution. Relationships are defined between these models to describe the web of dependencies between the models. These relationships are used to assist in the generation and reasoning of the final solution.

Here, we base our approach on the Model Driven Architecture (MDA) framework [71]. MDA encapsulates a number of technologies that provide for the formal specification of the structure or function of a system, where the modeling language is the programming language. In this way the models created are used to generate the program code. This approach enables us to represent each of the distribution patterns, presented in Chapter 5, using UML 2.0. These models are then used to subsequently generate an executable system. We define our modeling platform as Web service specific. This means from the MDA perspective, that all our modeling notations are Platform Independent Models (PIMs).

Each of the eight notations, introduced in Section 6.4 below, are compatible with the MDA framework as their abstract syntax is defined in ECore. These notations are considered meta-models in MDA terminology. Alternatives to the MDA modeling stack, including ontologies and abstract syntax approaches, are considered by Kurtev et al. in [104].

6.4 Notations

We use eight languages or notations, outlined below, and illustrated in Figure 6.1. Each is described in detail in the following sections. The notations were first introduced in Section 4.3.2. Three of these notations: UML 2.0 Notation, Collaboration Notation and XML Notation have been previously defined elsewhere. The remaining 5 notations are defined by us.

- UML 2.0 Notation
- Distribution Pattern Language UML Profile (DPLProfile)
- Distribution Pattern Language Notation (DPL)
- Collaboration Notation
- Interface Notation
- Deployment Descriptor Notation
- Deployment Catalog Notation
- XML Notation

Relationships between each of the languages is outlined in Figure 6.2. The boxes with grey shading are extensions to existing languages or are entirely new languages, as is the case for DPL. Boxes without shading are direct representations of existing languages.

The abstract syntax for each of these languages is defined using ECore, based on the specification of each language as defined by their respective designers. ECore was chosen because it is closely aligned with a subset of MOF called EMOF, a standards based domain specific language for defining languages. ECore, a domain specific language itself, is

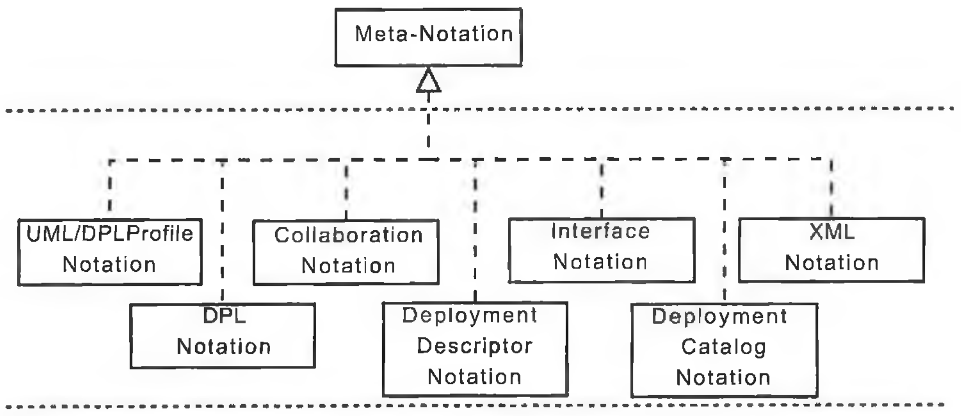


Figure 6.1: Notations used in our modeling approach.

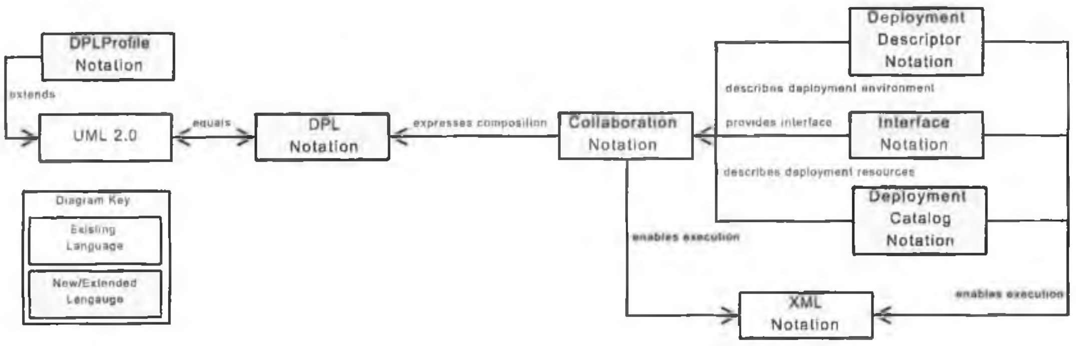


Figure 6.2: Relationships between the notations used in our modeling approach.

supported by the Eclipse Modeling Framework (EMF) and its associated toolset [39]. The use of ECore guarantees interoperability with a range of tools. ECore is compatible with our chosen framework MDA. The use of ECore in an MDA context facilitates our goal of executable system generation. An additional motivation for the use of ECore was the availability of ECore based implementations of some of the notations, specifically UML 2.0, WS-BPEL, WSDL and XML.

The concrete syntax for the UML 2.0 notation is graphical. UML features a number of graphical icons, as outlined and illustrated by Eriksson in [60]. The remaining seven notations have no user friendly concrete syntax. To assist the end user of these languages they may be manipulated using an EMF based editor, which is capable of manipulating ECore based languages. However, in the context of our work these notations are never directly manipulated, and instead are abstracted by the framework.

The semantic domain for the notations is defined here using structured textual semantics. Detailed textual semantics are also available for each of the existing languages in their respective language specifications, referenced in the sections below. No axiomatic or denotational semantics have been defined for the existing languages. These mathematical formalisms encourage the reduction of language specification overlap and allow properties such as completeness and precision to be assessed. Instead the authors of these languages have deemed the textual semantics adequate for their usage. Textual semantics can be sufficiently complete and precise if unambiguously defined. Here, we follow this trend and utilise semantic mappings to relate newly defined languages to existing languages. These semantic mappings between languages are provided by relations, as discussed in Chapter 7.

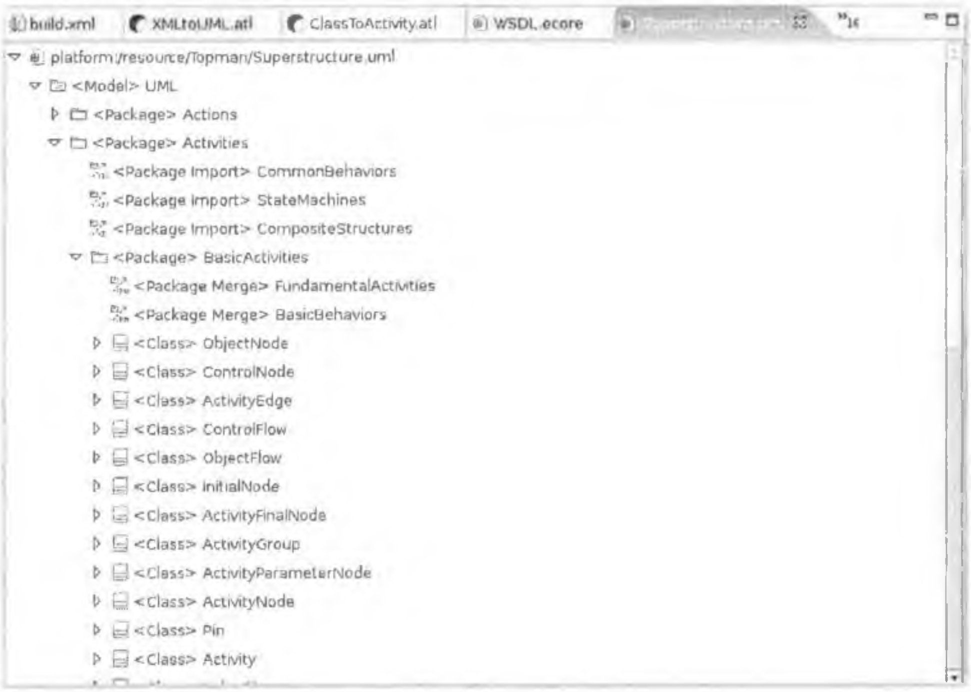
All of the eight notations are defined using only one language, Ecore, which is closely aligned to the EMOF, a subset of the MOF language. ECore is considered a meta-meta language in MDA terminology.

6.4.1 UML 2.0 Notation

The UML 2.0 notation utilises a subset of the standard UML 2.0 specification [140] and is used to describe distribution patterns using standard UML 2.0 Activity diagram constructs. We also utilise the UML 2.0 Class diagram constructs to represent the interfaces of the discrete Web service participants.

The UML notation is used to describe distribution patterns because it is capable of providing a visual representation of the patterns, which should be easily understood by a software architect. In addition, UML, because of its MOF compliance, is interoperable with many tools and other notations. This interoperability provides for improved reuse of data and models, as outlined by Moreno at al. in [123]. However, this flexibility is at the cost of the complexity of working with such a large specification as UML 2.0.

The abstract and concrete structure of the UML 2.0 notation is specified by the OMG using MOF, bootstrapped by utilising Class diagrams from the UML. The abstract structure can be manipulated using an Eclipse EMF based editor, as illustrated in Figure 6.3.



```
platform:/resource/Topman/Superstructure.uml
  <Model> UML
    <Package> Actions
    <Package> Activities
      <Package Import> CommonBehaviors
      <Package Import> StateMachines
      <Package Import> CompositeStructures
      <Package> BasicActivities
        <Package Merge> FundamentalActivities
        <Package Merge> BasicBehaviors
        <Class> ObjectNode
        <Class> ControlNode
        <Class> ActivityEdge
        <Class> ControlFlow
        <Class> ObjectFlow
        <Class> InitialNode
        <Class> ActivityFinalNode
        <Class> ActivityGroup
        <Class> ActivityParameterNode
        <Class> ActivityNode
        <Class> Pin
        <Class> Activity
```

Figure 6.3: Excerpt of the UML2 abstract syntax as viewed in Eclipse.

The semantic domain of UML 2.0 is specified by the OMG, using plain English. We consider only a subsection of the notation - UML Activity diagrams and Class diagrams. The significant subsection of the UML Activity diagram notation is illustrated in Figure 6.4, whilst the important textual semantics of constructs are outlined in Table 6.1. This notation is deemed adequate as it defines all the constructs necessary for distribution pattern modeling.

Activity diagram models provide a number of important modeling artifacts necessary for the modeling of distribution patterns. UML ActivityPartitions, also known as swim-lanes, are used to group a number of actions within an Activity diagram. In our model, these actions will represent WSDL operations. Any given interface has one or more ports that will have one or more operations, all of which will reside in a single swim-lane. To provide for a rich model we use a particular type of UML action to model the operations of the WSDL interface. These actions, called CallBehaviorActions, model process invocations and can

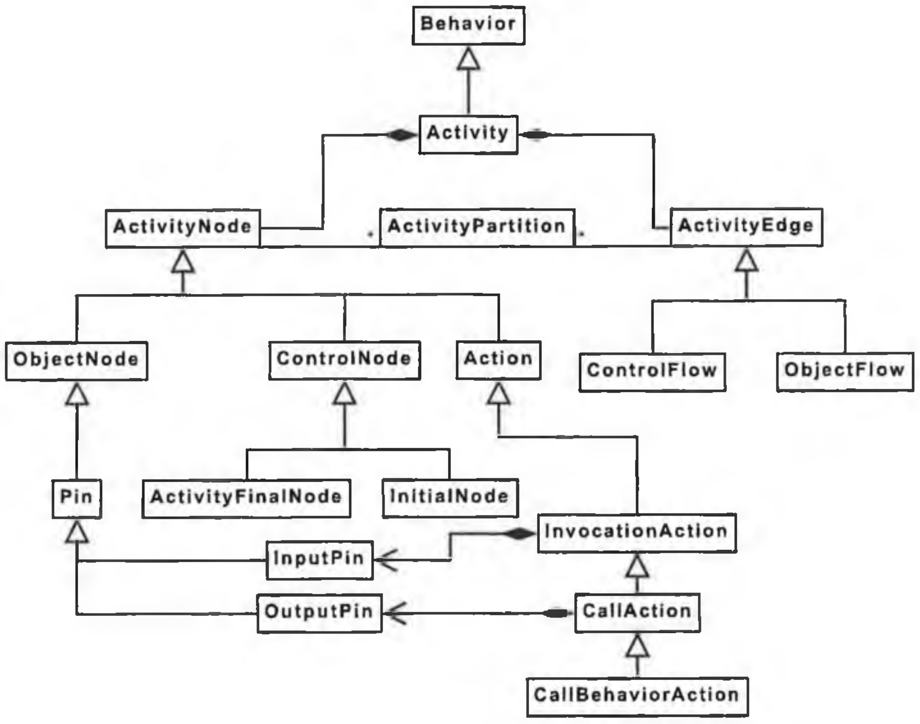


Figure 6.4: Activity diagram subset of UML2 notation expressed using a UML Class diagram.

have an additional modeling constructs applied to them called pins. There are two types of pins, InputPins and OutputPins, which map directly to the parts of the WSDL messages going into and out of a WSDL operation. Bock and Eriksson provide more information on these artifacts in [36] and [60].

In MDA terminology this notation is a meta-model and is defined at the PIM level, as UML is not tied to any specific platform technology. The Eclipse UML2 project provides an open source ECore based implementation of the UML notation, derived from the UML specification [65], by defining the constructs which may be used in UML 2.0 models.

An example of the UML 2.0 notation being used to model a centralised distribution pattern, using Eclipse, can be seen in Figure 6.5. This illustration is an implementation of the centralised shared-hub pattern that is represented as a UML Activity diagram in Figure 5.2.

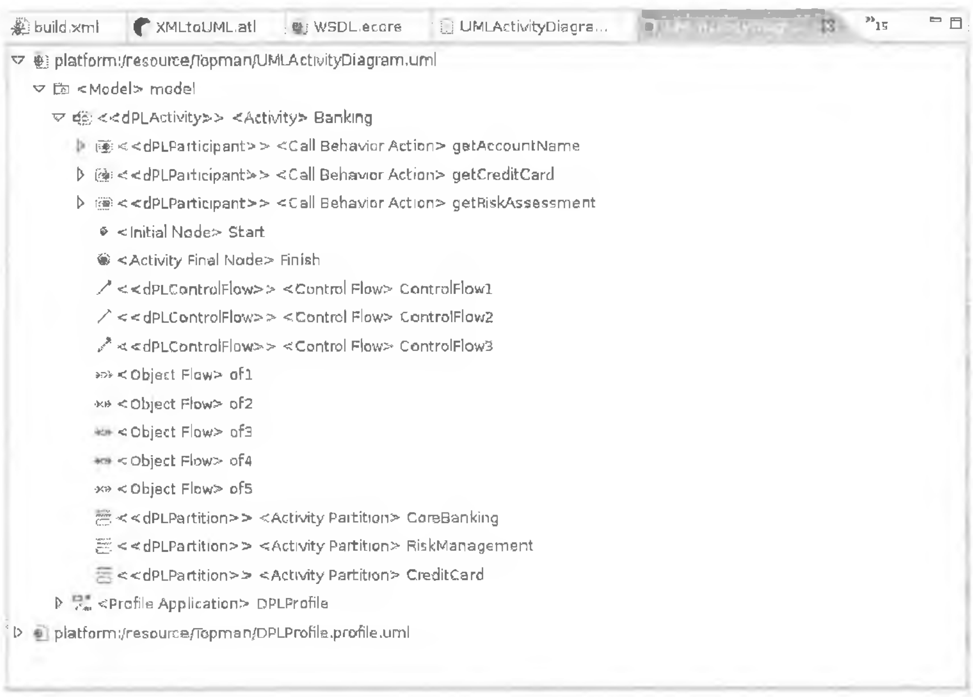
Table 6.1: Structured textual semantics of important UML abstract syntax constructs.

Construct	Description
Activity	An observable effect defined as a workflow
Action	A unit of observable effect, which forms part of an activity
ActivityPartition	An area in an Activity diagram where activities/actions are located
ControlFlow	Connector between two activities
ObjectFlow	Connector carrying data between activities
Pin	Input or output region for accepting or returning data
CallBehiorAction	A form of action which has input and output arguments
InitialNode	Start point of an activity
ActivityFinalNode	End point of an activity

6.4.2 Distribution Pattern Language UML Profile (DPLProfile)

The Distribution Pattern Language UML Profile, or DPLProfile, is our novel extension of the UML 2.0 notation, which allows extra distribution pattern specific information to be applied to a UML model. The profile is an extension to an existing notation, rather that a new notation in itself. As the UML is a general purpose software engineering modeling language it is often necessary to extend the language to enable it to adequately describe specific scenarios, such as distribution pattern modeling. UML profiles are a standard extension mechanism of UML, as discussed in Section 2.5.5. Profiles define stereotypes and subsequently tag definitions that extend a number of UML constructs defined in the UML 2.0 notation. Each time one of these derived constructs is used in our model we may assign values to its tag definitions. The profile is not strictly a notation; it is an extension to the existing abstract notation of the UML 2.0 language.

The motivation for using a UML profile is that it restricts the set of UML constructs by extending appropriate constructs of the UML notation, which need to be used to model the distribution domain. This reduces the complexity of the resultant models. The profile is used to mark an Activity diagram so it can be used to describe a distribution pattern.



```
platform:/resource/Topman/UMLActivityDiagram.uml
  <Model> model
    <<dPLActivity>> <Activity> Banking
      <<dPLParticipant>> <Call Behavior Action> getAccountName
      <<dPLParticipant>> <Call Behavior Action> getCreditCard
      <<dPLParticipant>> <Call Behavior Action> getRiskAssessment
        <Initial Node> Start
        <Activity Final Node> Finish
        <<dPLControlFlow>> <Control Flow> ControlFlow1
        <<dPLControlFlow>> <Control Flow> ControlFlow2
        <<dPLControlFlow>> <Control Flow> ControlFlow3
        <Object Flow> of1
        <Object Flow> of2
        <Object Flow> of3
        <Object Flow> of4
        <Object Flow> of5
        <<dPLPartition>> <Activity Partition> CoreBanking
        <<dPLPartition>> <Activity Partition> RiskManagement
        <<dPLPartition>> <Activity Partition> CreditCard
    <Profile Application> DPLProfile
platform:/resource/Topman/DPLProfile.profile.uml
```

Figure 6.5: The UML2 notation being used to model a distribution pattern as viewed in Eclipse.

Without this profile extension the UML would not be able to adequately describe the various distribution possibilities of Web service compositions. As previously noted, using UML, and by extension using UML profiles to focus the language to our needs, allows us to leverage software architect's comprehension of UML and the plethora of existing tools which support UML. An overview of our profile can be seen in Figure 6.6. The stereotypes are placed in a UML context in Table 6.2.

It should be noted that although we apply role tag definition to UML CallBehaviourAction constructs, via our DPLProfile, this is not strictly correct and may break the UML 2 semantics unintentionally [140]. We have placed the role tag definition on UML CallbehaviorAction constructs as different roles are often applied to different operations within a particular Web service interface. An alternative solution which does not break the UML semantics would be to move the role tag definition to the UML ActivityPartition. A combination of horizontal and vertical ActivityPartition constructs could then be used to partition different role tag definitions for a given interface. The drawback of this approach is the

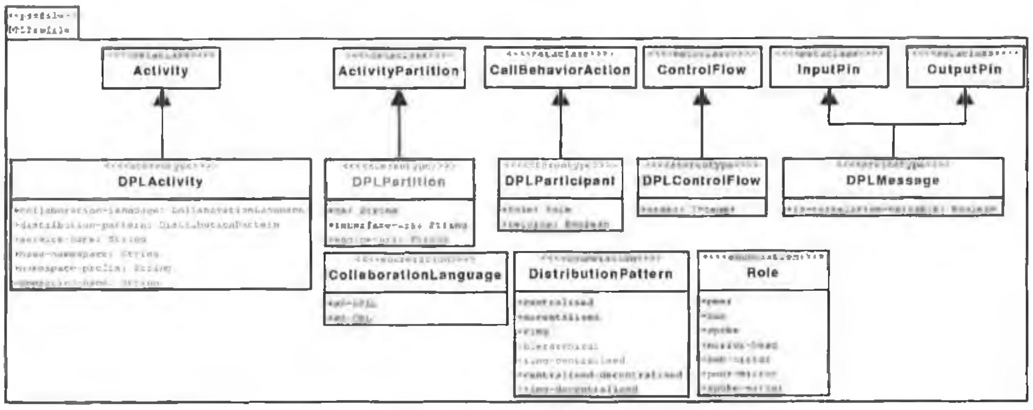
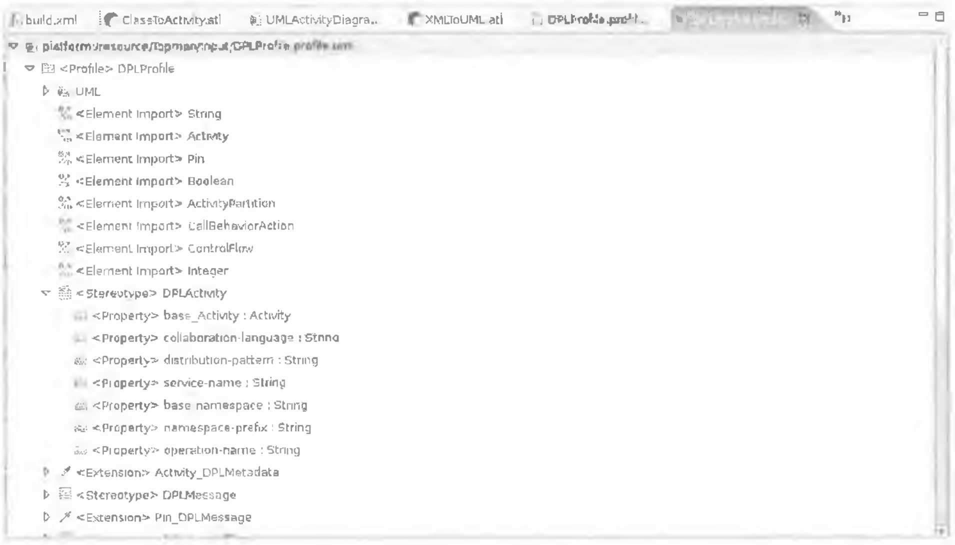


Figure 6.6: UML profile for modeling distribution patterns.

added visual complexity of the models which feature both vertical and horizontal partitions.

The profile definition that is outlined above was implemented using Eclipse EMF [39], which uses ECore for profile definition. The concrete syntax of the profile can be manipulated using an Eclipse EMF based editor. The definition of the language in the Eclipse editor can be seen in Figure 6.7.



```
platform:/resource/Topmanyinput/DPLProfile.profile.uml
  <Profile> DPLProfile
    UML
    <Element Import> String
    <Element Import> Activity
    <Element Import> Pin
    <Element Import> Boolean
    <Element Import> ActivityPartition
    <Element Import> CallBehaviorAction
    <Element Import> ControlFlow
    <Element Import> Integer
    <Stereotype> DPLActivity
      <Property> base_Activity : Activity
      <Property> collaboration-language : String
      <Property> distribution-pattern : String
      <Property> service-name : String
      <Property> base namespace : String
      <Property> namespace-prefix : String
      <Property> operation-name : String
    <Extension> Activity_DPLMetadata
    <Stereotype> DPLMessage
    <Extension> Pin_DPLMessage
```

Figure 6.7: The DPLProfile definition as viewed in Eclipse.

We can see from Figure 6.6 that the profile extends the Activity, ActivityPartition, CallBehaviorAction, ControlFlow, InputPin and OutputPin UML constructs. This extension allows distribution pattern metadata to be applied to the constructs via the tag definitions.

Table 6.2: DPLProfile abstract syntax or stereotype attributes.

DPL Attribute	UML Base Element	Stereotype
distribution-pattern	Activity	<<DPLActivity>>
collaboration-language	Activity	<<DPLActivity>>
service-name	Activity	<<DPLActivity>>
base-namespace	Activity	<<DPLActivity>>
namespace-prefix	Activity	<<DPLActivity>>
operation-name	Activity	<<DPLActivity>>
ns	ActivityPartition	<<DPLPartition>>
interface-uri	ActivityPartition	<<DPLPartition>>
engine-uri	ActivityPartition	<<DPLPartition>>
role	CallBehaviorAction	<<DPLParticipant>>
is-correlation-variable	Pin	<<DPLMessage>>
order	ControlFlow	<<DPLControlFlow>>

For example, the distribution pattern is chosen by selecting a pattern from the DistributionPattern enumeration and assigning it to the distribution-pattern tag definition on the DPLActivity construct. A full description of the semantics, using plain English, for all the profile tag definitions is outlined in Table 6.3.

Together, the UML 2.0 notation and DPLProfile provide the notational syntax and semantics necessary for the software architect to model any of the distribution patterns outlined in our catalog, using a number of UML 2.0 based tools. The constructs used in the UML should be familiar to software architects, making this language ideal for defining distribution pattern based compositions. Figure 6.8 illustrates the Eclipse tool showing a UML model of a distribution pattern with the DPLProfile applied. The stereotypes applied to the UML constructs can be seen in the main pane of the figure, while the profile values can be seen in the properties pane at the bottom of the figure.

Instances of the UML 2.0 notation and DPLProfile are output from step two of our methodological framework.

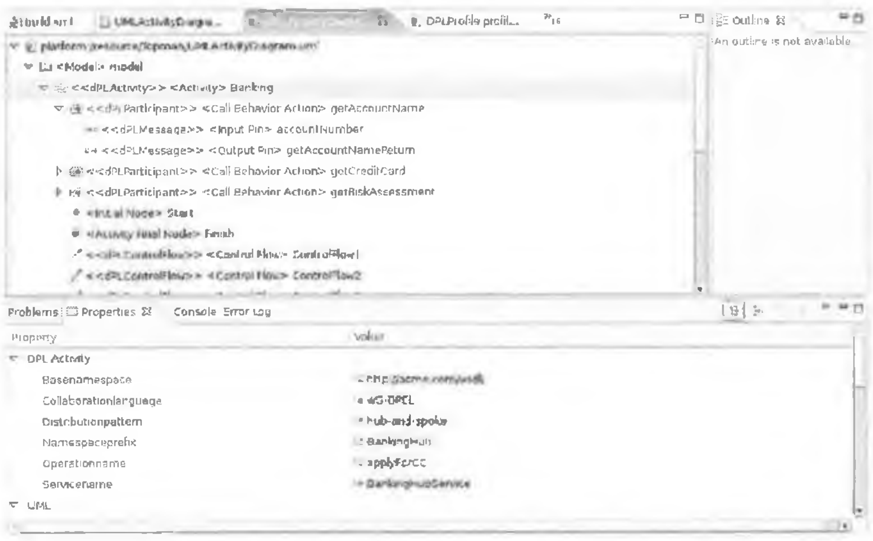
Table 6.3: Structured textual semantics of DPLProfile stereotypes attributes.

Attribute	Description
distribution-pattern	Choice of distribution pattern to be applied to composition
collaboration-language	Choice of collaboration language to enact composition
service-name	Name used by clients to reference the composition
base-namespace	Namespace URI for the composition, avoids name clashes
namespace-prefix	Namespace alias for the composition, avoids name clashes
operation-name	Operation name used by clients to reference the service
ns	Namespace URI of the participant, avoids name clashes
interface-uri	URI specifying the location of the participant's interface
engine-uri	URI specifying the location of the enactment engine
role	Choice of roles for the participant from the Role enumeration
is-correlation-variable	Unique identifier field for a composition
order	Execution order assigned to action

6.4.3 Distribution Pattern Language Notation

The Distribution Pattern Language notation provides the constructs for the internal representation of a distribution pattern. DPL is a novel Domain Specific Language (DSL) for distribution pattern description. DPL, from a semantic point of view is equivalent to the UML 2.0 notation/DPLProfile combination previously discussed.

The motivation for implementing a new language was to provide for ease of analysis, verification and transformation of distribution patterns. This is achieved because DPL is a precise language for representing the distribution domain. These motivations are outlined by Mernik et al. in [117] as reasons for developing domain specific languages. Further motivation for the use of DSLs, including productivity improvements and appropriate levels of abstraction, are also provided by Mernik et al. The DPL notation provides a purpose built simple notation for the description of distribution patterns. However, this precision is at the cost of interoperability with tools, such as those supporting UML, as noted by Moreno et al. in [123]. For this reason we only use the language internally to assist in the analysis and



```
<Model> model
  <<dPLActivity>> <Activity> Banking
    <<dPLParticipant>> <Call Behavior Action> getAccountName
      <<dPLMessage>> <Input Pin> accountNumber
      <<dPLMessage>> <Output Pin> getAccountNameReturn
    <<dPLParticipant>> <Call Behavior Action> getCreditCard
    <<dPLParticipant>> <Call Behavior Action> getRiskAssessment
    <Initial Node> Start
    <Activity Final Node> Finish
    <<dPLControlFlow>> <Control Flow> ControlFlow1
    <<dPLControlFlow>> <Control Flow> ControlFlow2
```

An outline is not available

Property	Value
DPL Activity	
Basenamespace	[illegible]
Collaborationlanguage	WS-BPEL
Distributionpattern	hub-and-spoke
Namespaceprefix	BankingHub
Operationname	applyForCC
Servicename	BankingHubService
UML	

Figure 6.8: A UML model with the DPLProfile applied as viewed in Eclipse.

transformation of distribution patterns.

The language's abstract syntax was defined using Eclipse EMF [39], which uses ECore for language definition. The abstract syntax of the language can be manipulated using an Eclipse EMF based editor, as illustrated in Figure 6.10. No concrete graphical syntax has been defined by us because we believe the UML provides an adequate graphical representation of distribution patterns using Activity diagrams and also due to UML's wide spread acceptance in the software engineering community. However, having defined a specific DSL for distribution patterns the possibility of later developing a novel graphical representation is feasible. What this representation would look like has not been considered here. The notation is deemed adequate as it defines succinctly all the constructs necessary for distribution pattern modeling. The notation, illustrated in Figure 6.9, defines the constructs which may be used in the DPL Model.

The structured textual semantics, using plain English, for all the DPL constructs is outlined in Table 6.4 below. As previously stated, the DPL language is semantically equivalent to the UML 2.0 notation/DPLProfile combination already outlined. To enforce these semantics we provide semantic mappings between the UML language and the DPL language using relations, as discussed in Section 7.4.1. Semantic mappings from DPL to other lan-

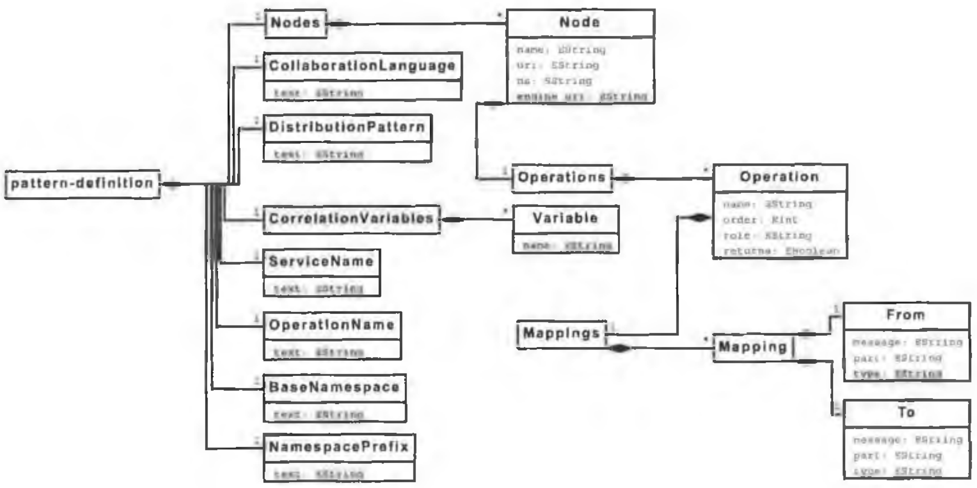


Figure 6.9: DPL notation expressed using a UML Class diagram.

guages, like WS-BPEL, are also defined using relations in Section 7.4.2.

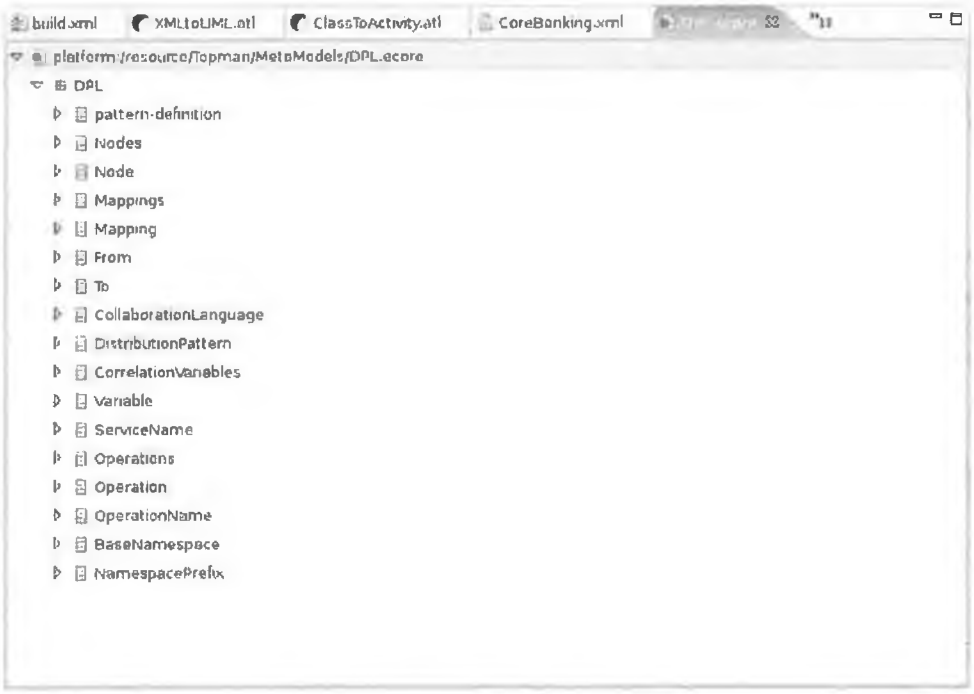
The DPL notation uses static semantics to validate the correctness of instances describing a given distribution pattern. For example, if the centralised distribution pattern has been chosen, the validation process must ensure that all Operation attributes have either the hub or spoke role applied to them. An example of the definition of static semantics can be seen in Section 9.2.4.2.

It should also be considered that the DPL notation has no reliance on UML, thus allowing alternatives to the UML modeling notation approach, such as π calculus [118] and Architecture Description Languages (ADL) [114], discussed in Section 3.3. Both of these languages can be used to describe software architectures, and so potentially distribution patterns. The DPL notation avoids using the constructs associated with existing modeling notations, instead it provides a notation specifically targeted at modeling distribution pattern concerns. These existing modeling notations can have relations and transformations defined towards the DPL notation, as we have done for UML, enabling their use in our methodological framework.

In MDA terminology this notation is a meta-model and is defined at the PIM level, as DPL is not tied to any specific platform technology. As previously noted, this language is novel and so has been defined specifically for our purposes in ECore. Instances of the DPL

Table 6.4: Structured textual semantics of DPL notation constructs.

Attribute	Description
pattern-definition	Holder for all pattern information
Nodes	Holder for Node constructs
Node	A node represents each participant in the composition
Mappings	Holder for Mapping constructs
Mapping	A mapping represents a data exchange between participants
To	Information about where data is going to in an interaction
From	Information about where data is coming from in an interaction
CollaborationLanguage	Choice of collaboration language to enact composition
DistributionPattern	Choice of distribution pattern to be applied to composition
CorrelationVariables	Holder for Variable constructs
Variable	Name of variables used to identify a service instance
ServiceName	Name of the service assigned to the composition
Operations	Holder for Operation constructs
Operation	Details of an operation on a Node
OperationName	Name of the operation to be called to execute the composition
BaseNamespace	URI for the composition
NamespacePrefix	Prefix used to reference the URI for the composition



```
platform:/resource/Topman/MetaModels/DPL.ecore
  DPL
    pattern-definition
    Nodes
    Node
    Mappings
    Mapping
    From
    To
    CollaborationLanguage
    DistributionPattern
    CorrelationVariables
    Variable
    ServiceName
    Operations
    Operation
    OperationName
    BaseNamespace
    NamespacePrefix
```

Figure 6.10: Excerpt of the DPL abstract syntax as viewed in Eclipse.

notation are output from step three of our methodological framework, discussed in Section 9.2.3.

6.4.4 Collaboration Notation

The Collaboration notation provides the constructs necessary to define the choreography of a distribution pattern based composition. The Collaboration notation is a Domain Specific Language (DSL) for choreography description. The notation could be based on a number of collaboration languages such as WS-BPEL or WS-CDL, discussed in Section 2.3.1. However, both of these languages have a different abstract and concrete syntax, along with different semantics, as they are designed for different compositional contexts, namely orchestration and choreography. In our context, the notation is necessary to enable the enactment of a distribution pattern via a composition engine. Our collaboration notation specifically considers one language, WS-BPEL. This notation was chosen because execution engines are readily available for WS-BPEL based language instances, unlike WS-CDL which currently has no execution engine. The notation is based, from an abstract syntax and semantic domain point of view, on WS-BPEL.

The abstract structure of the Collaboration notation is specified by OASIS in [128] using XML Schema. The concrete syntax of the language is expressed using XML or by using numerous graphical WS-BPEL based tools like ActiveEndpoints' BPEL Designer [3]. However, as we only use WS-BPEL internally there is no need to use any concrete graphical representation here. The more common text-based concrete representation is XML. The semantic domain of the Collaboration notation is also specified by OASIS, using plain English in [128]. Semantics are also defined via mappings from the DPL language to the Collaboration notation using relations, as discussed in Section 7.4.2. Semantic mappings from the Collaboration notation to XML are also defined using relations in Section 7.4.6.

We consider only a subsection of the notation, which is specifically related to choreography. As WS-BPEL is an orchestration based language it provides constructs for scenarios that will not be required when considering only choreographies. The notation is deemed adequate as it defines all the constructs necessary for distribution pattern based choreography description. The notation is illustrated in Figure 6.11, while Table 6.5 provides structured textual semantics of the constructs which may be used in the Collaboration notation.

In MDA terminology this notation is a meta-model and is defined at the PIM level, meaning it can be run on any platform that provides a WS-BPEL compliant engine. The Eclipse BPEL Project provides an open source ECore based implementation of the Collaboration notation, derived from the WS-BPEL specification [128]. The language's abstract syntax can be manipulated using an Eclipse EMF based editor [39], as illustrated in Figure 6.12.

Instances of the Collaboration notation are an intermediate output from step five of our methodological framework, discussed in Section 9.2.5. This enables the realisation of a distribution pattern on a compositional engine.

Table 6.5: Structured textual semantics of important Collaboration notation constructs.

Attribute	Description
Process	Holder for constructs which define the process
PartnerLinks	Holder for PartnerLink constructs
PartnerLink	Definition of process service participant
Variables	Holder for Variable constructs
Variable	Definition of data to be held by process
Sequence	Holder for constructs which define a sequential collection of activities
Flow	Holder for constructs which define a concurrent collection of activities
Invoke	Execute an operation on a partner service
Receive	Wait for a matching message to arrive
Assign	Holder for constructs which will update variables
Copy	Update the data held in variables
To	Source of variable copy operation
From	Destination of variable copy operation
Reply	Send a message as a reply to a call received already
Correlations	Holder for Correlation constructs
Correlation	Description of an identifier so an instance can be identified from other instances

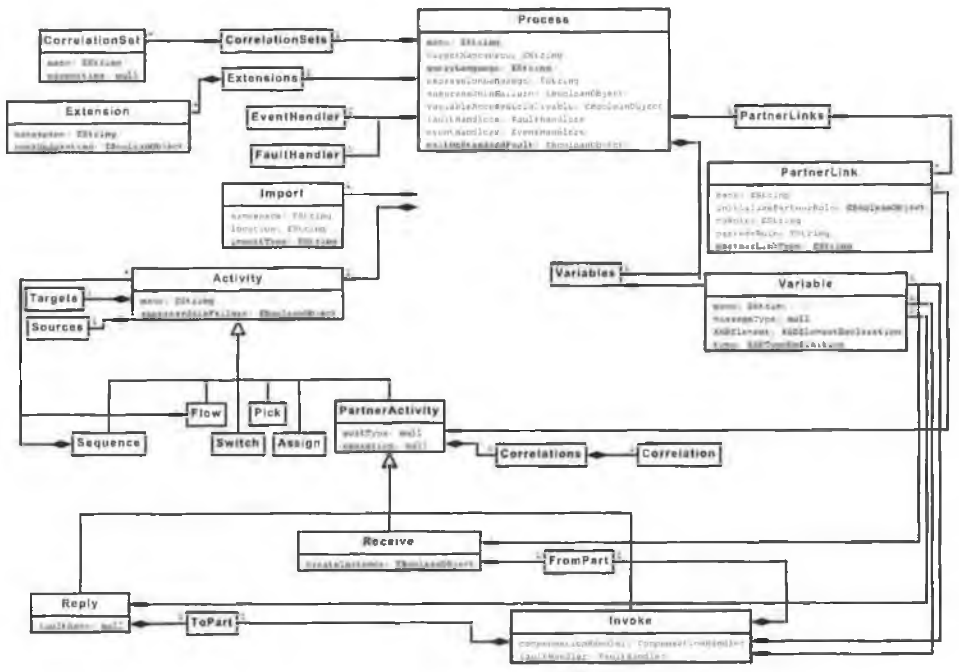
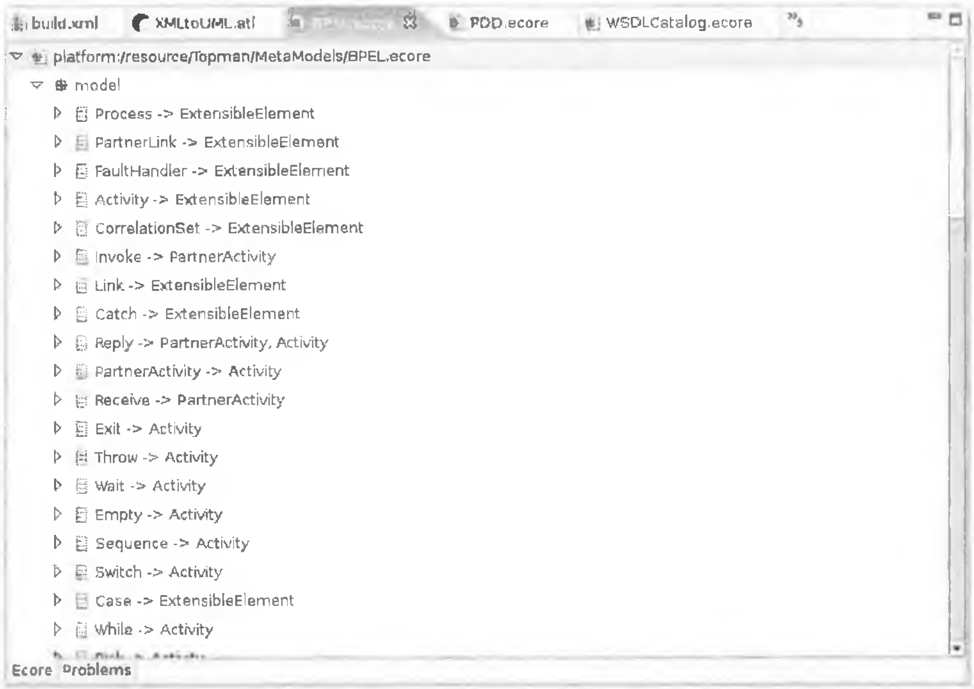


Figure 6.11: Excerpt of Collaboration Notation expressed using a UML Class diagram.

6.4.5 Interface Notation

The Interface notation provides the constructs necessary to define a distribution pattern based compositional interface, which is to be exposed as a Web service, as discussed in Section 2.3. The Interface notation is a Domain Specific Language (DSL) for Web service interface description. The notation is based, from an abstract syntax and semantic domain point of view, on the WSDL specification [184]. The WSDL specification was chosen because it is standards based and compatible with the collaboration notation, previously introduced. The notation is necessary to provide an entry point for the execution of a Collaboration notation instance previously generated.

The abstract structure of the Interface notation is specified by the W3C in [184] using XML Schema. The concrete syntax of the language can be expressed using XML or using UML, as outlined by Provost in [151]. However, because we only use WSDL internally, there is no need to use any concrete graphical representation here. The more common text-based concrete representation is XML. The semantic domain of the Interface notation is



```
platform:/resource/Topman/MetaModels/BPEL.ecore
  model
    Process -> ExtensibleElement
    PartnerLink -> ExtensibleElement
    FaultHandler -> ExtensibleElement
    Activity -> ExtensibleElement
    CorrelationSet -> ExtensibleElement
    Invoke -> PartnerActivity
    Link -> ExtensibleElement
    Catch -> ExtensibleElement
    Reply -> PartnerActivity, Activity
    PartnerActivity -> Activity
    Receive -> PartnerActivity
    Exit -> Activity
    Throw -> Activity
    Wait -> Activity
    Empty -> Activity
    Sequence -> Activity
    Switch -> Activity
    Case -> ExtensibleElement
    While -> Activity
```

Figure 6.12: The Collaboration notation's abstract syntax as viewed in Eclipse.

also specified by the W3C, using plain English in [184]. Semantics are also defined via mappings from the DPL language to the Interface notation using relations, as discussed in Section 7.4.3. Semantic mappings from the Interface notation to XML are also defined using relations in Section 7.4.7.

We consider the notation to be incomplete as it does not define all the constructs necessary for representing an interface of a distribution pattern based choreography description. We were required to extend the language to encompass a number of additional constructs, such as PartnerLinkType and Role. These constructs enable us to expose a WS-BPEL based compositional interface. The notation is illustrated in Figure 6.13, while Table 6.6 provides the structured textual semantics of the constructs that may be used in the Interface notation. Extensions are explicitly noted with an asterisk.

In MDA terminology this notation is a meta-model and is defined at the PIM level, meaning it can be run on any platform that provides a Web service execution environment. The Eclipse Web Standard Tools (WST) provides an open source ECore based implemen-

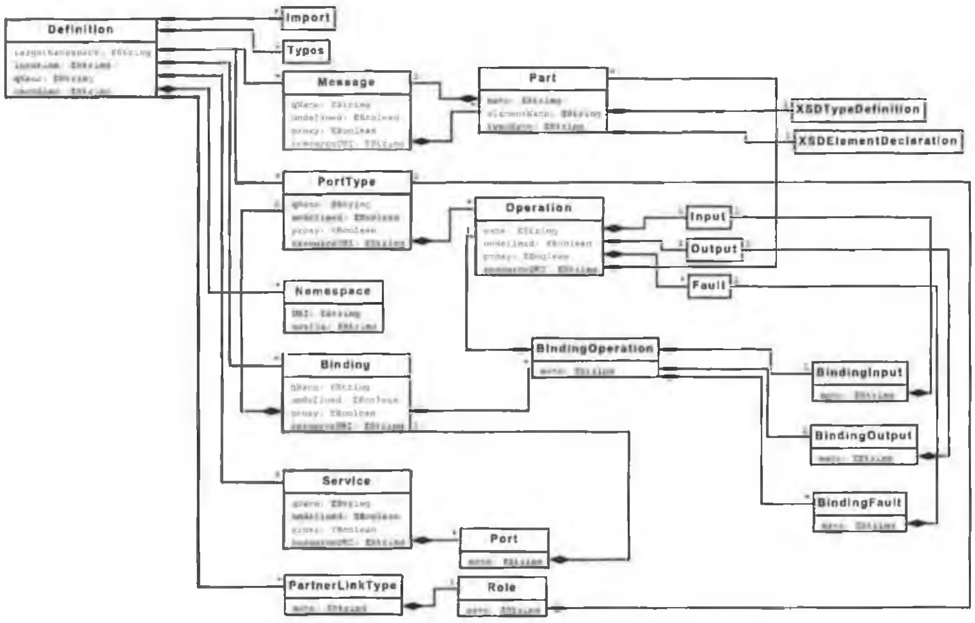


Figure 6.13: Interface Notation expressed using a UML Class diagram.

tation of the Interface notation, derived from the WSDL specification [67]. However, as previously noted, WST had to be extended to enable it to describe compositional interfaces by adding the PartnerLinkType and the Role constructs. The language's abstract syntax can be manipulated using an Eclipse EMF based editor [39], as illustrated in Figure 6.14.

Instances of the Interface notation are an intermediate output from step five of our methodological framework, discussed in Section 9.2.5. This output enables the execution of a distribution pattern on a compositional engine.

6.4.6 Deployment Descriptor Notation

The Deployment Descriptor notation provides the constructs to define a distribution pattern based deployment, to be enacted on a composition engine. The notation is a Domain Specific Language (DSL) for Web service composition deployment. This notation provides the link between Collaboration notation instance(s) and the Interface notation instance(s). The notation could be based on a number of composition engine specific deployment languages.

In our context the notation is necessary to enable the deployment of a distribution pat-

Table 6.6: Structured textual semantics of important Interface notation constructs.

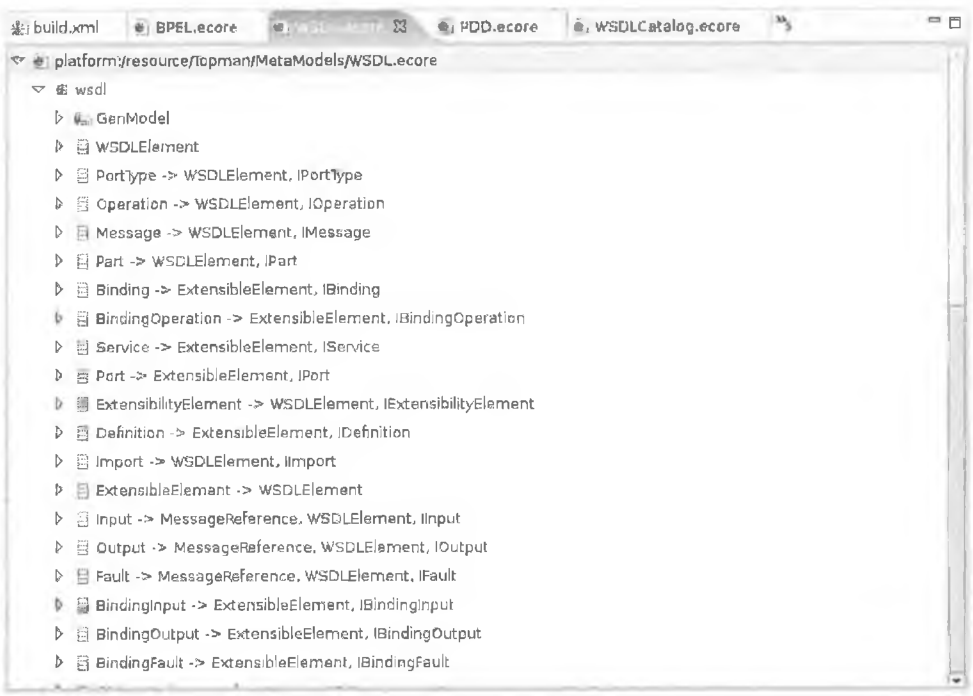
Attribute	Description
Definition	Holder for service interface description
Message	Description of the input or output from or to a service
PortType	Description of a service's abstract interface
Operation	Description of an action provided by a PortType
Part	Description of parameters for a service operation
Binding	Description of the concrete message formats/protocols used by a service
PartnerLinkType*	Details of participants referenced by a given service
Role*	Details of roles played by a participant
Service	Description of how the service can be accessed

tern via a composition engine. Our deployment descriptor notation specifically considers one language, the PDD deployment descriptor, which is part of the Active BPEL composition engine specification [2]. The notation is based, from an abstract syntax and semantic domain point of view, on the PDD language.

The abstract structure of the Deployment notation is outlined by Active Endpoints using an XML Schema, pdd.xsd [2], which itself is defined using EBNF. The concrete syntax of the language is text-based and is expressed using XML. The semantic domain of the Deployment notation is also specified by the Active Endpoints, using plain English in [2]. Semantics are defined via mappings from the DPL language to the Deployment notation using relations, as discussed in Section 7.4.4. Semantic mappings from the Deployment notation to XML are also defined using relations in Section 7.4.8.

The notation is deemed adequate as it defines all the constructs necessary for distribution pattern based deployments. The notation is illustrated in Figure 6.15, while Table 6.7 provides the structured textual semantics of the constructs used in the Deployment notation.

In MDA terminology this notation is a meta-model and is defined at the PIM level, meaning it can be run on any platform that supports the ActiveBPEL engine. We have writ-



```
build.xml   BPEL.ecore   [illegible]   PDD.ecore   WSDLCatalog.ecore
platform:/resource/Topman/MetaModels/WSDL.ecore
  wsdl
    GenModel
    WSDLElement
    PortType -> WSDLElement, IPortType
    Operation -> WSDLElement, IOperation
    Message -> WSDLElement, IMessage
    Part -> WSDLElement, IPart
    Binding -> ExtensibleElement, IBinding
    BindingOperation -> ExtensibleElement, IBindingOperation
    Service -> ExtensibleElement, IService
    Port -> ExtensibleElement, IPort
    ExtensibilityElement -> WSDLElement, IExtensibilityElement
    Definition -> ExtensibleElement, IDefinition
    Import -> WSDLElement, IImport
    ExtensibleElemant -> WSDLElement
    Input -> MessageReference, WSDLElement, IInput
    Output -> MessageReference, WSDLElement, IOutput
    Fault -> MessageReference, WSDLElement, IFault
    BindingInput -> ExtensibleElement, IBindingInput
    BindingOutput -> ExtensibleElement, IBindingOutput
    BindingFault -> ExtensibleElement, IBindingFault
```

Figure 6.14: Excerpt of the Interface notation's abstract syntax as viewed in Eclipse.

ten an ECore based implementation of the ActiveBPEL's deployment descriptor notation, derived from an XML Schema. The language's abstract syntax can be manipulated using an Eclipse EMF based editor [39], as illustrated in Figure 6.16.

Instances of the Deployment Descriptor notation are an intermediate output from step five of our methodological framework, discussed in Section 9.2.5.

6.4.7 Deployment Catalog Notation

The Deployment Catalog notation provides the constructs to enumerate the interfaces of a distribution pattern based deployment to be enacted on a composition engine. The notation is a Domain Specific Language (DSL) for enumerating the interfaces of a Web service based composition deployment. The DSL is necessary to enable a composition engine to find the interface and dependent resources within a deployment archive.

The notation is necessary to enable the deployment of a distribution pattern via a composition engine. Our deployment catalog notation specifically considers one language, WS-

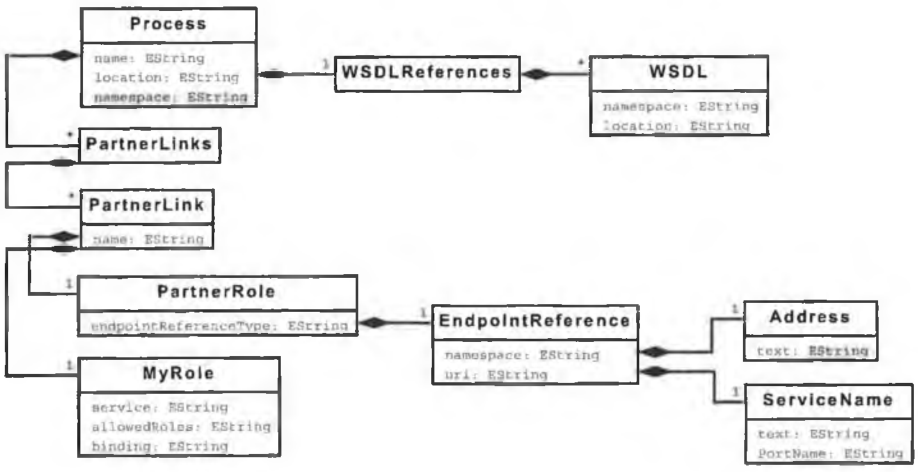


Figure 6.15: Deployment Descriptor Notation expressed using a UML Class diagram.

DLCatalog deployment catalog, which is part of the Active BPEL composition engine specification [2].The notation is based, from an abstract syntax and semantic domain point of view, on the WSDLCatalog language.

The abstract structure of the Catalog notation is not outlined by Active Endpoints. The concrete syntax of the language is text-based and is expressed using XML [2]. The semantic domain of the Catalog notation is also specified by the Active Endpoints, using plain English in [2]. Semantics are defined via mappings from the DPL language to the Catalog notation using relations, as discussed in Section 7.4.5. Semantic mappings from the Catalog notation to XML are also defined using relations in Section 7.4.9.

The notation is deemed adequate as it defines all the constructs necessary for distribution pattern based deployments. The notation is illustrated in Figure 6.17, while Table 6.8 provides the structured textual semantics of the constructs used in the Deployment Catalog notation.

In MDA terminology this notation is a meta-model and is defined at the PIM level, meaning it can be run on any platform that supports the ActiveBPEL engine. We have written an ECore based implementation of the ActiveBPEL's deployment catalog notation, derived from XML examples of ActiveBPEL's deployment catalogs in the engine's provided documentation. The language's abstract syntax can be manipulated using an Eclipse

Table 6.7: Structured textual semantics of important Deployment Descriptor notation constructs.

Attribute	Description
Process	Holder for constructs which describe the deployment
PartnerLinks	Holder for PartnerLink constructs
PartnerLink	Details of the role played by a participant in the composition
MyRole	Details of the composition participant which initiates the composition
PartnerRole	Details of the participants which are not the composition initiator
EndpointReference	Holder for Address and ServiceName constructs
Address	The URI location of a composition participant
ServiceName	Name of the composition participant
WSDLReference	Holder for WSDL constructs
WSDL	The location/classpath of an interface in the deployment archive

Table 6.8: Structured textual semantics of important Deployment Catalog notation constructs.

WSDLCatalog	Holder for WSDLReference and XSDReference constructs
Entry	Generic construct for location and classpath data storage
WSDLEntry	The location and classpath of a WSDL interface in the deployment archive
SchemaEntry	The location and classpath of an XML Schema in the deployment archive

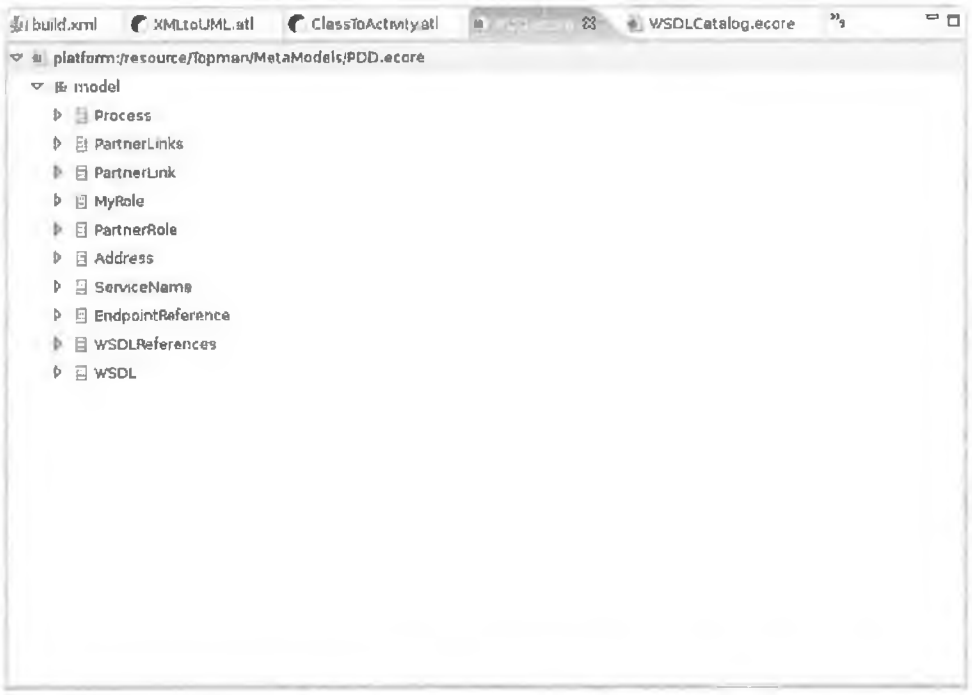


Figure 6.16: The Deployment notation's abstract syntax as viewed in Eclipse.

EMF based editor [39], as illustrated in Figure 6.18.

Instances of the Deployment Descriptor notation are an intermediate output from step five of our methodological framework, discussed in Section 9.2.5.

6.4.8 XML Notation

The XML notation provides the constructs to define an XML based document. The XML notation is a General Purpose Language (GPL) for structuring data. The notation is based on the XML specification [188]. The notation is necessary as the Collaboration, Interface, Deployment and Deployment Catalog notations must be translated to XML for execution.

The abstract structure of the XML notation is specified by the W3C in [188] using EBNF. The concrete syntax of the language is tree based plain text. The semantic domain of the XML notation is specified by the W3C, using plain English in [188]. Semantics are also defined via mappings from the Collaboration, Interface, Deployment and Deployment Catalog notations using relations, as discussed in Chapter 7.

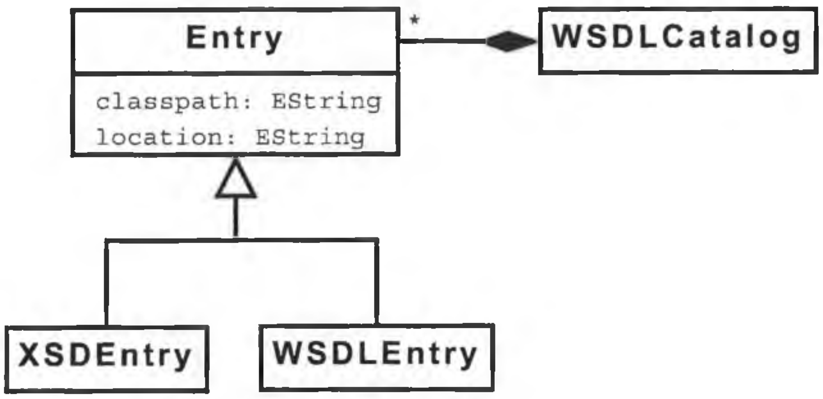
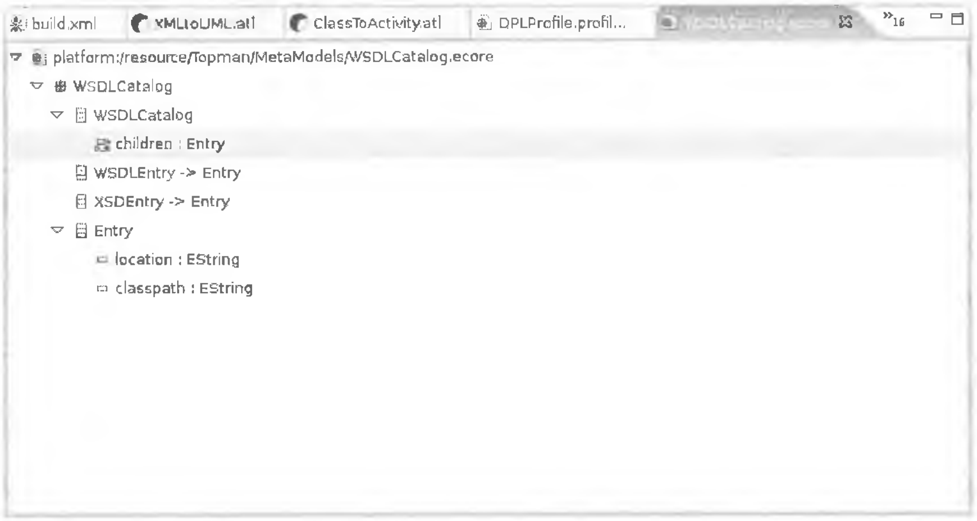


Figure 6.17: Deployment Catalog Notation expressed using a UML Class diagram.



```
platform:/resource/Topman/MetaModels/WSDLCatalog.ecore
  WSDLCatalog
    WSDLCatalog
      children : Entry
    WSDLEntry -> Entry
    XSDEntry -> Entry
    Entry
      location : EString
      classpath : EString
```

Figure 6.18: The Catalog notation's abstract syntax as viewed in Eclipse.

One particular form of XML is XMI (XML Metadata Interchange). XMI, discussed in Section 2.5.7.1 is the serialisation format for all the languages already introduced. There is a semantic difference between the two languages, which can be reconciled using a tool called AMMA [33]. This tool has semantic mappings between the languages, enabling XML injectors and extractors to transform XMI to XML and XML to XMI.

The notation was deemed adequate as the notation is capable of representing a wide range of structured information, including all the other notations defined here. The notation is illustrated in Figure 6.19, while Table 6.9 provides the structured textual semantics of the constructs which may be used in the XML notation.

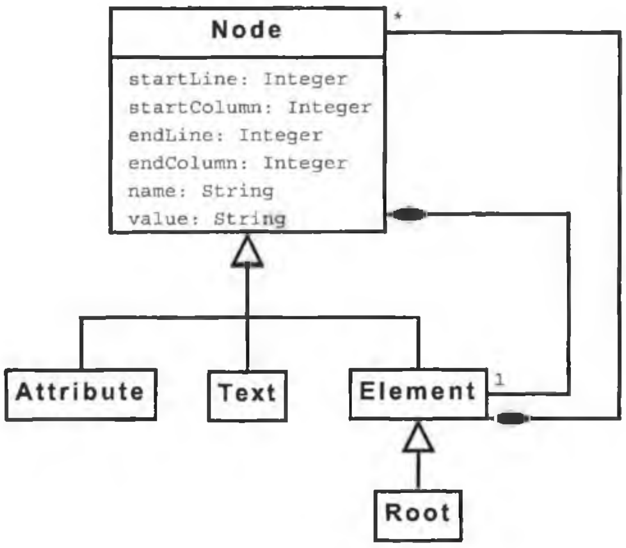


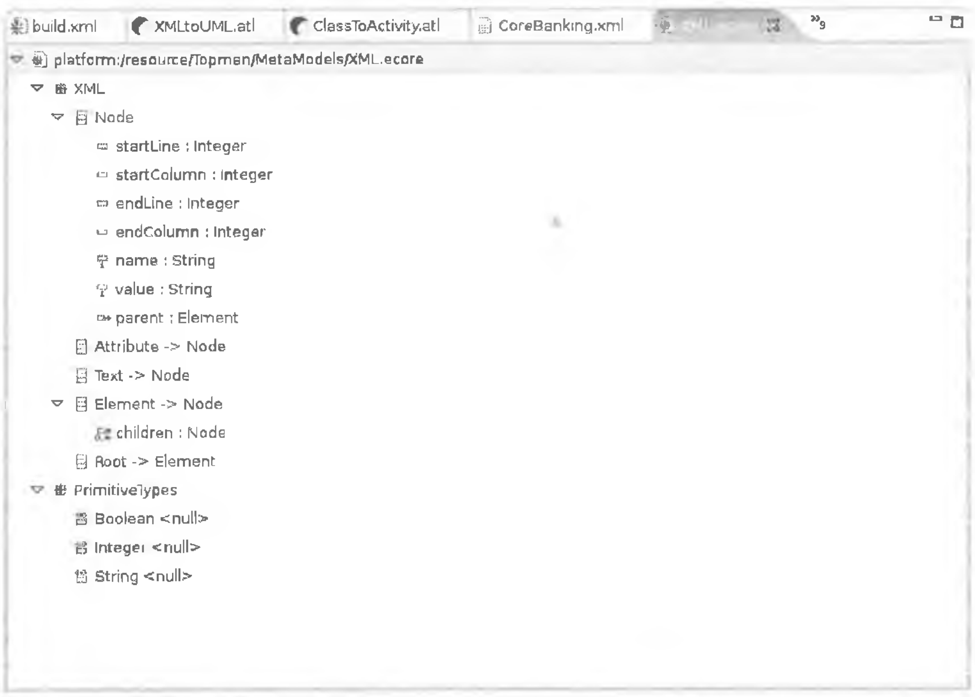
Figure 6.19: XML Notation expressed using a UML Class diagram.

Table 6.9: Structured textual semantics of important XML notation constructs.

Node	Generic construct for data representation
Root	Holder for Node constructs, there is only ever one root
Element	Named structure which can contain both Text and Attribute constructs
Attribute	Name value pair for data
Text	Free area for text data

In MDA terminology this notation is a meta-model and is defined at the PIM level, meaning it can be run on any platform that that provides an XML parser. An open source example provided by the ATL project provides an ECore based meta-model, derived from the XML specification [70]. The language's abstract syntax can be manipulated using an Eclipse EMF based editor [39], as illustrated in Figure 6.20.

Instances of the XML notation are an intermediate output from step five of our methodological framework, discussed in Chapter 9.



```
build.xml   XMLtoUML.atl   ClassToActivity.atl   CoreBanking.xml   »9
▽ platform:/resource/Topmen/MetaModels/XML.ecore
  ▽ XML
    ▽ Node
        startLine : Integer
        startColumn : Integer
        endLine : Integer
        endColumn : Integer
        name : String
        value : String
        parent : Element
      Attribute -> Node
      Text -> Node
    ▽ Element -> Node
        children : Node
      Root -> Element
  ▽ PrimitiveTypes
      Boolean <null>
      Integer <null>
      String <null>
```

Figure 6.20: The XML notation's abstract syntax as viewed in Eclipse.

6.5 Tool Support

A number of tools are currently available to assist in the generation of modeling infrastructure. Eclipse Modeling Framework (EMF), discussed in Section 2.5.7.2, is one such open source tool. EMF provides editors to create ECore models and code generation facilities to generate model editors based on meta-models. The IBM tool Rational Software Architect (RSA) [107] provides a visual modeling tool, which can be used to assist in the generation of ECore meta-models, along with EMF based models and model instances.

Another open source tool, NetBeans Meta Data Repository (Netbeans MDR), implements the MOF specification, providing facilities like MOF meta-model storage, import and export [111]. Meta-data in the MDR repository can be created, stored, retrieved and interchanged using the standardised Java Metadata Interface (JMI) Java API [171]. The MOF based meta-models can be designed in tools such as Poseidon [76] or MagicDraw [87], and subsequently exported as XMI, before being stored and manipulated in the MDR repository. A UML2MOF tool is also provided by MDR to enable the definition of meta-models using UML, and subsequent conversion to a MOF compliant meta-metamodel.

A MOF based tool, called MOmo, is also considered by Bicher in [34]. Here, a MOF based meta-model is defined and exported by a tool such as ArgoUML [178]. The MOmo tool then converts this MOF representation into an object model, enabling end users to create instances of the model in code.

6.6 Summary

In this chapter we have presented eight notations, or languages, required to facilitate the modeling of distribution patterns, and the subsequent generation of an executable system. Two of these languages, DPL and DPLProfile, are novel, while one, WSDL, has been extended to enable it to be used to expose the interface of a WS-BPEL based composition. Each language has been motivated, its syntax defined, and its semantics outlined. The languages, once defined, have been placed in an MDA based context to enable code generation. Each of the languages are required by the five step methodological framework, outlined in Chapter 9.

Chapter 7

Model Relations

7.1 Introduction

In this chapter we present the third component in our modeling and transformation framework, model relations. Relations provide the semantic mappings between the modeling notations previously defined in Chapter 6. These semantic mappings define the web of dependencies that must hold between source and target modeling notation. The model relations vary slightly depending on the distribution pattern chosen by the software architect.

In Section 7.2 we motivate the use of model relations. The MDA framework within which the relations must be defined is outlined in Section 7.3. A full set of relations are presented in Section 7.4. We conclude the chapter by investigating the tool support for defining model relations in Section 7.5.

7.2 Model Relation Definitions and Semantics

Model relations enable preservation of information between notations. The goal of defining relations between notations is to record the process by which information is related between notations, thus ensuring modeling information is preserved from one model instance to the next. These relations are considered abstract specifications or constraints, in that they are not themselves executable. Model relations can be used as a template for model transformations, which are executable.

Information preservation is achieved using tracing techniques. Traceability can ensure system quality by establishing the purpose for a given software development artifact, in our context modeling artifacts, existing in a given system [1]. During system development traceability enables developers to monitor the effect of changes throughout the system, potentially increasing the quality of the system produced. Aizenbud-Reshef et al. suggest a more general definition of traceability, where explicit relationships are defined between software artifacts during system development [124]. We consider the most basic form of traceability, where the semantics of relational links are not considered. Here, links indicate a relationship between modeling artifacts, without implying any specific type of relationship. We achieve traceability by separating the treatment of abstract semantic relations, discussed in this chapter, and the executable implementation of top-down transformations in Chapter 8.

Relations can be defined informally using plain English or through examples, while formal definitions include graph transformations and mathematics [82]. Relations can also be defined between a language with no semantic domain and a language with a semantic domain. The undefined language then takes on the semantics of the related language [105].

7.3 Framework

In Section 6.3 we discussed how we leverage the MDA framework to enable code generation from high level models. The model relations defined in this chapter must be compatible with this MDA approach. Consequently, the relations defined between notations utilise the recently standardised QVT (Query/View/Transformation) language [133] outlined in Section 2.5.7.7.

We have chosen to use QVT graphical notation to declaratively illustrate the relations that must hold for transformations between candidate models to be performed correctly. The QVT notation lends itself to such definitions as it is based upon UML Object diagrams

[60], which provides for an intuitive view of relations along with their selection patterns. UML Object diagrams have been extended to allow for the specification of patterns within a relation, along with the use of a new diamond shaped symbol to denote a transformation. We envisage that software architects will be able to easily comprehend these diagrams. These relations will later be used to assist in the declaration of transformations as discussed in Chapter 8. It should be noted that the same language has not been used to define both relations and transformations. Separate languages were required to adequately express the relations graphically and to define executable transformations. The QVT language is used to illustrate relations graphically but has only immature tool support to enable execution. There is currently no tool available for defining QVT relations graphically, making the definitions in this chapter difficult to define. We have used a general purpose drawing tool, Dia [77], to draw the relations. Our transformation language of choice, ATL, has no graphical representation but has well supported tools and an execution environment.

7.4 Relations

Relations provide the semantic mappings between seven of the eight modeling notations defined in Chapter 6. The DPLProfile notation is not related to other notations as it is directly applied to the UML 2.0 notation. Each of these notations describes part of a Web service composition from a different perspective. The relations define the web of dependencies that must hold between pairs of candidate notations, a source and a target notation. These candidate notations are called meta-models in MDA terminology.

Each "relations" box in Figures 7.1 through 7.3 maps to a set of QVT relations. A QVT relation requires the definition of two or more domains. The source domain refers to a particular artifact in the source notation, whilst the target domain refers to an artifact in the target notation. The source domain may have a source pattern defined, which restricts the search space over the source artifacts, and also binds source model values to variables. A destination pattern may also be defined, to apply source model matches, and subsequently bind variables to destination model variables. These variables represent model instance artifacts of both the source and target models. In addition, a *when* and *where* predicate can be defined in relations. The *when* predicate specifies pre-conditions or relations which must have been previously executed before this relation will hold. The *where* predicate specifies post-conditions or relations which should be run after this relation.

Using seven of the notations outlined in Chapter 6, we consider nine relation sets which define relations from the high level UML notation to the lower level executable notations. These nine relation sets can be categorised into three subsets as follows.

- UML 2.0 Notation (with DPLProfile applied) to Distribution Pattern Language (DPL) Notation
- DPL Notation to Executable System Notations (Collaboration, Interface, Deployment Descriptor, Deployment Catalog)
- Executable System Notations to XML Notation

Figures 7.1 through 7.3, below, illustrate how the relations between the notations in our modeling approach will be defined.

The UML 2.0 notation (with DPLProfile applied) to DPL notation relation outlined in the left of Figure 7.1 is a bi-directional relationship. The two notations are equivalent as they both represent the same level of modeling information. We note in Section 6.4.3 the motivation for implementing a new language is to provide for ease of analysis, verification and transformation of distribution patterns. The target DPL notation is a simpler representational format than the source UML 2.0 notation when used to model distributions, making transformation definitions simpler.

The DPL to Executable System models in Figure 7.2 are uni-directional relationships. The figure illustrates a more detailed view of the previously illustration relations between the DPL notation and the Executable system notation in Figure 7.1. The source DPL notation is related to different target notations depending on the information requirements of the target notations. Attempting to reverse engineer any one of these target models in iso-

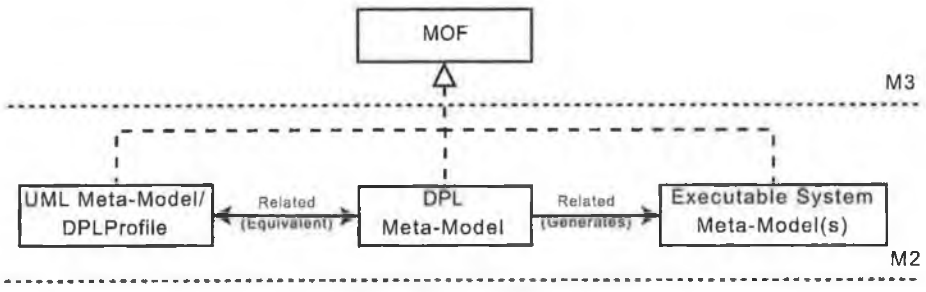


Figure 7.1: Relations between UML 2.0 (with DPLProfile applied), DPL and Executable system notations.

lation would result in an incomplete DPL source model. However, combining all the target models together would result in a complete DPL source model.

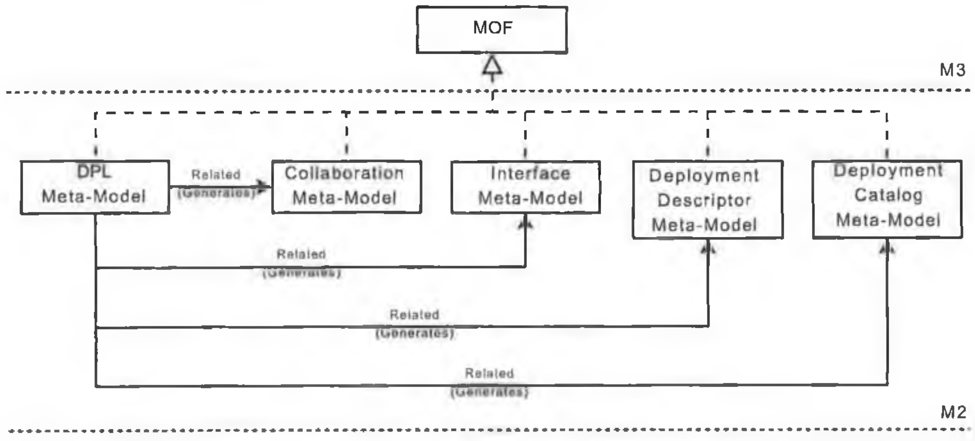


Figure 7.2: Relations between DPL and Executable System notations.

Finally the Executable System to XML models in Figure 7.3 are bi-directional relationships. The two notations are equivalent as they both represent the same level of modeling information, albeit in different formats. The target XML notation represents the ubiquitous XML format.

We consider relations at the notation or meta-model layer, whereby relations are defined between source and destination notations or meta-models. Specific source notation elements are identified, using selection patterns, and related to destination notation elements, according to a set of rules or relations.

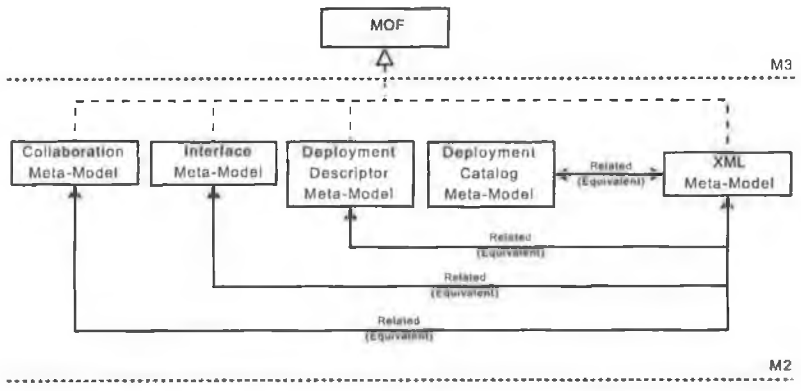


Figure 7.3: Relations between Executable System and XML notations.

Each set of model relations are either distribution pattern independent or distribution pattern dependent. Relations which are pattern dependent define relationships between source artifacts and different target artifacts, depending on the distribution pattern chosen. Pattern independent relations define relationships between source artifacts and target artifacts, which are always the same regardless of the chosen pattern. The UML to Distribution Pattern relations and the Executable System to XML relations are pattern independent, whilst the remaining relations are pattern dependent. These pattern dependent relations differ slightly depending on the individual pattern requirements.

In addition to relations we also define functions. Functions are simple operations performed over the source model to return either collections of model artifacts, individual model artifacts or simple return values i.e. strings, booleans or integers. These functions do not relate source artifacts to target artifacts. Instead the functions assist in the definition of relations. All functions are prefixed with either "get" or "convert".

To demonstrate the definition of model relations from a UML model to an executable system, we present in the following subsections a centralised shared hub worked example. This example features a centralised shared hub distribution pattern, outlined in Section 5.4.1. The relations are complete, meaning they cover all the important constructs of all the notations outlined in Chapter 6. The relations enable us to semantically relate newly defined languages to existing languages, such as from UML to DPL. The relations are implemented using transformations in Chapter 8. The effectiveness of the relations, and subsequently the transformations, is assessed in Chapter 10.

7.4.1 Relating UML 2.0 Notation/DPLProfile to Distribution Pattern Notation

UML 2.0 is a standards based graphical language for the modeling of software systems, as discussed in Section 2.5.2. DPL is our internal representation format for distribution patterns. The first relation set is from the UML 2.0 notation to the Distribution Pattern notation, where the UML 2.0 and the Distribution Pattern notations are the candidate models. The UML 2.0 notation is used to describe distribution patterns using standard UML 2.0 Activity diagram constructs and a novel profile DPLProfile, while the DPL notation provides the constructs for the internal representation of a distribution pattern. These relations are bi-directional as the two models are representations of the same information.

The transformation declaration, which holds all the relations, is expressed using QVT textual notation, as illustrated in Figure 7.4. This relation set is pattern independent meaning these relations, unless otherwise indicated, hold across all distribution patterns. There is no graphical representation of this transformation definition as it is simply a declaration of the notations. The transformation is defined in Figure 7.4.

```
transformation umlactivityTodpl(ua1:UMLActivity, dpl1:DPL)
{
....
}
```

Figure 7.4: Textual QVT umlactivityTodpl transform declaration.

The five relations in this set are outlined as follows, and described in more detail below. The five relations show how the metaclasses that are stereotyped in the DPLProfile are mapped to the DPL notation.

- ActivityToPatternDefinition
- ActivityPartitionToNode
- CallBehaviorActionToNode

- ObjectFlowToMapping
- PinToCorrelationVariables

The first relation, ActivityToPatternDefinition, defines the relation between a UML Activity artifact and a DPL PatternDefinition artifact. The relation is a top level relation and so will be matched rather than being called directly by another relation. We can see the UML Activity has had the dplActivity stereotype applied to it, allowing for distribution specific values to be applied and subsequently related to the DPL notation. The properties of both notations are bound using the same variable names, meaning they share the same values. The *where* clause indicates that a function getActivityPartitions (retrieves all the UML ActivityPartitions in the source model) must be subsequently executed. The relation is expressed graphically in Figure 7.5.

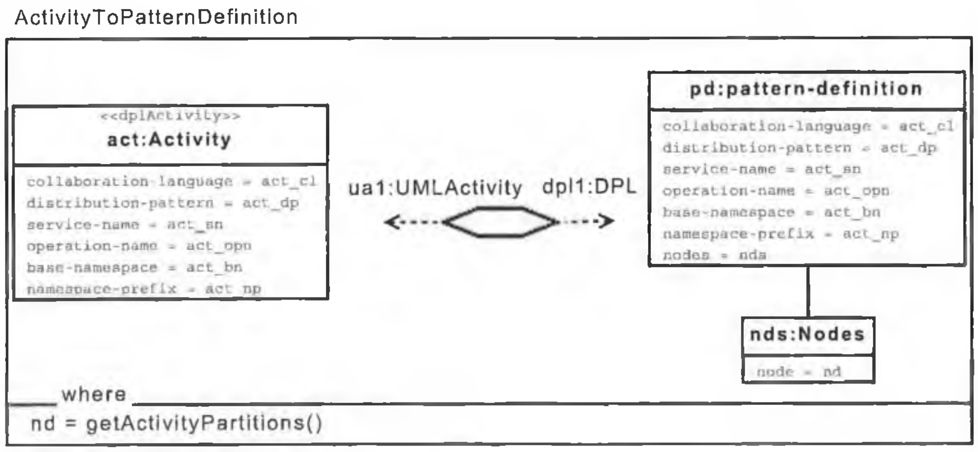


Figure 7.5: Graphical QVT ActivityToPatternDefinition relation declaration.

The second relation, ActivityPartitionToNode, defines the relation between a UML ActivityPartition artifact and a DPL Node artifact. Both the UML ActivityPartitions and CallBehaviorActions artifacts have applied stereotypes, in this case dplPartition and dplParticipant respectively. The relation is top level thus negating the need for a *when* clause. The *where* clause indicates that the relation CBAToOperation must be subsequently executed. The relation is expressed graphically in Figure 7.6.

The third relation, CallBehaviorActionToNode, defines the relation between a UML

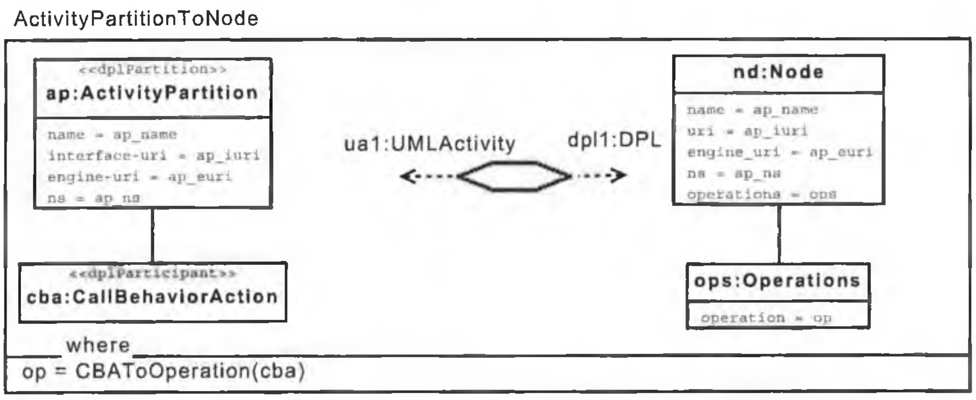


Figure 7.6: Graphical QVT ActivityPartitionToNode relation declaration.

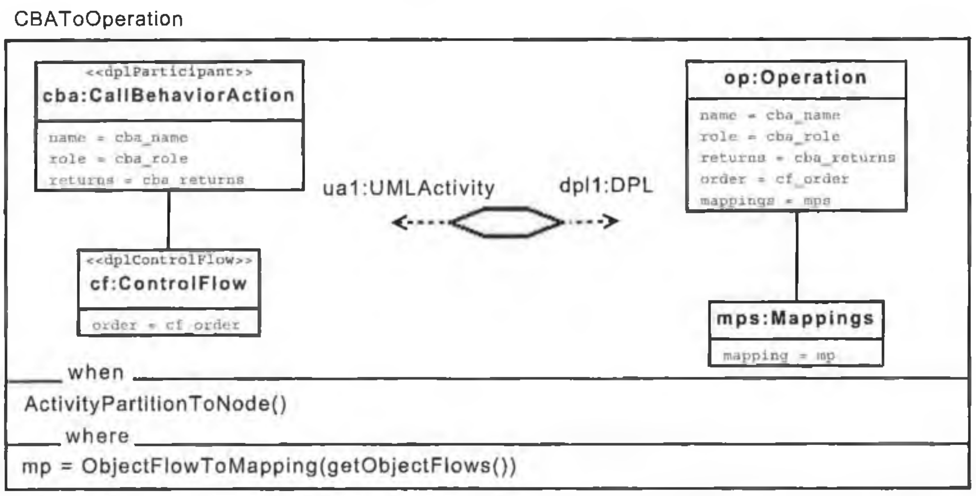
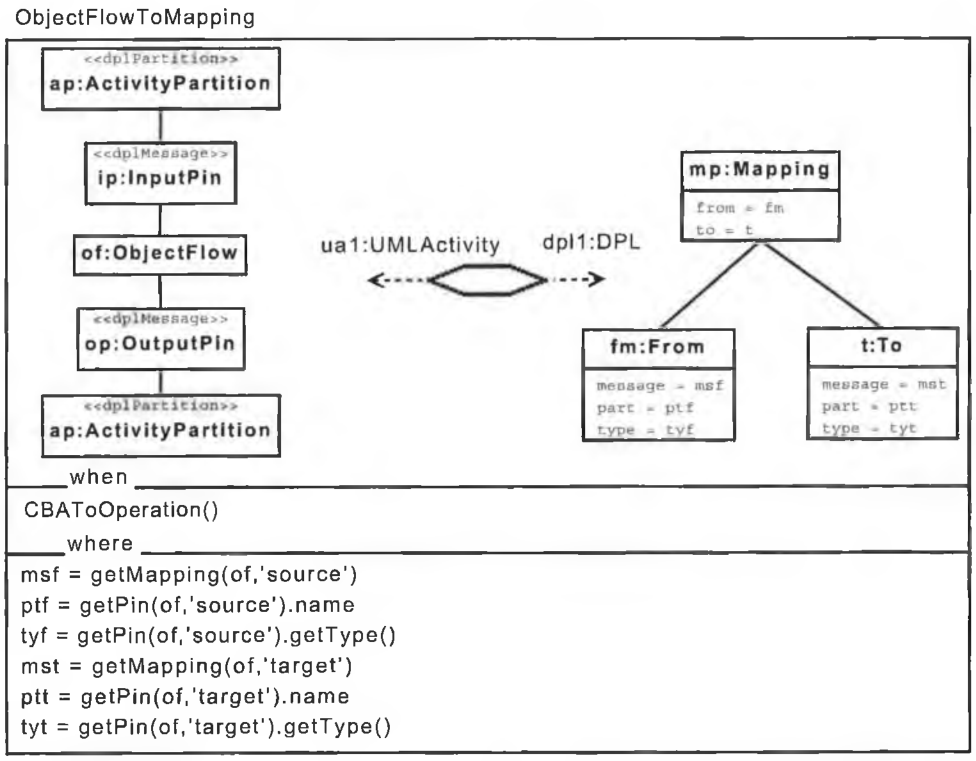


Figure 7.7: Graphical QVT CBAToOperation relation declaration.

CallBehaviorAction artifact and a DPL Operation artifact. As in the previous relations, stereotypes have been applied to the UML CallBehaviorActions and ControlFlows artifacts. The relation is called by the second relation, ActivityPartitionToNode, which is specified in the *when* clause. The *where* clause specifies that the relation ObjectFlowToMapping and the function getObjectFlows (retrieves all the UML ObjectFlows in the source model) must be subsequently executed. The relation is expressed graphically in Figure 7.7.

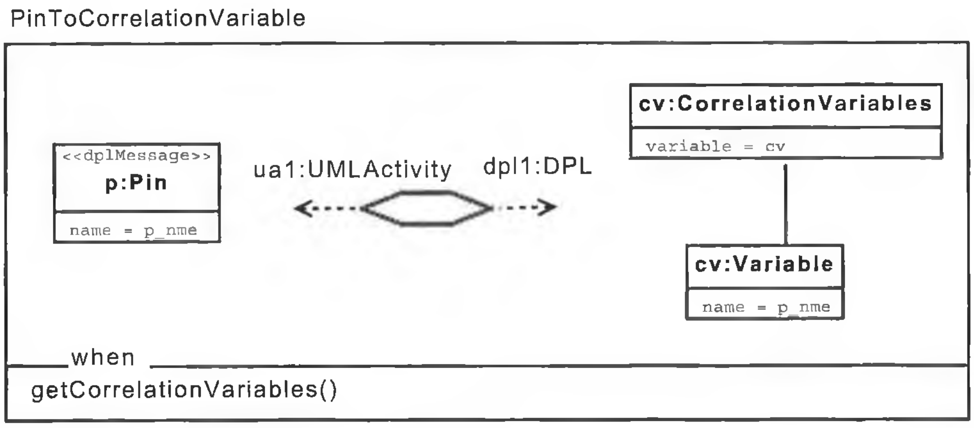
The fourth relation, ObjectFlowToMapping, defines the relation between a UML ObjectFlow connector artifact and a DPL Mapping artifact. The relation matches UML Input/Output Pin information, and relates them to DPL Mappings. The relation is called



```
CBAToOperation()
```

```
msf = getMapping(of,'source')
ptf = getPin(of,'source').name
tyf = getPin(of,'source').getType()
mst = getMapping(of,'target')
ptt = getPin(of,'target').name
tyt = getPin(of,'target').getType()
```

Figure 7.8: Graphical QVT ObjectFlowToMapping relation declaration.



```
getCorrelationVariables()
```

Figure 7.9: Graphical QVT PinToCorrelationVariables relation declaration.

by the third relation, CallBehaviorActionToNode, which is specified in the *when* clause. The *where* clause specifies a number of functions which be subsequently executed, such as getMapping (evaluates the source or destination pin of a UML ObjectFlow mapping), getPin (evaluates the source or target UML Pin for a UML ObjectFlow) and finally getType (evaluates the type of a UML Pin). The relation is expressed graphically in Figure 7.8.

The fifth relation, PinToCorrelationVariables, defines the pattern specific relation between a UML Pin artifact and a DPL Variable artifact. The relation is called by the getCorrelationVariables function for patterns that require CorrelationVariables, as specified in the *when* clause i.e. decentralised distribution patterns. This relation is not called for centralised distribution patterns. We have included the relation here to exemplify distribution pattern conditional relations. The relation is expressed graphically in Figure 7.9.

7.4.2 Relating DPL Notation to Collaboration Notation

The second relation set is from the DPL notation to the Collaboration notation, where the DPL notation and the Collaboration notation are the candidate models. The DPL notation provides the constructs for the internal representation of a distribution pattern, while the Collaboration notation provides the constructs necessary to define the choreography of a distribution pattern based composition. These relations are uni-directional as the target model only represents some of the candidate model information. We assume in this example the collaboration language, and subsequently target notation, will be WS-BPEL 2.3.2. It should be noted that other collaboration languages, including WS-CDL, could have been used as an alternative to WS-BPEL. We have chosen to use WS-BPEL because it has a number of execution engines unlike WS-CDL.

This relation set is pattern dependent because the target notation's artifacts, along with the models outputted, depend on the chosen distribution pattern. This pattern dependency is caused by the fact that different target artifacts must be considered by the relations, as different distribution patterns have differing mechanisms for maintaining state, amongst other concerns. For example, centralised distributions maintain state at a central hub, while decentralised distributions distribute state amongst the participants and utilise the DPL language's CorrelationVariables construct as a unique composition session identifier. Also, centralised distributions require only one Collaboration model to describe a composition, while decentralised distributions require a Collaboration model for each participant and the composition originator. As previously stated, for the purpose of our worked example, we assume the use of a centralised distribution pattern. The relation is expressed in Figure 7.10.

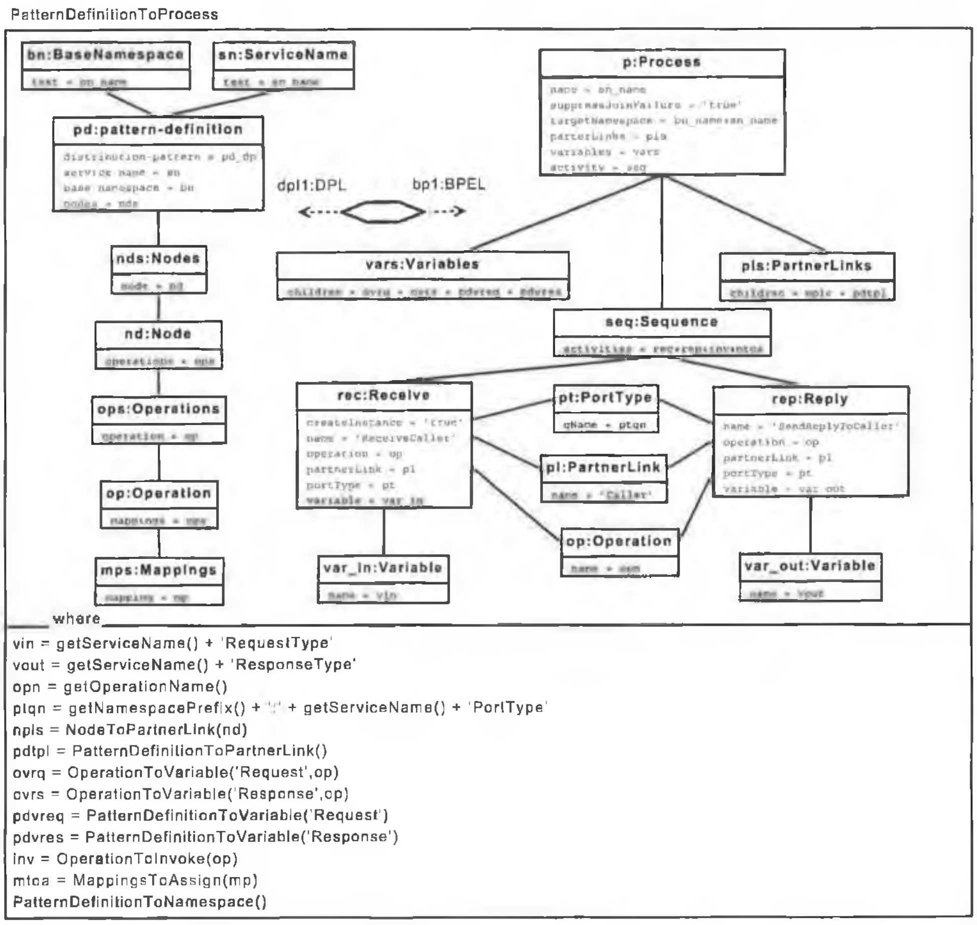
```
transformation dplTobpel(dpl1:DPL, bpel1:BPEL)
{
....
}
```

Figure 7.10: Textual QVT dplTobpel transform declaration.

The ten relations in this set are outlined as follows, and described in more detail below. The ten relations show how the DPL artifacts are mapped to the BPEL notation.

- PatternDefinitionToProcess
- NodeToNamespace
- OperationToVariable
- NodeToPartnerLink
- PatternDefinitionToNamespace
- OperationToInvoke
- MappingToAssign
- PatternDefinitionToVariable
- PatternDefinitionToNamespace
- PatternDefinitionToPartnerLink

The first relation PatternDefinitionToProcess defines the relation between a DPL PatternDefinition artifact and a BPEL Process artifact. The relation is a top level relation and



```
vin = getServiceName() + 'RequestType'
vout = getServiceName() + 'ResponseType'
opn = getOperationName()
ptqn = getNamespacePrefix() + ':' + getServiceName() + 'PortType'
npls = NodeToPartnerLink(nd)
pdtpl = PatternDefinitionToPartnerLink()
ovrq = OperationToVariable('Request',op)
ovrs = OperationToVariable('Response',op)
pdvreq = PatternDefinitionToVariable('Request')
pdvres = PatternDefinitionToVariable('Response')
inv = OperationToInvoke(op)
mtoa = MappingsToAssign(mp)
PatternDefinitionToNamespace()
```

Figure 7.11: Graphical QVT PatternDefinitionToProcess relation declaration.

so will be matched rather than being called directly by another relation. As in the first relation set the properties of both notations are bound using the same variable names, meaning they share the same values. The *where* clause indicates that a number of functions and relations must be subsequently executed. The functions are getServiceName (retrieve the name assigned to the composition), getOperationName (retrieve the operation name assigned to the composition) and getNamespacePrefix (retrieve the namespace prefix for the composition). The relations to be executed are NodeToPartnerLink, OperationToInvoke, MappingsToAssign, PatternDefinitionToPartnerLink, PatternDefinitionToVariable, PatternDefinitionToNamespace and OperationToVariable. The relation is expressed graphically in Figure 7.11.

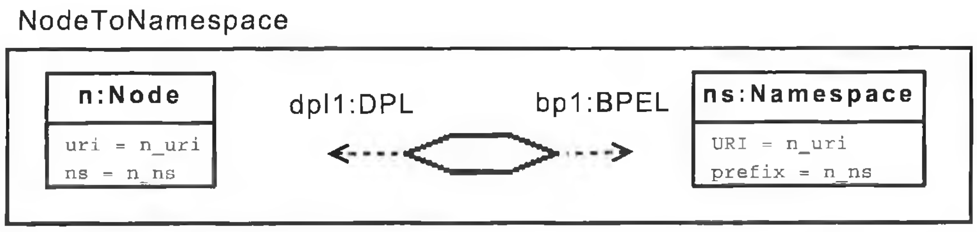


Figure 7.12: Graphical QVT NodeToNamespace relation declaration.

The second relation, NodeToNamespace, defines the relation between a DPL Node artifact and a BPEL Namespace artifact. The relation is a top level relation and so will be matched rather than being called directly by another relation. No *when* or *where* clause is specified. The relation is expressed graphically in Figure 7.12.

The third relation, OperationToVariable, defines the relation between a DPL Operation artifact and BPEL Variable artifacts. The relation is called by another relation, PatternDefinitionToProcess, as specified in the *when* clause. No *where* clause is specified. The relation is expressed graphically in Figure 7.13.

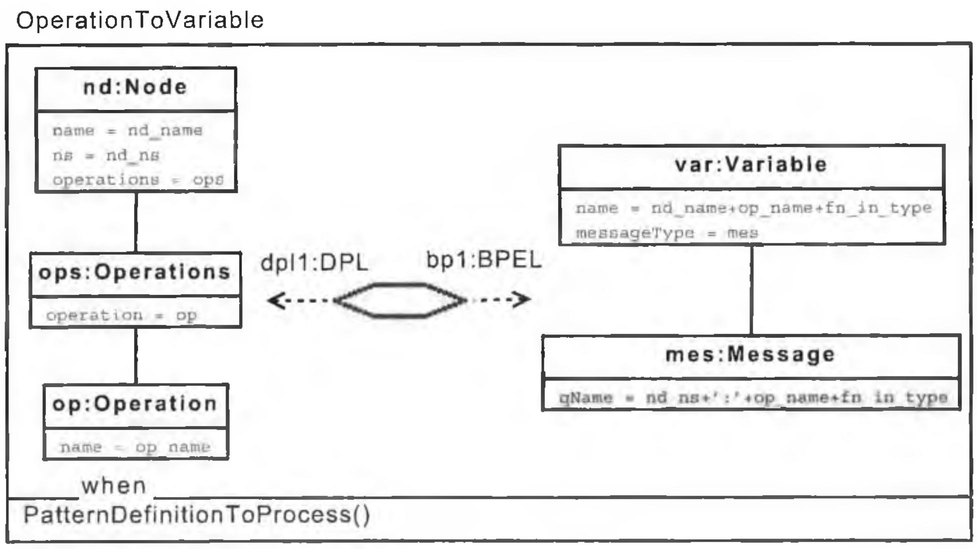


Figure 7.13: Graphical QVT OperationToVariable relation declaration.

The fourth relation, NodeToPartnerLink, defines the relation between a DPL Node artifact and a BPEL PartnerLink artifact. The relation is called by another relation, PatternDefinitionToProcess, as specified in the *when* clause. The *where* clause specifies the function

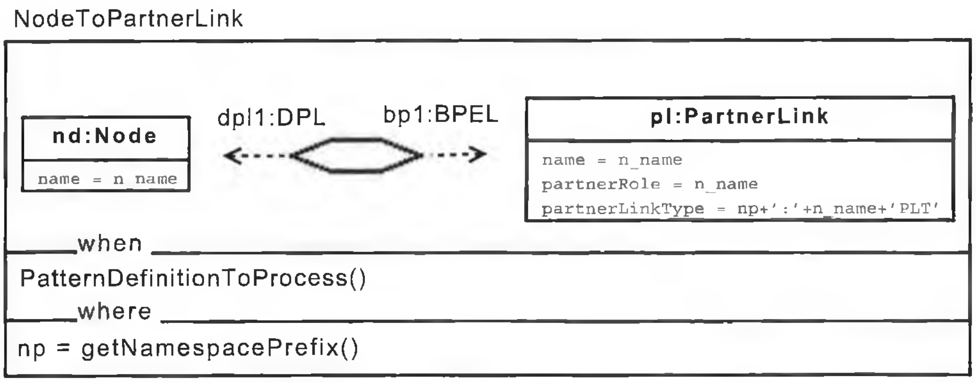


Figure 7.14: Graphical QVT NodeToPartnerLink relation declaration.

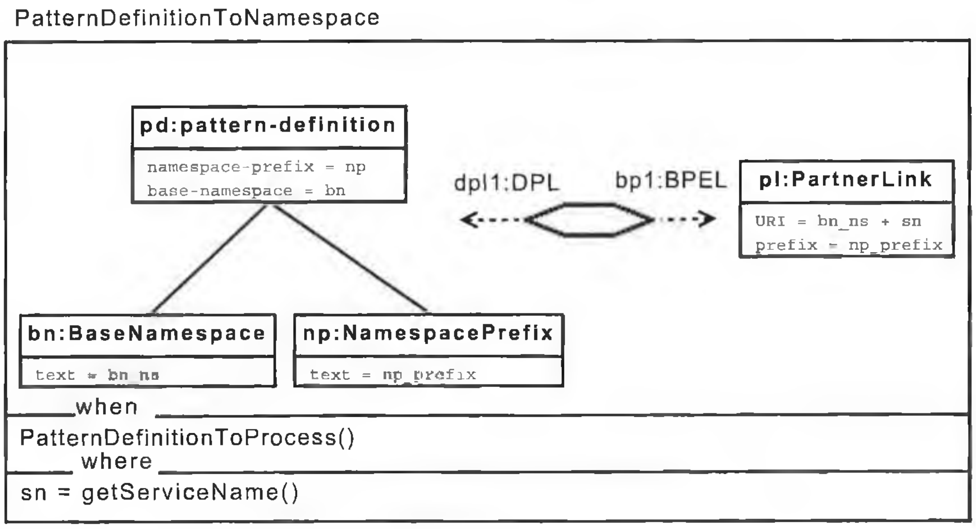


Figure 7.15: Graphical QVT PatternDefinitionToNamespace relation declaration.

getNamespacePrefix (retrieve the namespace prefix for the composition) must be executed subsequent to this relation. The relation is expressed graphically in Figure 7.14.

The fifth relation, PatternDefinitionToNamespace, defines the relation between a DPL PatternDefinition artifact and a BPEL Namespace artifact. The relation is called by another relation, PatternDefinitionToProcess, which is specified in the *when* clause. The *where* clause specifies the function getServiceName (retrieve the name for the composition) must be executed subsequent to this relation. The relation is expressed graphically in Figure 7.19.

The sixth relation, OperationToInvoke, defines the relation between a DPL Operation artifact and a BPEL Invoke artifact. The relation is called by another relation, PatternDefi-

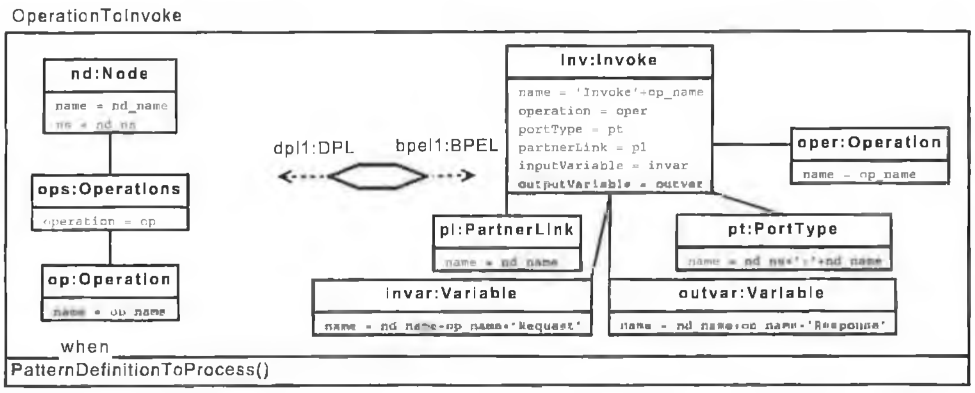


Figure 7.16: Graphical QVT OperationToInvoke relation declaration.

nitionToProcess, as specified in the *when* clause. No *where* clause is specified. The relation is expressed graphically in Figure 7.16.

The seventh relation, MappingToAssign, defines the relation between a DPL Mapping artifact and a BPEL Assign artifact. The *when* clause specifies that the PatternDefinition-ToProcess relation must have been previously executed. No *where* clause is specified. The relation is expressed graphically in Figure 7.17.

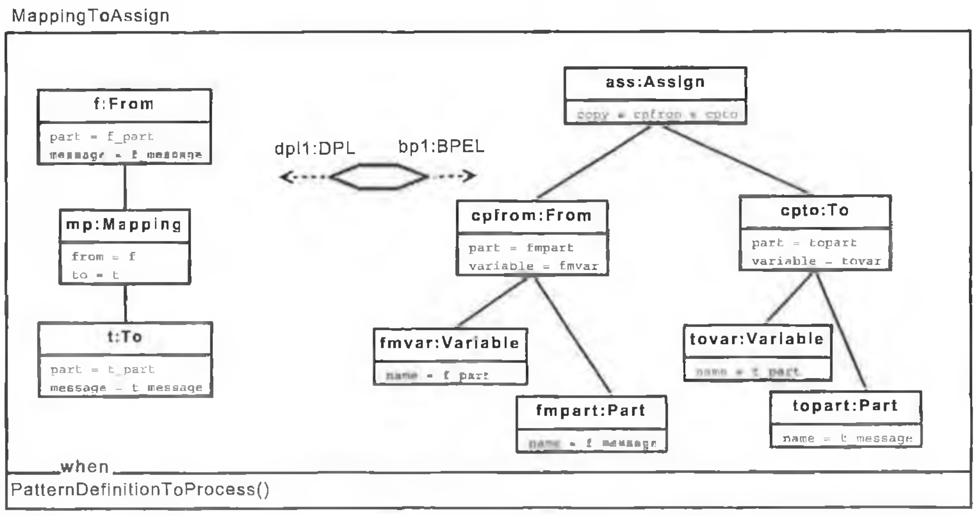


Figure 7.17: Graphical QVT MappingToAssign relation declaration.

The eighth relation, PatternDefinitionToVariable, defines the relation between a DPL PatternDefinition artifact and a BPEL Variable artifact. The fn_in_type variable is bound at runtime to either the "request" or "response" literal string values. The *when* clause specifies that the PatternDefinitionToProcess relation must have been previously executed. No *where* clause is specified. The relation is expressed graphically in Figure 7.18.

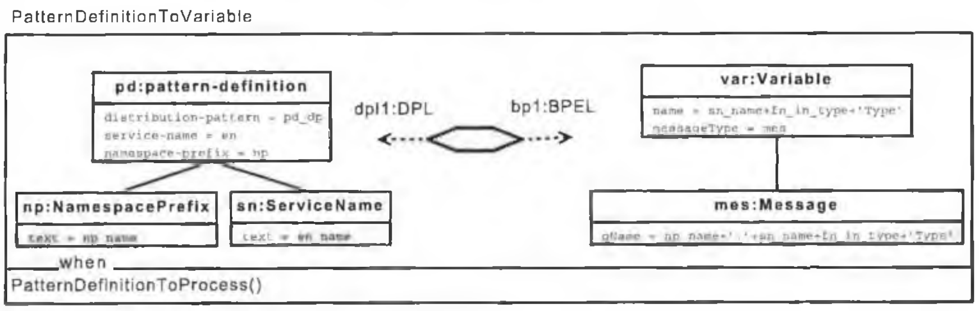


Figure 7.18: Graphical QVT PatternDefinitionToVariable relation declaration.

The ninth relation, PatternDefinitionToNamespace, defines the relation between a DPL PatternDefinition artifact and a BPEL Namespace artifact. The *when* clause specifies that the PatternDefinitionToProcess relation must have been previously executed. No *where* clause is specified. The relation is expressed graphically in Figure 7.19.

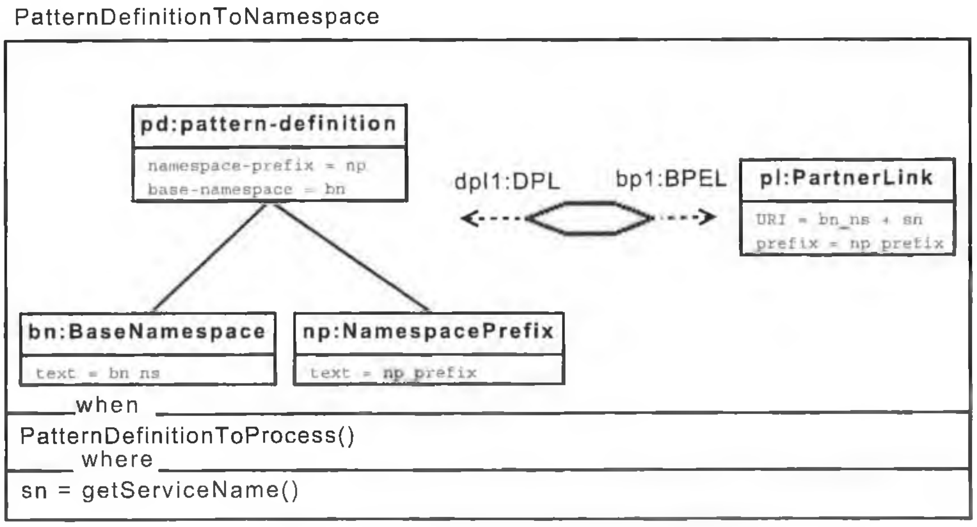


Figure 7.19: Graphical QVT PatternDefinitionToNamespace relation declaration.

The tenth relation, PatternDefinitionToPartnerLink, defines the relation between a DPL PatternDefinition artifact and a BPEL PartnerLink artifact. The *when* clause specifies that the PatternDefinitionToProcess relation must have been previously executed. No *where* clause is specified. The relation is expressed graphically in Figure 7.20.

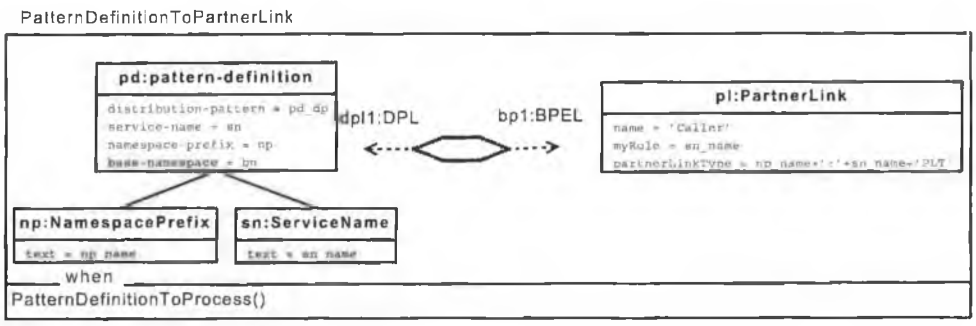


Figure 7.20: Graphical QVT PatternDefinitionToPartnerLink relation declaration.

7.4.3 Relating DPL Notation to Interface Notation

The third relation set is from the DPL notation to the Interface notation, where the DPL notation and the Interface notation are the candidate models. The DPL notation provides the constructs for the internal representation of a distribution pattern, while the Interface notation provides the constructs necessary to define a distribution pattern based compositional interface. These relations are uni-directional as the target model only represents some of the candidate model information. We assume the Web service interface language will be WSDL, and thus utilise the WSDL notation, discussed in Section 2.3.

This relation set is pattern dependent because the target notation's artifacts, along with the models outputted, depend on the chosen distribution pattern. For example, centralised distributions require only a single Interface model to expose the composition hub, while decentralised distributions require an Interface model for each composition participant along with the composition initiator. Again we assume the use of a centralised distribution pattern. The relation is expressed in Figure 7.21.

```
transformation dplTowsdl(dpl1:DPL, ws1:WSDL)
{
....
}
```

Figure 7.21: Textual QVT dplTowsdl transform declaration.

The five relations in this set are outlined as follows, and described in more detail below. The five relations show how the DPL artifacts are mapped to the WSDL notation.

- PatternDefinitionToDefinition
- NodeToNamespace
- NodeToPartnerLinkType
- FromToPart
- ToToPart

The first relation, PatternDefinitionToDefinition, defines the relation between a DPL PatternDefinition artifact and a WSDL Definition artifact. The relation is a top level relation and so will be matched rather than being called directly by another relation. As before the properties of both notations are bound using the same variable names, meaning they share the same values. The *where* clause indicates that a number of functions and relations must be subsequently executed. These functions are getBasenamespace (retrieve the namespace for the composition), getServiceName (retrieve the name assigned to the composition), getOperationName (retrieve the operation name assigned to the composition) and getNamespacePrefix (retrieve the namespace prefix for the composition). The relations to be executed are NodeToNamespace, NodeToPartnerLinkType, ToToPart and ToFromPart. The relation is expressed graphically in Figure 7.22.

The second relation, NodeToNamespace, defines the relation between a DPL Node artifact and a WSDL Namespace artifact. The relation is called by the first relation, PatternDefinitionToDefinition, as specified in the *when* clause. No *where* clause is specified. The relation is expressed graphically in Figure 7.23.

The third relation, NodeToPartnerLinkType, defines the relation between a DPL Node artifact and a WSDL PartnerLinkType artifact. The relation is called by the first relation, PatternDefinitionToDefinition, which is specified in the *when* clause. No *where* clause is specified. The relation is expressed graphically in Figure 7.24.

The fourth relation, FromToPart, defines the relation between a DPL From artifact and a WSDL Part artifact. The relation is called by the first relation, PatternDefinitionToDefini-

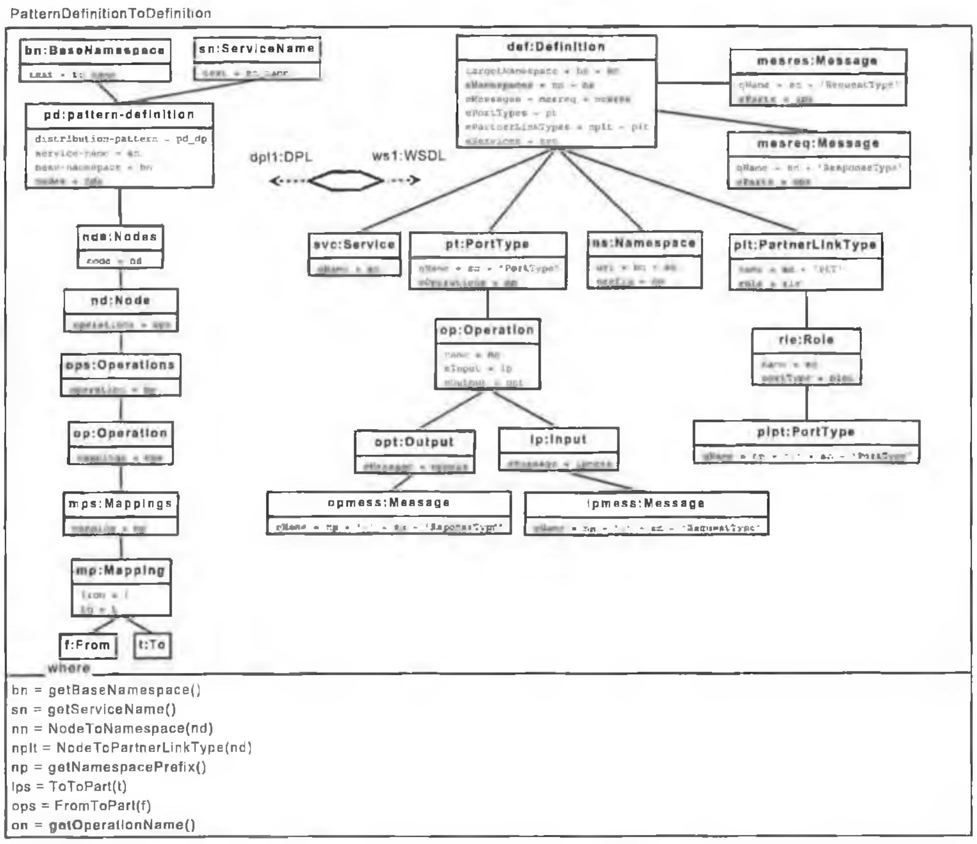


Figure 7.22: Graphical QVT PatternDefinitionToDefinition relation declaration.

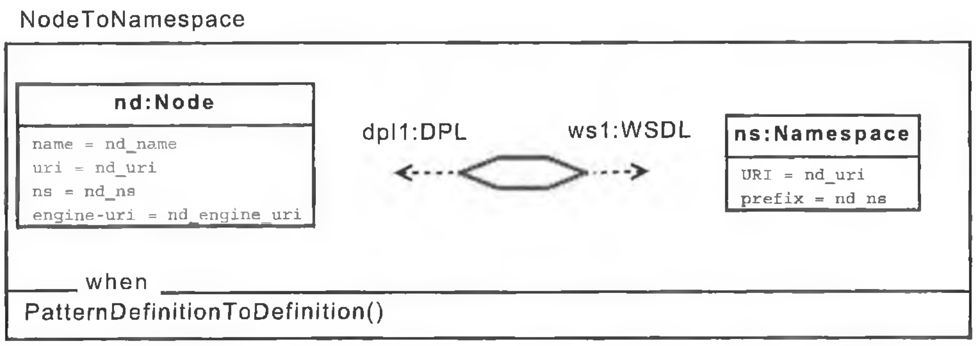


Figure 7.23: Graphical QVT NodeToNamespace relation declaration.

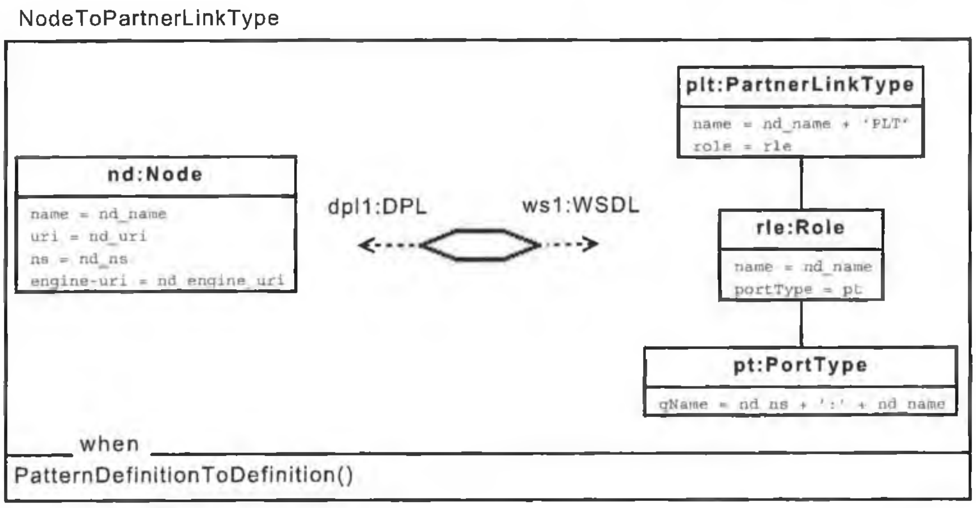


Figure 7.24: Graphical QVT NodeToPartnerLinkType relation declaration.

tion, as specified in the *when* clause. The *where* clause indicates the function convertEType-ToWSDLType (converts an ECore data type to a WSDL data type) must run subsequent to this relation. The relation is expressed graphically in Figure 7.25.

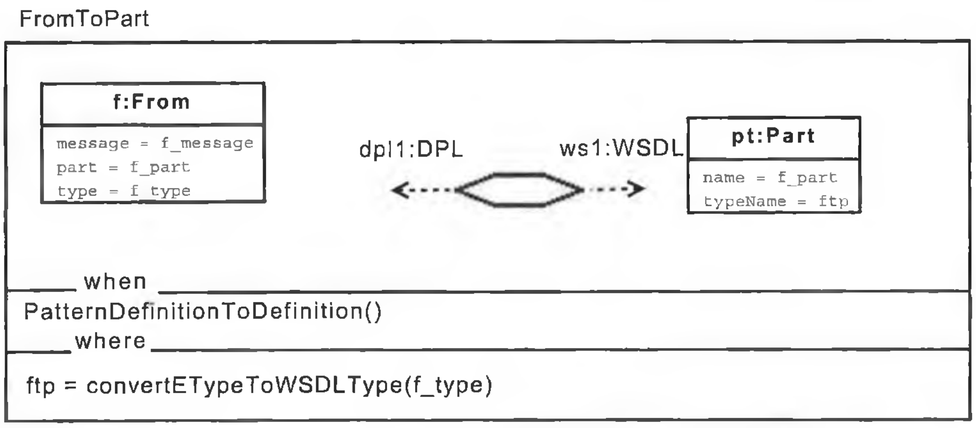


Figure 7.25: Graphical QVT FromToPart relation declaration.

The fifth relation, ToToPart, defines the relation between a DPL To artifact and a WSDL Part artifact. The relation is called by the first relation, PatternDefinitionToDefinition, which is specified in the *when* clause. The *where* clause indicates the function convertEType-ToWSDLType (converts an ECore data type to a WSDL data type) must run subsequent to this relation. The relation is expressed graphically in Figure 7.26.

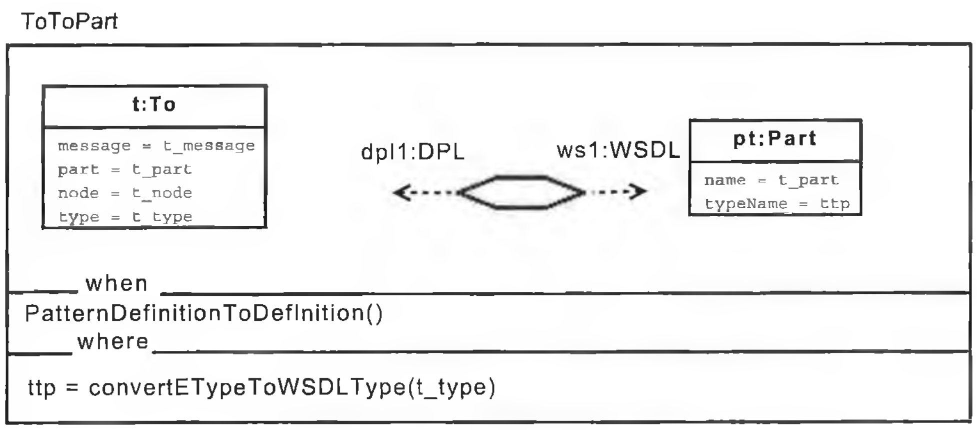


Figure 7.26: Graphical QVT ToToPart relation declaration.

7.4.4 Relating DPL Notation to Deployment Descriptor Notation

The fourth relation set is from the DPL notation to the Deployment Descriptor notation, where the DPL notation and the Deployment Descriptor notation are the candidate models. The DPL notation provides the constructs for the internal representation of a distribution pattern, while the Deployment Descriptor notation provides the constructs to define a distribution pattern based deployment, to be enacted on a composition engine. These relations are uni-directional as the target model only represents some of the candidate model information. We previously indicated that the WS-BPEL collaboration language will be used. It follows that we must choose a WS-BPEL compliant deployment environment. One such platform is ActiveBPEL. ActiveBPEL has a deployment descriptor format PDD (Process Deployment Descriptor), which is the basis for the deployment descriptor notation.

This relation set is pattern dependent because the target notation's artifacts, along with a number of models outputted, depend on the chosen distribution pattern. For example, centralised distributions require only a single Deployment Descriptor model to expose the composition hub, while decentralised distributions require each participant to have a Deployment Descriptor model as well as the composition initiator. As previously stated we assume the use of a centralised distribution pattern. The relation is expressed in Figure 7.27.

The five relations in this set are outlined as follows, and described in more detail below. The five relations show how the DPL artifacts are mapped to the PDD notation.

```
transformation dplTopdd(dpl1:DPL, pdd1:PDD)
{
....
}
```

Figure 7.27: Textual QVT dplTopdd transform declaration.

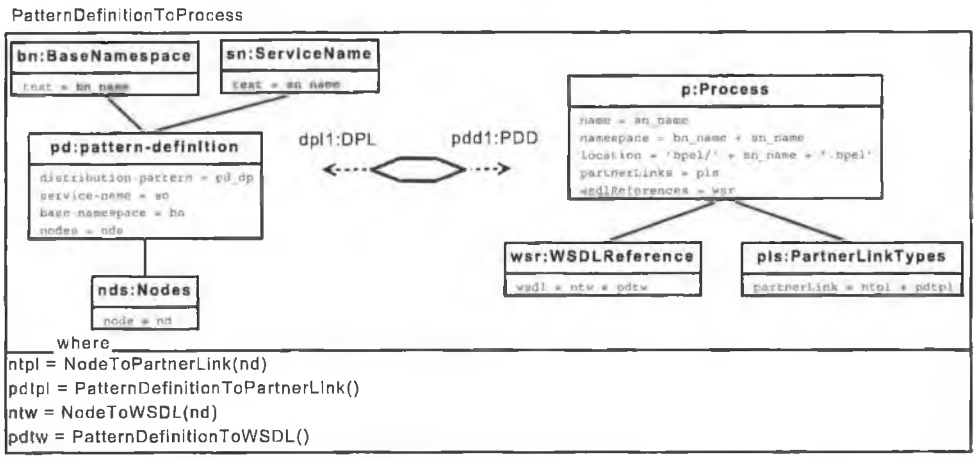


Figure 7.28: Graphical QVT PatternDefinitionToProcess relation declaration.

- PatternDefinitionToProcess
- NodeToPartnerLink
- NodeToWSDL
- PatternDefinitionToPartnerLink
- PatternDefinitionToWSDL

The first relation, PatternDefinitionToProcess, defines the relation between a DPL PatternDefinition artifact and a PDD Process artifact. The relation is a top level relation and so will be matched rather than being called directly by another relation. As in the other relation sets the properties of both notations are bound using the same variable names meaning they share the same values. The *where* clause indicates that a number of relations must be subsequently executed. The relations to be executed are NodeToPartnerLink, NodeToWSDL, PatternDefinitionToPartnerLink and PatternDefinitionToWSDL. The relation is expressed graphically in Figure 7.28.

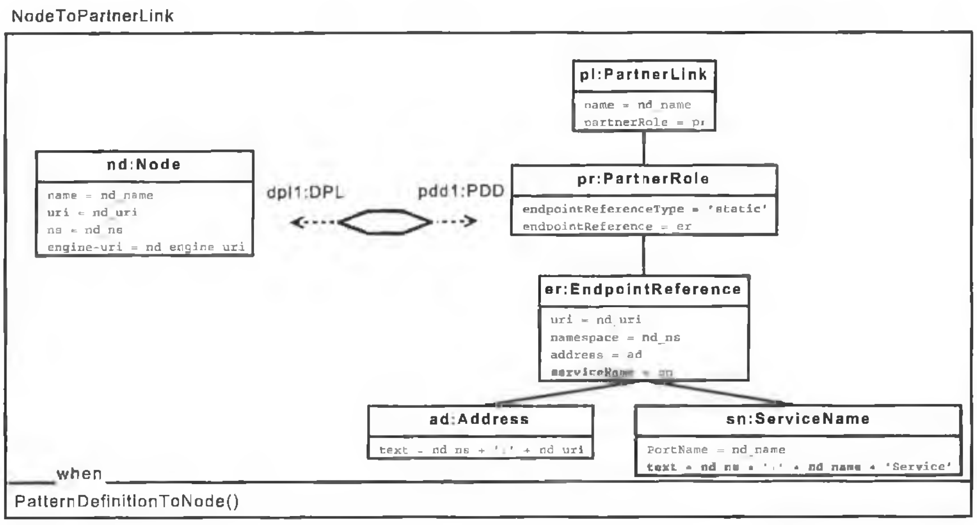


Figure 7.29: Graphical QVT NodeToPartnerLink relation declaration.

The second relation, NodeToPartnerLink, defines the relation between a DPL Node artifact and a PDD PartnerLink artifact. The relation is called by the first relation, Pattern-DefinitionToProcess, which are specified in the *when* clause. No *where* clause is specified. The relation is expressed graphically in Figure 7.29.

The third relation, NodeToWSDL, defines the relation between a DPL Node artifact and a PDD WSDL artifact. The relation is called by the first relation, PatternDefinition-ToProcess, as specified in the *when* clause. No *where* clause is specified. The relation is expressed graphically in Figure 7.30.

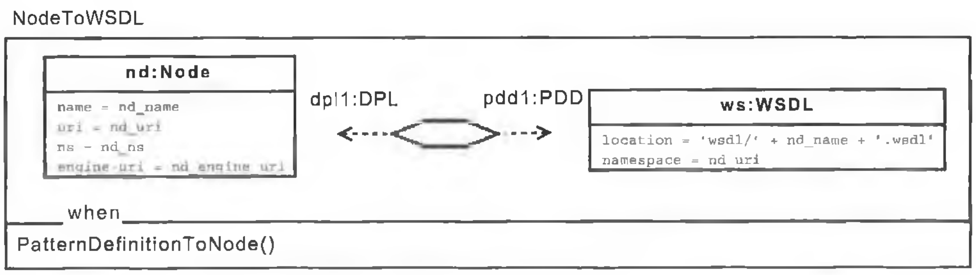


Figure 7.30: Graphical QVT NodeToWSDL relation declaration.

The fourth relation, PatternDefinitionToPartnerLink, defines the relation between a DPL PatternDefinition artifact and a PDD PartnerLink artifact. The relation is called by the first relation, PatternDefinitionToProcess, as specified in the *when* clause. No *where* clause is specified. The relation is expressed graphically in Figure 7.31.

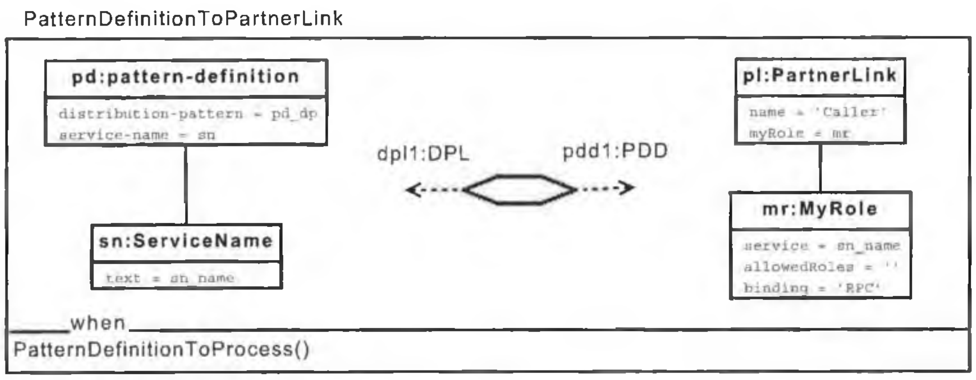


Figure 7.31: Graphical QVT PatternDefinitionToPartnerLink relation declaration.

The fifth relation, PatternDefinitionToWSDL, defines the relation between a DPL PatternDefinition artifact and a PDD WSDL artifact. The relation is called by the first relation, PatternDefinitionToProcess, which are specified in the *when* clause. No *where* clause is specified. The relation is expressed graphically in Figure 7.32.

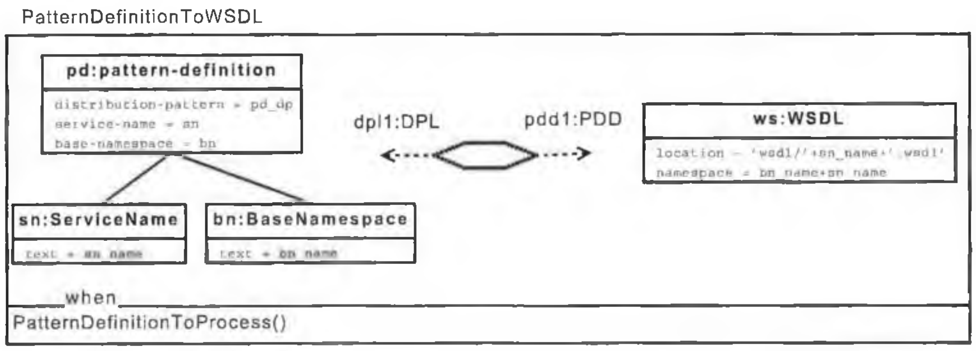


Figure 7.32: Graphical QVT PatternDefinitionToWSDL relation declaration.

7.4.5 Relating DPL Notation to Deployment Catalog Notation

The fifth relation set is from the DPL notation to the Deployment Catalog notation, where the DPL notation and the Deployment Catalog notation are the candidate models. The DPL notation provides the constructs for the internal representation of a distribution pattern, while the Deployment Catalog notation provides the constructs to enumerate the interfaces of a distribution pattern based deployment, to be enacted on a composition engine.

These relations are uni-directional as the target model only represents some of the candidate model information. As previously stated, we assume the use of a centralised distribution pattern, the WS-BPEL collaboration language and the ActiveBPEL execution platform. ActiveBPEL has a deployment catalog format wsdlCatalog, which is the basis for the Deployment Catalog notation.

This relation set is pattern dependent because the target notation's artifacts, along with the models outputted, depend on the chosen distribution pattern. For example centralised distributions require only a single Deployment Catalog model to describe the interfaces required by the composition hub, while decentralised distributions require each participant to have a Deployment Catalog model along with the composition initiator. The relation is expressed in Figure 7.33.

```
transformation dplTowdc(dpl1:DPL, wdc1:WSDLCatalog)
{
....
}
```

Figure 7.33: Textual QVT dplTowdc transform declaration.

The three relations in this set are outlined as follows, and described in more detail below. The three relations show how the DPL artifacts are mapped to the WSDLCatalog notation.

- PatternDefinitionToWSDLCatalog
- NodeToWSDLEntry
- PatternDefinitionToWSDLEntry

The first relation, PatternDefinitionToWSDLCatalog, defines the relation between a DPL PatternDefinition artifact and a WSDLCatalog WSDLCatalog artifact. The relation is a top level relation and so will be matched rather than being called directly by another relation. As in the other relation sets the properties of both notations are bound using the same variable names meaning they share the same values. The *where* clause indicates that a function and a relation must be subsequently executed. The relation to be executed is DistributionPatternToWSDLEntry, whilst the function is getAllNodes (retrieves all the DPL Nodes from the source model). The relation is expressed graphically in Figure 7.34.

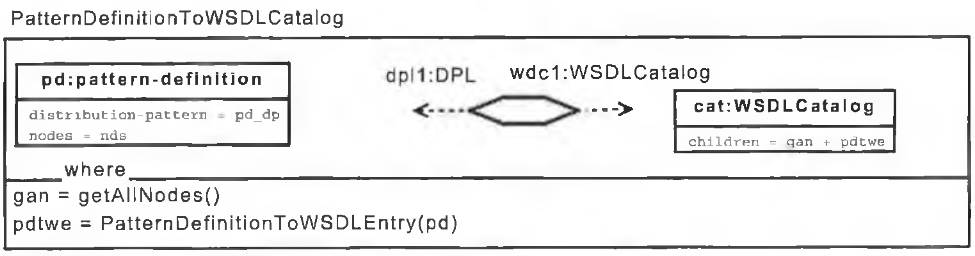


Figure 7.34: Graphical QVT PatternDefinitionToWSDLCatalog relation declaration.

The second relation, NodeToWSDLEntry, defines the relation between a DPL Node artifact and a WSDLCatalog WSDLEntry artifact. The relation is a top level relation and so will be matched rather than being called directly by another relation. No *when* or *where* clause is specified. The relation is expressed graphically in Figure 7.35.

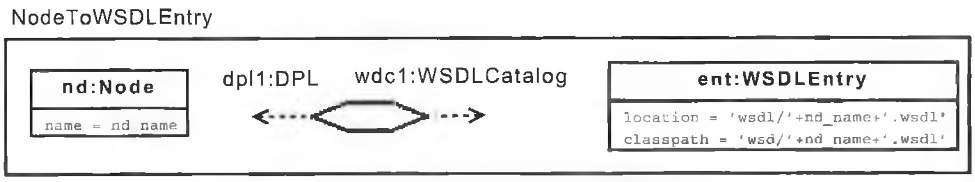
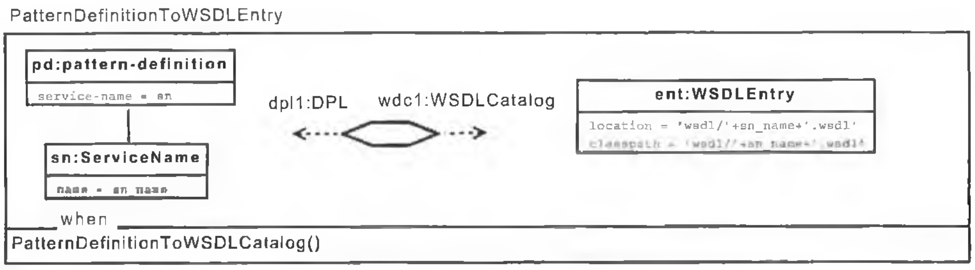


Figure 7.35: Graphical QVT NodeToWSDLEntry relation declaration.

The third relation, PatternDefinitionToWSDLEntry, defines the relation between a DPL PatternDefinition artifact and a WSDLCatalog WSDLEntry artifact. The relation is called by the first relation, PatternDefinitionToWSDLCatalog, as specified in the *when* clause. No *where* clause is specified. The relation is expressed graphically in Figure 7.36.



PatternDefinitionToWSDLEntry
pd:pattern-definition
service-name = sn
sn:ServiceName
name = sn_name
dpl1:DPL
wdc1:WSDLCatalog
ent:WSDLEntry
location = 'wsdl/'+sn_name+'.wsdl'
classpath = 'wsdl/'+sn_name+'.wsdl'
when
PatternDefinitionToWSDLCatalog()

Figure 7.36: Graphical QVT PatternDefinitionToWSDLEntry relation declaration.

7.4.6 Relating Collaboration Notation to XML Notation

The sixth relation set is from the Collaboration notation to the XML notation, where the Collaboration notation and the XML notation are the candidate models. The Collaboration notation provides the constructs necessary to define the choreography of a distribution pattern based composition, while the XML notation provides the constructs to define an XML based document necessary for execution. These relations are bi-directional because the target and source models represent the same information, albeit in different formats. Again we assume the Collaboration notation is targeted to the WS-BPEL collaboration language. This relation set specifies how a WS-BPEL notation is related to an XML based notation, enabling models based upon the WS-BPEL notation to be serialised as XML text. The relation set is pattern independent as these relations hold across all distribution patterns. The relation is expressed in Figure 7.37.

```
transformation bpelToxml(bpel1:BPEL, xml1:XML)
{
....
}
```

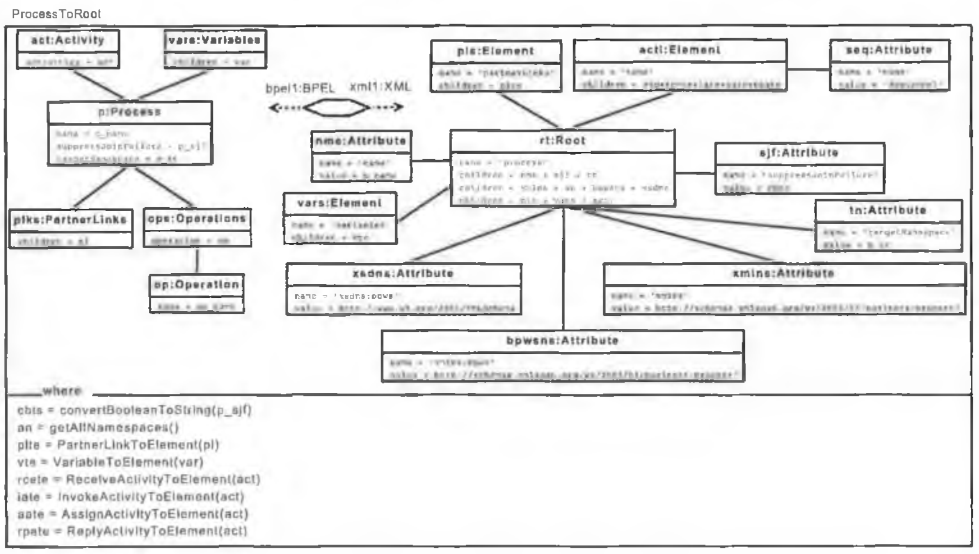
Figure 7.37: Textual QVT bpelToxml transform declaration.

The eight relations in this set are outlined as follows, and described in more detail below. The eight relations show how the BPEL artifacts are mapped to the XML notation.

- ProcessToRoot
- NamespaceToAttribute
- InvokeActivityToElement
- ReceiveActivityToElement
- ReplyActivityToElement
- AssignActivityToElement
- VariableToElement

- PartnerLinkToElement

The first relation, ProcessToRoot, defines the relation between a BPEL Process artifact and an XML Root artifact. The relation is a top level relation and so will be matched rather than being called directly by another relation. As in the other relation sets, the properties of both notations are bound using the same variable names meaning they share the same values. The *where* clause indicates that a number of functions and relations must be subsequently executed. The relations to be executed are PartnerLinkToElement, VariableToElement, ReceiveActivityToElement,InvokeActivityToElement, AssignActivityToElement and ReplyActivityToElement. The functions are convertBooleanToString (converts a Boolean value to its string equivalent) and getAllNamespaces (retrieves all the Namespaces in a source BPEL model). The relation is expressed graphically in Figure 7.38.



```
cbts = convertBooleanToString(p_sjf)
an = getAllNamespaces()
plte = PartnerLinkToElement(pl)
vte = VariableToElement(var)
rcete = ReceiveActivityToElement(act)
iate = InvokeActivityToElement(act)
aate = AssignActivityToElement(act)
rpate = ReplyActivityToElement(act)
```

Figure 7.38: Graphical QVT ProcessToRoot relation declaration.

The second relation, NamespaceToAttribute, defines the relation between a BPEL Namespace artifact and an XML Attribute artifact. The relation is a top level relation and so will be matched rather than being called directly by another relation. No *when* or *where* clause are specified. The relation is expressed graphically in Figure 7.39.

The third relation, InvokeActivityToElement, defines the relation between a BPEL In-

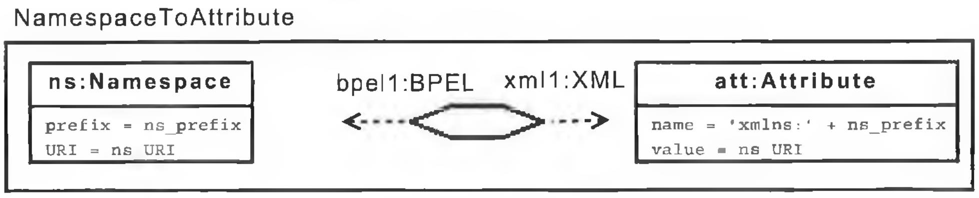


Figure 7.39: Graphical QVT NamespaceToAttribute relation declaration.

voke artifact and an XML Element artifact. The relation is called by the first relation, ProcessToRoot, as specified in the *when* clause. The *where* clause specifies that the function getActivityType (resolves the type of a BPEL source model Activity) must be subsequently executed. The relation is expressed graphically in Figure 7.40.

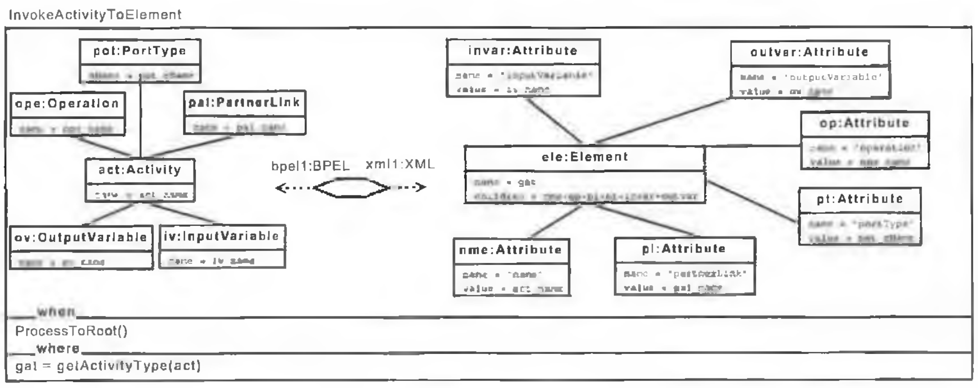


Figure 7.40: Graphical QVT InvokeActivityToElement relation declaration.

The fourth relation, ReceiveActivityToElement, defines the relation between a BPEL Receive artifact and an XML Element artifact. The relation is called by the first relation, ProcessToRoot, which is specified in the *when* clause. The *where* clause specifies that the function getActivityType (resolves the type of a BPEL source model Activity) and convertBooleanToString (converts a Boolean value to its string equivalent) must be subsequently executed. The relation is expressed graphically in Figure 7.41.

The fifth relation, ReplyActivityToElement, defines the relation between a BPEL Reply artifact and an XML Element artifact. The relation is called by the first relation, ProcessToRoot, as specified in the *when* clause. The *where* clause specifies that the function getActivityType (resolves the type of a BPEL source model Activity) must be subsequently

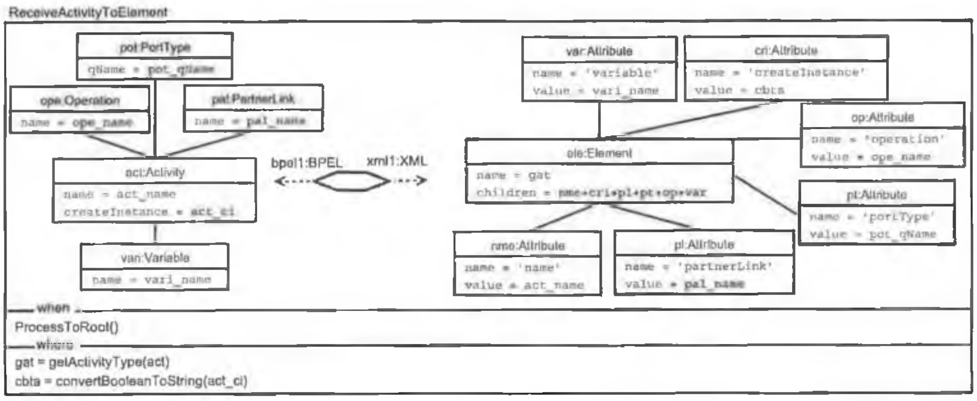


Figure 7.41: Graphical QVT ReceiveActivityToElement relation declaration.

executed. The relation is expressed graphically in Figure 7.42.

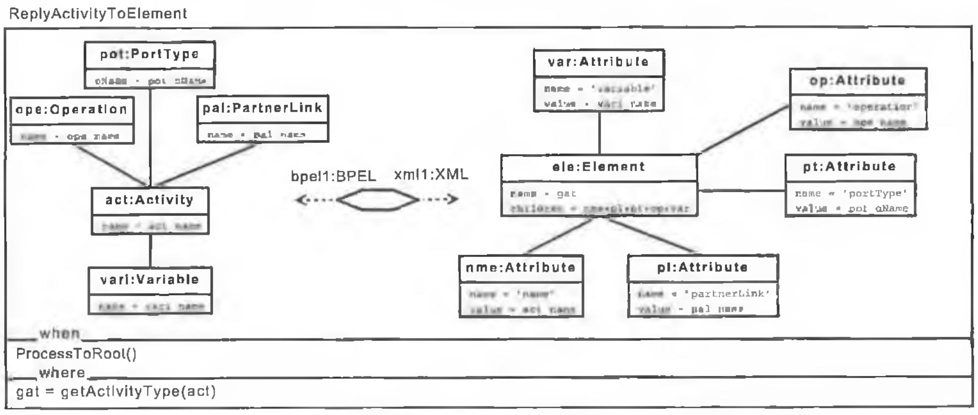


Figure 7.42: Graphical QVT ReplyActivityToElement relation declaration.

The sixth relation, AssignActivityToElement, defines the relation between a BPEL Assign artifact and an XML Element artifact. The relation is called by the first relation, ProcessToRoot, which is specified in the *when* clause. The *where* clause specifies that the function getActivityType (resolves the type of a BPEL source model Activity) must be subsequently executed. The relation is expressed graphically in Figure 7.43.

The seventh relation, VariableToElement, defines the relation between a BPEL Variable artifact and an XML Element artifact. The relation is called by the first relation, ProcessToRoot, as specified in the *when* clause. No *where* clause is specified. The relation is expressed graphically in Figure 7.44.

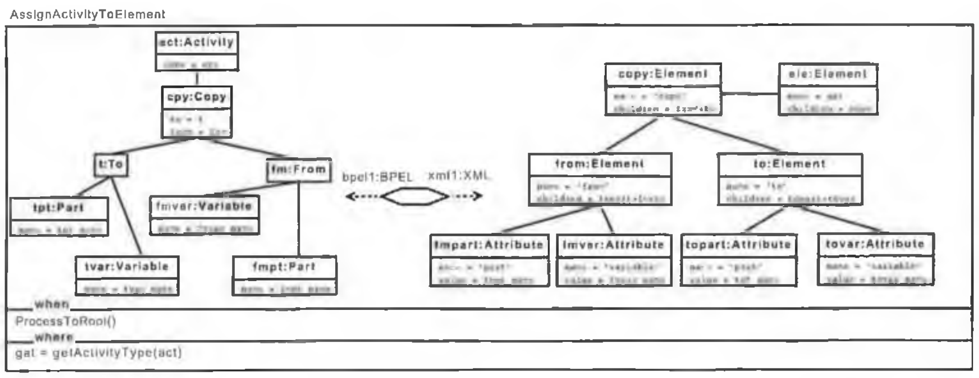


Figure 7.43: Graphical QVT AssignActivityToElement relation declaration.

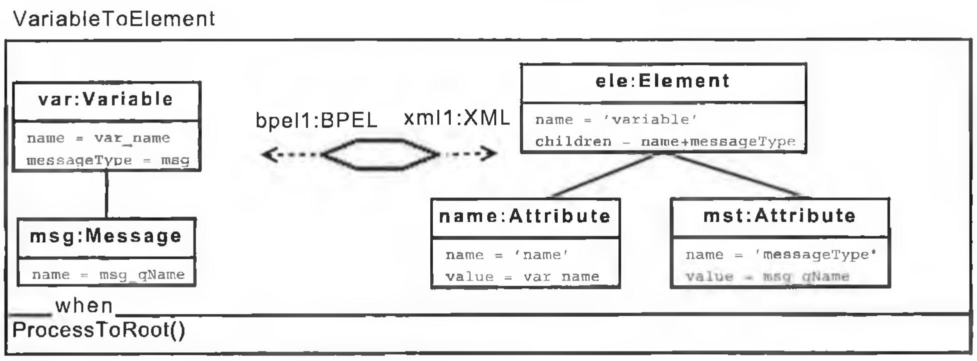


Figure 7.44: Graphical QVT VariableToElement relation declaration.

The eighth relation, PartnerLinkToElement, defines the relation between a BPEL PartnerLink artifact and an XML Element artifact. The relation is called by the first relation, ProcessToRoot, which is specified in the *when* clause. No *where* clause is specified. The relation is expressed graphically in Figure 7.45.

7.4.7 Relating Interface Notation to XML Notation

The seventh relation set is from the Interface notation to the XML notation, where the Interface notation and the XML notation are the candidate models. The Interface notation provides the constructs necessary to define a distribution pattern based compositional interface, while the XML notation provides the constructs to define an XML based document, necessary for execution. These relations are bi-directional as the target and source models

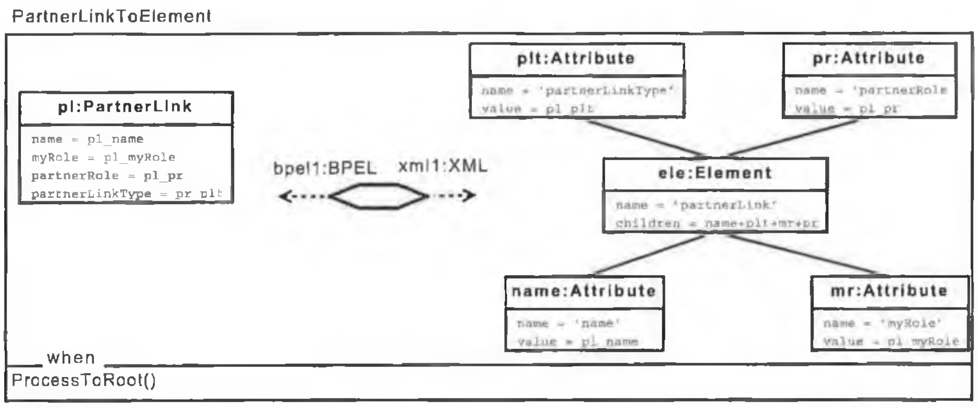


Figure 7.45: Graphical QVT PartnerLinkToElement relation declaration.

represent the same information, albeit in different formats. We assume the Interface notation is targeted to the WSDL interface language. This relation set specifies how a WSDL notation is related to an XML based notation, enabling models based upon the WSDL notation to be serialised to XML text. The relation set is pattern independent as these relations hold across all distribution patterns. The relation is expressed in Figure 7.46.

```
transformation wsdlToxml(ws1:WSDL, xml1:XML)
{
....
}
```

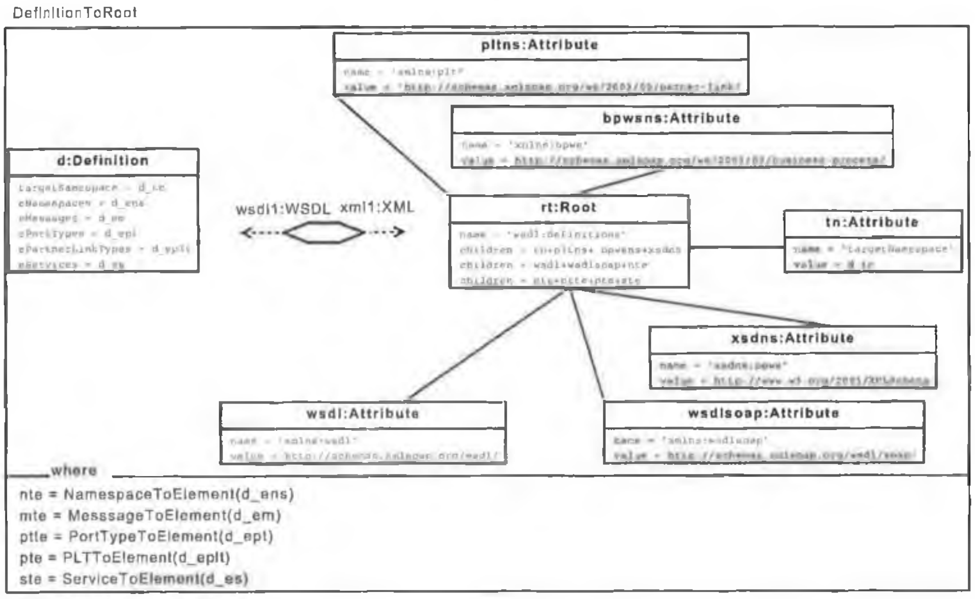
Figure 7.46: Textual QVT wsdlToxml transform declaration.

The twelve relations in this set are outlined as follows, and described in more detail below. The twelve relations show how the WSDL artifacts are mapped to the XML notation.

- DefinitionToRoot
- NamespaceToAttribute
- MessageToElement
- PartToElement
- PortTypeToElement
- OperationToElement

- InputToElement
- OutputToElement
- PLTToElement
- RoleToElement
- PLTPortTypeToElement
- ServiceToElement

The first relation, DefinitionToRoot, defines the relation between a WSDL Definition artifact and an XML Root artifact. The relation is a top level relation and so will be matched rather than being called directly by another relation. As in the other relation sets the properties of both notations are bound using the same variable names meaning they share the same values. The *where* clause indicates that a number of relations must be subsequently executed. The relations to be executed are NamespaceToElement, MessageToElement, PortTypeToElement, PLTToElement and ServiceToElement. The relation is expressed graphically in Figure 7.47.


DefinitionToRoot
pltns:Attribute
name = 'xmlns:plt'
value = 'http://schemas.xmlsoap.org/ws/2003/05/partner-link/'
bpwsns:Attribute
name = 'xmlns:bpws'
value = http://schemas.xmlsoap.org/ws/2003/03/business-process/
d:Definition
targetNamespace = d_tn
eNamespaces = d_ens
eMessages = d_em
ePortTypes = d_ept
ePartnerLinkTypes = d_eplt
eServices = d_es
wsdl1:WSDL
xml1:XML
rt:Root
name = 'wsdl:definitions'
children = tn+pltns+ bpwsns+xsdns
children = wsdl+wsdlsoap+nte
children = mte+ptte+pte+ste
tn:Attribute
name = 'targetNamespace'
value = d_tn
xsdns:Attribute
name = 'xsdns:bpws'
value = http://www.w3.org/2001/XMLSchema
wsdl:Attribute
name = 'xmlns:wsdl'
value = 'http://schemas.xmlsoap.org/wsdl/'
wsdlsoap:Attribute
name = 'xmlns:wsdlsoap'
value = http://schemas.xmlsoap.org/wsdl/soap/
where

```
nte = NamespaceToElement(d_ens)
mte = MesssageToElement(d_em)
ptte = PortTypeToElement(d_ept)
pte = PLTToElement(d_eplt)
ste = ServiceToElement(d_es)
```

Figure 7.47: Graphical QVT DefinitionToRoot relation declaration.

The second relation, NamespaceToAttribute, defines the relation between a WSDL Namespace artifact and an XML Attribute artifact. The relation is called by the first relation, DefinitionToRoot, as specified in the *when* clause. No *where* clause is specified. The relation is expressed graphically in Figure 7.48.

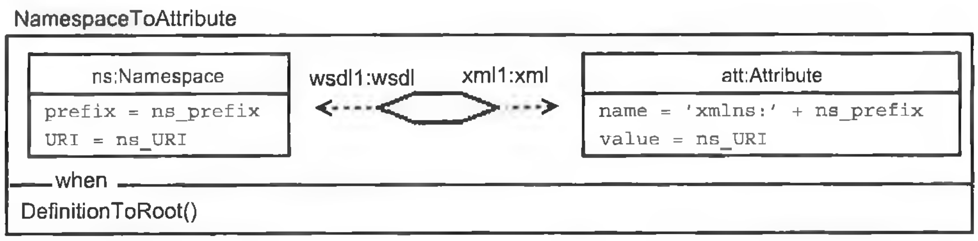


Figure 7.48: Graphical QVT NamespaceToAttribute relation declaration.

The third relation, MessageToElement, defines the relation between a WSDL Message artifact and an XML Element artifact. The relation is called by the first relation, DefinitionToRoot, which is specified in the *when* clause. The *where* clause specifies that the relation, PartToElement, must be subsequently executed. The relation is expressed graphically in Figure 7.49.

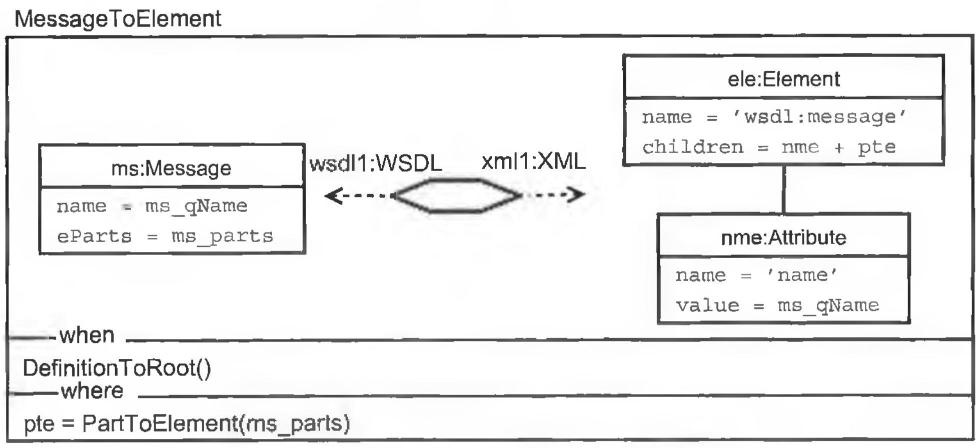


Figure 7.49: Graphical QVT MessageToElement relation declaration.

The fourth relation, PartToElement, defines the relation between a WSDL Part artifact and an XML Element artifact. The relation is called by the third relation, MessageToElement, as specified in the *when* clause. No *where* clause is specified. The relation is expressed graphically in Figure 7.50.

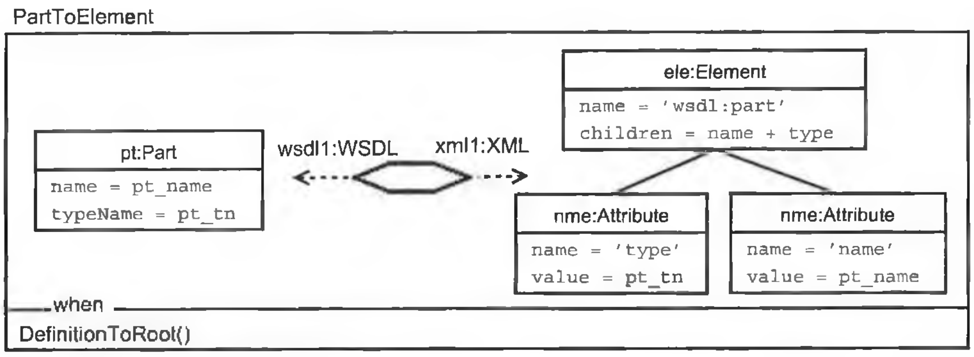


Figure 7.50: Graphical QVT PartToElement relation declaration.

The fifth relation, PortTypeToElement, defines the relation between a WSDL PortType artifact and an XML Element artifact. The relation is called by the first relation, Definition-ToRoot, which is specified in the *when* clause. The *where* clause specifies that the relation, OperationToElement, must be subsequently executed. The relation is expressed graphically in Figure 7.51.

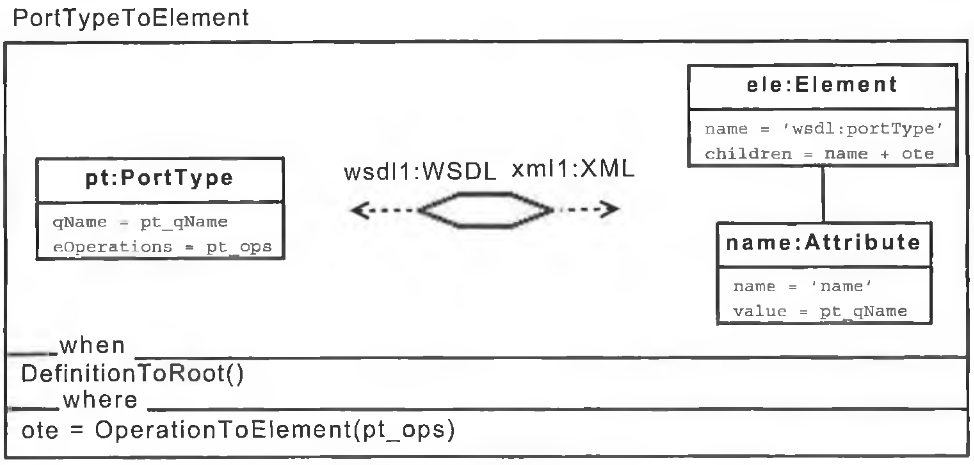


Figure 7.51: Graphical QVT PortTypeToElement relation declaration.

The sixth relation, OperationToElement, defines the relation between a WSDL Operation artifact and an XML Element artifact. The relation is called by the fifth relation, PortTypeToElement, as specified in the *when* clause. The *where* clause specifies that the relations, InputToElement and OutputToElement, must be subsequently executed. The relation is expressed graphically in Figure 7.52.

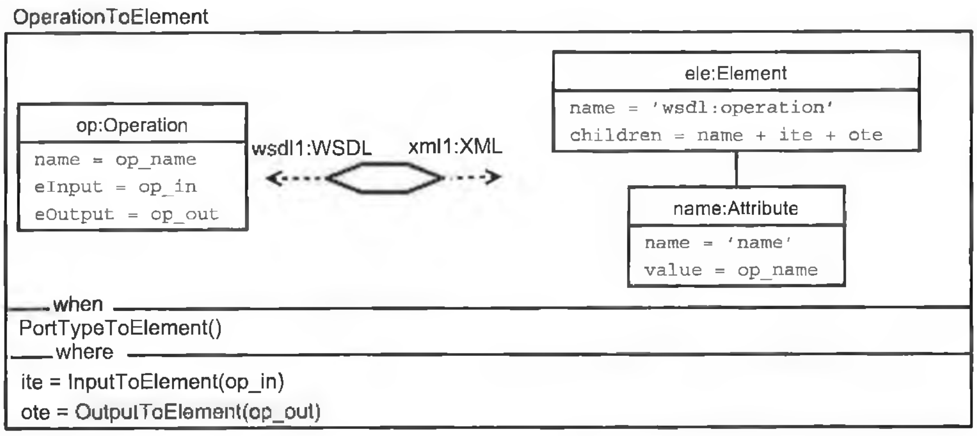


Figure 7.52: Graphical QVT OperationToElement relation declaration.

The seventh relation, InputToElement, defines the relation between a WSDL Input artifact and an XML Element artifact. The relation is called by the sixth relation, OperationToElement, which is specified in the *when* clause. There is no *where* clause specified. The relation is expressed graphically in Figure 7.53.

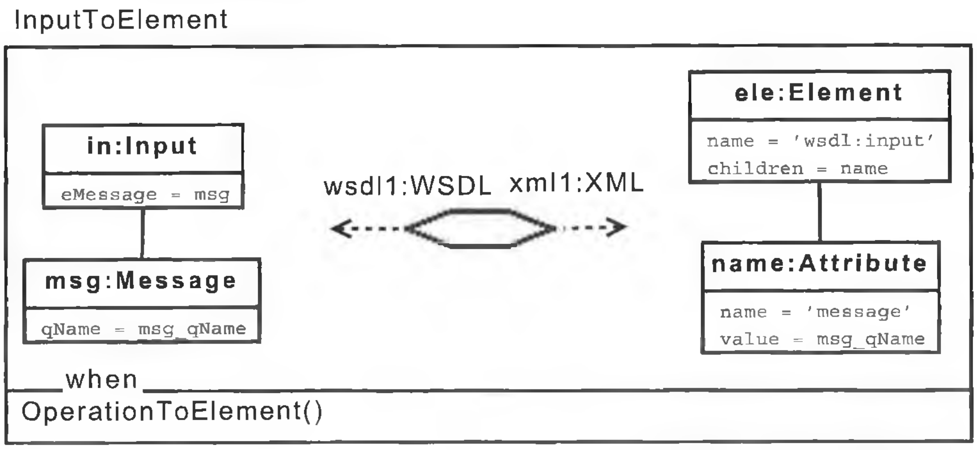


Figure 7.53: Graphical QVT InputToElement relation declaration.

The eighth relation, OutputToElement, defines the relation between a WSDL Ouput artifact and an XML Element artifact. The relation is called by the sixth relation, OperationToElement, as specified in the *when* clause. There is no *where* clause specified. The relation is expressed graphically in Figure 7.54.

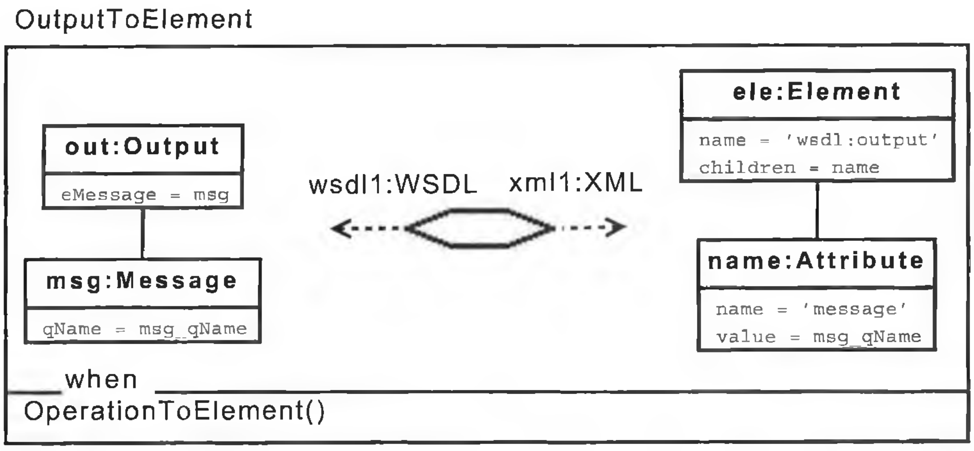


Figure 7.54: Graphical QVT OutputToElement relation declaration.

The ninth relation, PLTToElement, defines the relation between a WSDL PartnerLinkType artifact and an XML Element artifact. The relation is called by the sixth relation, DefinitionToRoot, which is specified in the *when* clause. The *where* clause specifies that the relation, RoleToElement, must be subsequently executed. The relation is expressed graphically in Figure 7.55.

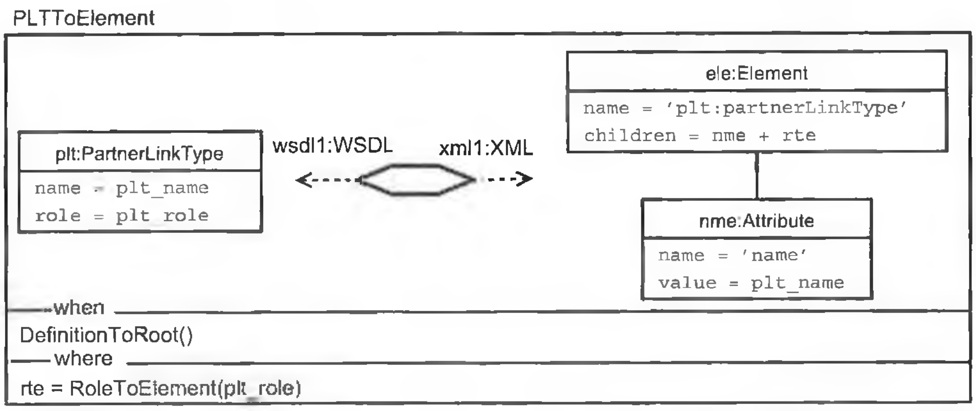


Figure 7.55: Graphical QVT PLTToElement relation declaration.

The tenth relation, RoleToElement, defines the relation between a WSDL Role artifact and an XML Element artifact. The relation is called by the ninth relation, PLTToElement, as specified in the *when* clause. The *where* clause specifies that the relation, PLTPortType-ToElement, must be subsequently executed. The relation is expressed graphically in Figure

7.56.

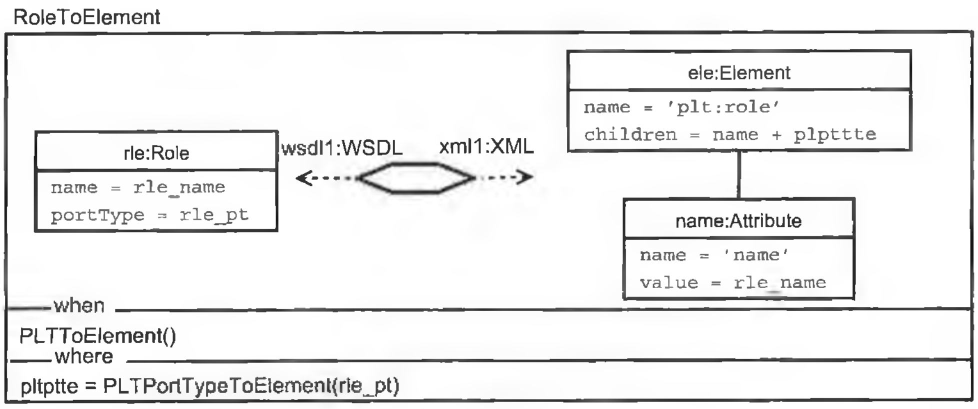


Figure 7.56: Graphical QVT RoleToElement relation declaration.

The eleventh relation, PLTPortTypeToElement, defines the relation between a WSDL PortType artifact and an XML Element artifact. The relation is called by the ninth relation, RoleToElement, which is specified in the *when* clause. No *where* clause is specified. The relation is expressed graphically in Figure 7.57.

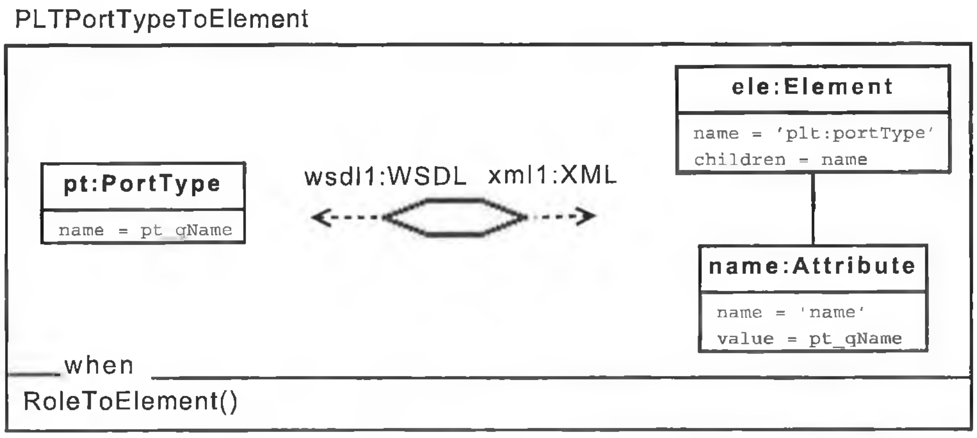


Figure 7.57: Graphical QVT PLTPortTypeToElement relation declaration.

The twelfth relation, ServiceToElement, defines the relation between a WSDL Service artifact and an XML Element artifact. The relation is called by the first relation, DefinitionToRoot, as specified in the *when* clause. No *where* clause is specified. The relation is expressed graphically in Figure 7.58.

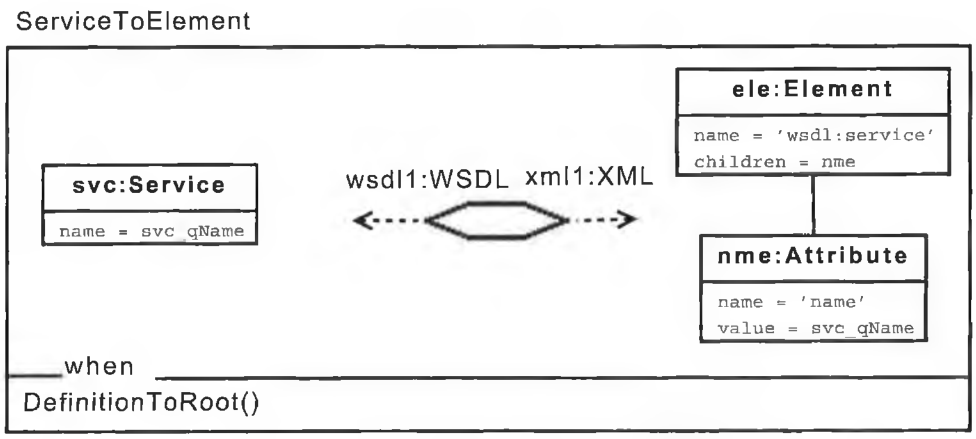


Figure 7.58: Graphical QVT ServiceToElement relation declaration.

7.4.8 Relating Deployment Descriptor Notation to XML Notation

The eighth relation set is from the Deployment Descriptor notation to the XML notation, where the Deployment Descriptor notation and the XML notation are the candidate models. The Deployment Descriptor notation provides the constructs to define a distribution pattern based deployment, to be enacted on a composition engine, while the XML notation provides the constructs to define an XML based document, necessary for execution. These relations are bi-directional as the target and source models represent the same information, albeit in different formats. We assume the Deployment Descriptor notation is targeted to the ActiveBPEL deployment environment, using the PDD deployment language. This relation set specifies how a PDD notation is related to an XML based notation, enabling models based upon the PDD notation to be serialised to XML text. The relation set is pattern independent as these relations hold across all distribution patterns. The relation is expressed in Figure 7.59.

```
transformation pddToxml(pdd1:PDD, xml1:XML)
{
....
}
```

Figure 7.59: Textual QVT pddToxml transform declaration.

The five relations in this set are outlined as follows, and described in more detail below.

The five relations show how the PDD artifacts are mapped to the XML notation.

- ProcessToRoot
- WSDLToElement
- PartnerLinkToElement
- MyRoleToElement
- PartnerRoleToElement

The first relation, ProcessToRoot, defines the relation between a PDD Process artifact and an XML Root artifact. The relation is a top level relation and so will be matched rather than being called directly by another relation. As in the other relation sets, the properties of both notations are bound using the same variable names meaning they share the same values. The *where* clause indicates that the functions, getAllWSDLReferences (retrieves all the WSDL references from a PDD source model) and getAllPartnerLinks (retrieves all the PartnerLinks from a PDD source model), must be subsequently executed. The relation is expressed graphically in Figure 7.60.

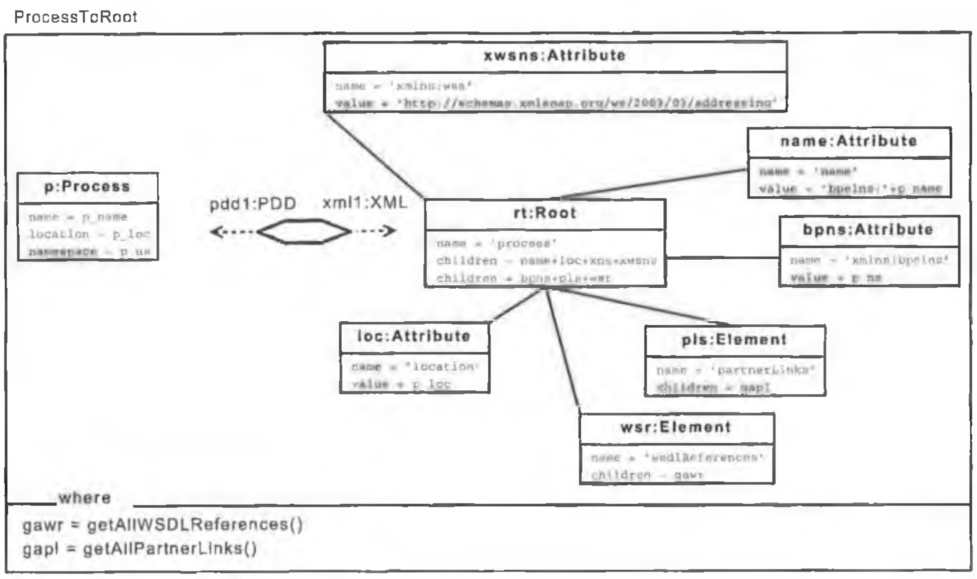


Figure 7.60: Graphical QVT ProcessToRoot relation declaration.

The second relation, WSDLToElement, defines the relation between a PDD WSDL artifact and an XML Element artifact. The relation is a top level relation and so will be matched rather than being called directly by another relation. No *when* or *where* clause are specified. The relation is expressed graphically in Figure 7.61.

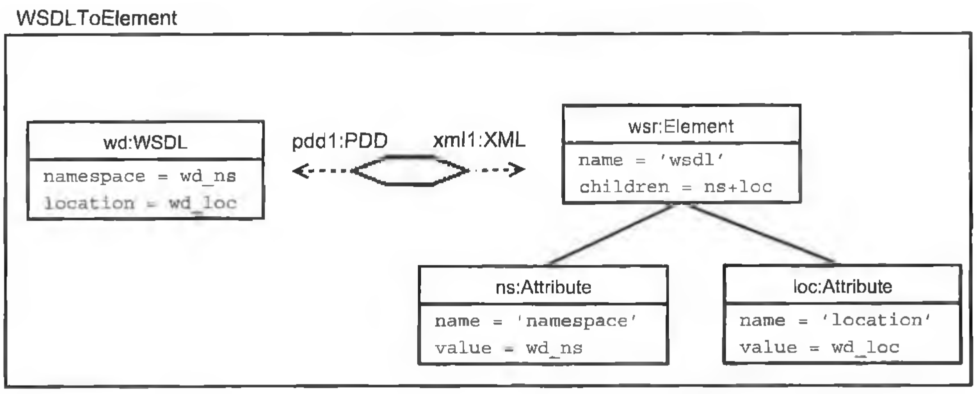


Figure 7.61: Graphical QVT WSDLToElement relation declaration.

The third relation, PartnerLinkToElement, defines the relation between a PDD PartnerLink artifact and an XML Element artifact. The relation is a top level relation and so will be matched rather than being called directly by another relation. The *where* clause specifies that the relations, MyRoleToElement and PartnerRoleToElement, must be subsequently executed. The relation is expressed graphically in Figure 7.62.

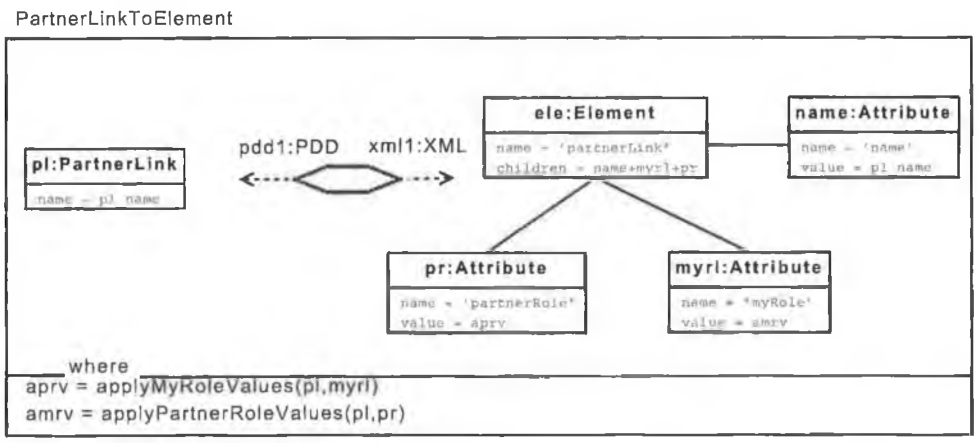


Figure 7.62: Graphical QVT PartnerLinkToElement relation declaration.

The fourth relation, MyRoleToElement, defines the relation between a PDD MyRole artifact and an XML Element artifact. The relation is called by the second relation, PartnerLinkToElement, which is defined in the *when* clause. No *where* clause is specified. The relation is expressed graphically in Figure 7.63.

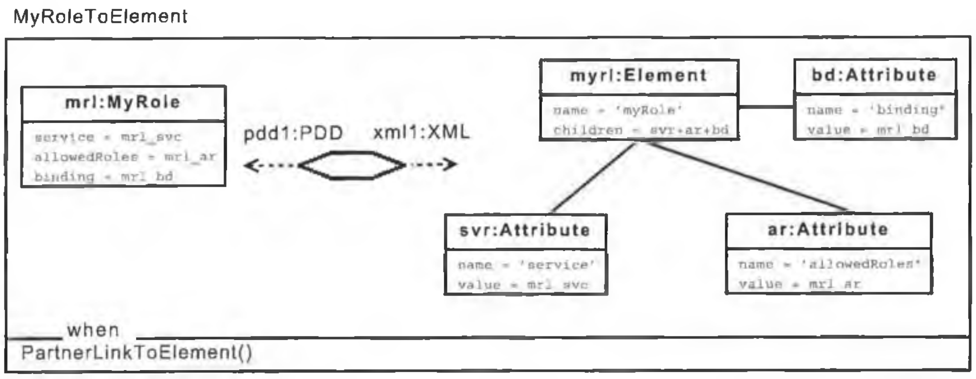


Figure 7.63: Graphical QVT MyRoleToElement relation declaration.

The fifth relation, PartnerRoleToElement, defines the relation between a PDD PartnerRole artifact and an XML Element artifact. The relation is called by the second relation, PartnerLinkToElement, as defined in the *when* clause. No *where* clause is specified. The relation is expressed graphically in Figure 7.64.

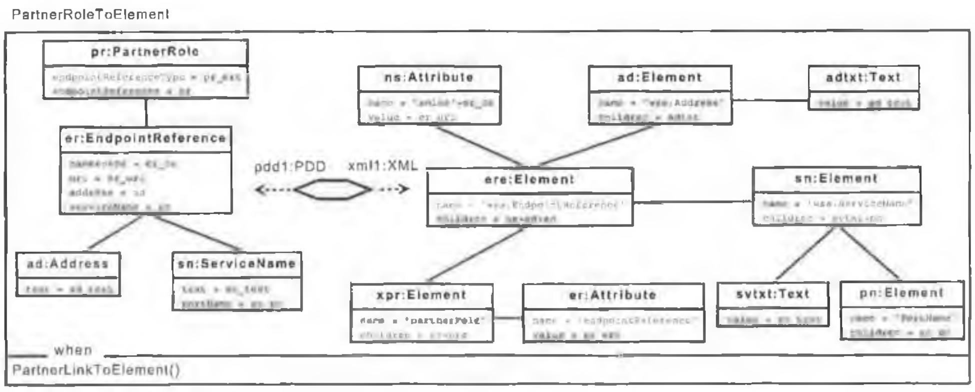


Figure 7.64: Graphical QVT PartnerRoleToElement relation declaration.

7.4.9 Relating Deployment Catalog Notation to XML Notation

The ninth relation set is from the Deployment Catalog notation to the XML notation, where the Deployment Catalog notation and the XML notation are the candidate models. The

Deployment Catalog notation provides the constructs to enumerate the interfaces of a distribution pattern based deployment, to be enacted on a composition engine, while the XML notation provides the constructs to define an XML based document, necessary for execution. These relations are bi-directional as the target and source models represent the same information, albeit in different formats. We assume the Deployment Catalog notation is targeted to the ActiveBPEL deployment environment, using the WSDLCatalog deployment language. This relation set specifies how a WSDLCatalog notation is related to an XML based notation, enabling models based upon the WSDLCatalog notation to be serialised to XML text. The relation is expressed in Figure 7.65.

```
transformation wdcToxml(wdc1:WSDLCatalog, xml1:XML)
{
....
}
```

Figure 7.65: Textual QVT wdcToxml transform declaration.

The two relations in this set are outlined as follows, and described in more detail below. The two relations show how the WSDLCatalog artifacts are mapped to the XML notation.

- WSDLCatalogToRoot
- WSDLEntryToElement

The first relation, WSDLCatalogToRoot, defines the relation between a WSDLCatalog WSDLCatalog artifact and an XML Root artifact. The relation is a top level relation and so will be matched rather than being called directly by another relation. As in the other relation sets the properties of both notations are bound using the same variable names meaning they share the same values. The *where* clause indicates that the function, getAllWSDLEntries (retrieves all the WSDLEntries references from a WSDLCatalog source model), must be subsequently executed. The relation is expressed graphically in Figure 7.66.

The second relation, defines the relation between a WSDLCatalog WSDLEntry artifact and an XML Element artifact. The relation is a top level relation and so will be matched rather than being called directly by another relation. No *when* or *where* clause are specified.

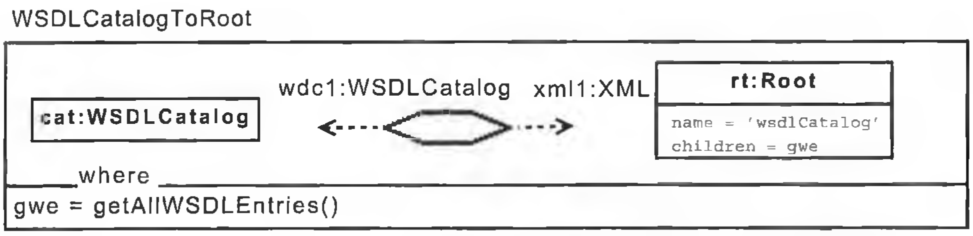


Figure 7.66: Graphical QVT WSDLCatalogToRoot relation declaration.

The relation is expressed graphically in Figure 7.67.

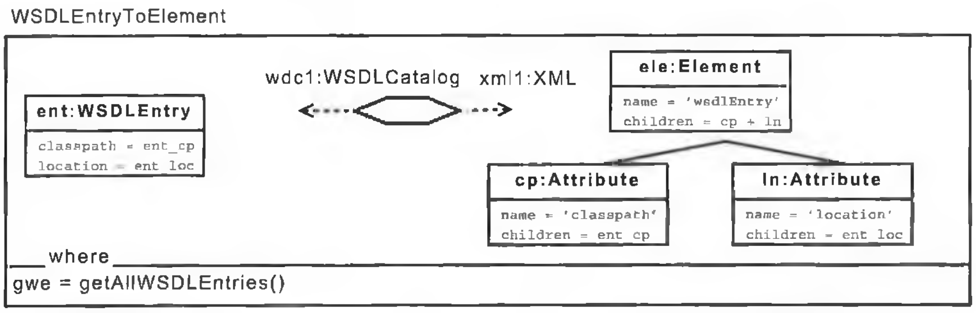


Figure 7.67: Graphical QVT WSDLEntryToElement relation declaration.

7.5 Tool Support

Currently there are no tools supporting the definition of QVT relations, either textual or graphical. However, a recently released open source tool, SmartQVT, supports the QVT-Operational language [176]. This tool provides an Eclipse environment to define QVT operations. These operations are then compiled using the Python language [177] to produce Java code, which can be used to execute the operations as transformations. This tool represents the first efforts to provide tool support for the QVT specification.

Here, we have used the freely available Dia tool [77] to define QVT graphical relations. Dia supports the creation of UML Object diagrams. A text editor was used to define the transformation declarations. No validation tool is currently available to validate the relations defined.

7.6 Summary

In this chapter we have presented the third component in our modeling and transformation framework, model relations. We have motivated their use as a template for the creation of model transformations. We have outlined nine relation sets that describe the web of dependencies between the seven languages defined in Chapter 6, from UML distribution pattern model to executable system XML. The relations outlined are for a centralised shared hub distribution pattern, however the model relations vary slightly depending on the distribution pattern chosen by the software architect.

Chapter 8

Model Transformations

8.1 Introduction

In this chapter we present our model transformations. Model transformations are the fourth component in our modeling and transformation framework. These transformations define how a source model is converted into a target model, whilst respecting the relations defined in Chapter 7. The relations defined in Chapter 7 define the abstract semantic mappings between artifacts of different notations. Here, we use these relations as the basis for the definition of executable transformations. The goal of transformations is to create a new model based upon a previously defined model, where the two candidate models have different notations. As with model relations, model transformations vary slightly depending on the distribution pattern chosen by the software architect.

The techniques used to define our transformations are discussed in Section 8.2. The MDA framework within which the transformations must be defined is discussed in Section 8.3. The transformations between candidate models are discussed in Section 8.4. Finally in Section 8.5, we discuss the MDA based tool support available for defining transformations between models.

8.2 Model Transformations Specification

Model transformations are the programs that define a mechanism for traversing the web of dependencies that have been previously been defined between a set of relations. They are implemented using transformation languages which transform a source model to a target model. This transformation process must respect the relational rules for a given domain, such as those previously defined in Chapter 7. Model transformations realise model relation templates and are executable.

8.2.1 ATL Transformation Language

Here, we use the Atlas Transformation Language (ATL) to define executable model transformation rules [96]. ATL is a hybrid model transformation language featuring both declarative and imperative constructs, which conforms to the MOF meta-meta-model. The language and related open source tools, ATL Development Tools (ADT), have no reliance on the UML serialisation format, XMI [80]. This independence from XMI is important as the XMI format can, and does, change considerably from version to version. ATL instead manipulates models using patterns and meta-models. These meta-models and patterns can be easily updated. This results in a considerably more extensible, maintainable and user friendly transformation solution. A wider context for the classification of model transformation languages, is provided by Czarnecki et al. in [51], and more recently in [52].

ATL transformations are realised using files that consist of a number of related discrete transformations, which may be considered as a set. Each ATL transformation set is implemented in a module, where each module has a header section, an optional import section and a number of optional helpers, and at least one transformation rule [96]. ATL has three rule definition formats. The module header declares the source and target models for the transformation. An import section details external types used in the transformation. The helper section outlines functions that navigate a source model and return a result or a set of results. Finally, transformation rules express the transformation logic to transform a source model to a target model.

There are a number of formats for transformation rules in ATL. The first, matched rules are used to define source and target patterns in the models to be transformed. Matched rules may also feature a guard to restrict the output from the source pattern. Secondly, lazy rules are called from matched rules or other lazy/called rules. These rule types will never be executed directly and are normally called with a parameter consisting of source model artifacts. Lazy rules may be tagged as unique, meaning they will only execute once for a given match. The third format is termed called rules, which are similar to lazy rules except they may take in any number of parameters. They must explicitly return a result or else nothing will be outputted. Figure 8.1 illustrates an ATL matched rule. In this example a construct A, from the Model1 notation, is transformed to a construct B, in the Model2 notation. Two attributes, attrib1 and attrib2 are then copied from construct A and applied to construct B.

```
module Model1ToModel2; -- Module Template
create OUT : Model2 from IN : Model1;
--Transform a Model1 construct (A) to Model2 construct (B)
rule ConstructAToConstructB{
  from
    a . Model1!A
  to
    b : Model2!B(
    attrib1 <- a.attrib1 ,
    attrib2 <- a.attrib2
  )
}
```

Figure 8.1: Example ATL transformation definition.

Transformation rules are specified declaratively where possible. Declarative rules are favoured as they clearly show the links between source and target models, whilst hiding complex details like selection, sorting, ordering and subsequent rule triggering of a transformation. However, many complex transformations cannot be declared solely using the declarative syntax. Imperative blocks may be used in any of the three rule forms. This construct is useful when the order of transformation rule execution must be explicitly managed, or target model artifacts are conditionally created based upon some logic not expressed in the source pattern or guard.

Traceability, discussed in detail in Section 7.2, of ATL transformations is considered by

Jouault in [93]. Jouault discusses how trace information is persisted after a transformation is executed, without altering the transformations themselves. Within the ATL transformation engine, transformation links are automatically created by relating a rule, a source match and a newly created target.

As previously discussed in Chapter 7, we use the QVT graphical language to declaratively illustrate the relationships between source and target notations. However, the QVT declarative language is currently only a specification, meaning it cannot be executed. To enable model transformations we implement the relations previously defined using QVT in Chapter 7 in ATL. ATL has no graphical format equivalent to the QVT graphical language. A comparison of ATL and QVT is investigated by Jouault et al. in [95]. Here, the author notes that different transformation languages are appropriate for different domains. They also note that interoperability between QVT and ATL is possible, depending on the conceptual levels of the defined transformations. For example ATL to QVT Operational Mappings are possible. This interoperability between languages means the decision to use ATL or QVT is not critical.

8.3 Framework

In Section 6.3 we discussed how we leverage the MDA framework to enable code generation from high level models. As with the model relations defined in Chapter 7, the model transformations must be compatible with this MDA approach. Both the ATL transformation language and the QVT transformation language are compatible with this approach.

8.4 Transformations

ATL uses transformation rules to define transformations between candidate models. Each "Relation" in Figures 7.1 through 7.3 is implemented using a set of ATL transformations. Each ATL transformation rule requires the definition of two or more domains. The source domain refers to a particular artifact in the source notation, whilst the target domain refers to an artifact in the target notation. The source domain may have a source pattern defined, which restricts the search space over the source artifacts. A destination pattern may also be defined to apply source model matches, and subsequently bind variables, to destination model variables. These variables represent model instance artifacts of both the source and target models. The assignment of these variables from source to target models, via transformational rules, is the basis for the transformation process.

Transformational rules which have pre-conditions, indicated by the *when* clause in relations, are marked with the keyword *lazy* or have a parameter list in their transformational prototype. These rules must be called by other rules. Transformational rules with post-conditions, indicated by the *where* clause in relations, will execute lazy rules, called rules or helper functions. These function calls are marked by the *thisModule* identifier.

Using seven of the notations defined in Chapter 6, we consider nine transformation sets that define transformations from high level UML model to lower level executable model. These nine transformation sets can be categorised into three subsets as outlined below.

- UML 2.0 Notation (with DPLProfile applied) to Distribution Pattern Language (DPL) Notation
- DPL Notation to Executable System Notations (Collaboration, Interface, Deployment Descriptor, Deployment Catalog)
- Executable System Notations to XML Notation

Although some of the relations defined in Chapter 7 are bi-directional, all the transformations defined here are uni-directional, from source to target model. We do not consider backward transformations from target to source models as the focus here is the creation of an executable system from high level models, rather than a reverse engineering effort. In fact, ATL is inherently uni-directional in that the source model is read only and so can be navigated but not transformed itself.

We use ATL exclusively in its standard mode of operation. This mode requires a transformation for every artifact that is to be outputted from a transformation. An alternative mode, refining, is provided by ATL. This mode causes unmatched artifacts to be automatically outputted. The refining mode is not utilised here.

Each set of model transformations is either distribution pattern independent or distribution pattern dependent. Transformations which are pattern dependent define transformation rules between source artifacts and different target artifacts, depending on the distribution pattern chosen. Pattern independent transformations define relationships between source artifacts and target artifacts that are always the same regardless of the chosen pattern. The UML to DPL relations and the Executable System to XML transformations are pattern independent, whilst the remaining transformations are pattern dependent. These pattern dependent transformations differ slightly depending on the individual pattern requirements. Pattern specific parts of the code are clearly highlighted in the transformation definitions. A new release of the ATL language supports transformation inheritance, which would assist in the modularisation of the transformatio code base into pattern specific and pattern independent modules.

Helpers, as previously noted, are simple operations performed over the source model to return either collections of model artifacts, individual model artifacts or simple return values i.e. strings, Booleans or integers. These helpers do not transform source artifacts to target artifacts, instead they assist in the definition of transformations. All helpers are prefixed with either "get" or "convert". The helpers are defined in Appendix A. All of the helpers have been fully tested in our tool implementation.

To demonstrate the definition of model transformations from a UML model to an executable system, for a given distribution pattern, we consider a worked example. This example features a centralised shared hub distribution pattern, outlined in Section 5.4.1, and whose relations are defined in Chapter 7. Here, we will define the transformations that implement the relations of the worked example for a centralised shared hub distribution pattern. Although we consider a specific worked example in this chapter, the transformations are generic and may be applied to any application. If a different distribution pattern was required the same process would be followed to define the generic transformations. The effectiveness of the transformation based approach is assessed in Chapter 10.

In the following subsections we enumerate all the transformations. Each transformation set implements a worked example relation set, as defined in Chapter 7. Each transformation is briefly introduced before presenting the code of the ATL transformation itself. Additional details are provided in the text for complex transformations. The transformations also include comments in the code where necessary.

8.4.1 Transforming UML 2.0 Model/DPLProfile to DPL Model

The first transform set is from a UML 2.0 model to a DPL model, where the UML 2.0 and the DPL models are the candidate models. These transformations are uni-directional as the transformation uses the source model to generate the target model, but does not define rules for transforming the target to the source model. UML 2.0 is a standards based graphical language for the modeling of software systems, as discussed in Section 6.4.1. DPL is our internal representation format for distribution patterns, which is discussed in Section 6.4.3. This transformation set is pattern independent as these transformations, unless otherwise indicated, hold across all distribution patterns.

The ATL module transformation header declares the name for the transformational set and declares two models, the source model UML and the target model DPL. The module is expressed in Figure 8.2.

```
module UMLActivityToDPL; -- Module Template
create OUT : DPL from IN : UML;
```

Figure 8.2: ATL UMLActivityToDPL transform module declaration.

The five transformations in this set are outlined as follows, and described in more detail below.

- ActivityToPatternDefinition
- ActivityPartitionToNode
- CallBehaviorActionToNode

- ObjectFlowToMapping
- PinToCorrelationVariables

The first transformation, defined in Figure 8.3, transforms a UML Activity artifact to a DPL pattern-definition artifact and its child artifacts CollaborationLanguage, DistributionPattern, ServiceName, OperationName, BaseNamespace, NamespacePrefix, Nodes and CorrelationVariables. The transformation implements the relation illustrated in Figure 7.5.

```
--Transform a UML Activity(there is one per UML diagram) to a DPL pattern-definition
rule ActivityToPatternDefinition {
  from
    act : UML!Activity
  to
    pd : DPL!"pattern-definition"("collaboration-language" <-cl,"distribution-pattern" <-dp,
      "service-name" <- sn,"operation-name" <- opn,"base-namespace" <- bn,
      "namespace-prefix" <- np,nodes <- nds,"correlation-variables" <- cvs),
    cl : DPL!CollaborationLanguage(text <- act.getValue(act.getAppliedStereotypes().first()
    ,'collaboration-language')),
    dp : DPL!DistributionPattern(text <- act.getValue(act.getAppliedStereotypes().first()
    ,'distribution-pattern')),
    sn : DPL!ServiceName(text <- act.getValue(act.getAppliedStereotypes().first(),'service-name')),
    opn : DPL!OperationName(text <- act.getValue(act.getAppliedStereotypes().first(),'operation-name')),
    bn : DPL!BaseNamespace(text <- act.getValue(act.getAppliedStereotypes().first(),'base-namespace')),
    np : DPL!NamespacePrefix(text <- act.getValue(act.getAppliedStereotypes().first(),'namespace-prefix')),
    nds : DPL!Nodes(node <- thisModule.getActivityPartitions()),
    cvs : DPL!CorrelationVariables(variable <- thisModule.getCorrelationVariables())
}
```

Figure 8.3: ATL ActivityToPatternDefinition transformation definition.

The second transformation, defined in Figure 8.4, transforms a UML ActivityPartition to a DPL Node artifact. There can be many ActivityPartition artifacts per UML diagram. The transformation implements the relation illustrated in Figure 7.6.

```
--Transform all UML ActivityPartitions to DPL Nodes
rule ActivityPartitionToNode{
  from
    ap : UML!ActivityPartition
  to
    nd : DPL!Node (name <- ap.name,
      uri <- ap.getValue(ap.getAppliedStereotypes().first(),'interface_uri'),
      engine_uri <- ap.getValue(ap.getAppliedStereotypes().first(),'engine_uri'),
      ns <- ap.getValue(ap.getAppliedStereotypes().first(),'ns'),
      operations <- ops),
    ops : DPL!Operations(
     --get all the UML CallBehaviorActions situated in this UML ActivityPartition and
     --assign them to DPL Node's operations collection
      operation <- ap.getNodes()->select(e|e.oclType()=UML!CallBehaviorAction)
     ->collect(e|thisModule.CBAToOperation(e)))
}
```

Figure 8.4: ATL ActivityPartitionToNode transformation definition.

The third transformation, defined in Figure 8.5, transforms a UML CallBehaviorAction to a DPL Operation artifact and its child artifact Mappings. There can be many CallBehaviorAction artifacts in each UML ActivityPartition. The order value of the DPL Operation target artifact is determined by the source UML ControlFlow artifact coming into the source UML CallBehaviorAction. Mappings output artifacts are created based on the position of UML InitialNode, ActivityFinalNode source artifacts, along with CallBehaviorAction source artifacts with UML ObjectFlow artifacts going into them. The transformation implements the relation illustrated in Figure 7.7.

```
--Transform (when called) a UML CallBehaviorActions to a DPL Operation
lazy rule CBAToOperation{
  from
    cba : UML!CallBehaviorAction
  to
    op : DPL!Operation(name <- cba.name,
      role <- cba.getValue(cba.getAppliedStereotypes().first(),'role').name,
      returns<-cba.getValue(cba.getAppliedStereotypes().first(),'returns'),
     --get the operation's execution order value based on the UML ControlFlow
     --connection coming into the CBA
      order <- cba.getIncomings()->select(e|e.oclType()= UML!ControlFlow).at(1).getValue
      (cba.getIncomings()->select(e|e.oclType()= UML!ControlFlow).at(1)
      .getAppliedStereotypes().first(),'order'),
      mappings <- mps),
   --convert all the UML ObjectFlows on each CBA to a DPL Mapping
    mps : DPL!Mappings(
     --create a DPL Mapping for the start point of the activity i.e. InitialNode
      mapping <- UML!ObjectFlow.allInstances()->select(e|e.getSource().oclType() = UML!
      InitialNode and e.getTarget().getOwner().name.debug('cba') = cba.name)
      ->collect(e|thisModule.ObjectFlowToMapping(e)),
     --create a DPL Mapping for each CBA which has an ObjectFlow going into it
      mapping <- UML!ObjectFlow.allInstances()->select(e|e.getTarget().getOwner().name
       = thisModule.getNextCBA(cba))->collect(e|thisModule.ObjectFlowToMapping(e)),
     --create a Mapping for the end point of the activiy i.e. ActivityFinalNode
      mapping <- UML!ObjectFlow.allInstances()->select(e|e.getTarget().oclType()
      = UML!ActivityFinalNode and e.getSource().getOwner().name.debug('cba')
      = cba.name)->collect(e|thisModule.ObjectFlowToMapping(e)))
}
```

Figure 8.5: ATL CBAToOperation transformation definition.

The fourth transformation, defined in Figure 8.6, transforms a UML ObjectFlow to a DPL Mapping artifact and its child artifacts From and To. There is always one ObjectFlow artifact per UML Pin, InitialNode and ActivityFinalNode artifact. The transformation implements the relation illustrated in Figure 7.8.

The fifth transformation, defined in Figure 8.7, transforms a UML Pin to a DPL Variable artifact. There can be many Pin artifacts per UML CallBehaviorAction. Only UML Pin artifacts which have been marked as correlation variables should be passed into this rule. This rule is only required for distribution patterns requiring correlation variables and is ignored by patterns not requiring this construct. The transformation implements the relation illustrated in Figure 7.9.

```
--Transform (when called) a UML ObjectFlow connections to a DPL Mapping
lazy rule ObjectFlowToMapping{
  from
    of: UML!ObjectFlow
  to
    mp : DPL!Mapping ("from"<-fm,"to" <-t),
    fm : DPL!From(
      --retrieve the ObjectFlow values
       message <- thisModule.getMapping(of,'source'),
       part <- thisModule.getPin(of,'source').name,
       type <- thisModule.removeMMPrefix(thisModule.getPin(of,'source').getType())),
    t : DPL!To(
      --retrieve the ObjectFlow values
       message <- thisModule.getMapping(of,'target'),
       part <- thisModule.getPin(of,'target').name,
       type <- thisModule.removeMMPrefix(thisModule.getPin(of,'target').getType()))
}
```

Figure 8.6: ATL ObjectFlowToMapping transformation definition.

```
--Transform (when called) a UML Pin connections to a DPL Variable. This rule should
--only be called for pins with is_correlation_variable set to true
lazy rule PinToCorrelationVariable{
  from
    p : UML!Pin
  to
    var : DPL!Variable(name <- p.name)
}
```

Figure 8.7: ATL PinToCorrelationVariable transformation definition.

8.4.2 Transforming DPL Model to Collaboration Model

The second transformation set is from a DPL model to a Collaboration model, where the DPL model and the Collaboration model are the candidate models. These transformations are uni-directional as the transformation uses the source model to generate the target model, but does not define rules for transforming the target to the source model. It should be noted that bi-directionality would be impossible as the target model is only a subset of the source model. We assume in this example that the collaboration language, and subsequently target model, will be WS-BPEL, discussed in Section 2.3.2. We previously noted that other collaboration languages, such as WS-CDL, could have been used as an alternative to WS-BPEL. This transformation set is pattern dependent because the target model artifacts to be created depend on the chosen distribution pattern. Pattern specific parts of the code are clearly highlighted in the transformation definitions.

The ATL module transformation header declares the name for the transformational set and declares two models, the source model DPL and the target model BPEL. The module is expressed in Figure 8.8.

```
module DPLtoBPEL; -- Module Template
create OUT : BPEL from IN : DPL;
```

Figure 8.8: ATL DPLtoBPEL transform module declaration.

The ten transformations in this set are outlined as follows, and described in more detail below.

- PatternDefinitionToProcess
- NodeToNamespace
- OperationToVariable
- NodeToPartnerLink
- PatternDefinitionToNamespace
- OperationToInvoke
- MappingToAssign
- PatternDefinitionToVariable
- PatternDefinitionToNamespace
- PatternDefinitionToPartnerLink

The first transformation, defined in Figure 8.9, transforms a DPL pattern-definition artifact to a BPEL Process artifact and its child artifacts PartnerLinks, Variables, Sequence, Receive and Reply. The transform subsequently transforms the Reply artifact's children Operation, PortType, PartnerLink and Variable. There is only one pattern-definition artifact per DPL model. The order of BPEL activities is important to ensure the sequence of events within the composition occurs as specified by the software architect in the UML model. To ensure the BPEL target model conforms to this sequence we order the source Operation artifacts using their order attribute, and imperatively control the creation of appropriate Invoke and Assign target artifacts. The transformation implements the relation illustrated in Figure 7.11.

```
--Transform a DPL pattern-definition(there is one per DPL model) type to a BPEL process
 rule PatternDefinitionToProcess {
  from
    pd : DPL!"pattern-definition"
  to
    p : BPEL!Process(
      name<-pd."service-name".text, suppressJoinFailure<-true, --We assume this is always true
      targetNamespace<-pd."base-namespace".text +p.name, partnerLinks<-pls, variables<-vars, activity<-seq),
    pls : BPEL!PartnerLinks(--**Pattern Specific**
     --each DPL Node is a BPEL PartnerLink type
      children <- pd.nodes.node->collect(e|thisModule.NodeToParterLink(e)),
     --consider distribution pattern dependent partnerLinks
      children <- thisModule.PatternDefinitonToPartnerLink()),
    vars : BPEL!Variables(--**Pattern Specific**
     --need BPEL Variable for each DPL From and To type
      children <- pd.nodes.node->collect(e|e.operations)->collect(e|e.operation)
     ->flatten()->collect(e|thisModule.OperationToVariable('Request',e)),
      children <- pd.nodes.node->collect(e|e.operations)->collect(e|e.operation)
     ->flatten()->collect(e|thisModule.OperationToVariable('Response',e)),
     --consider distribution pattern dependent variables as they need BPEL Variables too
      children <- thisModule.PatternDefinitionToVariable('Request'),
      children <- thisModule.PatternDefinitionToVariable('Response')),
    seq : BPEL!"Sequence"(),
    rec : BPEL!Receive(createInstance<-true,name<-'ReceiveCaller',operation<-op,     partnerLink<-pl,portType
     <-pt,variable<-var_in),
    rep : BPEL!Reply(name<-'SendReplyToCaller',operation<-op,partnerLink<-pl,portType<-pt,variable<-var_out),
   --the BPEL Ecore references the WSDL ecore for operations, we still ref the BPEL MM though
    op : BPEL!Operation(name <- thisModule.getOperationName()),
   --the BPEL Ecore references the WSDL ecore for porttype, we still ref the BPEL MM though
    pt : BPEL!PortType(qName <- thisModule.getNamespacePrefix()+':'+thisModule.getServiceName() + 'PortType')
    pl : BPEL!PartnerLink(name<-'Caller'),
    var_in : BPEL!Variable(name <- thisModule.getServiceName()+'RequestType'),
    var_out : BPEL!Variable(name <- thisModule.getServiceName()+'ResponseType')
    do--to ensure the activities are sorted correctly we must process the DPL operations imperatively
    {
      seq.activities <- rec;
     --create mappings for all the InitialNode messages
      seq.activities <- pd.nodes.node->collect(e|e.operations)->collect(e|e.operation)
     ->flatten()->collect(e|e.mappings)->flatten()->collect(e|e.mapping)->flatten()->
      select(e|e."from".message = 'InitialNode')->collect(e|thisModule.MappingsToAssign(e));

     --interleave the invoke and assigns based on their order
      for(op in pd.nodes.node->collect(e|e.operations)->collect(e|e.operation)
->flatten()->sortedBy(e|e.order))
      {
        seq.activities <- thisModule.OperationToInvoke(op);

            for(mp in op.mappings.mapping->select(e|e."from".message <> 'InitialNode'
        and e."to".message <> 'FinalNode'))
            {
              seq.activities <- thisModule.MappingsToAssign(mp);
            }
          }
     --create mappings for all the FinalNode messages
      seq.activities <- pd.nodes.node->collect(e|e.operations)->collect(e|e.operation)
     ->flatten()->collect(e|e.mappings)->flatten()->collect(e|e.mapping)->flatten()
     ->select(e|e."to".message = 'FinalNode')->collect(e|thisModule.MappingsToAssign(e));
      seq.activities <- rep;--create a reply activity
      thisModule.PatternDefinitionToNamespace();
    }
}
```

Figure 8.9: ATL PatternDefinitionToProcess transformation definition.

```
--Transform each DPL node type into a BPEL namespace type
rule NodeToNamespace{
  from
    n: DPL!Node
  to
    ns : BPEL!Namespace(URI<-n.uri, prefix<-n.ns)
}
```

Figure 8.10: ATL NodeToNamespace transformation definition.

```
--Create (when explicitly called) BPEL variable holders for
--the values input and output during execution of the DPL Operation
rule OperationToVariable(type : String, op : DPL!Operation){
  to
    var: BPEL!Variable(
      name <- op.eContainer().eContainer().name + op.name + type, messageType <- mes),
    mes : BPEL!Message(qName <- op.eContainer().eContainer().ns+':'+op.name+type)
    --note:called rules MUST explicitly return a result
    do{var;}
}
```

Figure 8.11: ATL OperationToVariable transformation definition.

```
--Transform (when called) each DPL Node type to a BPEL PartnerLink type
lazy rule NodeToParterLink{
  from
    n: DPL!Node
  to
    pl : BPEL!PartnerLink(name<-n.name, partnerRole<-n.name, partnerLinkType<-thisModule.getNamespacePrefix()
      +':'+n.name+'PLT')
}
```

Figure 8.12: ATL NodeToPartnerLink transformation definition.

The second transformation, defined in Figure 8.10, transforms a DPL Node artifact to a BPEL Namespace artifact. There may be many Node artifact per DPL pattern-definition. The transformation implements the relation illustrated in Figure 7.12.

The third transformation, defined in Figure 8.11, transforms a DPL Operation artifact to a BPEL Message artifact. There may be many Operation artifacts per DPL Node. The transformation implements the relation illustrated in Figure 7.13.

The fourth transformation, defined in Figure 8.12, transforms a DPL Node artifact to a BPEL PartnerLink artifact. There may be multiple Node artifacts per DPL pattern-definition. The transformation is distribution pattern specific. The transformation implements the relation illustrated in Figure 7.14.

The fifth transformation, defined in Figure 8.13, creates a BPEL PartnerLink. The transformation is distribution pattern specific. The transformation implements the relation illustrated in Figure 7.20.

The sixth transformation, defined in Figure 8.14, transforms a DPL Operation artifact to a BPEL Invoke artifact, and its child artifacts Operation, PortType, PartnerLink and Variable. There may be many Operation artifacts per DPL Node. The transformation implements the relation illustrated in Figure 7.16.

```
--Create (when explicitly called) a BPEL PartnerLink type
--**Pattern Specific**
rule PatternDefinitonToPartnerLink() {
  to
    pl : BPEL!PartnerLink(name <- 'Caller', myRole <- thisModule.getServiceName()
      ,partnerLinkType <- thisModule.getNamespacePrefix()+':'+thisModule.getServiceName()+'PLT')
    do
    {pl;}
}
```

Figure 8.13: ATL PatternDefinitonToPartnerLink transformation definition.

```
--Transform (when called) a DPL Operation type to a BPEL Invoke type
lazy rule OperationToInvoke{
  from
    op : DPL!Operation
  to
    inv : BPEL!Invoke(name <- 'Invoke'+op.name,operation <- oper,portType <- pt,
      partnerLink <- pl,inputVariable <- invar,outputVariable <- outvar),
    oper : BPEL!Operation(name <- op.name ),
    pt : BPEL!PortType(qName <- op.eContainer().eContainer().ns +':' + op.eContainer().eContainer().name),
    pl : BPEL!PartnerLink(name <- op.eContainer().eContainer().name),
    invar : BPEL!Variable(name <- op.eContainer().eContainer().name + op.name + 'Request'),
    outvar : BPEL!Variable(name <- op.eContainer().eContainer().name + op.name + 'Response')
}
```

Figure 8.14: ATL OperationToInvoke transformation definition.

The seventh transformation, defined in Figure 8.15, transforms a DPL Mapping artifact to a BPEL Assign artifact, and its child artifacts From and To, and subsequently to their child artifacts Part and Variable. There may be many Mapping artifacts per DPL Operation. An imperative block is used here to make the rule usable in two contexts, for handling "Request" and "Response" source Mapping artifact types. The transformation implements the relation illustrated in Figure 7.17.

```
--Transform (when called) each DPL Mapping type to a BPEL Assign type
lazy rule MappingToAssign{
  from
    mp : DPL!Mapping
  to
    ass : BPEL!Assign(copy <- cpfrom,copy <- cpto),
    cpfrom :BPEL!From(part <- fmpart,variable <- fmvar),
    cpto : BPEL!To(part <- topart,variable <- tovar),
    fmpart : BPEL!Part(name <- mp."from".part),
    topart : BPEL!Part(name <- mp."to".part),
    fmvar : BPEL!Variable(name <-
      if mp."from".message = 'InitialNode' then
        thisModule.getServiceName() + 'RequestType'
      else
        mp."from".message
      endif
    ),
    tovar : BPEL!Variable(name <-
      if mp."to".message = 'FinalNode' then
        thisModule.getServiceName() + 'ResponseType'
      else
        mp."to".message
      endif
    )
}
```

Figure 8.15: ATL MappingToAssign transformation definition.

The eighth transformation, defined in Figure 8.16, creates a BPEL Variable artifact and associated child Message artifact. The transformation is distribution pattern specific. The transformation implements the relation illustrated in Figure 7.18.

```
--Create (when explicitly called) BPEL variable holders for
--the distribution pattern specific requirements
--**Pattern Specific**
rule PatternDefinitionToVariable(type : String){
  to
    var: BPEL!Variable(name <- thisModule.getServiceName()+type+'Type',messageType<-mes  -- this is from
the WSDL Message ECore EClass),
    mes : BPEL!Message(qName <- thisModule.getNamespacePrefix()+':'+thisModule.getServiceName()+type+'Type')
    --note:called rules MUST explicitly return a result
    do{var;}
}
```

Figure 8.16: ATL PatternDefinitionToVariable transformation definition.

The ninth transformation, defined in Figure 8.17, creates a BPEL Namespace. The transformation is distribution pattern specific and implements the relation illustrated in Figure 7.19.

```
--Create (when explicitly called) a BPEL Namespace type
--**Pattern Specific**
rule PatternDefinitionToNamespace() {
  to
    ns : BPEL!Namespace(URI <- thisModule getBaseNamespace() + thisModule.getServiceName(),
      prefix <- thisModule.getNamespacePrefix())
    do
    {ns;}
}
```

Figure 8.17: ATL PatternDefinitionToNamespace transformation definition.

8.4.3 Transforming DPL Model to Interface Model

The third transformation set is from a DPL model to an Interface model, where the DPL model and the Interface model are the candidate models. These transformations are unidirectional because the transformation uses the source model to generate the target model, but does not define rules for transforming the target to the source model. It should be noted that bi-directionality would be impossible as the target model is only a subset of the source model. We assume the Web service interface language will be WSDL, and thus utilise the WSDL notation, discussed in Section 2.3. This transformation set is pattern dependent as the target model artifacts to be created depend on the chosen distribution pattern. Pattern specific parts of the code are clearly highlighted in the transformation definitions.

The ATL module transformation header declares the name for the transformational set and declares two models, the source model DPL and the target model WSDL. The module is expressed in Figure 8.18.

```
module DPLtoWSDL; -- Module Template
create OUT : WSDL from IN : DPL;
```

Figure 8.18: ATL DPLtoWSDL transform module declaration.

The five transformations in this set are outlined as follows, and described in more detail below.

- PatternDefinitionToDefinition
- NodeToNamespace
- NodeToPartnerLinkType
- FromToPart
- ToToPart

The first transformation, defined in Figure 8.19, transforms a DPL pattern-definition artifact to a WSDL Definition artifact and its child artifacts Namespace, Message, Port-Type, PartnerLinkType and Service. The transformation subsequently transforms the Port-Type's child artifact Operation, the PartnerLinkType's child artifact Role, its child artifact PortType, the Operation's child artifacts Input and Output, and finally their child Message artifacts. There is only one pattern-definition artifact per DPL model. The transformation implements the relation illustrated in Figure 7.22.

The second transformation, defined in Figure 8.20, transforms a DPL Node artifact to a WSDL Namespace artifact. There can be many DPL Node artifacts per DPL model. The transformation implements the relation illustrated in Figure 7.23.

The third transformation, defined in Figure 8.21, transforms a DPL Node artifact to a WSDL PartnerLinkType artifact, its child artifact Role, and subsequently to its child artifact

```
--Transform a DPL pattern-definition type to a WSDL Definition type
rule PatternDefinitionToDefinition{
  from
    pd : DPL!"pattern-definition"
  to
  "def" : WSDL!Definition(
    targetNamespace <- thisModule.getBaseNamespace () + thisModule.getServiceName(),
   --Delegate the creation of the WSDL namespaces
    eNamespaces <- DPL!Node.allInstances()->collect(e|thisModule.NodeToNamespace(e)),
    eNamespaces <- ns,eMessages <- mesreq,eMessages <- mesres,ePortTypes <- pt,
   --delegate the creation of the WSDL PLTs
    ePartnerLinkTypes <- DPL!Node.allInstances()->collect(e|thisModule.NodeToPartnerLinkType(e)),
    ePartnerLinkTypes <- plt,eServices <- svc

  ),
 --namespace for this Service
  ns : WSDL!Namespace(URI <- thisModule.getBaseNamespace ()+thisModule.getServiceName()
    ,prefix <- thisModule.getNamespacePrefix()),
  mesreq : WSDL!Message(--**Pattern Specific**
    qName <- thisModule.getServiceName()+'RequestType',
    eParts <- DPL!From.allInstances()->select(e|e.message='InitialNode')->collect(e|thisModule.FromToPart(e))
  ),
  mesres : WSDL!Message(--**Pattern Specific**
    qName <- thisModule.getServiceName()+'ResponseType',
    eParts <- DPL!To.allInstances()->select(e|e.message='FinalNode')->collect(e|thisModule.ToToPart(e))
  ),
  pt : WSDL!PortType(qName <- thisModule.getServiceName() + 'PortType',eOperations <- op),
 --Definition for Hub PLT
  plt : WSDL!PartnerLinkType(name <- thisModule.getServiceName() + 'PLT',role <- rle),
  rle : WSDL!Role(name <- thisModule.getServiceName(),portType <- plpt),
  plpt : WSDL!PortType(qName <- thisModule.getNamespacePrefix() + ':'
    + thisModule.getServiceName() + 'PortType'),
  op : WSDL!Operation(name <- thisModule.getOperationName(),eInput <- ip,eOutput <- opt),
  ip : WSDL!Input(eMessage <- ipmess),
  ipmess : WSDL!Message(qName <- thisModule.getNamespacePrefix()+':'
    +thisModule.getServiceName()+'RequestType'),
  opt : WSDL!Output(eMessage <- opmess),
  opmess : WSDL!Message(qName <- thisModule.getNamespacePrefix()+':'
    +thisModule.getServiceName()+'ResponseType'),
  svc : WSDL!Service(qName <- thisModule.getServiceName())
}
```

Figure 8.19: ATL PatternDefinitionToDefinition transformation definition.

```
--Transform (when called) a DPL Node type to a WSDL Namespace type
lazy rule NodeToNamespace{
  from
    nd : DPL!Node
  to
    ns : WSDL!Namespace(URI <- nd.uri,prefix <- nd.ns)
}
```

Figure 8.20: ATL NodeToNamespace transformation definition.

```
--Transform (when called) a DPL Node type to a WSDL PartnerLinkType type
lazy rule NodeToPartnerLinkType{
  from
    nd : DPL!Node
  to
    plt : WSDL!PartnerLinkType(name <- nd.name + 'PLT',role <- rle),
    rle : WSDL!Role(name <- nd.name,portType <- pt),
    pt : WSDL!PortType(qName <- nd.ns + ':' + nd.name )
}
```

Figure 8.21: ATL NodeToPartnerLinkType transformation definition.

PortType. There can be many DPL Node artifacts per DPL model. The transformation implements the relation illustrated in Figure 7.24.

The fourth transformation, defined in Figure 8.22, transforms a DPL From artifact to a WSDL Part artifact. There is only one From artifact per DPL Mapping. The transformation implements the relation illustrated in Figure 7.25.

```
--Transform (when called) a DPL From type to a WSDL Part type
lazy rule FromToPart{
  from
    fm : DPL!From
  to
    pt : WSDL!Part(name <- fm.part ,typeName <- thisModule.convertETypeToWSDLType(fm.type))
}
```

Figure 8.22: ATL FromToPart transformation definition.

The fifth transformation, defined in Figure 8.23, transforms a DPL To artifact to a WSDL Part artifact. There is only one To artifact per DPL Mapping. The transformation implements the relation illustrated in Figure 7.26.

```
--Transform (when called) a DPL To type to a WSDL Part type
lazy rule ToToPart{
  from
    "to" : DPL!To
  to
    pt : WSDL!Part(name <- "to".part.typeName <- thisModule.convertETypeToWSDLType("to".type))
}
```

Figure 8.23: ATL ToToPart transformation definition.

8.4.4 Transforming DPL Model to Deployment Descriptor Model

The fourth transformation set is from a DPL model to a Deployment Descriptor model, where the DPL model and the Deployment Descriptor model are the candidate models. These transformations are uni-directional as the transformation uses the source model to generate the target model, but does not define rules for transforming the target to the source model. It should be noted that bi-directionality would be impossible as the target model is only a subset of the source model. We previously assumed the collaboration language will be WS-BPEL. It follows that we must choose a WS-BPEL compliant deployment environment. Once such platform is ActiveBPEL. ActiveBPEL has a deployment descriptor format PDD (Process Deployment Descriptor), which is the basis for the deployment descriptor notation. This transformation set is pattern dependent because the target model artifacts to be created depend on the chosen distribution pattern. Pattern specific parts of the code are clearly highlighted in the transformation definitions.

The ATL module transformation header declares the name for the transformational set and declares two models, the source model DPL and the target model PDD. The module is expressed in Figure 8.24.

```
module DPLtoPDD; -- Module Template
create OUT : PDD from IN : DPL;
```

Figure 8.24: ATL DPLtoPDD transform module declaration.

The five transformations in this set are outlined as follows, and described in more detail below.

- PatternDefinitionToProcess
- NodeToPartnerLink
- NodeToWSDL
- PatternDefinitionToPartnerLink
- PatternDefinitionToWSDL

The first transformation, defined in Figure 8.25, transforms a DPL pattern-definition artifact to a PDD Process artifact, and its child artifacts PartnerLinks and WSDLReferences. There is only one pattern-definition artifact per DPL model. The transformation implements the relation illustrated in Figure 7.28.

The second transformation, defined in Figure 8.26, transforms a DPL Node artifact to a PDD PartnerLink artifact, its child artifact PartnerRole, its child artifact EndpointReference, and finally its children Address and ServiceName. There can be many DPL Node artifacts per DPL model. The transformation implements the relation illustrated in Figure 7.29.

```
--Transform a DPL pattern-definition to a PDD Process
rule PatternDefinitionToProcess {
  from
    pd : DPL!"pattern-definition"
  to
    p : PDD!Process(name <- thisModule.getServiceName(),namespace <- thisModule.getBaseNamespace()
      + thisModule.getServiceName(),
      location <- 'bpel/'+ thisModule.getServiceName() + '.bpel',partnerLinks <- pls,wsdlReferences <- wsr),
    pls : PDD!PartnerLinks(partnerLink <- pd.nodes.node->collect(c|thisModule.NodeToParterLink(c))
      ,partnerLink <- thisModule.PatternDefinitionToPartnerLink()),
    wsr : PDD!WSDLReferences(wsdl <- pd.nodes.node->collect(c|thisModule.NodeToWSDL(c))
      ,wsdl <- thisModule.PatternDefinitionToWSDL())
}
```

Figure 8.25: ATL PatternDefinitionToProcess transformation definition.

```
--Transform (when called) each DPL Node to a PDD PartnerLink
lazy rule NodeToParterLink{
  from
    n: DPL!Node
  to
    pl : PDD!PartnerLink(name<-n.name,partnerRole<-pr),
    pr : PDD!PartnerRole(endpointReferenceType <- 'static',endpointReference <- cr),
    cr : PDD!EndpointReference(uri <- n.uri,namespace <- n.ns,address <- ad,serviceName <- sn),
    ad : PDD!Address(text <- n.ns + ':' + n.uri),
    sn : PDD!ServiceName(PortName <- n.name,text <- n.ns + ':' + n.name + 'Service')
}
```

Figure 8.26: ATL NodeToPartnerLink transformation definition.

The third transformation, defined in Figure 8.27, transforms a DPL Node artifact to a PDD WSDL artifact. There can be many DPL Node artifact per DPL model. The transformation implements the relation illustrated in Figure 7.30.

```
--Transform (when called) each DPL Node to a PDD WSDL type
lazy rule NodeToWSDL {
  from
    nd : DPL!Node
  to
    ws : PDD!WSDL(location <- 'wsdl/'+nd.name + '.wsdl',namespace <-  nd.uri)
}
```

Figure 8.27: ATL NodeToWSDL transformation definition.

The fourth transformation, defined in Figure 8.28, creates a PDD PartnerLink artifact, and its child artifact MyRole. The transformation implements the relation illustrated in Figure 7.31.

The fifth transformation, defined in Figure 8.29, creates a PDD WSDL artifact. The transformation implements the relation illustrated in Figure 7.32.

```
--Create (when explicitly called) a PDD PartnerLink
--**Pattern Specific**
rule PatternDefinitionToPartnerLink() {
  to
    pl : PDD!PartnerLink(name <- 'Caller', myRole <- mr),
    mr : PDD!MyRole(service <- thisModule.getServiceName(),allowedRoles <- '',binding <- 'RPC')
    do
    {pl;}
}
```

Figure 8.28: ATL PatternDefinitionToPartnerLink transformation definition.

```
--Create (when explicitly called) a PDD WSDL
--**Pattern Specific**
rule PatternDefinitionToWSDL() {
  to
    ws : PDD!WSDL(location <- 'wsdl/' + thisModule.getServiceName() + '.wsdl'
      ,namespace <- thisModule.getBaseNamespace() + thisModule.getServiceName())
    do{ws;}
}
```

Figure 8.29: ATL PatternDefinitionToWSDL transformation definition.

8.4.5 Transforming DPL Model to Deployment Catalog Model

The fifth transformation set is from a DPL model to a Deployment Catalog model, where the DPL model and the Deployment Catalog model are the candidate models. These transformations are uni-directional as the transformation uses the source model to generate the target model, but does not define rules for transforming the target to the source model. It should be noted that bi-directionality would be impossible as the target model is only a subset of the source model. We previously stated that we assume the use of a centralised distribution pattern, the WS-BPEL collaboration language and the ActiveBPEL execution platform. ActiveBPEL has a deployment catalog format WSDLCatalog, which is the basis for the Deployment Catalog notation. This transformation set is pattern dependent as the target model artifacts to be created depend on the chosen distribution pattern. Pattern specific parts of the code are clearly highlighted in the transformation definitions.

The ATL module transformation header declares the name for the transformational set and declares two models, the source model DPL and the target model WSDLCatalog. The module is expressed in Figure 8.30.

The three relations in this set are outlined as follows, and described in more detail below.

- PatternDefinitionToWSDLCatalog

```
module DPLtoWSDLCatalog; -- Module Template
create OUT : WSDLCatalog from IN : DPL;
```

Figure 8.30: ATL DPLtoWSDLCatalog transform module declaration.

- NodeToWSDLEntry
- PatternDefinitionToWSDLEntry

The first transformation, defined in Figure 8.31, transforms a DPL pattern-definition artifact to a WSDLCatalog WSDLCatalog artifact. There is only one pattern-definition artifact per DPL model. The transformation implements the relation illustrated in Figure 7.34.

```
--Transform a DPL pattern-definition to a WSDLCatalog WSDLCatalog
rule PatternDefinitionToWSDLCatalog{
  from
    pd : DPL!"pattern-definition"
  to
    cat : WSDLCatalog!WSDLCatalog(children <- thisModule.getAllNodes()
      ,children <- thisModule.PatternDefinitionToWSDLEntry(pd))
}
```

Figure 8.31: ATL PatternDefinitionToWSDLCatalog transformation definition.

The second transformation, defined in Figure 8.32, transforms a DPL Node artifact to a WSDLCatalog WSDLEntry artifact. There can be many DPL Node artifacts per DPL model. The transformation implements the relation illustrated in Figure 7.35.

```
--Transform each DPL Node to a WSDLCatalog WSDLEntry type
rule NodeToWSDLEntry {
  from
    nd : DPL!Node
  to
    ent : WSDLCatalog!WSDLEntry(location <- 'wsdl/'+nd.name + '.wsdl',classpath <-
'wsdl/'+nd.name + '.wsdl')
}
```

Figure 8.32: ATL NodeToWSDLEntry transformation definition.

The third transformation, defined in Figure 8.33, transforms a DPL pattern-definition artifact to a WSDLCatalog WSDLEntry artifact. There is only one pattern-definition artifact per DPL model. The transformation implements the relation illustrated in Figure 7.36.

```
--Transform (when called) a DPL "pattern-definition" to a WSDLCatalog WSDLEntry type
lazy rule PatternDefinitionToWSDLEntry {
  from
    pd : DPL!"pattern-definition"
  to
    ent : WSDLCatalog!WSDLEntry(location <- 'wsdl/'+ thisModule.getServiceName() + ' wsdl'
      ,classpath <-  'wsdl/'+ thisModule.getServiceName() + '.wsdl')
}
```

Figure 8.33: ATL PatternDefinitionToWSDLEntry transformation definition.

8.4.6 Transforming Collaboration Model to XML Model

The sixth transformation set is from a Collaboration model to an XML model, where the BPEL model and the XML model are the candidate models. These transformations are unidirectional as the transformation uses the source model to generate the target model, but does not define rules for transforming the target to the source model. Again we assume the Collaboration notation is targeted to the WS-BPEL collaboration language. This transformation set specifies how a WS-BPEL model is transformed to an XML based model, which can be serialised to WS-BPEL compliant XML text. This transformation set is pattern independent as the target model artifacts do not depend on the chosen distribution pattern. Pattern specific parts of the code are clearly highlighted in the transformation definitions.

The ATL module transformation header declares the name for the transformational set and declares two models, the source model BPEL and the target model XML. The module is expressed in Figure 8.34.

```
module BPELtoXML; -- Module Template
create OUT : XML from IN : BPEL;
```

Figure 8.34: ATL BPELtoXML transform module declaration.

The eight transformations in this set are outlined as follows, and described in more detail below.

- ProcessToRoot
- NamespaceToAttribute
- InvokeActivityToElement

- ReceiveActivityToElement
- ReplyActivityToElement
- AssignActivityToElement
- VariableToElement
- PartnerLinkToElement

The first transformation, defined in Figure 8.35, transforms a BPEL Process artifact to an XML Root artifact along with to a number of nested XML Element and Attribute artifacts. There is only one Process artifact per BPEL model. An imperative block is used to ensure the source model artifacts Receive, Reply, Invoke and Assign are output to XML based artifacts in a predetermined order. The transformation implements the relation illustrated in Figure 7.38.

The second transformation, defined in Figure 8.36, transforms a BPEL Namespace artifact to an XML Attribute artifact. There can be many Namespace artifacts per BPEL Process. The transformation implements the relation illustrated in Figure 7.39.

The third transformation, defined in Figure 8.37, transforms a BPEL Activity artifact to an XML Element artifact, and subsequently to a number of child XML Attribute artifacts. There can be many Activity artifacts per BPEL Sequence. The transformation implements the relation illustrated in Figure 7.40.

The fourth transformation, defined in Figure 8.38, transforms a BPEL Activity artifact to an XML Element artifact, and subsequently to a number of child XML Attribute artifacts. There can be many Activity artifacts per BPEL Sequence. The transformation implements the relation illustrated in Figure 7.41.

The fifth transformation, defined in Figure 8.39, transforms a BPEL Activity artifact to an XML Element artifact, and subsequently to a number of child XML Attribute artifacts. There can be many Activity artifacts per BPEL Sequence. The transformation implements the relation illustrated in Figure 7.42.

```
--Transform the BPEL Process to an XML Root type
rule ProcessToRoot (
  from
    p : BPEL!Process
  to
    rt : XML!Root(name <- 'process',
      children <- nme,children <- sjf,children <- tn,
      children <- xmlns,children <- thisModule.getAllNamespaces(),
      children <- bpwsns,children <- xsdns,children <- pls,children <- vars,children <- acti),
    nme:XML!Attribute (name <- 'name',value <- p.name),
    sjf:XML!Attribute (name <- 'suppressJoinFailure'
      ,value <- thisModule.convertBooleanToString(p.suppressJoinFailure)),
    --this is the BPEL process specific namespace for the composition
    tn:XML!Attribute (name <- 'targetNamespace',value <- p.targetNamespace),
    --this is the BPEL namespace which is always the same
    xmlns:XML!Attribute (name <- 'xmlns',value <- 'http://schemas.xmlsoap.org/ws/2003/03/business-process/'),
    --this is the BPEL namespace which is always the same
    bpwsns:XML!Attribute (name <- 'xmlns:bpws'
      ,value <- 'http://schemas.xmlsoap.org/ws/2003/03/business-process/'),
    --this is the XSD namespace which is always the same
    xsdns:XML!Attribute(name <- 'xmlns:xsd',value <- 'http://www.w3.org/2001/XMLSchema'),
    pls:XML!Element(name <- 'partnerLinks'
      ,children <- p.partnerLinks.children->collect(c|thisModule.PartnerLinkToElement(c))),
    vars:XML!Element(name <- 'variables'
      ,children <- p.variables.children->collect(c|thisModule.VariableToElement(c))),
    acti:XML!Element(name <- 'sequence',children <- seq),
    seq:XML!Attribute(name <- 'name',value <- 'Sequence1')
    do
    {
      --we need to control the order of the activities so we must use a for loop instead
      --of just have a matched rule. Matched rules cannot be used if have to order the output
      for(act in p.activity.activities)
          {
            if(act.oclType() = BPEL!Receive)
            {
              acti.children <- thisModule.ReceiveActivityToElement(act);
            }
            else if(act.oclType() = BPEL!Invoke)
            {
              acti.children <- thisModule.InvokeActivityToElement(act);
            }
            else if(act.oclType() = BPEL!Assign)
            {
              acti.children <- thisModule.AssignActivityToElement(act);
            }
            else if(act.oclType() = BPEL!Reply)
            {
              acti.children <- thisModule.ReplyActivityToElement(act);
            }
          }
    }
}
```

Figure 8.35: ATL ProcessToRoot transformation definition.

```
--Transform a BPEL Namespace to an XML Attribute type
rule NamespaceToAttribute{
  from
    ns: BPEL!Namespace
  to
    att : XML!Attribute(name <-'xmlns:'+ns.prefix,value <- ns.URI)
}
```

Figure 8.36: ATL NamespaceToAttribute transformation definition.

```
--Transform (when called) each BPEL Activity type to an XML Element type
lazy rule InvokeActivityToElement{
  from
    act: BPEL!Activity
  to
    ele : XML!Element(name <- thisModule.getActivityType(act.oclType()),children <- nme,
      children <- op,children <- pl,children <- pt,children <- invar,children <- outvar
    ),
    nme : XML!Attribute(name <- 'name',value <- act.name),
    op : XML!Attribute(name <- 'operation',value <- act.operation.name ),
    pl : XML!Attribute(name <- 'partnerLink',value <- act.partnerLink.name),
    pt : XML!Attribute(name <- 'portType',value <- act.portType.qName),
    invar : XML!Attribute(name <- 'inputVariable',value <- act.inputVariable.name),
    outvar : XML!Attribute(name <- 'outputVariable',value <- act.outputVariable.name)
}
```

Figure 8.37: ATL InvokeActivityToElement transformation definition.

The sixth transformation, defined in Figure 8.40, transforms a BPEL Activity artifact to an XML Element artifact, and subsequently to a number of child XML Attribute and Element artifacts. There can be many Activity artifacts per BPEL Sequence. The transformation implements the relation illustrated in Figure 7.43.

The seventh transformation, defined in Figure 8.41, transforms a BPEL Variable artifact to an XML Element artifact, and subsequently to a number of child XML Attribute artifacts. There can be many Variable artifacts per BPEL Process. The transformation implements the relation illustrated in Figure 7.44.

The eighth transformation, defined in Figure 8.42, transforms a BPEL PartnerLink artifact to an XML Element artifact, and subsequently to a number of child XML Attribute artifacts. There can be many PartnerLink artifacts per BPEL Process. An imperative conditional block is used to control the creation of target model artifacts, depending on whether source model artifacts MyRole and PartnerRole exist. The transformation implements the relation illustrated in Figure 7.45.

```
--Transform (when called) each BPEL Receive type to an XML Element type
lazy rule ReceiveActivityToElement{
  from
    act: BPEL!Activity
  to
    ele : XML!Element(name <- thisModule.getActivityType(act.oclType())
      ,children <- nme,children <- cri ,children <- pl,children <- pt
      ,children <- op,children <- var),
    nme : XML!Attribute(name <-'name',value <- act.name
    ),
    cri : XML!Attribute(name <-'createInstance',value
     <- thisModule.convertBooleanToString(act.createInstance)),
    pl : XML!Attribute(name <-'partnerLink',value <- act.partnerLink.name),
    pt : XML!Attribute(name <-'portType',value <- act.portType.qName),
    op : XML!Attribute(name <-'operation',value <- act.operation.name),
    var : XML!Attribute(name <-'variable',value <- act.variable.name)
}
```

Figure 8.38: ATL ReceiveActivityToElement transformation definition.

```
--Transform (when called) each BPEL Reply type to an XML Element type
lazy rule ReplyActivityToElement{
  from
    act: BPEL!Activity
  to
    ele : XML!Element(name <- thisModule.getActivityType(act.oclType()),
      children <- nme,children <- pl,children <- pt,children <- op,children <- var),
    nme : XML!Attribute(name <-'name',value <- act.name),
    pl : XML!Attribute(name <-'partnerLink',value <- act.partnerLink.name),
    pt : XML!Attribute(name <-'portType',value <- act.portType.qName),
    op : XML!Attribute(name <-'operation',value <- act.operation.name),
    var : XML!Attribute(name <-'variable',value <- act.variable.name)
}
```

Figure 8.39: ATL ReplyActivityToElement transformation definition.

8.4.7 Transforming Interface Model to XML Model

The seventh transformation set is from an Interface model to an XML model, where the WSDL model and the XML model are the candidate models. These transformations are uni-directional as the transformation uses the source model to generate the target model, but does not define rules for transforming the target to the source model. We assume the Interface model is targeted to the WSDL interface language. This transformation set specifies how a WSDL model is transformed to an XML based model, which can be serialised as WSDL compliant XML text. This transformation set is pattern independent because the target model artifacts to be created do not depend on the chosen distribution pattern. Pattern specific parts of the code are clearly highlighted in the transformation definitions.

The ATL module transformation header declares the name for the transformational set and declares two models, the source model WSDL and the target model XML. The module is expressed in Figure 8.43.

```
--Transform (when called) each BPEL Assign type to an XML Element type
lazy rule AssignActivityToElement{
  from
    act : BPEL!Activity
  to
    ele : XML!Element(name <- thisModule.getActivityType(act.oclType()),children <- copy),
    copy : XML!Element(name <- 'copy',children <- "from",children <- "to"),
    "from" : XML!Element(name <- 'from',children <- fmpart,children <- fmvar),
    "to" : XML!Element(name <- 'to',children <- topart,children <- tovar),
    fmpart : XML!Attribute(name <-'part',value <- act.copy.at(1).part.name),
    fmvar : XML!Attribute(name <-'variable',value <- act.copy.at(1).variable.name),
    topart : XML!Attribute(name <-'part',value <- act.copy.at(2).part.name),
    tovar: XML!Attribute(name <-'variable',value <- act.copy.at(2).variable.name)
}
```

Figure 8.40: ATL AssignActivityToElement transformation definition.

```
--Transform (when called) each BPEL Variable type to an XML Element type
lazy rule VariableToElement{
  from
    var : BPEL!Variable
  to
    ele : XML!Element(name <-'variable',children <- name,children <- mst),
    name : XML!Attribute(name <- 'name',value <- var.name),
    mst : XML!Attribute(name <- 'messageType',value <- var.messageType.qName)
}
```

Figure 8.41: ATL VariableToElement transformation definition.

The twelve transformations in this set are outlined as follows, and described in more detail below.

- DefinitionToRoot
- NamespaceToAttribute
- MessageToElement
- PartToElement
- PortTypeToElement
- OperationToElement
- InputToElement
- OutputToElement
- PLTToElement
- RoleToElement

```
--Transform (when called) each BPEL PartnerLink type to an XML Element type
lazy rule PartnerLinkToElement{
  from
    pl: BPEL!PartnerLink
  to
    ele : XML!Element(name <-'partnerLink ',children <- name,children <- plt),
    --need to see if the attribute exists before running this
    mr : XML!Attribute(name <- 'myRole',value <- pl.myRole),
    name : XML!Attribute(name <- 'name',value <- pl.name),
    plt : XML!Attribute(name <- 'partnerLinkType',value <- pl.partnerLinkType),
    pr : XML!Attribute(name <- 'partnerRole ',value <- pl.partnerRole)
    do --imperative block
    {
      --some attributes are not mandatory, we must check to see if they exist before trying to output them
      if(pl.myRole oclIsUndefined() <> true)
      {
        ele.children <- mr;
      }
      if(pl.partnerRole.oclIsUndefined() <> true)
      {
        ele.children <- pr;
      }
    }
}
```

Figure 8.42: ATL PartnerLinkToElement transformation definition.

```
module WSDLtoXML; -- Module Template
create OUT : XML from IN : WSDL;
```

Figure 8.43: ATL WSDLtoXML transform module declaration.

- PLTPortTypeToElement

- ServiceToElement

The first transformation, defined in Figure 8.44, transforms a WSDL Definition artifact to an XML Root artifact along with a number of nested XML Element and Attribute artifacts. There is only one Definition artifact per WSDL model. The transformation implements the relation illustrated in Figure 7.47.

The second transformation, defined in Figure 8.45, transforms a WSDL Namespace artifact to an XML Attribute artifact. There can be many Namespace artifacts per WSDL Definition. The transformation implements the relation illustrated in Figure 7.48.

The third transformation, defined in Figure 8.46, transforms a WSDL Message artifact to an XML Element artifact, and subsequently to a child XML Attribute artifact. There can be many Message artifacts per WSDL Definition. The transformation implements the relation illustrated in Figure 7.49.

The fourth transformation, defined in Figure 8.47, transforms a WSDL Part artifact to an XML Element artifact, and subsequently to a number of child XML Attribute artifacts.

```
--Transform each WSDL Definition type to an XML Root type
rule DefinitionToRoot {
  from
    d : WSDL!Definition
  to
    rt : XML!Root(name <- 'wsdl:definitions',children <- tn,children <- pltns,children <- bpwsns
      ,children <- xsdns,children <- wsdl,children <- wsdlsoap,
      children <- d.eNamespaces->collect(e|thisModule.NamespaceToAttribute(e)),
      children <- d.eMessages->collect(e|thisModule.MessageToElement(e)),
      children <- d.ePortTypes->collect(e|thisModule.PortTypeToElement(e)),
      children <- d.ePartnerLinkTypes->collect(e|thisModule.PLTToElement(e)),
      children <- d.eServices->collect(e|thisModule.ServiceToElement(e))
    ),
    tn :XML!Attribute (name <- 'targetNamespace',value <- d.targetNamespace),
    pltns :XML!Attribute (name <- 'xmlns:plt',
      value <- 'http://schemas.xmlsoap.org/ws/2003/05/partner-link/'),
    bpwsns :XML!Attribute (name <- 'xmlns:bpws',
      value <- 'http://schemas.xmlsoap.org/ws/2003/03/business-process/'),
    wsdl :XML!Attribute (name <- 'xmlns:wsdl',
      value <- 'http://schemas.xmlsoap.org/wsdl/'),
    wsdlsoap :XML!Attribute (name <- 'xmlns:wsdlsoap',
      value <- 'http://schemas.xmlsoap.org/wsdl/soap/'),
    xsdns :XML!Attribute (name <- 'xmlns:xsd',
      value <- 'http://www.w3.org/2001/XMLSchema')
}
```

Figure 8.44: ATL DefinitionToRoot transformation definition.

```
--Transform (when called) each WSDL Namespace type to an XML Attribute type
lazy rule NamespaceToAttribute{
  from
    ns: WSDL!Namespace
  to
    att : XML!Attribute(name <- 'xmlns:'+ns.prefix,value <- ns.URI)
}
```

Figure 8.45: ATL NamespaceToAttribute transformation definition.

```
--Transform (when called) each WSDL Message type to an XML Element type
lazy rule MessageToElement{
  from
    ms: WSDL!Message
  to
    ele : XML!Element(name <- 'wsdl:message',children <- nme
      ,children <- ms.eParts->collect(e|thisModule.PartToElement(e))),
    nme : XML!Attribute(name <- 'name',value <- ms.qName)
}
```

Figure 8.46: ATL MessageToElement transformation definition.

There can be many Part artifacts per WSDL Message. The transformation implements the relation illustrated in Figure 7.50.

```
--Transform (when called) each WSDL Part type to an XML Element type
lazy rule PartToElement{
  from
    pt : WSDL!Part
  to
    ele : XML!Element(name <- 'wsdl:part',children <- name,children <- type).
    name : XML!Attribute(name <- 'name',value <- pt.name),
    type : XML!Attribute(name <- 'type',value <- pt.typeName)
}
```

Figure 8.47: ATL PartToElement transformation definition.

The fifth transformation, defined in Figure 8.48, transforms a WSDL PortType artifact to an XML Element artifact, and subsequently to a child XML Attribute artifact. There can be many PortType artifacts per WSDL Definition. The transformation implements the relation illustrated in Figure 7.51.

```
--Transform (when called) each WSDL PortType type to an XML Element type
lazy rule PortTypeToElement{
  from
    pt: WSDL!PortType
  to
    ele : XML!Element(name <- 'wsdl:portType',children <- name,
      children <- pt.eOperations->collect(e|thisModule.OperationToElement(e))),
    name : XML!Attribute(name <- 'name',value <- pt.qName)
}
```

Figure 8.48: ATL PortTypeToElement transformation definition.

The sixth transformation, defined in Figure 8.49, transforms a WSDL Operation artifact to an XML Element artifact, and subsequently to a child XML Attribute artifact. There can be many Operation artifacts per WSDL PortType. The transformation implements the relation illustrated in Figure 7.52.

The seventh transformation, defined in Figure 8.50, transforms a WSDL Input artifact

```
--Transform (when called) each WSDL Operation type to an XML Element type
lazy rule OperationToElement{
  from
    op : WSDL!Operation
  to
    ele : XML!Element(name <- 'wsdl:operation',children <- name,
      children <- Sequence{op.eInput}->collect(e|thisModule.InputToElement(e)),
      children <- Sequence{op.eOutput}->collect(e|thisModule.OutputToElement(e))).
    name : XML!Attribute(name <- 'name',value <- op.name)
}
```

Figure 8.49: ATL OperationToElement transformation definition.

```
--Transform (when called) each WSDL Input type to an XML Element type
lazy rule InputToElement{
  from
    inp: WSDL!Input
  to
    ele : XML!Element(name <- 'wsdl:input',children <- name),
    name : XML!Attribute(name <- 'message',value <- inp.eMessage.qName)
}
```

Figure 8.50: ATL InputToElement transformation definition.

to an XML Element artifact, and subsequently to a child XML Attribute artifact. There can be many Input artifacts per WSDL Operation. The transformation implements the relation illustrated in Figure 7.53.

The eighth transformation, defined in Figure 8.51, transforms a WSDL Input artifact to an XML Element artifact, and subsequently to a child XML Attribute artifact. There can be many Output artifacts per WSDL Operation. The transformation implements the relation illustrated in Figure 7.54.

```
--Transform (when called) each WSDL Output type to an XML Element type
lazy rule OutputToElement{
  from
    out: WSDL!Output
  to
    ele : XML!Element(name <- 'wsdl:output',children <- name),
    name : XML!Attribute(name <- 'message', value <- out.eMessage.qName)
}
```

Figure 8.51: ATL OutputToElement transformation definition.

The ninth transformation, defined in Figure 8.52, transforms a WSDL PartnerLinkType artifact to an XML Element artifact, and subsequently to a child XML Attribute artifact. There can be many PartnerLinkType artifacts per WSDL Definition. The transformation implements the relation illustrated in Figure 7.55.

```
--Transform (when called) each WSDL PartnerLinkType type to an XML Element type
lazy rule PLTToElement{
  from
    plt: WSDL!PartnerLinkType
  to
    ele : XML!Element(name <- 'plt:partnerLinkType',children <- nme,
      children <- Sequence{plt.role}->collect(e|thisModule.RoleToElement(e))),
    nme : XML!Attribute(name <- 'name',value <- plt.name)
}
```

Figure 8.52: ATL PLTToElement transformation definition.

The tenth transformation, defined in Figure 8.53, transforms a WSDL Role artifact to an XML Element artifact, and subsequently to a child XML Attribute artifact. There is only one Role artifact per WSDL PartnerLinkType. The transformation implements the relation illustrated in Figure 7.56.

```
--Transform (when called) each WSDL Role type to an XML Element type
lazy rule RoleToElement{
  from
    rle: WSDL!Role
  to
    ele : XML!Element(name <- 'plt:role',children <- name,
      children <- Sequence{rle.portType}->collect(e|thisModule.PLTPortTypeToElement(e))),
    name : XML!Attribute(name <- 'name',value <- rle.name)
}
```

Figure 8.53: ATL RoleToElement transformation definition.

The eleventh transformation, defined in Figure 8.54, transforms a WSDL PortType artifact to an XML Element artifact, and subsequently to a child XML Attribute artifact. There is only one PortType artifact per WSDL Role. The transformation implements the relation illustrated in Figure 7.57.

```
--Transform (when called) each WSDL PortType type to an XML Element type
lazy rule PLTPortTypeToElement{
  from
    pt: WSDL!PortType
  to
    ele : XML!Element(name <- 'plt:portType',children <- name),
    name : XML!Attribute(name <- 'name',value <- pt.qName)
}
```

Figure 8.54: ATL PLTPortTypeToElement transformation definition.

The twelfth transformation, defined in Figure 8.55, transforms a WSDL PortType artifact to an XML Element artifact, and subsequently to a child XML Attribute artifact. There is only one Service artifact per WSDL Definition. The transformation implements the relation illustrated in Figure 7.58.

```
--Transform (when called) each WSDL Service type to an XML Element type
lazy rule ServiceToElement{
  from
    s: WSDL!Service
  to
    ele : XML!Element(name <- 'wsdl:service',children <- nme),
    nme : XML!Attribute(name <- 'name',value <- s.qName)
}
```

Figure 8.55: ATL ServiceToElement transformation definition.

8.4.8 Transforming Deployment Descriptor Model to XML Model

The eighth transformation set is from a Deployment Descriptor model to an XML model, where the PDD model and the XML model are the candidate models. These transformations are uni-directional as the transformation uses the source model to generate the target model, but does not define rules for transforming the target to the source model. We assume the Deployment Descriptor model is targeted to the ActiveBPEL deployment environment, using the PDD deployment language. This transformation set specifies how a PDD model is transformed to an XML based model, which can be serialised to PDD compliant XML text. This transformation set is pattern independent as the target model artifacts to be created do not depend on the chosen distribution pattern.

The ATL module transformation header declares the name for the transformational set and declares two models, the source model PDD and the target model XML. The module is expressed in Figure 8.56.

```
module PDDtoXML; -- Module Template
create OUT : XML from IN : PDD;
```

Figure 8.56: ATL PDDtoXML transform module declaration.

The five transformations in this set are outlined as follows, and described in more detail below.

- ProcessToRoot
- WSDLToElement
- PartnerLinkToElement
- MyRoleToElement
- PartnerRoleToElement

The first transformation, defined in Figure 8.57, transforms a PDD Process artifact to an XML Root artifact well as to a number of nested XML Element and Attribute artifacts.

There is only one Process artifact per PDD model. The transformation implements the relation illustrated in Figure 7.60.

```
--Transform the PDD Process type to an XML Root type
rule ProcesstoRoot{
  from
    p: PDD!Process
  to
    rt : XML!Root(name <- 'process',children <- name,children <- loc,children <- xns,
      children <- xwsns,children <- bpns,children <- pls,children <-wsr),
    name : XML!Attribute(name <- 'name',value <- 'bpelns:'+p.name),
    loc : XML!Attribute(name <- 'location',value <- p.location),
    xns : XML!Attribute(name <- 'xmlns',
      value <- 'http://schemas.active-endpoints.com/pdd/2004/09/pdd.xsd'),
    xwsns : XML!Attribute(name <- 'xmlns:wsa',
      value <- 'http://schemas.xmlsoap.org/ws/2003/03/addressing'),
    bpns : XML!Attribute(name <- 'xmlns:bpelns',value <- p.namespace),
    pls : XML!Element(name <- 'partnerLinks',children <- thisModule.getAllPartnerLinks()),
    wsr : XML!Element(name <- 'wsdlReferences',children <- thisModule.getAllWSDLReferences())
}
```

Figure 8.57: ATL ProcesstoRoot transformation definition.

The second transformation, defined in Figure 8.58, transforms a PDD WSDL artifact to an XML Element artifact along with a number of nested XML Attribute artifacts. There can be many WSDL artifacts per PDD Definition. The transformation implements the relation illustrated in Figure 7.61.

The third transformation, defined in Figure 8.59, transforms a PDD PartnerLink artifact to an XML Element artifact as well as to a number of nested XML Attribute and Element artifacts. There can be many PartnerLink artifacts per PDD Definition. An imperative conditional block is used to control the creation of target model artifacts, depending on whether source model artifacts MyRole and PartnerRole exist. The transformation implements the relation illustrated in Figure 7.62.

The fourth transformation, defined in Figure 8.60, transforms a PDD MyRole artifact to an XML Element artifact along with a number of nested XML Attribute artifacts. There is

```
--Transform a PDD WSDL type to an XML Element type
rule WSDLtoElement{
  from
    wd: PDD!WSDL
  to
    wsr : XML!Element(name <- 'wsdl',children <- ns,children <- loc),
    ns : XML!Attribute(name <- 'namespace',value <- wd.namespace),
    loc : XML!Attribute(name <- 'location',value <- wd.location)
}
```

Figure 8.58: ATL WSDLtoElement transformation definition.

```
--Transform a PDD PartnerLink type to an XML Element type
rule ParterLinktoElement{
  from
    pl: PDD!PartnerLink
  to
    ele : XML!Element(name <- 'partnerLink ',children <- name),
    name : XML!Attribute(name <- 'name',value <- pl.name),
    myrl : XML!Element(name <- 'myRole'),
    pr : XML!Element(name <- 'partnerRole ')
    do --imperative block
    {
      --some attributes are not mandatory, we must check to see if they
      --exist before trying to output them
      if(pl.myRole.oclIsUndefined() <> true)
      {
        ele.children <- myrl;
        thisModule.MyRoleToElement(pl.myRole,myrl);
      }
      if(pl.partnerRole.oclIsUndefined() <> true)
      {
        ele.children <- pr;
        thisModule.PartnerRoleToElement(pl.partnerRole ,pr);
      }
    }
}
```

Figure 8.59: ATL ParterLinktoElement transformation definition.

only one MyRole artifact per PDD PartnerLink. The transformation implements the relation illustrated in Figure 7.63.

```
--Transform a PDD MyRole type to an XML Element type
--This is a called rule. only use called rules when the transform is only
--decided at runtime based upon the source metamodel element type
rule MyRoleToElement (mr: PDD!MyRole, xrl : XML!Element){
  to
  svr : XML!Attribute(name <- 'service ',value <- mr.service),
  ar : XML!Attribute(name <- 'allowedRoles ',value <- mr.allowedRoles),
  bd : XML!Attribute(name <- 'binding ',value <- mr.binding)
  do{
    xrl.children <- svr;xrl.children <- ar;xrl.children <- bd;
  }
}
```

Figure 8.60: ATL MyRoleToElement transformation definition.

The fifth transformation, defined in Figure 8.61, transforms a PDD PartnerRole artifact to an XML Element artifact as well as to a number of nested XML Attribute, Element and Text artifacts. There is only one PartnerRole artifact per PDD PartnerLink. The transformation implements the relation illustrated in Figure 7.64.

8.4.9 Transforming Deployment Catalog Model to XML Model

The ninth transformation set is from a Deployment Catalog model to an XML model, where the WSDLCatalog model and the XML model are the candidate models. These transformations are uni-directional as the transformation uses the source model to generate the

```
--Transform a PDD PartnerRole type to an XML Element type
--This is a called rule. only use called rules when the transform is only
--decided at runtime based upon the source metamodel element type
rule PartnerRoleToElement (pr: PDD!PartnerRole, xpr: XML!Element){
  to
  er : XML!Attribute(name <- 'endpointReference',value <- pr.endpointReferenceType),
  ere : XML!Element(name <- 'wsa:EndpointReference',children <- ns,children <- ad,children <- sn),
  ns : XML!Attribute(name <- 'xmlns:' + pr.endpointReference.namespace,value <- pr.endpointReference.uri),
  ad : XML!Element(name <- 'wsa:Address',children <- adtxt),
  adtxt : XML!Text(value <- pr.endpointReference.address.text),
  sn : XML!Element(name <- 'wsa:ServiceName',children <- svtxt,children <- pn),
  svtxt : XML!Text(value <- pr.endpointReference.serviceName.text),
  pn : XML!Attribute(name <- 'PortName',value <- pr.endpointReference.serviceName.PortName)
  do{
    xpr.children <- er;xpr.children <- ere;
  }
}
```

Figure 8.61: ATL PartnerRoleToElement transformation definition.

target model, but does not define rules for transforming the target to the source model. We assume the Deployment Catalog model is targeted to the ActiveBPEL deployment environment, using the WSDLCatalog deployment language. This transformation set specifies how a WSDLCatalog model is transformed to an XML based notation, which can be serialised to WSDLCatalog compliant XML text. This transformation set is pattern independent as the target model artifacts to be created do not depend on the chosen distribution pattern.

The ATL module transformation header declares the name for the transformational set and declares two models, the source model WSDLCatalog and the target model XML. The module is expressed in Figure 8.62.

```
module WSDLCatalogtoXML; -- Module Template
create OUT : XML from IN : WSDLCatalog;
```

Figure 8.62: ATL WSDLCatalogtoXML transform module declaration.

The two transformations in this set are outlined as follows, and described in more detail below.

- WSDLCatalogToRoot
- WSDLEntryToElement

The first transformation, defined in Figure 8.63, transforms a WSDLCatalog WSDLCatalog artifact to an XML Root artifact. There is only one WSDLCatalog artifact per WSDLCatalog model. The transformation implements the relation illustrated in Figure

7.66.

```
--Transform a WSDLCatalog WSDLCatalog type to an XML Root type
rule WSDLCatalogToRoot{
  from
    cat: WSDLCatalog!WSDLCatalog
  to
    rt : XML!Root(name <-'wsdlCatalog',children <- thisModule.getWSDLEntries()
  )
}
```

Figure 8.63: ATL WSDLCatalogToRoot transformation definition.

The second transformation, defined in Figure 8.64, transforms a WSDLCatalog WSDLEntry artifact to an XML Element artifact along with a number of nested XML Attribute artifacts. There can be many WSDLEntry artifacts per WSDLCatalog. The transformation implements the relation illustrated in Figure 7.67.

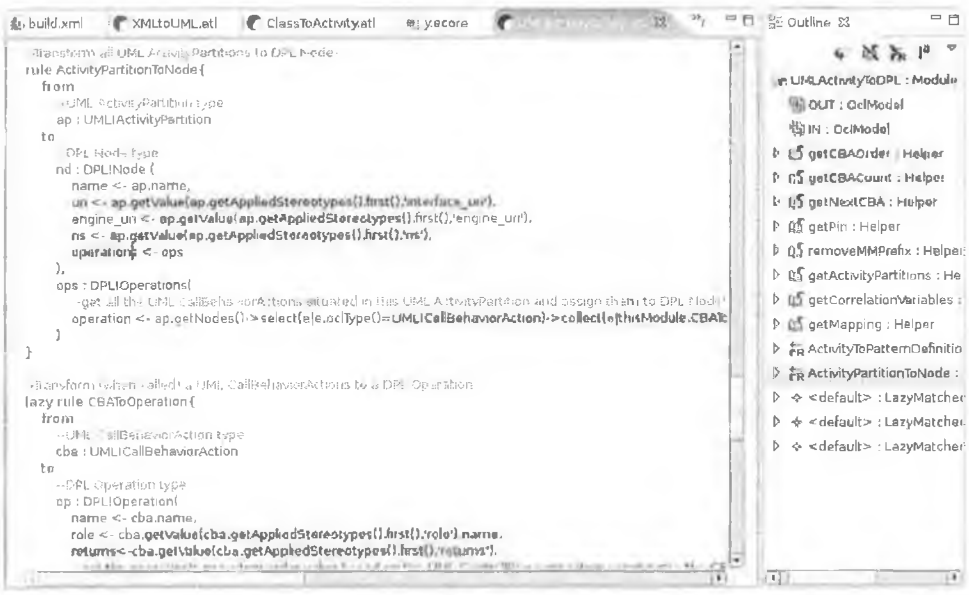
```
--Transform a WSDLCatalog WSDLEntry type to an XML Element type
rule WSDLEntryToElement{
  from
    ent : WSDLCatalog!WSDLEntry
  to
    ele : XML!Element(name <- 'wsdlEntry',children <- cp,children <- ln),
    cp : XML!Attribute(name <- 'classpath',value <- ent.classpath),
    ln : XML!Attribute(name <- 'location',value <- ent.location)
}
```

Figure 8.64: ATL WSDLEntryToElement transformation definition.

8.5 Tool Support

The ATL project provides an Integrated Development Environment (IDE), for the creation of ATL based transformations [8]. The ATL project builds on the open source Eclipse Modeling Framework (EMF), discussed in Section 2.5.7.2. Both EMF and ATL are plugins to the open source Eclipse development platform [64]. Once installed the ATL plugin provides an environment for ATL code syntax highlighting and outlining, a source code debugger and execution support. The execution engine provides ATL to bytecode compilation complete with a virtual machine to interpret the bytecode. Figure 8.65 illustrates the ATL plugin in action.

It should be noted that at the time of writing a QVT based tool, SmartQVT [176], which supports the QVT-Operational language had just been launched. The QVT-Operational lan-



```
--transform all UML ActivityPartitions to DPL Node
rule ActivityPartitionToNode{
   from
      --UML ActivityPartition type
      ap : UML!ActivityPartition
   to
      --DPL Node type
      nd : DPL!Node (
         name <- ap.name,
         uri <- ap.getValue(ap.getAppliedStereotypes().first(),'interface_uri'),
         engine_uri <- ap.getValue(ap.getAppliedStereotypes().first(),'engine_uri'),
         ns <- ap.getValue(ap.getAppliedStereotypes().first(),'ns'),
         operations <- ops
      ),
      ops : DPL!Operations(
         --get all the UML CallBehaviorActions situated in this UML ActivityPartition and assign them to DPL Node
         operation <- ap.getNodes()->select(e|e.oclType()=UML!CallBehaviorAction)->collect(e|thisModule.CBAt
      )
}

--transform (when called) a UML CallBehaviorActions to a DPL Operation
lazy rule CBAToOperation{
   from
      --UML CallBehaviorAction type
      cba : UML!CallBehaviorAction
   to
      --DPL Operation type
      op : DPL!Operation(
         name <- cba.name,
         role <- cba.getValue(cba.getAppliedStereotypes().first(),'role').name,
         returns<-cba.getValue(cba.getAppliedStereotypes().first(),'returns'),
```

```
UMLActivityToDPL : Module
   OUT : OclModel
   IN : OclModel
 getCBAOrder : Helper
 getCBACount : Helper
 getNextCBA : Helper
 getPin : Helper
 removeMMPrefix : Helper
 getActivityPartitions : He
 getCorrelationVariables :
 getMapping : Helper
 ActivityToPatternDefinitio
 ActivityPartitionToNode :
 <default> : LazyMatche
 <default> : LazyMatche
 <default> : LazyMatcher
```

Figure 8.65: ATL code environment in Eclipse.

guage can be used, like ATL, for the declarative and imperative description of a transformation. The tool, like the ATL tool, builds on the open source Eclipse Modeling Framework (EMF), and is an Eclipse based plugin. However, the tool is as yet untested on a large scale to be considered stable. For this reason we continued to use the well tested ATL tool.

8.6 Summary

In this chapter we have presented the fourth component in our modeling and transformation framework, model transformations. We have discussed how model transformations, defined using ATL, implement the model relations defined in Chapter 7. These transformations, when executed, create a new model based upon a previously defined model, where the two candidate models have different notations. We have outlined nine transformation sets that, in combination, transform a UML distribution pattern model to an XML based executable system. We have rigorously and systematically defined the transformations in this chapter by directly implementing the relations in Chapter 7 as transformations. Each transformation is an implementation of a previously defined relation. This process helps ensure the consistency and correctness of our modeling approach. It should be noted that the transformations outlined in this chapter are pattern specific. The transformations outlined here are for a centralised shared hub distribution pattern. Different distribution patterns require different numbers of output artifacts for executable system generation. For example, a decentralised distribution pattern would require an interface for each compositional participant. These variations also include small differences in the output artifacts themselves. Specifically, a decentralised distribution pattern would require the passing of state between participants in a composition.

Chapter 9

Methodological Framework and Case Study

9.1 Introduction

This chapter presents the fifth and final component of our modeling and transformation framework, our methodological framework. The goal of the methodological framework is to detail the modeling activities, which ensure that non-functional attribute quality control is no longer an afterthought of the Web service composition generation process. This is achieved by using distribution pattern models within the methodological framework, which consider non-functional quality attributes as the driver for the executable system generation effort. These distribution pattern based models guide our code generation effort based upon previous experience of systems expressing a given distribution pattern, documented in our pattern catalog, which achieves certain non-functional QoS properties. The distribution pattern catalog discussed in Chapter 5 is used to assist the software architect when modeling the Web service composition. Three other components of our framework, the notations, relations and transformations, enable the transformation of the distribution pattern model to an executable system.

In Section 9.2, we introduce our methodology before enumerating its five steps in the subsequent subsections. Each step in the methodological framework is exemplified by a banking case study and details of the tool support, termed TOPMAN, we provide to automate the step.

9.2 The Methodology

Our approach to distribution pattern modeling and subsequent Web service composition generation consists of five steps, which are illustrated in Figure 9.1, and subsequently described below. This approach is based on the Model Driven Software Development (MDSD) approach, where models are used to assist in the generation and reasoning of software systems [182, 30]. Our methodology commences with a number of discrete Web service interfaces and terminates with an executable composition of the discrete Web services, exhibiting the QoS attributes of a chosen distribution pattern. In the following subsections we present our five step methodology, which provides guidelines and support for compositional modeling with distribution patterns. The five steps, or activities, are as follows.

- Step 1 - Transform Interfaces To UML Model(s)
- Step 2 - Distribution Pattern Definition
- Step 3 - Transform UML Activity Diagram Model to DPL Model
- Step 4 - Validate DPL Model
- Step 5 - Transform DPL Model to Executable System

The methodological framework is supported by three specific techniques, listed below, illustrated in Figure 9.2, and elaborated in the five specific steps that follow. These techniques are motivated by our use case, first outlined in Section 4.3.5, and supported by our tool implementation, TOPMAN (TOPology MANager), outlined in the steps below.

- UML Activity diagram/Profile extension (step 1,2)

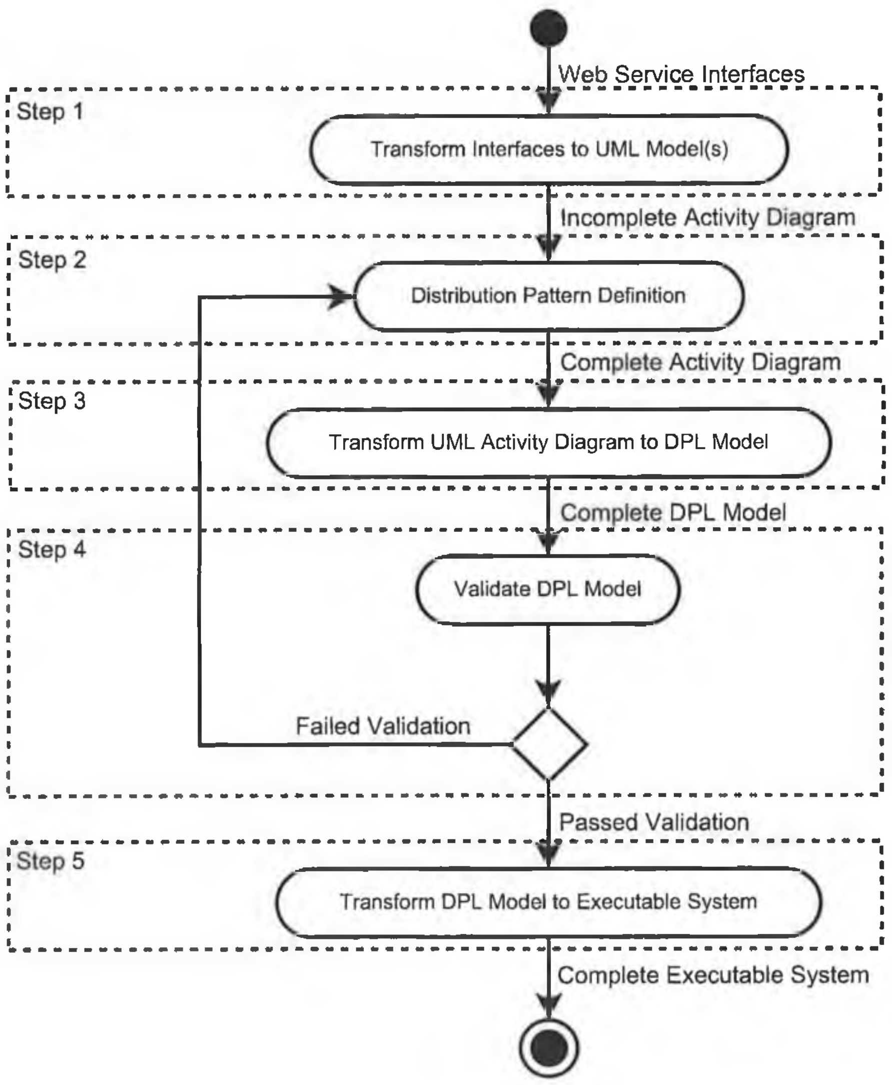

Web Service Interfaces
Step 1
Transform Interfaces to UML Model(s)
Incomplete Activity Diagram
Step 2
Distribution Pattern Definition
Complete Activity Diagram
Step 3
Transform UML Activity Diagram to DPL Model
Complete DPL Model
Step 4
Validate DPL Model
Failed Validation
Passed Validation
Step 5
Transform DPL Model to Executable System
Complete Executable System

Figure 9.1: UML Activity diagram of the methodological framework.

- DPL generator/DPL validator (step 3,4)
- Generators (step 1,3,5)

The methodological framework is accompanied by a small scale case study that motivates our technique. The case study is a banking system with three interacting business processes. We choose a banking system as it is susceptible to changes in organisational structure while requiring stringent controls over data management, two important criteria when choosing a distribution pattern. The scenario, illustrated in Figure 9.3, involves a

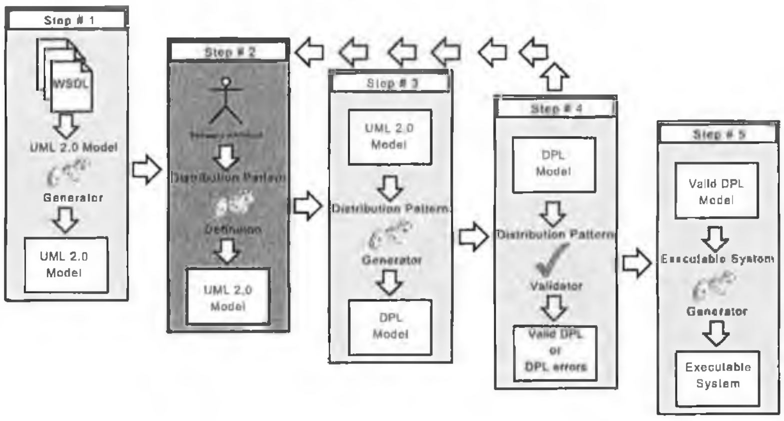


Figure 9.2: Overview of the modeling activities in the methodological framework.

bank customer requesting a credit card. The customer applies to the bank for a credit card, the bank checks the customer's credit rating with a risk assessment agency before passing the credit rating on to a credit card agency for processing. The customer's credit card application is subsequently approved or declined. The case study description is integrated into the five step methodological framework. Each step of the methodology is also accompanied by details of the tool support that we supply.

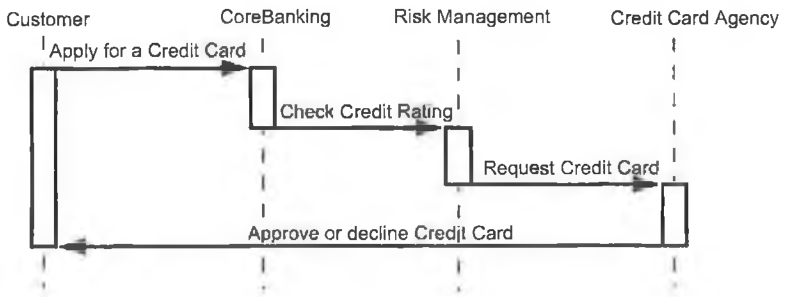


Figure 9.3: UML sequence diagram representing case study.

9.2.1 Step 1 - Transform Interfaces To UML Model(s)

The initial step in the methodology is to take a number of Web service interfaces as input and transform them to the UML 2.0 Activity diagram model, using a UML 2.0 model generator. The Activity diagram model generated is logically separated as no composition has yet been defined. A UML Class diagram representing the interfaces is also created at an intermediate step. The step is illustrated in Figure 9.4 and further outlined in the following sections. The majority of this step can be automated using our tool as outlined in Section 9.2.1.6.

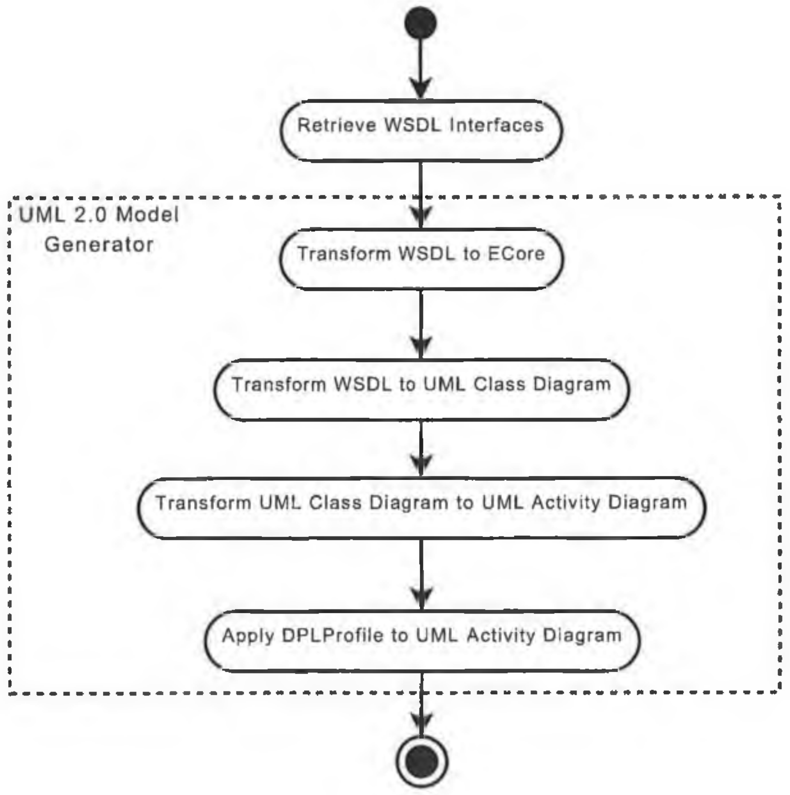

Retrieve WSDL Interfaces
UML 2.0 Model Generator
Transform WSDL to ECore
Transform WSDL to UML Class Diagram
Transform UML Class Diagram to UML Activity Diagram
Apply DPLProfile to UML Activity Diagram

Figure 9.4: UML Activity diagram of transformation from WSDL to UML Activity diagram model.

9.2.1.1 Convert WSDL Interfaces to ECore

A number of WSDL interfaces are retrieved from either a local file-system or from a URI. These interfaces represent the Web service composition participants. These interfaces must be converted to an EMF compatible language, like ECore, so that they can be manipulated by our tool.

9.2.1.2 Transform WSDL to UML Class Diagram Model

The WSDL interfaces represented using the ECore language are transformed, using a generator, into a UML 2.0 Class diagram model. The UML 2.0 Class diagram model is useful for documenting the system because the diagram clearly expresses the discrete interfaces of the Web services to be composed. Class diagram models in this context express the static structures of interfaces in the system. The transformation from WSDL to UML is possible as both WSDL and UML have well defined structures i.e. the WSDL specification [184] and the UML 2.0 specification [140].

9.2.1.3 Transform UML Class Diagram Model to UML Activity Diagram Model

The UML Class diagram model, generated in the previous step, is transformed, using a generator, into a UML 2.0 Activity diagram model. This transformation is possible as diagrams should conform to the UML 2.0 specification [140]. This approach is also considered by Skogan et al. in [53]. The Activity diagram model generated contains many of the new features of UML 2.0, such as Pins, CallBehaviorActions and ControlFlows [103]. The UML Activity diagram is chosen to model distribution patterns as it provides a number of features that assist in clearly illustrating the distribution pattern, while providing sufficient information to drive the generation of the executable system. Activity diagram models show the sequential flow of actions, which are the basic unit of behaviour within a system, and are typically used to illustrate workflows.

9.2.1.4 Apply DPLProfile to UML Activity Diagram Model

For our UML Activity diagram model to effectively model distribution patterns, we require the model to be more descriptive than the standard UML dialect allows. To this end we use a standard extension mechanism of UML, called a profile [71]. Profiles, discussed in Section 2.5.5, define stereotypes and subsequently tag definitions that extend a number of UML constructs. The profile we utilise, called the DPLProfile, is discussed in detail in Section 6.4.2. Figure 9.5, illustrates an example of a UML Activity diagram model comprising three services each containing one operation, indicative of what is output from this step.

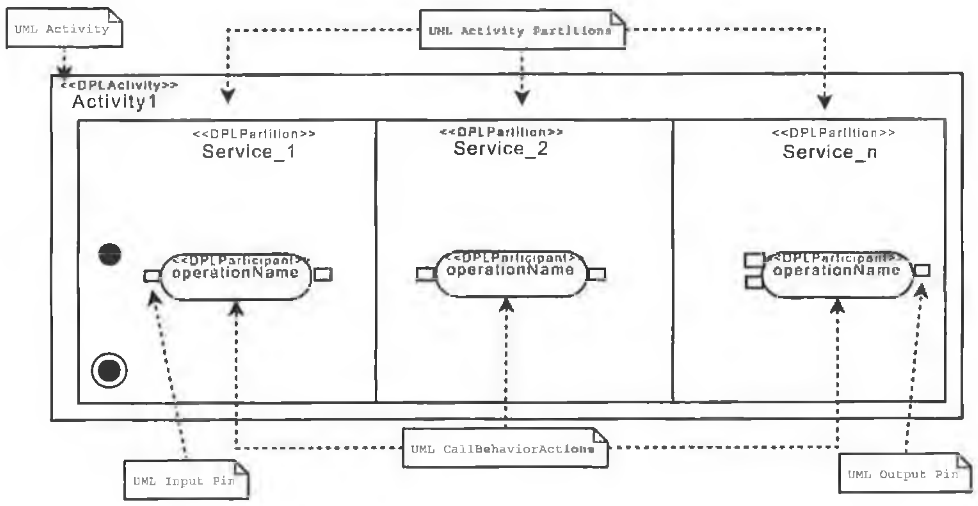


Figure 9.5: Example of a UML Activity diagram with profile applied.

9.2.1.5 Case Study

To illustrate this step we consider our case study scenario. The banking case study provides three WSDL interfaces as input to the UML 2.0 model generator. The interfaces are illustrated in Figure 9.6. These interfaces represent the bank (CoreBanking), the risk assessment agency (RiskManagement) and the credit card agency (CreditCard).

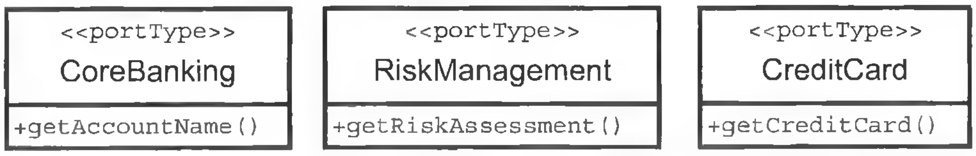


Figure 9.6: Banking case study Web service interfaces as UML Class diagram model.

All three interfaces are represented in a UML Activity diagram model, illustrated in Figure 9.7, albeit without any connections between them. A swim-lane is provided for each interface. Each interface has one operation, represented by a CallBehaviorAction, which is placed in the appropriate swim-lane. The message parts associated with each operation are represented using InputPins and OutputPins. These pins are placed on the appropriate CallBehaviorAction. No model intervention from the software architect is required at this step,

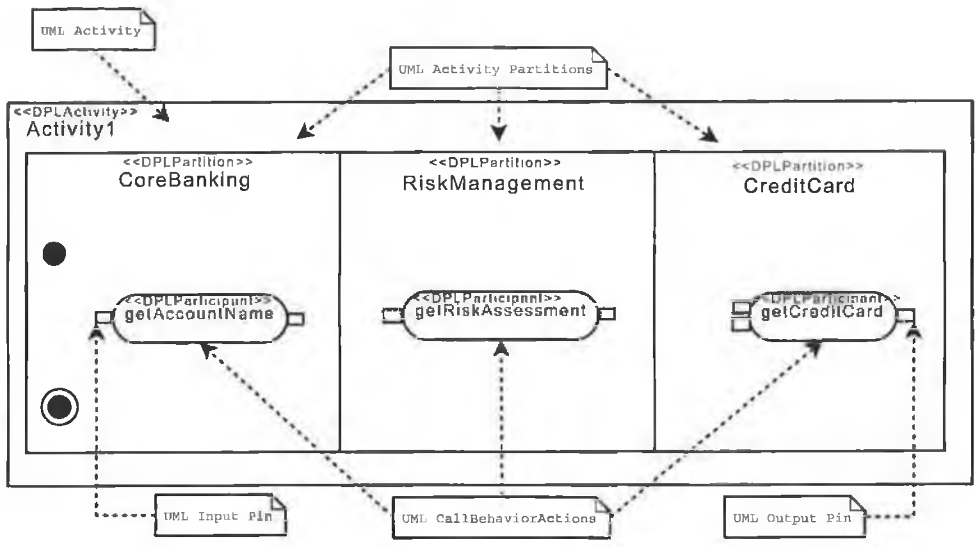


Figure 9.7: UML Activity diagram model output from Step 1.

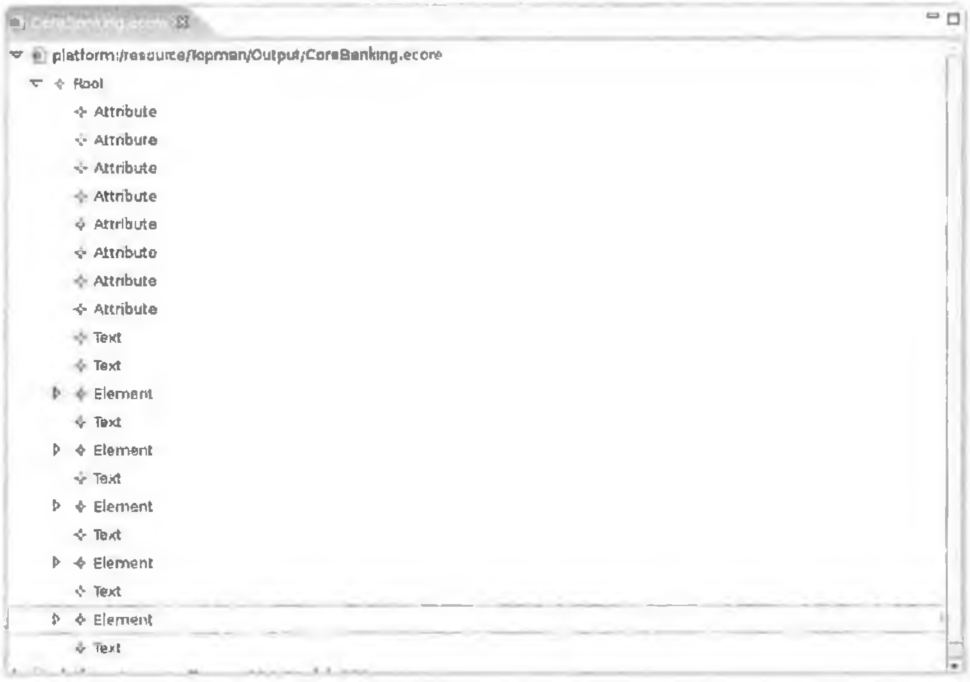
as this conversion from WSDL interfaces to UML Activity diagram model is automated by our tool, as outlined in Section 9.2.1.6 below.

9.2.1.6 Tool Support

The transformation from interfaces to UML models is supported by four specific steps. These four steps can be automated using our tool, as discussed in detail below.

Convert WSDL Interfaces to ECore The three WSDL interfaces passed as input to the UML 2.0 model generator are converted to ECore using the ANT script outlined in Figure C.1 of Appendix C. The ANT script makes use of the AM3 XML injector [10]. This injector converts the text based WSDL interfaces to XML based ECore models. The CoreBanking ECore based model output from the ANT task is illustrated in Figure 9.8.

Transform WSDL to UML Class Diagram Model The three XML based ECore models of the three WSDL interfaces are now converted to a single UML 2.0 Class diagram model, using the ANT script outlined in Figure C.2 of Appendix C. The ANT script takes an XML based model as input and outputs a UML 2.0 based model by executing an ATL



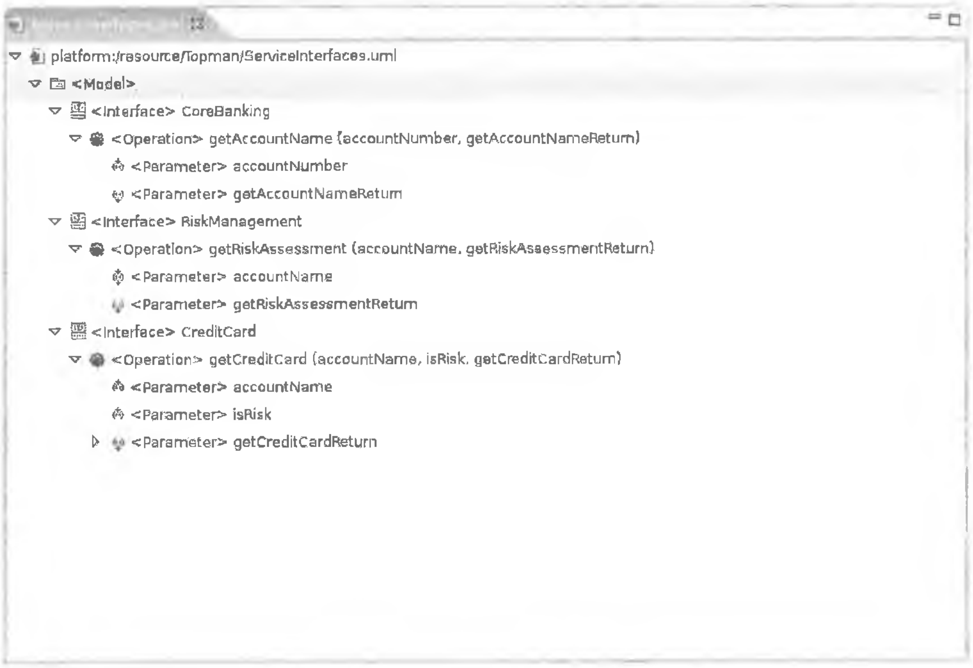
```
CoreBanking.ecore
▽ platform:/resource/lopman/Output/CoreBanking.ecore
  ▽ Root
      Attribute
      Attribute
      Attribute
      Attribute
      Attribute
      Attribute
      Attribute
      Attribute
      Text
      Text
    ▷ Element
      Text
    ▷ Element
      Text
    ▷ Element
      Text
    ▷ Element
      Text
    ▷ Element
      Text
```

Figure 9.8: CoreBanking XML based ECore model of the CoreBanking WSDL interfaces.

transformation, as outlined in Figure B.1 of Appendix B. The output from the ANT task is illustrated in Figure 9.9.

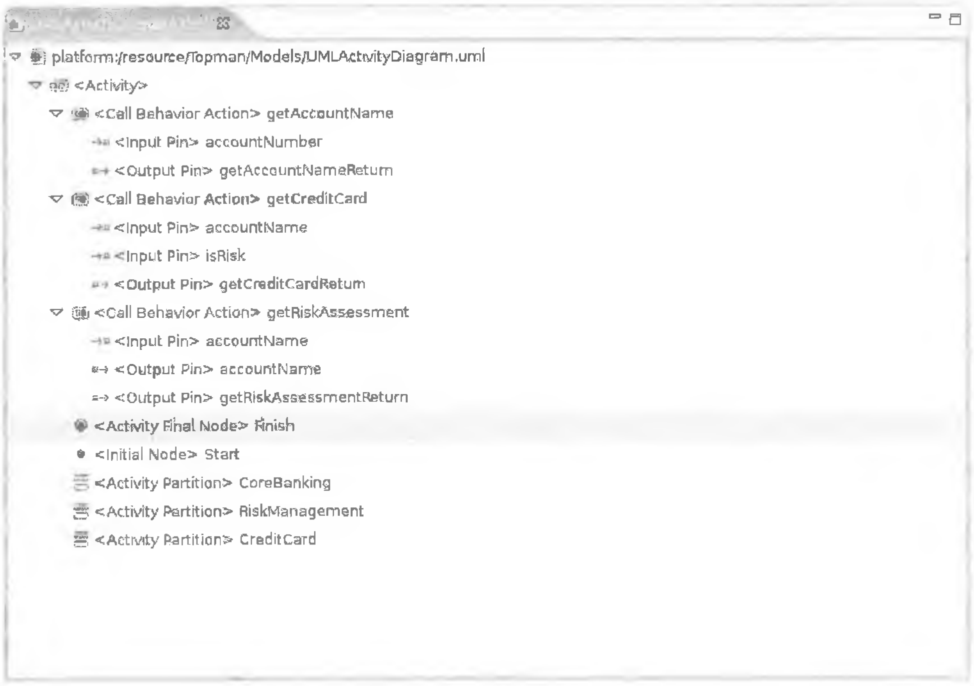
Transform UML Class Diagram Model to UML Activity Diagram Model The single UML Class diagram model representing the three WSDL interfaces is now converted to a single UML 2.0 Activity diagram model, using the ANT script outlined in Figure C.3 of Appendix C. The ANT script takes a UML 2.0 based Class diagram model as input and outputs a UML 2.0 Activity diagram model by executing an ATL transformation, as outlined in Figure B.2 of Appendix B. The output from the ANT task is illustrated in Figure 9.10. It should be noted that the UML InputPins and OutputPins in Figure 9.10 have been manually added using the Eclipse UML2 editor because the ATL script does not currently support the creation of Pins. The script can be enhanced to automate this effort.

Apply DPLProfile to UML Activity Diagram Model We now apply the DPLProfile to the UML Activity diagram just created, to enable the model to be decorated with extra



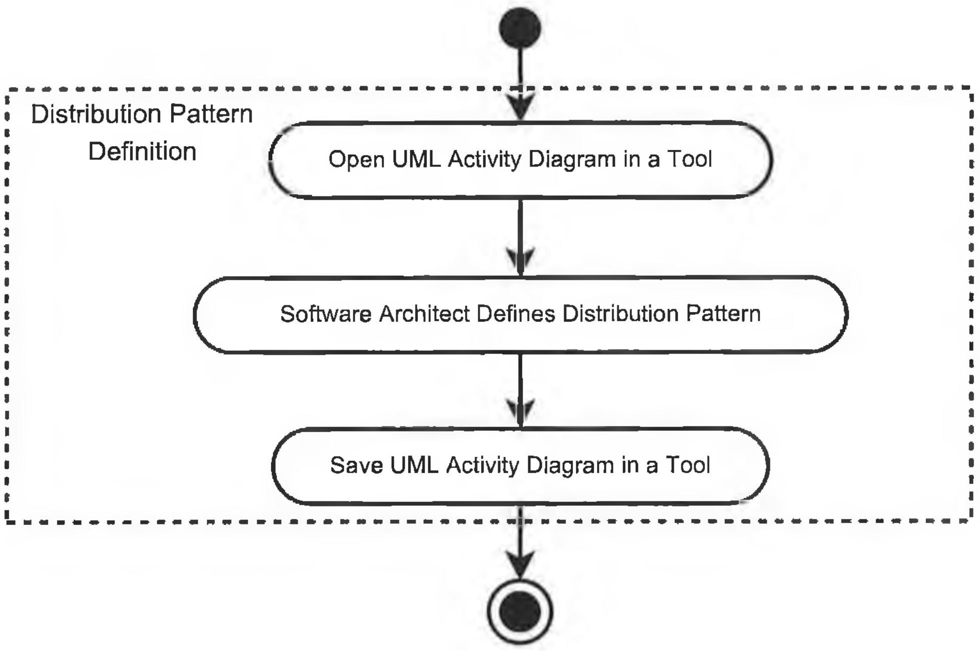
```
▽ platform:/resource/Topman/ServiceInterfaces.uml
  ▽ <Model>
    ▽ <Interface> CoreBanking
      ▽ <Operation> getAccountName (accountNumber, getAccountNameReturn)
          <Parameter> accountNumber
          <Parameter> getAccountNameReturn
    ▽ <Interface> RiskManagement
      ▽ <Operation> getRiskAssessment (accountName, getRiskAssessmentReturn)
          <Parameter> accountName
          <Parameter> getRiskAssessmentReturn
    ▽ <Interface> CreditCard
      ▽ <Operation> getCreditCard (accountName, isRisk, getCreditCardReturn)
          <Parameter> accountName
          <Parameter> isRisk
        ▷ <Parameter> getCreditCardReturn
```

Figure 9.9: UML 2.0 Class diagram model of the WSDL interfaces.



```
▽ platform:/resource/Topman/Models/UMLActivityDiagram.uml
  ▽ <Activity>
    ▽ <Call Behavior Action> getAccountName
        <Input Pin> accountNumber
        <Output Pin> getAccountNameReturn
    ▽ <Call Behavior Action> getCreditCard
        <Input Pin> accountName
        <Input Pin> isRisk
        <Output Pin> getCreditCardReturn
    ▽ <Call Behavior Action> getRiskAssessment
        <Input Pin> accountName
        <Output Pin> accountName
        <Output Pin> getRiskAssessmentReturn
      <Activity Final Node> Finish
      <Initial Node> Start
      <Activity Partition> CoreBanking
      <Activity Partition> RiskManagement
      <Activity Partition> CreditCard
```

Figure 9.10: UML 2.0 Activity diagram model of the WSDL interfaces.



Distribution Pattern Definition
Open UML Activity Diagram in a Tool
Software Architect Defines Distribution Pattern
Save UML Activity Diagram in a Tool

Figure 9.11: Distribution pattern definition.

distribution pattern specific information. This step can be performed manually using either the IBM's Rational Software Architect [86] or the Eclipse UML2 plug-in [65]. A full step by step guide detailing the application of a profile, such as the DPLProfile, in RSA is provided by Misic in [120], while a full guide to UML2 application is provided by Hussey in [85].

This task can also be performed automatically using an ANT script in combination with an ATL transformation using the latest version of ATL. This process requires the use of ATL Superimposition [69], where a number of ATL modules are layered on top of each other to perform a transformation consisting of a copy and subsequent apply operation. A use case demonstrating the approach is made available as part of the ModelPlex project [38]. We have not integrated this feature into our tool yet because we have developed and tested our tool using an older version of ATL.

9.2.2 Step 2 - Distribution Pattern Definition

The UML Activity diagram model produced in Step 1, requires additional modeling, as illustrated in Figure 9.11. This entire step must be performed manually by a software architect because the WSDL interfaces do not supply enough information to be able to connect up the discrete Web services into a composition. Tool support is provided to assist the software architect, as outlined in Section 9.2.2.5.

9.2.2.1 Open UML Activity Diagram in a Tool

Initially the software architect must open the UML Activity diagram model previously generated in Step 1. A number of tools support editing of such diagrams including IBM's Rational Software Architect [86] and the Eclipse UML2 plug-in [65].

9.2.2.2 Software Architect Defines Distribution Pattern

Having opened the model in a tool the architect must select a distribution pattern from the pattern catalog, based on the non-functional requirements of the composition. The pattern is chosen from an enumeration of available patterns, outlined in Chapter 5. The architect must also set some distribution pattern specific variables on the model, which will be used to generate a distribution pattern model. These variables are outlined in Section 6.4.2.

Based on the chosen distribution pattern, the architect defines the sequence of actions by connecting CallBehaviorActions to one another, using UML ControlFlow connectors. Each CallBehaviorAction is assigned a role and each ControlFlow is assigned an order value to define the composition sequence. The InitialNode and ActivityFinalNode must then be assigned to appropriate ActivityPartitions, and connected appropriately using UML ControlFlow connectors. The architect then connects up the UML InputPins and OutputPins of the model, using UML ObjectFlows connectors, so data can be passed through the composition. Additional constructs deemed necessary can be added by the software architect as appropriate. This entire workflow is illustrated in Figure 9.12.

An example UML Activity diagram model, indicative of what is output from this step,

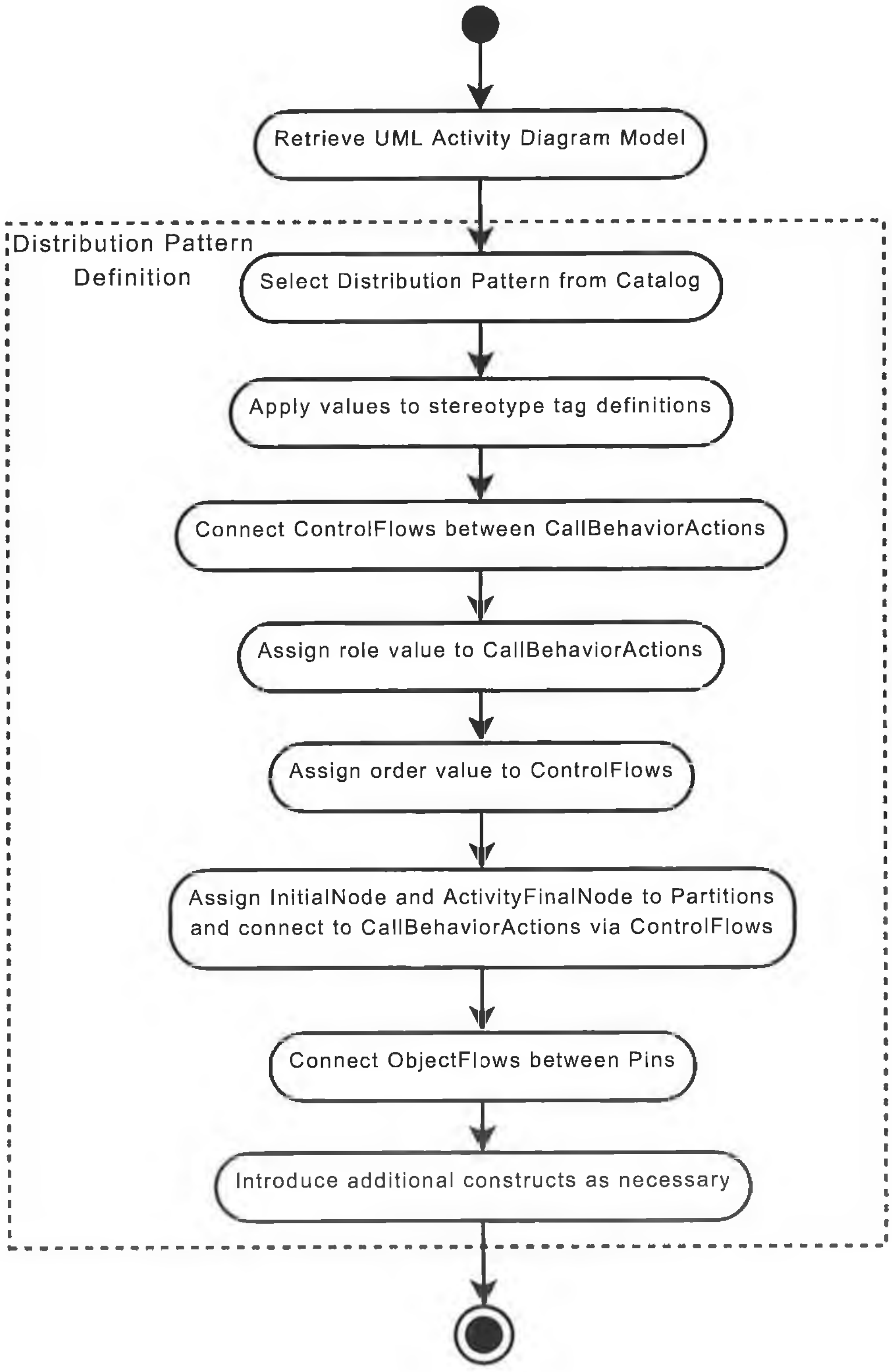

Retrieve UML Activity Diagram Model
Distribution Pattern Definition
Select Distribution Pattern from Catalog
Apply values to stereotype tag definitions
Connect ControlFlows between CallBehaviorActions
Assign role value to CallBehaviorActions
Assign order value to ControlFlows
Assign InitialNode and ActivityFinalNode to Partitions and connect to CallBehaviorActions via ControlFlows
Connect ObjectFlows between Pins
Introduce additional constructs as necessary

Figure 9.12: UML Activity diagram of application of distribution pattern by Software Architect.

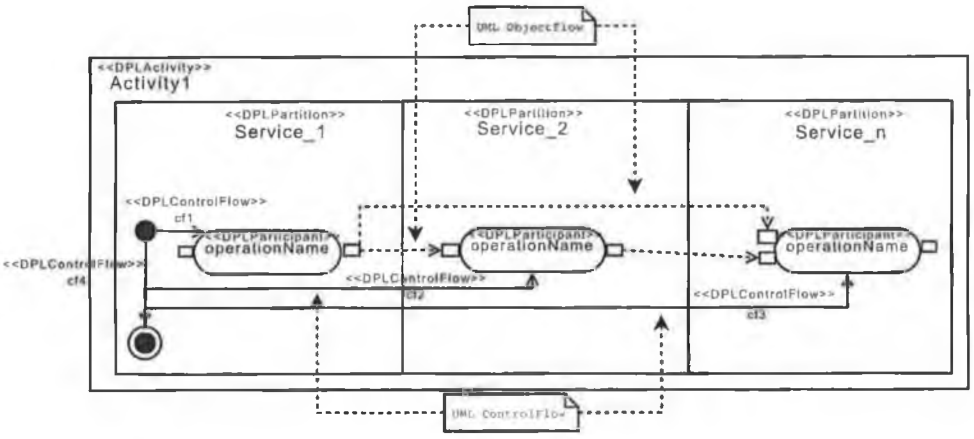


Figure 9.13: Example of a UML Activity diagram model with connections defined.

is illustrated in Figure 9.13. In this illustration we can see the UML ControlFlows connecting the three Web service operations, along with the UML ObjectFlows connecting up the UML InputPins and OutputPins. This particular example illustrates the decentralised shared peer distribution pattern. It should be noted that in Figure 9.13 the initial input for the composition and the final output of the composition are not connected up via their InputPin and OutputPin connectors as our tool automatically works out these connections based upon the chosen distribution pattern.

9.2.2.3 Save UML Activity Diagram in a Tool

Once the software architect has completed the UML Activity diagram model using their tool of choice it should be saved before proceeding to Step 3.

9.2.2.4 Case Study

With regards to our case study scenario the software architect selects the centralised shared hub distribution pattern. We choose this pattern because it should be the most familiar pattern in the catalog to readers, and can most easily illustrate our approach. The software architect after choosing a pattern must then manipulate the UML 2.0 Activity diagram model to appropriately model the chosen pattern across the three Web services. The DPL-Profile values for the case study are outlined in the Tables 9.1 through 9.10 below. These

Table 9.1: Case study values applied to DPLActivity stereotypes attributes.

Attribute	Value
distribution-pattern	hub-and-spoke
collaboration-language	WS-BPEL
service-name	BankingHubService
base-namespace	http://acme.com/wsdl/
namespace-prefix	BankingHub
operation-name	applyForCC

Table 9.2: Case study values applied to getAccountName DPLParticipant stereotypes attributes.

Attribute	Value
role	hub

values can be applied using the tool support discussed in Section 9.2.2.5 below.

All of the DPLMessage stereotype attributes are set to false as this attribute has no effect on centralised distribution patterns.

The case study UML Activity diagram model with distribution pattern applied is illustrated in Figure 9.14 and Figure 9.15. It should be noted that in Figure 9.14 the initial input for the composition and the final output of the composition are not connected up via their InputPin and OutputPin connectors as our tool automatically works out these connections based upon the chosen distribution pattern.

Table 9.3: Case study values applied to getRiskAssessment DPLParticipant stereotypes attributes.

Attribute	Value
role	spoke

Table 9.4: Case study values applied to getCreditCard DPLParticipant stereotypes attributes.

Attribute	Value
role	spoke

Table 9.5: Case study values applied to the first DPLControlFlow stereotypes attributes.

Attribute	Value
order	1

Table 9.6: Case study values applied to the second DPLControlFlow stereotypes attributes.

Attribute	Value
order	2

Table 9.7: Case study values applied to the third DPLControlFlow stereotypes attributes.

Attribute	Value
order	3

Table 9.8: Case study values applied to the CoreBanking DPLParticipant stereotypes attributes.

Attribute	Value
ns	engine_uri1
interface-uri	http://localhost:1234/axis/services/CoreBanking
engine-uri	CoreBanking

Table 9.9: Case study values applied to the RiskManagement DPLParticipant stereotypes attributes.

Attribute	Value
ns	engine_uri2
interface-uri	http://localhost:1234/axis/services/RiskManagement
engine-uri	RiskManagement

Table 9.10: Case study values applied to the CreditCard DPLParticipant stereotypes attributes.

Attribute	Value
ns	engine_uri3
interface-uri	http://localhost:1234/axis/services/CreditCard
engine-uri	CreditCard

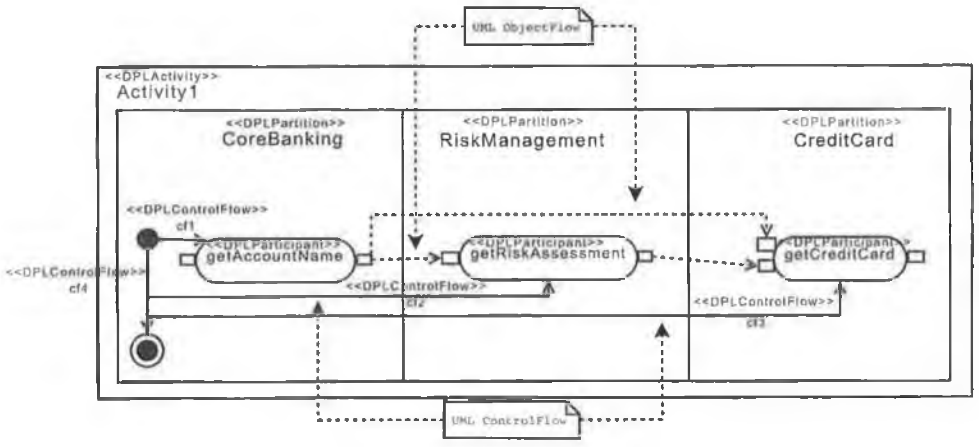
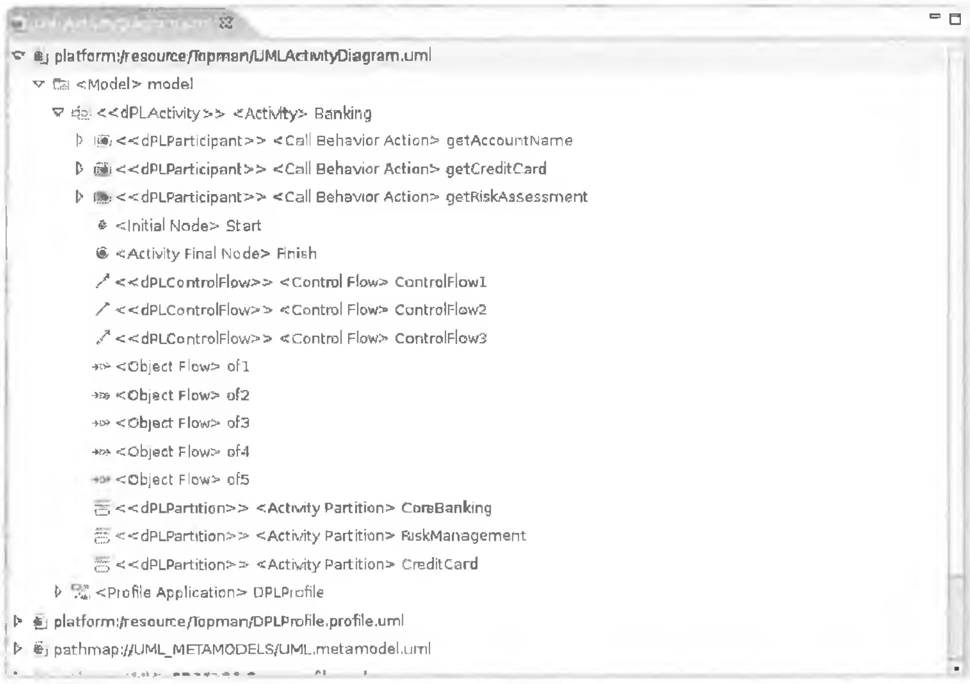


Figure 9.14: UML Activity diagram model output from step 2.



```
platform:/resource/Topman/UMLActivityDiagram.uml
  <Model> model
    <<dPLActivity>> <Activity> Banking
      <<dPLParticipant>> <Call Behavior Action> getAccountName
      <<dPLParticipant>> <Call Behavior Action> getCreditCard
      <<dPLParticipant>> <Call Behavior Action> getRiskAssessment
       <Initial Node> Start
       <Activity Final Node> Finish
       <<dPLControlFlow>> <Control Flow> ControlFlow1
       <<dPLControlFlow>> <Control Flow> ControlFlow2
       <<dPLControlFlow>> <Control Flow> ControlFlow3
       <Object Flow> of1
       <Object Flow> of2
       <Object Flow> of3
       <Object Flow> of4
       <Object Flow> of5
       <<dPLPartition>> <Activity Partition> CoreBanking
       <<dPLPartition>> <Activity Partition> RiskManagement
       <<dPLPartition>> <Activity Partition> CreditCard
    <Profile Application> DPLProfile
platform:/resource/Topman/DPLProfile.profile.uml
pathmap://UML_METAMODELS/UML.metamodel.uml
```

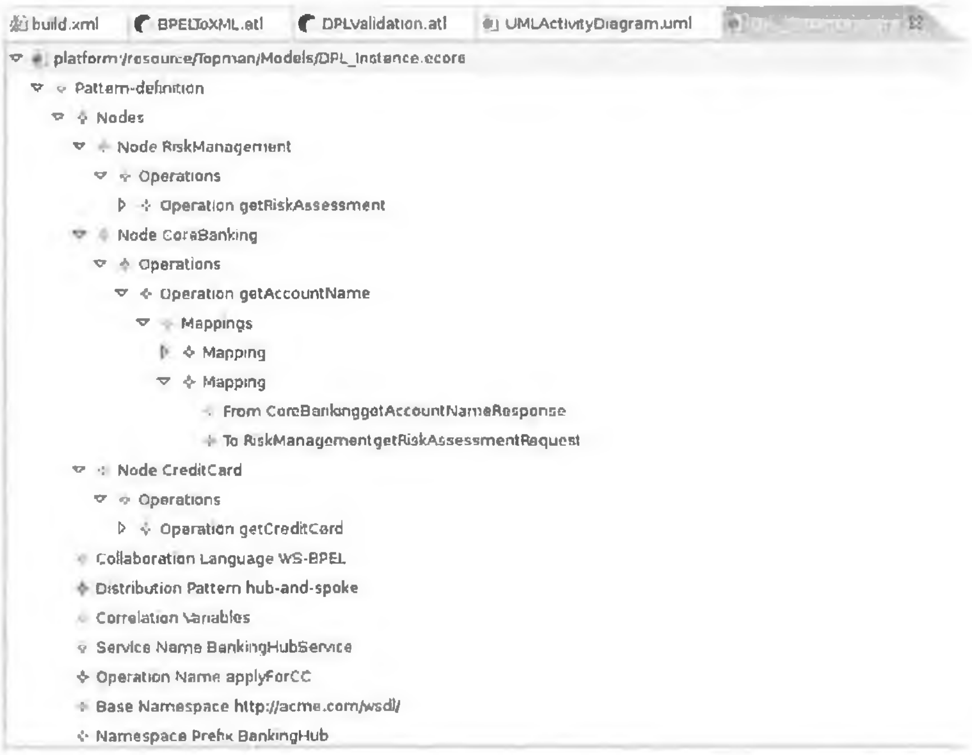
Figure 9.15: UML 2.0 Activity diagram model output from Step 2.

9.2.2.5 Tool Support

Application of the distribution pattern to the model generated in Step 1 is facilitated by a number of tools. Within the course of this work we have considered two such tools, IBM's RSA and the Eclipse UML2 editor. Both tools are discussed in Section 2.5.2. We use the Eclipse UML2 tool to apply the case study DPLProfile values to the UML 2.0 Activity diagram model.

9.2.3 Step 3 - Transform UML Activity Diagram Model to DPL Model

Using the model output from step two as input, the model is transformed to a distribution pattern model, using the distribution pattern generator. This model, expressed using our novel specification language Distribution Pattern Language (DPL), discussed in Section 6.4.3, is called a DPL model. The DPL specification, defined using EMOF, discussed in Section 2.5.7.2, has no reliance on UML and so any number of modeling techniques may be inputted. In fact we envisage that alternative languages such as Architectural Description



```
platform:/resource/Topman/Models/DPL_Instance.ecore
  Pattern-definition
    Nodes
      Node RiskManagement
        Operations
          Operation getRiskAssessment
      Node CoreBanking
        Operations
          Operation getAccountName
            Mappings
              Mapping
              Mapping
                From CoreBankinggetAccountNameResponse
                To RiskManagementgetRiskAssessmentRequest
      Node CreditCard
        Operations
          Operation getCreditCard
    Collaboration Language WS-BPEL
    Distribution Pattern hub-and-spoke
    Correlation Variables
    Service Name BankingHubService
    Operation Name applyForCC
    Base Namespace http://acme.com/wsdl/
    Namespace Prefix BankingHub
```

Figure 9.16: DPL model for case study in Eclipse tool.

Languages and the π calculus, may be used in the future in place of UML as the transformation source. This entire step can be automated using our tool as outlined in Section 9.2.3.2.

9.2.3.1 Case Study

Once again considering the case study scenario, the UML 2.0 Activity diagram model of the three composed Web services output from step two of the methodological framework is converted to a DPL model. This conversion to DPL facilitates the validation of the pattern applied and simplifies the transformations to an executable system. A DPL model for our case study is illustrated in Figure 9.16.

9.2.3.2 Tool Support

The DPL model is created by the DPL generator using an ANT script defined in Figure C.4 of Appendix C. The ANT script executes the transformations outlined in Section 8.4.1.

9.2.4 Step 4 - Validate DPL Model

The DPL model, representing the distribution pattern modeled by the software architect, is verified at this step by the distribution pattern validator. This validation ensures that the values entered in step two are valid. The verification process ensures the distribution pattern selected by the software architect is compatible with the model and applied profile settings. The only valid values for the role and distribution-pattern attributes are the enumerated values as defined in the UML profile enumeration in Figure 6.6. Validation of the distribution pattern model is essential to avoid the generation of an invalid system. The validation process is illustrated in Figure 9.17. This entire step can be automated using our tool as outlined in Section 9.2.4.2.

9.2.4.1 Case Study

Step four considers validation of the generated DPL model. In our case study, because the centralised distribution pattern has been chosen, the validation process must ensure that all the operations on a node have either the hub or spoke role applied. The validation process also checks that there are at least two node operations, which is necessary for a composition. If incorrect values have been entered the architect must correct these values at step two before proceeding to the next step. The validation of the DPL model can be seen in action in Figure 9.18.

9.2.4.2 Tool Support

The DPL model produced by the DPL generator is validated by a DPL validator using the ANT script outlined in Figure C.5 of Appendix C. This script utilises ATL to validate the DPL model, using the ATL transformation detailed in Figure B.3 of Appendix B.

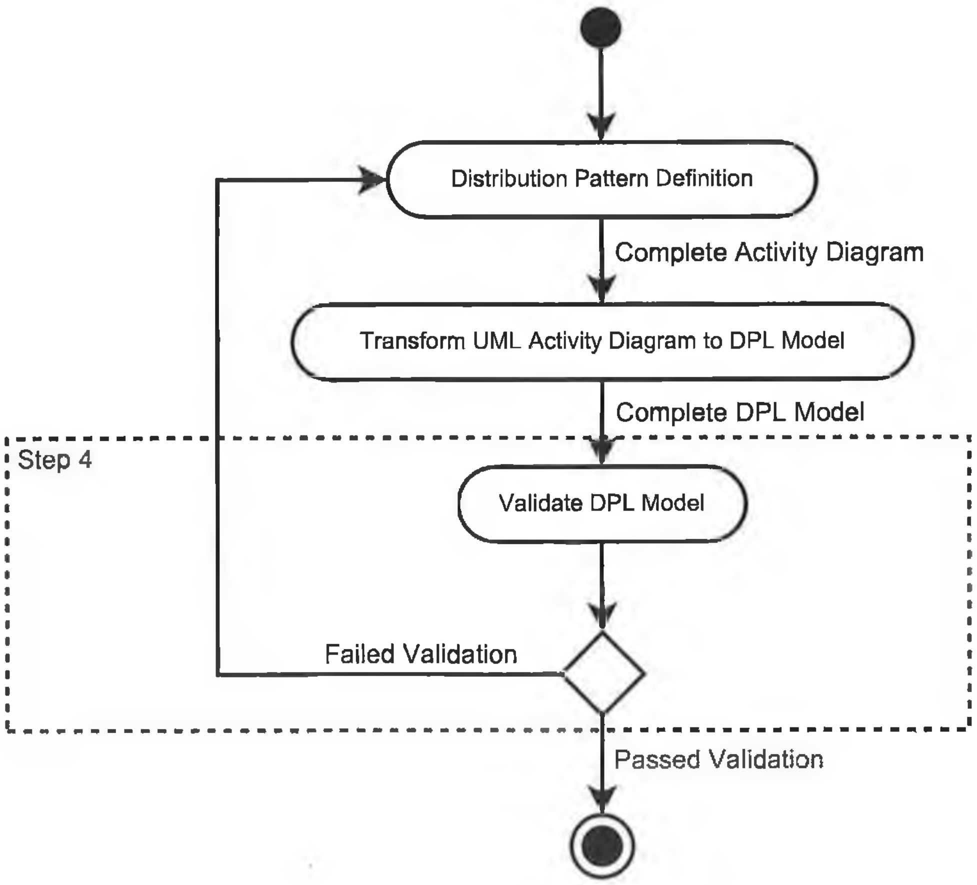
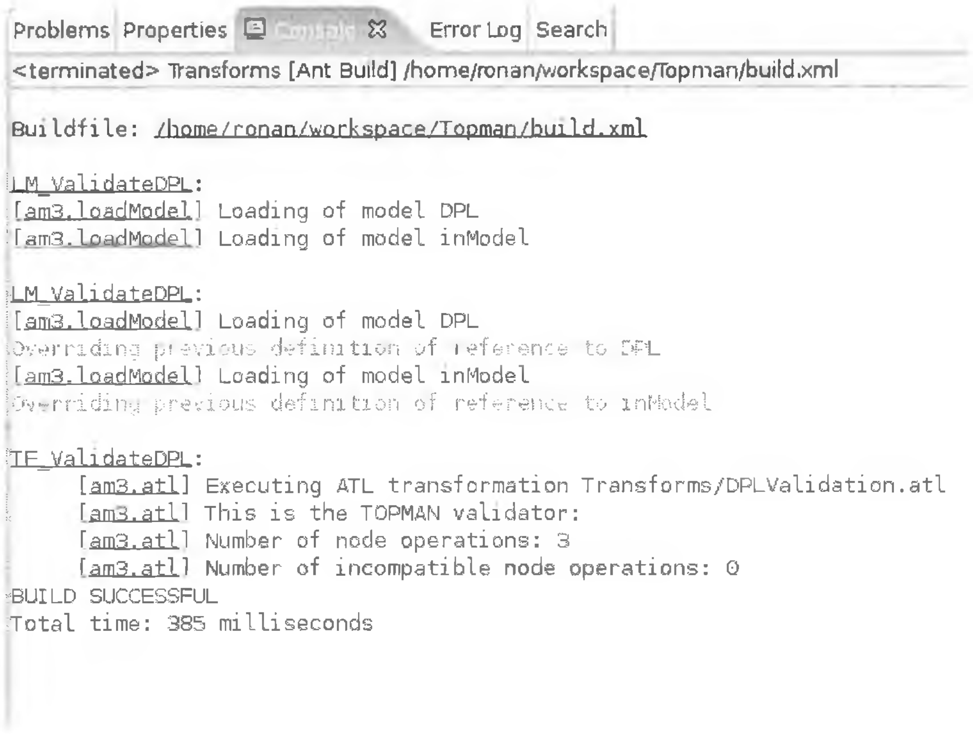

Distribution Pattern Definition
Complete Activity Diagram
Transform UML Activity Diagram to DPL Model
Complete DPL Model
Step 4
Validate DPL Model
Failed Validation
Passed Validation

Figure 9.17: Validation of DPL model.



```
Problems  Properties  Console  Error Log  Search
<terminated> Transforms [Ant Build] /home/ronan/workspace/Topman/build.xml

Buildfile: /home/ronan/workspace/Topman/build.xml

LM_ValidateDPL:
[am3.loadModel] Loading of model DPL
[am3.loadModel] Loading of model inModel

LM_ValidateDPL:
[am3.loadModel] Loading of model DPL
Overriding previous definition of reference to DPL
[am3.loadModel] Loading of model inModel
Overriding previous definition of reference to inModel

TF_ValidateDPL:
     [am3.atl] Executing ATL transformation Transforms/DPLValidation.atl
     [am3.atl] This is the TOPMAN validator:
     [am3.atl] Number of node operations: 3
     [am3.atl] Number of incompatible node operations: 0
BUILD SUCCESSFUL
Total time: 385 milliseconds
```

Figure 9.18: DPL ATL validation script run in Eclipse editor.

9.2.5 Step 5 - Transform DPL Model to Executable System

Finally, the executable system generator takes the validated DPL model and generates all the code and supporting document instances required for a fully executable system. This executable system will realise the Web service composition using the distribution pattern applied by the software architect. All that remains is to deploy the generated artifacts and supporting infrastructure to enable the enactment of the composed system. The step is illustrated in Figure 9.19 and further outlined in the following sections. Again, this entire step can be automated using our tool as outlined in Section 9.2.5.11.

9.2.5.1 Transform DPL Model to WS-BPEL Model

The valid DPL model is transformed to a WS-BPEL model, which represents the collaboration between the participants in the composition.

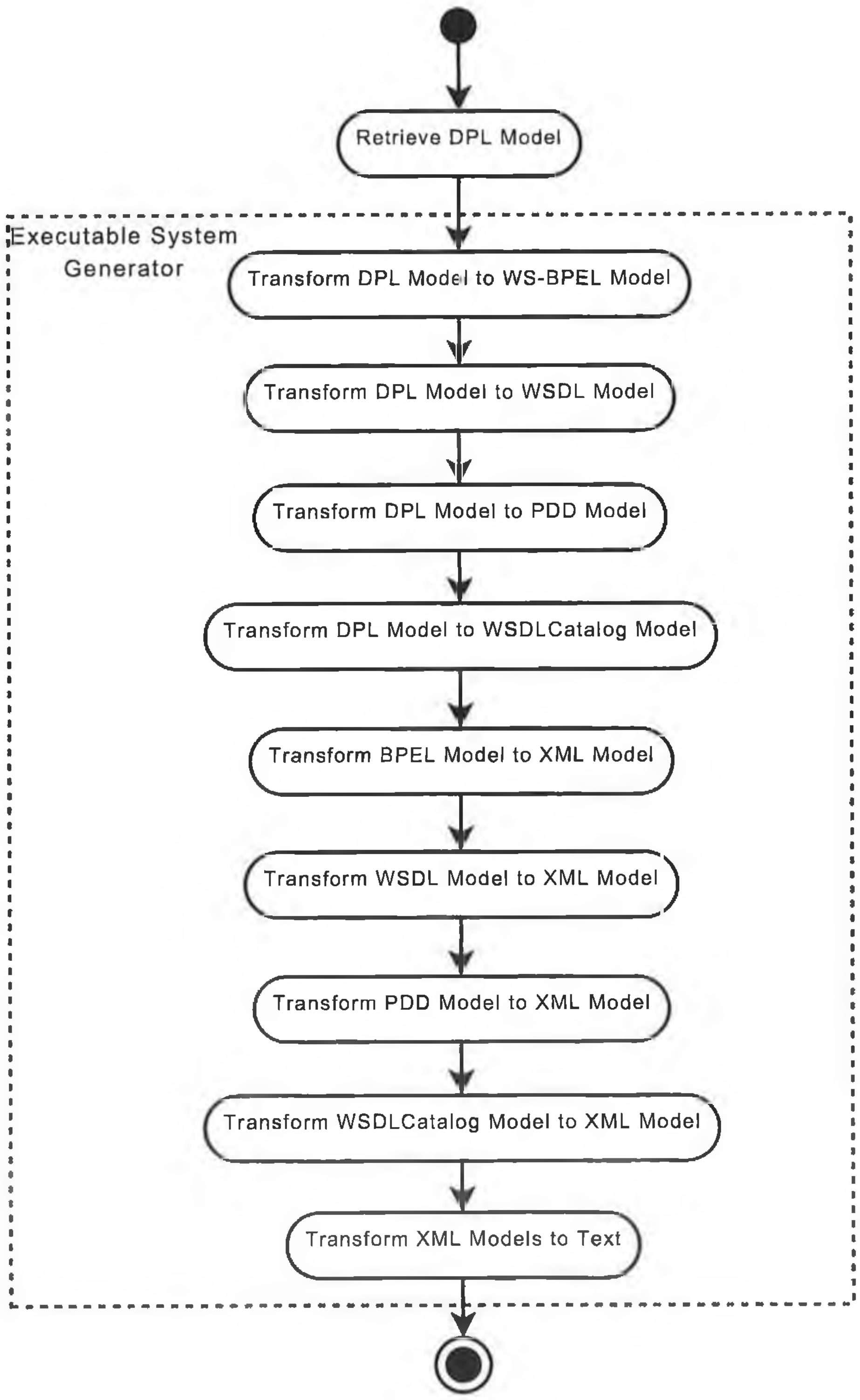

Retrieve DPL Model
Executable System Generator
Transform DPL Model to WS-BPEL Model
Transform DPL Model to WSDL Model
Transform DPL Model to PDD Model
Transform DPL Model to WSDLCatalog Model
Transform BPEL Model to XML Model
Transform WSDL Model to XML Model
Transform PDD Model to XML Model
Transform WSDLCatalog Model to XML Model
Transform XML Models to Text

Figure 9.19: UML Activity diagram of transformation from DPL Model to executable system files.

9.2.5.2 Transform DPL Model to WSDL Model

The valid DPL model is transformed to a WSDL model, which represents the interfaces of the participants in the composition.

9.2.5.3 Transform DPL Model to PDD Model

The valid DPL model is transformed to a PDD model, which represents an ActiveBPEL specific deployment descriptor detailing the resources of the composition.

9.2.5.4 Transform DPL Model to WSDLCatalog Model

The valid DPL model is transformed to a WSDLCatalog model, which represents an ActiveBPEL specific interface deployment descriptor detailing the interfaces of the composition.

9.2.5.5 Transform WS-BPEL Model to XML Model

The WS-BPEL model is transformed to an XML model. This XML model will later be serialised as WS-BPEL compliant XML text.

9.2.5.6 Transform WSDL Model to XML Model

The WSDL model is transformed to an XML model. This XML model will later be serialised as WSDL compliant XML text.

9.2.5.7 Transform PDD Model to XML Model

The PDD model is transformed to an XML model. This XML model will later be serialised as PDD compliant XML text.

9.2.5.8 Transform WSDLCatalog Model to XML Model

The WSDLCatalog model is transformed to an XML model. This XML model will later be serialised as WSDLCatalog compliant XML text.

9.2.5.9 Transform XML Models to Text

The four XML based models identified above are serialised to XML text, which can be executed on a composition engine.

9.2.5.10 Case Study

In step five in our case study example a WS-BPEL interaction logic document is created to represent the centralised distribution pattern. Additionally, a WSDL interface is created as a wrappers to the interaction logic document, enabling the composition to work in a centralised environment. A deployment descriptor and a deployment catalog file is also created. All that remains is for the system to be deployed to the target environment.

9.2.5.11 Tool Support

The transformation from a valid DPL model to an executable system is supported by nine specific steps. These nine steps can be automated using our tool as discussed in detail below.

Transform DPL Model to WS-BPEL Model The DPL model is now converted to a WS-BPEL model. The ANT script, outlined in Figure C.6 of Appendix C executes the transformations outlined in Section 8.4.2.

Transform DPL Model to WSDL Model The DPL model is now converted to a WSDL model. The ANT script, outlined in Figure C.7 of Appendix C executes the transformations outlined in Section 8.4.3.

Transform DPL Model to PDD Model The DPL model is now converted to a PDD model. The ANT script, outlined in Figure C.8 of Appendix C executes the transformations outlined in Section 8.4.4.

Transform DPL Model to WSDLCatalog Model The DPL model is now converted to a WSDLCatalog model. The ANT script, outlined in Figure C.9 of Appendix C executes the transformations outlined in Section 8.4.5.

Transform WS-BPEL Model to XML Model The WS-BPEL model is now converted to an XML based model by an ANT script. The script outlined in Figure C.10 of Appendix C executes the transformations outlined in Section 8.4.6.

Transform WSDL Model to XML Model The WSDL model is now converted to an XML based model by an ANT script. The script outlined in Figure C.11 of Appendix C executes the transformations outlined in Section 8.4.7.

Transform PDD Model to XML Model The PDD model is now converted to an XML based model by an ANT script. The script outlined in Figure C.12 of Appendix C executes the transformations outlined in Section 8.4.8.

Transform WSDLCatalog Model to XML Model The WSDLCatalog model is now converted to an XML based model by an ANT script. The script outlined in Figure C.13 of Appendix C executes the transformations outlined in Section 8.4.9.

Transform XML Models to Text The four XML based models are now converted to XML based text suitable for execution on a Web service composition engine. The ANT script outlined in Figure C.14 of Appendix C makes use of the AM3 XML extractor [10]. This extractor converts the XML based models to XML based text.

9.3 Summary

In this chapter we have presented our five step methodological framework. This framework enables the modeling of a Web service composition from a distribution pattern perspective, before generating an executable Web service composition. The framework uses the Model Driven Software Development (MDSD) process to automate, where possible, the executable system generation. This approach front loads the development effort enabling faster Web service composition development as well as providing mechanisms to reason about the system through models. These models are used here to apply distribution patterns with particular QoS attributes. These models can also be used to validate and verify, using OCL, the composition to ensure it will execute correctly, before it is generated. System properties such as deadlock and liveness can be proved using the MDSD process.

The distribution pattern catalog discussed in Chapter 5 is utilised by the methodological framework to assist the software architect when modeling the Web service composition. The methodological framework features three specific techniques, which assist in the generation of the executable system. All these techniques are supported by our tool implementation. Finally, we have included a case study featuring the centralised distribution pattern to illustrate the usage of the methodological framework.

Chapter 10

Evaluation

10.1 Introduction

Over the past six chapters we have presented our modeling and transformation framework. This framework containing five components is our solution to addressing the non-functional modeling of Web service compositions. Here, we assess how well our approach has met its objectives and compare our efforts to existing approaches and tools.

Initially we revisit the motivations for our modeling and transformation framework to Web service composition development, outlined in Chapter 1. We reiterate our problem statement and objectives in Section 10.2 before outlining the assumptions we have made in the scope of our work in Section 10.3. To assess our approach we compare and contrast our work to existing approaches such as the traditional handcrafted approach in Section 10.4, existing alternative frameworks in Section 10.5, and lastly to existing tools in Section 10.6. Finally, in Section 10.7 we discuss some of the issues encountered during the course of our research.

10.2 Problem and Objectives

Before evaluating our approach it is important to revisit our problem statement. In Chapter 1 we stated that there are a number of issues with traditional approaches to development of

Web service compositions. These issues are considered to be non-functional requirements, as outlined in ISO 9126 [88, 83]. Traditional development is often ad-hoc and requires considerable low level development effort for realisation. We also noted that this effort is increased in proportion to the number of Web services in a composition. Effort is also increased when there is a requirement that the composition participants must be flexible. Such systems often exhibit fixed architectures making maintenance difficult and error prone. Additionally, a number of the Quality of Service (QoS) requirements of a composition like efficiency and reliability cannot be assessed easily by examining low level code because of poor comprehensibility of the handcrafted artifacts. These quality requirements can be better managed using UML, as outlined by Lange et al. in [106].

Our objectives can be split into two categories, as follows.

- Development Process
- Product Output

Firstly, we intend to improve the development process for Web service compositions. We consider that the development effort for creating Web service compositions should be reduced. As a byproduct of reducing the development effort we intend to reduce the maintenance overheads traditionally associated with mutating compositions. More specifically, we intend to obviate the coding effort traditionally associated with Web service composition development thereby significantly reducing the development effort. Maintenance of the solution should be significantly reduced. In addition, improved maintainability should be achieved through a flexible architecture.

Secondly, we intend to improve the product output from the development process. Along with achieving improved maintainability we also intend to be able to manage the QoS of Web service compositions. Our approach to managing QoS is to expose non-functional properties, such as efficiency and reliability, of Web service compositions to ensure the resultant composition is of the required quality. This is achieved by refering the reader, where possible, to related works where the costs of a given pattern have been assessed using profiling techniques. Where explicit measure costs are not available the reader is referred to system implementations which are known to expose certain non-functional attributes.

10.3 Assumptions

A number of assumptions have been made in the scope of this work. These assumptions are listed below and elaborated in the following text.

- UML Familiarity
- Closed Environment
- Design Time Definition

The first assumption is as stated in Chapter 6 where we assume software architects will be familiar with the UML notation. For this reason we have chosen to model using UML where possible. We believe this reduces the initial learning overhead for architects modeling distribution patterns. If a Domain Specific Language (DSL) had been used the software architect would not be familiar with the notations used. In fact we have used a DSL to model distribution patterns, as discussed in Chapter 6, however this language is only used internally by our tools to facilitate validation and reduce the complexity of the transformations.

Our second assumption is that our solution will be deployed in a closed environment. This is necessary because many of the distribution patterns require a workflow engine to be installed to enable the execution of a given pattern. It would be unreasonable to assume that every Web service in an open environment has such a workflow engine available. Here we assume the deployer of our solution must have control over the deployment infrastructure.

Finally, our third assumption is that Web service compositions are defined at design time. We do not consider dynamic Web service compositions where the participants are selected at runtime. A design time approach has been chosen as our goal is to illustrate the benefits of distribution pattern modeling. Adapting our work for dynamic compositions, while useful, is considered to be future work.

10.4 Comparison to Handcrafted Approach

To evaluate the usefulness of our approach we must compare it to a traditional approach for defining Web service compositions. For simplicity we compare our approach to a handcrafted approach, which is often used for defining Web service compositions. The handcrafted approach is the most flexible way to define code of any type, because the code may be tailored exactly to requirements. However, this flexibility comes at a price as we will see in the following sections. We assume handcrafting to be the use of a text based editor to define the composition artifacts manually. To this end we compare both approaches with respect to the following criteria.

- Development Effort
- Maintainability
- Comprehensibility
- QoS

10.4.1 Development Effort

Considerable work is required in building our modeling and transformation framework as it is based on a generative programming methodology. Generative programming front loads most of the development effort. In our case this required the definition of languages, relations, transformations and a methodological framework. This development effort is a once off overhead. However, this effort is offset by a greatly reduced development time for generating code to realise Web service compositions, as our tool is capable of auto generating all of the composition artifacts with only limited human intervention required.

If a composition's architecture was going to be permanently fixed, with no flexibility with regards to participants and distribution pattern, our solution would require far more effort than just handcrafting the composition artifacts without the aid of a framework. However, we do not consider that Web service compositions are by their nature fixed in this manner. We believe that Web services are by their nature flexible and loosely coupled, thus motivating our modeling and code generation approach.

The amount of work required to define a composition using our modeling and transformation framework is considerably less than handcrafting the composition artifacts. Using our approach the only manual work the software architect must complete is step two of our methodological framework, as outlined in Section 9.2.2. Here, the architect defines the distribution pattern via a UML Activity diagram using a UML modeling tool. The rest of the work is automated using our tool. No handcrafting of any code, XML or otherwise, is required.

Considerable handcrafting of XML based composition artifacts is required if no framework is utilised. In our case study scenario a considerable amount of XML code, 137 lines to be precise, must be handcrafted to define an executable Web service composition. The breakdown of this figure is outlined in Table 10.1. Such handcrafting is both time consuming and error prone, requiring considerable testing and validation. The amount of work required to define such compositions increases significantly in relation to the number of composition participants and the distribution pattern being handcrafted. Our modeling and transformation framework is capable of generating fully executable code, which is comparable to that which would have had to be handcrafted previously.

It should be noted that more complex distribution patterns than the centralised pattern examined in the case study would require far more handcrafting of XML for realisation. For example, the decentralised distribution patterns would require one compositional interface, one collaboration description, one deployment descriptor and one deployment catalog per compositional participant. Thus the amount of handcrafting effort is directly proportional to the number of compositional participants.

Table 10.1: Single lines of code required for handcrafted approach.

Compositional Interface	36 lines
Collaboration Description	56 lines
Deployment Descriptor	38 lines
Deployment Catalog	7 lines
Total	137 lines

The software architecture of Web service compositions is often ignored or designed on a case by case basis when developed using a handcrafted approach. Ignoring architectural patterns such as the distribution pattern of a Web service composition results in a number of issues, such as the hiding of QoS attributes. Designing Web service composition architectures from scratch is a time intensive process and may result in an unnecessarily bespoke system. In contrast, our modeling and transformation framework, in association with our pattern library, provides a reusable framework for creating Web service compositions. This framework results in considerably lower design/development time and effort because the framework generates the compositional code. The pattern library ensures well documented solutions to particular deployment scenarios, which are well understood and reused where appropriate.

Web service compositions can be realised using a number of technologies. For example, WS-BPEL or WS-CDL could be used as the collaboration language to realise any of the distribution patterns discussed in Chapter 5. A number of deployment environments or compositional engines also exist. If a Web service composition is handcrafted it must be decided at an early stage which of these languages or environments will be targeted, resulting in poor portability. In contrast, using our modeling and transformation framework all the code is generated. This means that, provided transformations and notations have been fully defined, it is only a matter of changing a tagged value on the model to generate the desired language code. Although creating such transformations and notations is a time and labour intensive effort this is a once off labour outlay. This highly portable approach is augmented by the fact that our modeling approach is based on a platform and language neutral modeling language UML 2.

10.4.2 Maintainability

To assess the maintainability of both our modeling and transformation framework and the handcrafted approach we apply the ALMA (Architecture Level Modifiability Analysis) method [90, 142]. The ALMA method is a scenario based method designed specifically for evaluating risk assessment, maintenance and costs prediction. The method consists of five steps, outlined below.

- Set the Analysis Goal
- Describe Software Architecture
- Elicit Change Scenarios
- Evaluate the Change Scenarios
- Interpret Results

10.4.2.1 Set the Analysis Goal

Our analysis goal is to assess the maintainability of the two competing approaches as previously stated. This should result in a qualitative assessment of the effort required to maintain each approach.

10.4.2.2 Describe Software Architecture

The architecture of our modeling and transformation framework is expressed using UML 2, while the architecture of the handcrafted approach is expressed through raw XML artifacts.

10.4.2.3 Elicit Change Scenarios

We elicit the possible change scenarios as follows. These scenarios are possible maintenance events that may occur within the lifetime of the system.

- Addition of composition participant
- Removal of composition participant
- Updating of composition participant
- Change of distribution pattern
- Change of execution engine

10.4.2.4 Evaluate the Change Scenarios

With these scenarios in mind we evaluate the change scenarios for our modeling and transformation framework against the handcrafted approach to Web service composition development.

Addition of composition participant Often, Web service compositions are augmented by the addition of an additional participant. This additional participant may be used to enhance the functional or non-functional requirements of the composition. Traditionally this scenario required the developer or software architect to first assess the current deployment environment, and to then manually manipulate the XML composition code, as well as any deployment artifacts. This process is both error prone and time intensive. With our modeling and transformation framework the maintainer may open the existing UML Activity diagram of the composition, add the new participant, apply appropriate values to the participant and use our tool to regenerate the entire composition for deployment.

Removal of composition participant This scenario is similar to the previous scenario, the only difference being that we are removing a participant rather than adding a new one.

A participant may be removed from a composition due to an alteration in the functional business process of an enterprise, or perhaps because the service is failing to meet its non-functional requirement obligations. The same benefits as in the previous scenario can be realised by using our modeling and transformation framework.

Updating of composition participant Web services as loosely bound participants in a composition are likely to evolve and move location. This scenario considers that the end-point of one of the compositional participants needs to be changed. Such a change using the traditional handcrafted approach requires changes to the endpoint address in the collaboration, interface and deployment descriptor artifacts. Using our modeling and transformation framework requires only one change to the DPL profile applied to the UML Activity diagram. Here, only the interface-uri DPL Attribute must be changed. Subsequently our tool must be run to regenerate the entire composition for deployment.

Change of distribution pattern Changing the distribution pattern of a Web service composition may be motivated by a number of non-functional reasons, as outlined in the case studies of Chapter 5. Such a change is error prone and time intensive using the handcrafted approach. Firstly, the difficult task of assessing the current deployment scenario must be performed. Once the current environment is understood the workflow of the collaboration description must be edited manually to match the desired distribution pattern. Additional interfaces must be generated for certain patterns and the deployment descriptor and deployment catalog must be updated to reflect the changes. The larger the composition the more difficult it is to alter the distribution pattern because the complexity becomes so high. Even after these manual changes it is difficult to assess if the non-functional requirements of the composition will be met as no formal models exist. Using our modeling and transformation framework this complexity is significantly reduced as only a few changes to the DPL profile applied to the UML Activity diagram are required. Here, the distribution-pattern DPL Attribute must be changed, the roles of the CallBehaviourActions may be changed, ControlFlow and ObjectFlow connectors must be checked and possibly changed and finally additional constructs such as extra Pins may be added. Subsequently our tool must be run to regenerate the entire composition for deployment.

Change of execution engine Throughout this thesis we have considered only one execution engine, ActiveBPEL. However, it is possible that there may be a requirement after deployment to change to a different execution engine. Using the handcrafted approach, this would require the creation and testing of both the new engine's deployment descriptor and deployment catalog. As with the other maintenance tasks considered this is both error prone and time intensive. Our modeling and transformation framework also requires considerable work to support a new execution engine. Specifically, new notations must be defined to facilitate the new execution engine. New transformations between our DPL notation and these new notations must also be defined. The notations should be as rigorously defined as were the deployment descriptor and deployment catalog notations in Chapter 6. The transformations must also be related as were our relations is Chapter 7, and defined like our transformations in Chapter 8. These tasks are time and labour intensive. However, once the notations, relations and transformations have been defined they can be plugged into our modeling and transformation framework and reused many times, unlike the handcrafted approach. Once the notations, relations and transformations have been plugged into the framework our tool need only be run to regenerate the entire composition for deployment.

10.4.2.5 Interpret Results

Having enumerated over all the change scenarios we will now interpret the results. It should be considered that before a composition can be maintained the developer must be familiar with the current deployment scenario. Where models exist, as in our modeling and transformation framework, this discovery process is considerably shortened as the developer can view a model describing the architecture of the deployment. UML Activity diagrams are an excellent way to communicate the distribution pattern of a composition to a developer or software architect unfamiliar with a composition. Without such models the developer has to weed through XML code to discover the connections between the participants in the composition and build up a model of the distribution pattern before attempting any maintenance effort.

The scenarios illustrate that common maintenance issues such as adding, removing or updating a composition participant are handled with minimal effort using our modeling and transformational framework, when compared to the handcrafted approach. However, the biggest gains with respect to reduced labour overheads are seen when the distribution pattern is required to be changed. Our modeling and transformation framework is particularly suited to such scenarios as it was built with such tasks in mind. The final scenario noted the considerable effort required to enable our framework to support a new execution engine. This is the case as the framework is targeted at a particular platform. However, the framework is extensible and can be made to support any execution engine. The amount of time required to support alternative platforms would however be considerably higher than handcrafting the artifacts. Of course handcrafting the XML will result in the loss of the benefits already realised using our framework approach.

To summarise, our modeling and transformational framework provides a far more maintainable environment when compared to the handcrafted approach.

10.4.3 Comprehensibility

An essential goal of any modeling effort is comprehensibility, or readability, of the models produced. To this end we have used the standards based UML 2 modeling language to express our distribution patterns. We believe the models produced are easy to read and are at an appropriate level of granularity for software architects to comprehend the diagrams easily. Clutter in the diagrams has been avoided through the use of the UML Profile extension mechanism. Additional comprehensibility is provided by our pattern catalog in Chapter 5, which explains each of the patterns identified in this thesis in detail.

Our UML 2 models are far more comprehensible than the raw XML artifacts used to describe Web service compositions. Graphical notations are instantly more recognisable than terse text files. XML files although originally designed to be human readable are far too verbose to be human comprehensible when considering large Web service compositions.

Complexity often results in reduced comprehensibility. As the number of participants in a composition increases the complexity of the system increases. Large Web service compositions traditionally require unwieldy amounts of handcrafted XML for realisation. As already noted in Section 10.4.2, this creates a significant maintenance overhead. Our modeling and transformation framework negates the manipulation of XML files by abstracting the collaboration details into high level UML models. These models however may also get unwieldy given a large number of participants in a composition. The Eclipse UML2 editor used to illustrate our use case is not ideal for medium to large compositions because of its rudimentary user interface and lack of helper tools. Tools like IBM's RSA [86] have been designed to assist software architects in managing enterprise systems, which would include medium to large Web service compositions. We have noted in our future work that there are significant automation opportunities in the area of semantics, which may obviate the need for any human intervention at the modeling level. In this scenario the models are used as the primary software architecture artifact, used to drive the generation of fully executable compositions.

10.4.4 QoS

An often overlooked aspect of Web service compositions is the non-functional Quality of Service (QoS) requirements such as efficiency and reliability. It is very difficult to assess the QoS of a composition purely by examining low level code because of the poor comprehensibility of the handcrafted artifacts. We believe that QoS issues should be considered from the beginning of the development process. This assertion is supported by our catalog of distribution patterns in Chapter 5, which each express different QoS attributes. Using distribution pattern based models as the primary development artifact we can guide our code generation effort based upon clearly visible non-functional QoS properties. These patterns are not visible in raw XML artifacts resulting in compositions which have undetermined

QoS attributes.

10.5 Comparison to Existing Frameworks

In Chapter 3 we discussed the state of the art in modeling and transformation frameworks by comparing each of the approaches encountered in Figure 3.1. Here, we compare our framework to these other frameworks. Although the ultimate goal of our work is different to the frameworks we compare to, we believe it provides considerable insight into the usefulness of our approach. The updated comparison table is illustrated in Figure 10.1.

	Arief et al	DECS	SELF-SERV	Peer-Serv	UWE	OPENFlow	Web-ML	UMT	Not Traveler	Topman
CORBA Support						✓				
Web Application Support					✓					
Web Service Support		✓	✓	✓			✓	✓		✓
ADL Model Support			✓							
XML Model Support		✓	✓	?			✓*	✓	✓	✓*
UML Model Support	✓			?	✓&			✓		✓
BPMN Model Support							✓			
No. of Schemes Supported	n/a	2#	1	1	n/a	2~	2	n/a	1	9
Models Architecture	✓									
Models Orchestrations		✓	✓	?		✓	✓	✓	✓	
Models Choreographies									✓	✓
Code Generation Support	✓	✓	✓	✓	✓	✓	✓	✓		✓
Dynamic Reconfiguration		✓				✓				
Static Reconfiguration	✓		✓	✓	✓		✓	✓		✓

# Supports changing of distribution scheme but does not model it explicitly
~ Distribution scheme is defined in the orchestration
* XML support is via XMI
& UWE uses a conservative extension to UML called WebRE
n/a Feature not explicitly considered
? Not clear from paper

Figure 10.1: Comparison of existing frameworks to our modeling and transformation framework.

From Figure 10.1 we can see how our modeling and transformation framework, termed TOPMAN in the table, compares favourably with the feature set provided by the state of the art in modeling and transformation frameworks. Our approach, like a number of other tools, considers the Web service composition domain. Two tools, UWE and OpenFlow, consider the alternative domains of Web applications and CORBA respectively. We chose not to consider these domains as CORBA can be considered a precursor to Web services in that both technologies have the same objective, while Web applications are orthogonal in that they sit on top of the functionality provided by Web service compositions.

We note that SELF-SERV is the only framework to support the use of an Architecture Description Language (ADL) as its modeling language. We chose not to use ADL as our modeling language because its various notations may not be as familiar to software architects as the UML 2 notation. We do, however, future proof our approach by defining our own DSL, the Distribution Pattern Language (DPL), which is not tied to UML 2, enabling future work to address a perceived need to model using ADL instead of UML 2. Three of the frameworks - DECS, SELF-SERV and Net Traveler - provide only XML modeling of compositions. We believe that XML is not an ideal language for defining models because XML is overly verbose and is not human friendly when compared to a well designed visual representation of a composition. Our modeling approach using UML 2 provides a visual representation of the composition along with bindings to XML via the UML serialisation format XMI. This approach is also utilised successfully by UMT. An alternative approach to UML modeling is considered by WEB-ML where BPMN models are used instead of UML to model compositions. As previously noted, our approach could be amended to model with BPMN as we have ensured that our approach is not UML dependent.

A number of the identified frameworks support different distribution patterns. However, only one of these approaches Web-ML, explicitly models the distribution pattern in a similar way to our approach. Web-ML is however restricted to only two distribution patterns. Both OpenFlow and DECS also support two distribution patterns; however they do not explicitly model the distribution pattern, and merely provide a switch to alter the behaviour of the executing system. Our approach, in contrast, considers nine distribution patterns which are expressed in easy to comprehend high level UML models.

The majority of the frameworks model compositions from an orchestration point of view. This perspective considers the workflow of the compositions participants rather than the non-functional requirements, which is our objective. UMT is an example of a tool which meets many of our requirements but models the composition from an orchestration perspective. It is worth noting that Arief et al. [17] consider the modeling of system architectures using UML, an approach we also use. Also worthy of note is the Net Traveler approach, which considers both an orchestration and a choreography model, albeit using XML based models. Our modeling and transformation framework combines the UML modeling approach used by Arief et al. and the choreography models used by Net Traveler to address non-functional requirements of Web service compositions.

All of the frameworks, with the exception of Net Traveler, provide an implementation to prove their usefulness and evaluate the claims of their respective authors. Our modeling and transformation framework provides an implementation, TOPMAN, which outputs a fully executable system that verifies our solution.

Finally, only two of the frameworks discussed consider dynamic reconfiguration of the systems they model. These two systems, DECS and OPENFlow, are capable of altering the distribution pattern used at runtime. This functionality is desirable as it allows the distribution pattern to be changed in reaction to the execution environment. For example, under high load the decentralised distribution pattern is more favourable that the centralised pattern. As previously noted, we consider runtime alteration of distribution patterns on our modeling and transformation framework as future work.

10.6 Comparison to Existing Tools

Throughout this chapter we compare our modeling and transformation framework to a handcrafted approach. There are, however, a number of tools which facilitate the creation of the compositional artifacts. We have not considered these tools thus far as they do not address our objectives of addressing non-functional requirements. However, for completeness we compare and contrast a number of these tools here.

ActiveBPEL Designer [3] is a GUI tool that assists in the development of Web service compositions. The tool is capable of generating all the compositional artifacts, as well as performing debugging and simulation activities. However, the tool takes an orchestration perspective on compositions, ignoring non-functional requirements addressed by our approach. The ActiveBPEL Designer creates compositional artifacts that can be run on the

ActiveBPEL compositional engine.

The Oracle BPEL Process Manager [141] is comparable to the ActiveBPEL Designer product. The Oracle tool provides the same functions as the ActiveBPEL tool. It also takes the same orchestration perspective, once again ignoring non-functional requirements addressed by our approach. The Oracle BPEL Process Manager creates compositional artifacts that can be run on the Oracle BPEL compositional engine.

Artix Orchestration [89] is an Eclipse plugin developed by Iona. The tool is used to design orchestration based compositions, which can be realised in the Artix environment. Again the tool ignores non-functional requirements.

Finally, we consider the Eclipse BPEL Project [66]. This Eclipse plugin provides a GUI for the development of WS-BPEL orchestrations. Unlike the other three tools previously described the Eclipse BPEL Project does not include an execution environment for orchestrations to be realised. The tool concentrates purely on facilitating the generation of WS-BPEL orchestrations and does not consider the generation of supporting compositional artifacts like the other three tools, as well as our own modeling and transformation framework, do.

10.7 Discussion

Over the course of our research a number of issues were encountered. These issues are listed below and discussed in detail in the following sections.

- Maintainability
- Complexity
- Composition

10.7.1 Maintainability

Although ultimately we decided on using ATL as our model transformation language, as discussed in Chapter 8, we also considered another approach, XSLT based transformations.

These two approaches provide distinct advantages and disadvantages with respect to maintainability, as discussed below.

Initially, we envisaged using the XML based transformation language XSLT [183] to transform XMI representations of our distribution pattern models. A prototype was built to test the usefulness of this approach. The primary motivation for using this language is that it is popular amongst developers, especially in the Web community. This popularity means there is excellent support and numerous resources for the language. Additionally, the language is declarative making the transformation code easier to comprehend and maintain than imperative based languages.

Although we were successful in outputting a fully functional executable system based on XSLT transformation of distribution pattern models, the code base was considered too brittle to be considered maintainable. This brittleness was caused by a number of factors. Firstly, the transformations are tied to a particular XMI version. Different tools output different versions of XMI and there are a number of significant versions of XMI. Any future version of XMI will break the XSLT transformations causing a large maintenance overhead for future users of this approach. These problems are similar to the well documented "DLL hell" problems in Windows based computing [79]. We consider that XMI is not supposed to be manipulated as regular XML as the standard is far too verbose for such transformations. Secondly, the XSLT transformations were very verbose and complex due to the complexity of the XMI documents they were transforming.

With these issues in mind we looked for an alternative approach. We discovered that ATL does not incur these maintainability issues as it uses the stable metamodel UML2. ATL does not manipulate XMI directly; instead it reads the XMI into an object model. These transformations can be easily modified to handle new versions of the UML2 metamodel. Changes to the metamodel are often incremental and do not break implementations based on the same major revision e.g. UML 2.1 is backwards compatible with UML 2.0. This resolved the brittleness problem encountered by XSLT.

10.7.2 Complexity

The complexity of the XSLT transformations, discussed in the previous section, was also solved by ATL. ATL uses the UML2 metamodel to traverse object based representations of models rather than pattern matching within large XMI files. This approach results in far cleaner and easier to manage code which can be easily maintained.

Another area of complexity is that because Web services is still an emerging technology the Web service stack is in constant flux. The sheer number of standards or proposals for standardisation causes great complexity in the domain. This flux is caused by the number of large organisations vying for dominance in this emerging area. In fact, it is these organisations flexing their influence that have caused this fragmentation by submitting to different standards bodies such as the W3C and OASIS in an attempt to have their favoured proposals fast tracked to consumer acceptance. An example of this competition is the Oasis standardised WS-BPEL and the W3C standardised WS-CDL. Both are still being revised.

A number of tools, introduced in Chapter 2, were used to support our modeling and transformation framework. These tools exhibit considerable complexity and require considerable time to gain proficiency. Primarily, we used the Eclipse tool in association with the Eclipse Modeling Framework (EMF) and the ATL transformation language. EMF is a complex framework requiring considerable learning effort. Ecore, EMF's meta-model language is complex necessitating considerable study before use. We used Ecore to define our notations in Chapter 6. This process required learning the Ecore notation and idiosyncrasies. ATL is a modeling specific language requiring the user to learn a new language which is considerably different to traditional procedural languages such as C or Java. Both Ecore and ATL utilise an additional language OCL for defining types, constraints and functions, which also needed to be studied.

Our approach uses the UML2 Ecore meta-model implementation. This implementation of such a large specification as UML 2.0, although well documented, requires considerable time before the user becomes proficient in its usage. UML 2.0's extension mechanism, UML Profiles, also requires a large labour outlay before usage. All these technologies result in a complex development environment where dependencies exist between the tools, implementations and languages. Additionally, as the general area of MDSD is new these technologies are undergoing fast evolution cycles. For example, UML2 has gone through five revisions from 2.0 to 2.1.1 in one year.

A significant issue encountered during our research was that we touch on a number of complex research areas in order to produce an adequate solution to our research problem. Each of these research areas, such as MDSD, Web services, software architectures and even semantics contain many open questions in their own right. The combination of such complex research areas results in the highly technical solution presented in this thesis. Each of the areas required considerable research to see how they could help us define our non-functional modeling and transformation framework. Adding to the inherent complexity of these areas is the fact that they are, for the most part, relatively new and expanding research areas, making the foundations for our research a moving target as the areas continue to evolve at considerable pace. Keeping apace with these research areas has been a significant issue in the context of our work.

10.7.3 Composition

When discussing distribution patterns with those who are unfamiliar with the concept of distribution patterns there is often great confusion between what the patterns represent and what traditional workflows represent. Often the audience cannot distinguish between the non-functional requirements we wish to model and the functional business processes usually represented as Activity diagrams. As noted in Section 2.3.1, we consider the best way to distinguish these mutually independent modeling approaches is to clearly define the difference between orchestrations and choreography, where orchestration represents the internal workflow required to implement a businesses workflow, while choreography represents the external message exchange between participants that maps closely to our concept of distribution patterns.

10.8 Summary

In this chapter we assessed the value of our modeling and transformation framework. We noted that our framework front loads the development effort. However, this effort is offset by the gains in maintainability obtained using our approach, as proved by our validation using the ALMA method. Our approach realises the benefits of the Model Driven Software Development (MDSD) approach, where models are used to assist in the generation and reasoning of software systems [182, 30].

Often when systems use the MDSD process the end user expectation is that the entire system will be auto-generated from a primitive model to an executable system. However, as in our case this is often not true. Human intervention is sometimes required to manipulate models to encode some hard to model knowledge. Here, we require the experience and skill of software architects to make important architectural decisions about which distribution pattern is most appropriate for a given scenario. This kind of intelligence is very difficult to automate even through the use of semantics. Automating important architectural decisions like this would be neither desirable nor would it be accepted by practitioners.

We have found that our approach meets its objective of providing readable models of the non-functional properties of Web service compositions, an aspect previously ignored by framework and tool developers. This comprehensibility is essential to adequately communicate the non-functional QoS attributes of compositions to software architects at design time.

Having considered a number of different approaches to our modeling and transformation framework, and having overcome a number of difficult issues, we believe we have made a significant novel contribution to the area of Web service composition development, and to a wider extent the software architecture community.

Chapter 11

Conclusions

11.1 Summary

Over the course of this thesis we have presented our novel Web service based modeling and transformation framework. Our Model Driven Software Development (MDSD) based approach takes existing Web service interfaces as its input and generates an executable Web service composition, based on a distribution pattern chosen by the software architect. We have placed this framework into context by discussing related work and evaluating our framework against the state of the art in compositional modeling. Here, we summarise our contributions.

We have provided a catalog of seven distribution patterns that provide solutions to varying non-functional requirements, such as efficiency and reliability. These patterns, which have been found to be useful in a networking context, are applicable to Web service compositions. The effectiveness of a pattern catalog has been objectively evaluated against appropriate case studies using four specific criteria: usage, coverage, utility and precision. The catalog scored good values for the majority of the measures indicating that it is sufficiently complete to cover all the current distribution scenarios, whilst also providing adequate future proofing for future usage scenarios.

A novel modeling and transformation framework consisting of five components has been provided. This framework, unlike the existing tools, considers modeling of the distribution scheme from a choreography perspective. This approach considers non-functional quality aspects from the outset of the development process. The framework, and its associated implementation, enables the generation of Web service compositions, based upon a distribution pattern model. These models abstract complexity and enable high level reasoning about a solution from an early point in the development life-cycle. Attributes that can be observed include design-time non-observable architectural quality attributes of compositions, like mutability and reuse. Additionally, run-time attributes affected by the chosen distribution scheme can also be observed at design time using these models. These attributes, also known as Quality of Service (QoS) attributes, include efficiency and reliability.

Each of the five components in the framework has been motivated with respect to non-functional quality aspects and each is essential in our MDSD approach for generating Web service compositions based on a distribution pattern model. The eight notations, or languages, required to facilitate the modeling of distribution patterns, and the subsequent generation of an executable system have been provided. The abstract syntax of all of these languages has been defined using ECore to enable their use in an MDA context. Nine relation sets have been used to describe the web of dependencies between the languages, from a UML distribution pattern model to executable system XML. The corresponding nine transformation sets have also been defined. The five step methodological framework that ties together the first four components of the modeling and transformation framework has been thoroughly documented. This methodological framework enables the modeling of a Web service composition from a distribution pattern perspective, before generating an executable Web service composition. A case study featuring the centralised distribution pattern has been used to exemplify usage of the methodological framework. The case study output has been deployed and tested on a real execution engine.

We have contrasted our framework with a handcrafted approach, which is often used for defining Web service compositions. The approaches were contrasted by considering four specific measures: development effort, maintainability, comprehensibility and QoS.

Our framework compared favourably in all comparisons.

Development effort is significantly reduced as handcrafting is both time consuming and error prone, requiring considerable testing and validation. The amount of work required to define compositions increases significantly in relation to the number of composition participants and the distribution pattern being handcrafted. Our modeling and transformation framework is capable of generating fully executable code, which is comparable to that which would have had to be handcrafted previously.

Maintenance issues such as adding, removing or updating a composition participant are handled with minimal effort using our modeling and transformational framework, when compared to the handcrafted approach. However, the biggest gains with respect to reduced labour overheads are seen when the distribution pattern is required to be changed. Our modeling and transformation framework is particularly suited to such scenarios as it was built with these tasks in mind.

11.2 Discussion

We have utilised the MDSD based approach in our framework for a number of reasons. MDSD uses models as its primary development artifact. These models capture significant decisions made by the software architect while designing the system architecture [84]. The significance of these decisions should not be understated as they are often taken with non-functional requirements in mind. Often these decisions are not documented resulting in the loss of important knowledge that can be used to ascertain the non-functional attributes of the system. Distribution patterns are an example of how models can be used to document these non-functional attributes. These patterns have well known trade-offs. The benefits and consequences of these trade-offs can be assessed before they are used in a system, ensuring the architect makes an informed decision before selecting a particular pattern.

The use of the MDSD approach provides for partial automation of the development of Web service based compositions. This partial automation results in reduced development/

maintenance effort as well as costs over the lifetime of the system [99, 50]. The ability to re-architect a previously generated system after the system has been developed is also catered for within the MDSD approach. This flexibility enables the software architect to apply a different distribution pattern, possibly because of new customer driven non-functional requirements, to a composition and redeploy the system.

Our modeling and transformation framework is specifically targeted at Web service based compositions. We consider a number of compositional languages such as WS-BPEL [12] and WS-CDL [187], standardised by OASIS and the W3C respectively. To avoid tying our approach to any specific compositional language we utilise the UML to model our distribution patterns [60]. UML Activity diagrams are particularly appropriate for describing the connections between discrete Web services in a distributed composition.

Complexity often results in reduced comprehensibility. As the number of participants in a composition increases the complexity of the system increases. Large Web service compositions traditionally require unwieldy amounts of handcrafted XML for realisation. Our modeling and transformation framework obviates the manipulation of XML files by abstracting the collaboration details into high level UML models. Our UML 2 models are far more comprehensible than the raw XML artifacts used to describe Web service compositions. Graphical notations are instantly more recognisable than terse text files. XML files, although originally designed to be human readable, are far too verbose to be human comprehensible when considering large Web service compositions.

It is very difficult to assess the QoS of a composition purely by examining low level code because of the poor comprehensibility of the handcrafted artifacts. We believe that QoS issues should be considered from the beginning of the development process. Using distribution pattern based models as the primary development artifact we can guide our code generation effort based upon previous experience of systems expressing a given distribution pattern, documented in our pattern catalog, which achieves certain non-functional QoS properties. These patterns are not visible in raw XML artifacts resulting in compositions which have undetermined QoS attributes.

Our framework has also been contrasted to the state in the art of existing frameworks and tools. Although many of these frameworks and tools overcome the issues of the hand-crafted approach, none of them make full use of the distribution pattern based modeling approach we have motivated throughout this thesis. In fact, the majority of the frameworks model compositions from an orchestration point of view. This perspective considers the workflow of the compositions participants rather than the non-functional requirements. The few frameworks that do consider that distribution patterns may be useful consider only the two most basic forms, centralised and decentralised.

11.3 Future Work

A number of enhancements to our modeling and transformation framework have been identified as future work. These enhancements are the result of conversations with researchers at both international conferences and workshops. Some of the enhancements have been detailed in our published papers, others are included here to encourage others to continue our work. These enhancements are as follows.

- Alternative Modeling Languages
- Full Modeling Approach
- Workflow Based Semi-Automation
- Semantic Based Semi-Automation
- Automated Deployment
- Explicit Modeling of Measure Costs

11.3.1 Alternative Modeling Languages

We believe it would be interesting to consider alternatives to our UML 2 modeling language approach, based on π calculus [118] and Architecture Description Languages (ADL) [114].

These languages offer alternatives approaches to describing distribution patterns, including the modeling of the patterns themselves rather than pattern prototypes. This approach to pattern modeling would enhance our pattern catalog by describing the discrete parts of the patterns how the actual patterns are constructed.

The π calculus is often used to model mobile processes, where the configuration of executing systems may change. This configuration is similar in concept to distribution patterns. We believe it would be interesting to model distribution patterns using such a calculus so that we can describe and reason about the distribution pattern catalog. The π calculus has been previously used to this end to describe, verify and validate WS-BPEL by Lucchi et al. in [155] and WS-CDL by Zhou et al. in[198].

ADLs are used to specifically model software architectures. Having a language designed with purely architectures in mind should make for a powerful representation of distributed systems such as Web service compositions. Such representations may lend themselves to better descriptions of distribution patterns over UML 2. We noted our reasons for not using ADLs in Chapter 3. However, we believe a full comparison of the two modeling approaches would be of great interest.

11.3.2 Full Modeling Approach

As previously noted our modeling and transformation framework considers the modeling of distribution patterns. These distribution patterns address only non-functional requirements of Web service compositions. To have a really useful modeling approach we must also consider functional requirements such as business workflows. A number of existing approaches considered in Chapter 3 address these concerns. They do not however address distribution patterns. We believe a full modeling approach addressing functional and non-functional requirements of Web service compositions is appropriate. Perhaps our approach could be integrated with an existing workflow based approach such as UMT [153]. How this combined approach would address modeling of orthogonal concerns is an open question.

11.3.3 Workflow Based Semi-Automation

Our modeling and transformation framework assumes only that WSDL artifacts already exist when the software architect is tasked with modeling the distribution pattern of a Web service composition. We use these artifacts as input to our framework to help generate part of the UML Activity diagram which will describe the distribution pattern. It may however be the case that the software architect has access to collaboration documents such as WS-BPEL [12] or WS-CDL documents [187], which describe the functional workflow of a composition. These artifacts could be used to further help generate more of the UML Activity diagram at step 1 of our methodological framework, as outlined in Section 9.2.1. For example, workflow documents contain mappings which could be used to relate UML ObjectFlow connectors, reducing the amount of work the architect must do to build a distribution pattern at step 2 of our methodological framework, as outlined in Section 9.2.2.

11.3.4 Semantic Based Semi-Automation

In our paper [24] we have envisaged an enhancement to our modeling and transformation framework using semantics to help reduce the software architect's workload at step 2 of our methodological framework. Currently the connections and mappings between Web services must be manually defined. However, if semantics were present the architect would simply choose a distribution pattern, possibly from a pattern repository, and the connections and mappings would be made automatically. We propose using Web service semantic descriptions in addition to Web service interfaces, to assist in the semi-automatic generation of the distribution pattern model. Web services described using semantic languages, such as OWL-S [109, 110], can be automatically assessed for compatibility and their input and output messages can be mapped to each other.

We assume all of the Web services to be composed are semantically annotated using OWL-S. The semantic documents for each service are passed to the semantic matching engine for processing. Each service must have an atomic process model describing, using an ontology, the message input and output parts. OWL-S atomic process models are analogous to WSDL operations. These semantic descriptions enable the automated sequencing of actions and connection of CallBehaviorActions to one another, in our distribution pattern model, using UML ControlFlow connectors. Services are matched together based on their level of compatibility. Each service is checked against every other participant service to assess if their process models are compatible. Compatibility here is defined as one participant having output message part(s) that match the input message part(s) requirements of another participant. If a sufficiently similar match is found a UML ControlFlow connector is created between the two compatible services in the model. Subsequently, the inputs and output parts of these matched services can be mapped. This integration results in the connection of UML InputPins and OutputPins in the model, using UML ObjectFlows connectors, so data can flow through the composition. In some cases, additional pins may be added automatically to the output of CallBehaviorActions, to meet data input requirements of other services. Existing services are wrapped to support the new connections. Without semantic annotation this entire step would have to be completed manually by the software architect. At this stage the model is complete and fully expresses the distribution pattern selected by the software architect.

11.3.5 Automated Deployment

A considerable overhead related to realising a number of distribution patterns, such as decentralised distribution patterns, is the deployment effort related to placing the artifacts generated by our tool on participating services. In our paper [25] we consider a novel approach to automating this effort.

We consider an enhancement to the container of each composition participant called Interaction Logic Document Processor (ILDP). The ILDP enhancement must be installed on each participant, however this is a once off installation. ILDP enhanced participants are exposed as Web services, capable of receiving, processing and deploying these documents. These enhanced participants can receive documents from the deployment engine. Subsequently the documents are processed by ILDP to ensure they are valid before storing them on the participant. Finally the stored documents are deployed by ILDP on the participant and exposed for composition by a composition runtime interface. An enactment engine, independent of ILDP, is responsible for enacting the interaction logic and subsequently invoking the participant services, facilitating decentralised interaction amongst the participant services. This approach obviates any requirement of manually deploying documents to participant services. Moreover, as the mechanism enhances the container capability, it is non-intrusive to the existing Web service implementation or to the existing interfaces of the participant services.

11.3.6 Explicit Modeling of Measure Costs

The pattern catalog presented in Chapter 5 uses related work to assess the non-functional properties of the patterns. These properties are based upon Web service composition measures and networking measures if there were no measures for a Web service context. Ideally, we would like to have measures for each pattern in a Web service context. These measures would enhance our understanding of the patterns in our particular domain of interest. Having these measures could also enable us to extend our DPLProfile to consider quantifiable estimates of efficiency and reliability as has been done in profiles such as MARTE [62], as discussed in Section 3.9.

Bibliography

[1] IEEE Std 610.12-1990. IEEE Standard Glossary of Software Engineering Terminology. Technical report, IEEE, 1990.

[2] ActiveBPEL. ActiveBPEL 2.0, 2006. `http://www.activebpel.org/`.

[3] ActiveEndpoints. ActiveBPEL Designer, 2007. `http://www.active-endpoints.com/active-bpel-designer.htm`.

[4] A. Agrawal, G. Karsai, , and A. Lédeczi. An End-to-End Domain-Driven Software Development Framework. In *Proc. 18th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 03)*, pages 8–15, Anaheim, California, 2003. ACM Press.

[5] D. Akehurst. Transformations Based on Relations. In *Proc. of the Workshop on Model Driven Development (WMDD 2004)*, Oslo, Norway, 2004. ACM.

[6] Stephen Albin. *The Art of Software Architecture: Design Methods and Techniques*. John Wiley & Sons, Inc., New York, NY, USA, 2003.

[7] C. Alexander, S. Ishikawa, and M. Silverstein. *A Pattern Language*. Oxford University Press, 1977.

[8] F. Allilaire, J. Bézivin, F. Jouault, and I. Kurtev. ATL - Eclipse Support for Model Transformation. In *Proc. of the Eclipse Technology eXchange workshop (eTX) at the ECOOP 2006*, Nantes, France, 2006.

[9] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services: Concepts, Architecture and Applications*. Springer Verlag, 2004.

[10] AM3 Tool. ANT Task for AMMA, 2007. `http://www.eclipse.org/m2m/atl/doc/ANT_Task_AMMA.pdf`.

[11] T. Ambühler. UML 2.0 Profile for WS-BPEL with Mapping to WS-BPEL. M.Sc. (thesis), Universität Stuttgart, Universitätsstr. 38, 70569, Stuggart, 2005.

[12] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Lie, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business Process Execution Language for Web Services (BPEL4WS) version 1.1. Technical report, BEA, IBM, Microsoft, SAP, Siebel, 2003.

[13] Apache. Apache Axis, 2006. `http://ws.apache.org/axis/`.

[14] Apache. Apache Struts, 2006. `http://struts.apache.org/`.

[15] B. K. Appukuttan, T. Clark, S. Reddy, L. Tratt, and R. Venkatesh. A Model Driven Approach to Building Implementable Model Transformations. In *Workshop in Software Model Engineering (WiSME) 2003*, 2003.

[16] L. Arief and N. Speirs. Automatic Generation of Distributed System Simulations from UML. In *Proc. 13th European Simulation Multiconference (ESM'99)*, pages 126–136, Warsaw, Poland, 1999.

[17] L.B. Arief, M.C. Little, and S.K. Shrivastava et. al. Specifying Distributed System Services. *BT Technology Journal*, 17:126–136, 1999.

[18] P. Avgeriou and U. Zdun. Architectural Patterns Revisited - a Pattern Language. In *Proc. 10th European Conference on Pattern Languages of Programs (EuroPLoP 2005)*, Irsee, Germany, 2005.

[19] B. Bauer and J. Müller. MDA Applied: From Sequence Diagrams to Web Service Choreography. In *Proc. 4th International Conference (ICWE 2004)*, pages 132–136, Munich, Germany, July 2004.

[20] J. W. Backus, J. H. Wegstein, A. van Wijngaarden, M. Woodger, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, and B. Vauquois. Report on the Algorithmic Language ALGOL 60. *Communications of the ACM*, 3(5):299–314, 1960.

[21] R. Barrett and S.J. Delany. OpenMVC: A Non-Proprietary Component-Based Framework for Web Applications. In *Proc. International Conference on the World-Wide Web (WWW 04)*, New York, USA, May 2004.

[22] R. Barrett and C. Pahl. Model Driven Design of Distribution Patterns for Web Service Compositions. In *The International Workshop on Models for Enterprise Computing (IWMEC 06)*, Hong Kong, China, October 2006.

[23] R. Barrett and C. Pahl. Model Driven Design of Distribution Patterns for Web Service Compositions. In *Proc. International Conference on Web Services (ICWS 2006) (Work-in-Progress Track)*, Chicago, USA, September 2006.

[24] R. Barrett and C. Pahl. Semi-Automatic Distribution Pattern Modeling of Web Service Compositions using Semantics. In *Proc. Tenth IEEE International EDOC Conference*, Hong Kong, China, October 2006.

[25] R. Barrett, C. Pahl, L. Patcas, and J. Murphy. Model Driven Distribution Pattern Design for Dynamic Web Service Compositions. In *Proc. Sixth International Conference on Web Engineering*, Palo Alto, USA, July 2006.

[26] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice, Second Edition*. Addison-Wesley Professional, 2003.

[27] B. Benatallah, M. Dumas, and Q. Z. Sheng. Facilitating the Rapid Development and Scalable Orchestration of Composite Web Services. *Distributed and Parallel Databases*, 17:5–37, 2005.

[28] B. Benatallah, M. Dumas, Q. Z. Sheng, and A. H. H. Ngu. Declarative Composition and Peer-to-Peer Provisioning of Dynamic Web Services. In *Proc. 18th International Conference on Data Engineering (ICDE'02)*, pages 297–308, San Jose, CA, February 2002.

[29] B. Benatallah, Q. Z. Sheng, and M. Dumas. The Self-Serv Environment for Web Services Composition. *IEEE Internet Computing*, 7:40–48, 2003.

[30] S. Beydeda, M. Book, and V. Gruhn. *Model-Driven Software Development*. Springer, 2005.

[31] J. Bėzivin. In Search of a Basic Principle for Model Driven Engineering. *UPGRADE*, 2, 2004.

[32] J. Bėzivin. Model-Driven Development: Its Essence and Opportunities. In *Summer School on Generative and Transformational Techniques in Software Engineering, Braga, Portugal*, pages 1–40, Braga, Portugal, 2005.

[33] J. Bėzivin, H. Brunelière, F. Jouault, and I. Kurtev. Model Engineering Support for Tool Interoperability. In *Int. Workshop in Software Model Engineering (WiSME)*, Montego Bay, Jamaica, 2005.

[34] L. Bichler. Tool Support for Generating Implementations of MOF-Based Modeling Languages. In *Proc. Third OOPSLA Workshop on Domain-Specific Modeling*, Anaheim, California, USA, 2003.

[35] Ken Birman, Robbert van Renesse, and Werner Vogels. Adding High Availability and Autonomic Behavior to Web Services. In *Proc. 26th International Conference on*

Software Engineering(ICSE'04), pages 17–26, Washington, DC, USA, 2004. IEEE Computer Society.

[36] C. Bock. UML 2 Activity and Action Models. *Journal of Object Technology*, 2, 2003.

[37] M. Brambilla, S. Ceri, P. Fraternali, and I. Manolescu. Process Modeling in Web Applications. *ACM Transactions on Software Engineering and Methodology*, 15:360–409, 2006.

[38] Hugo Bruneliere. MoDisco Use Case - Performance-Annotated UML2 State Charts, 2007. `http://www.eclipse.org/gmt/modisco/useCases/PerformanceAnnotatedUmlStateCharts/`.

[39] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. J. Grose. *Eclipse Modeling Framework*. Prentice Hall, 2003.

[40] F. Buschmann. *Pattern-Oriented Software Architecture : A System of Patterns*. Wiley, 2000.

[41] F. Buschmann, K. Henney, and D. C. Schmidt. *Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing*. Wiley, 2007.

[42] H. Caituiro-Monge and M. Rodriguez-Martinez. Net Traveler: A Framework for Autonomic Web Services Collaboration, Orchestration and Choreography in E-Government Information Systems. In *Proc. IEEE International Conference on Web Services (ICWS'04)*, pages 2–10, San Diego, California, USA, June 2004.

[43] S. Ceri, P. Fraternali, and M. Matera. Conceptual Modeling of Data-Intensive Web Applications. *IEEE Internet Computing*, 6:20–30, 2002.

[44] G. B. Chafle, S. Chandra, V. Mann, and M. G. Nanda. Decentralized Orchestration of Composite Web Services. In *Proc. 13th international World Wide Web conference*, pages 134 – 143, New York, NY, USA, May 2004.

[45] L. Chen, Ma Xiujun, C. Guanhua, S. Yanfeng, and F. Xuebing. A Peer-to-Peer Architecture for Dynamic Executing GIS Web Service Composition. In *Proc. Geoscience and Remote Sensing Symposium, 2005 (IGARSS '05)*, pages 979– 982, Seoul, Korea, July 2005.

[46] Q. Chen and M. Hsu. Inter-Enterprise Collaborative Business Process Management. In *Proc. of the 17th International Conference on Data Engineering*, pages 253–260, Heidelberg, Germany, April 2001.

[47] P. Clements, R. Kazman, and M. Klein. *Evaluating Software Architectures: Methods and Case Studies*. Addison-Wesley Professional, 2002.

[48] V. Cortellessa, A. Di Marco, and P. Inverardi. Integrating Performance and Reliability Analysis in a Non-Functional MDA Framework. In *Proc. of the Fundamental Approaches to Software Engineering (FASE'07)*, pages 57–71, Braga, Portugal, 2007. Springer.

[49] M. Cutumisu, C. Onuczko, D. Szafron, J. Schaeffer, M. McNaughton, T. Roy, J. Siegel, and M. Carbonaro. Evaluating Pattern Catalogs: The Computer Games Experience. In *Proc. 28th international conference on Software engineering (ICSE'06)*, pages 132–141. ACM Press, 2006.

[50] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Techniques and Applications*. Addison Wesley, 2000.

[51] K. Czarnecki and S. Helsen. Classification of Model Transformation Approaches. In *Proc. of Workshop on Generative Techniques in the Context of Model-Driven Architecture at OOPSLA 2003*, Anaheim, California, 2003. ACM.

[52] K. Czarnecki and S. Helsen. Feature-Based Survey of Model Transformation Approaches. *IBM Syst. J.*, 45(3):621–645, 2006.

[53] D. Skogan and R. Grønmo and I. Solheim. Web Service Composition in UML. In *Proc. 8th International IEEE Enterprise Distributed Object Computing Conference (EDOC)*, pages 47–57, Monterey, California, September 2004.

[54] M. Daconta, L. Obrst, and K. Smith. *The Semantic Web: A Guide to the Future of XML, Web Services, and Knowledge Management.* Wiley, 2003.

[55] V. de Castro, E. Marcos, and M. Lopez Sanz. Service Composition Modeling: A Case Study. In *Proc. Seventh Mexican International Conference on Computer Science, 2006 (ENC '06)*, pages 101–108, San Luis Potosi, Mexico, September 2006.

[56] S. Demathieu, C. Griffin, and S. Sendall. Model Transformation with the IBM Model Transformation Framework, 2005. `http://www-128.ibm.com/developerworks/rational/library/05/503_sebas/index.html`.

[57] M. D. Derk. Towards a Simpler Method of Operational Semantics for Language Definition. *SIGPLAN Not.*, 40(5):39–44, 2005.

[58] Choon Hoong Ding, Sarana Nutanong, and Rajkumar Buyya. P2P Networks for Content Sharing. *CoRR*, cs.DC/0402018, 2004.

[59] D. Elenius, G. Denker, D. Martin, F. Gilham, J. Khouri, S. Sadaati, and R. Senanayake. The owl-s editor - a development tool for semantic web services. In *Proc. Second European Semantic Web Conference*, pages 78–92, Crete, Greece, 2005.

[60] H. E. Eriksson, M. Penker, B. Lyons, and D. Fado. *UML 2 Toolkit.* Wiley, 2003.

[61] T. Erl. *Service-Oriented Architecture: A Field Guide to Integrating XML and Web Services.* Prentice-Hall, 2004.

[62] H. Espinoza, H. Dubois, S. Gerard, J. L. Medina Pasaje, and C. Murray Woodside D.C. Petriu. Annotating UML Models with Non-functional Properties for Quanti-

tative Analysis. In *Proc. MoDELS 2005 International Workshop Satellite Events*, pages 79–90, Montego Bay, Jamaica, 2005.

[63] H. Foster, S. Uchitel, J. Magee, J. Kramer, and M. Hu. Using a Rigorous Approach for Engineering Web Service Compositions: A Case Study. In *Proc. IEEE International Conference on Services Computing (SCC05)*, pages 217– 224, Florida, USA, July 2005.

[64] The Eclipse Foundation. Eclipse - An open development platform, 2006. `http://www.eclipse.org/`.

[65] The Eclipse Foundation. EMF-based UML 2.0 Metamodel Implementation, 2006. `http://www.eclipse.org/uml2/`.

[66] The Eclipse Foundation. The Eclipse BPEL Project, 2006. `http://www.eclipse.org/bpel/`.

[67] The Eclipse Foundation. The Web Standard Tools Subproject, 2006. `http://www.eclipse.org/webtools/wst/main.html`.

[68] The Eclipse Foundation. XML Schema Infoset Model, 2006. `http://www.eclipse.org/xsd/`.

[69] The Eclipse Foundation. ATL Superimposition, 2007. `http://wiki.eclipse.org/ATL_Superimposition`.

[70] The Eclipse Foundation. Atlas Transformation Langauge, 2007. `http://www.eclipse.org/m2m/atl/`.

[71] D. S. Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing*. Wiley, 2004.

[72] S. Frolund and J. Koistinen. Quality of Service Specification in Distributed Object Systems Design. In *Proceedings of the 4th USENIX Conference on ObjectOriented Technologies and Systems (COOTS)*, New Mexico, USA, April 1998.

[73] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns:Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[74] F. Garcia, M. F. Bertoa, C. Calero, A. Vallecillo, F. Ruiz, M. Piattini, and M. Genero. Towards a Consistent Terminology for Software Measurement. *Information & Software Technology*, 48(8):631–644, 2006.

[75] T. Gardner. UML Modeling of Automated Business Processes with a mapping to BPEL4WS. In *Proc. First European Workshop on Object Orientation and Web Service (EOOWS)*, Darmstadt, Germany, July 2003.

[76] Gentleware. Poseidon for UML, 2006. `http://www.gentleware.com/`.

[77] Gnome. Dia - A Drawing Program, 2006. `http://www.gnome.org/projects/dia/`.

[78] J. Gray, Y. Lin, and J. Zhang. Automating Change Evolution in Model-Driven Engineering. *IEEE Computer*, 39:51–58, 2006.

[79] Richard Grimes. .Net and DLL Hell, 2003. `http://www.ddj.com/windows/184416837`.

[80] T. J. Grose. *Mastering XMI: Java Programming with XMI, XML, and UML*. Wiley, 2002.

[81] J.J. Halliday, S. K. Shrivastava, and S. M. Wheater. Flexible Workflow Management in the OPENflow system. In *Proc. 5th IEEE/OMG International EDOC Conference (EDOC 2001)*, pages 82–92, Washington, USA, September 2001.

[82] D. Harel and B. Rumpe. Meaningful Modeling: What's the Semantics of "Semantic"? *Computer*, 37(10):64–72, 2004.

[83] N. Harrison and P. Avgeriou. Leveraging Architecture Patterns to Satisfy Quality Attributes. In *Proc. 1st European Conference on Software Architecture*, Madrid, Spain, 2007. Springer Verlag.

[84] N. B. Harrison, P. Avgeriou, and U. Zdun. "using patterns to capture architectural decisions". *IEEE Software*, 24(4):38–45, 2007.

[85] K. Hussey. Introduction to UML2 Profiles, 2006. `http://www.eclipse.org/modeling/mdt/uml2/docs/articles/Introduction_to_UML2_Profiles/article.html`.

[86] IBM. Rational Software Architect, 2006. `http://www-306.ibm.com/software/awdtools/architect/swarchitect/`.

[87] No Magic Inc. Magicdraw, 2006. `http://www.magicdraw.com/`.

[88] International Organization for Standardization. ISO 9126 Software Engineering - Product Quality, 2001-2004. `http://www.issco.unige.ch/projects/ewg96/node1.html`.

[89] Iona. IONA Artix Orchestration, 2007. `http://www.iona.com/products/artix/`.

[90] M. T. Ionita, D. K. Hammer, and H. Obbink. Scenario-Based Software Architecture Evaluation Methods: An Overview. In *Workshop on Methods and Techniques for Software Architecture Review and Assessment at the International Conference on Software Engineering*, 2002.

[91] IST. MODELling solution for softWARE systems (MODELWARE) - IST Project 511731. Technical report, IST, 2005.

[92] J. Timm and G. Gannod. A Model-Driven Approach for Specifying Semantic Web Services. In *Proc. of the 3rd IEEE International Conference on Web Services (ICWS 2005)*, pages 313–320, Orlando, Florida, 2005. IEEE.

[93] F. Jouault. Loosely Coupled Traceability for ATL. In *Proc. of Workshop on Traceability, European Conference on Model Driven Architecture (ECMDA 2005)*, 2005.

[94] F. Jouault and J. Bézivin. KM3: a DSL for Metamodel Specification. In *IFIP Int. Conf. on Formal Methods for Open Object-Based Distributed Systems, LNCS 4037*, pages 171–185. Springer, 2006.

[95] F. Jouault and I. Kurtev. On the Architectural Alignment of ATL and QVT. In *Proc. of the 2006 ACM Symposium on Applied Computing (SAC 06)*, Dijon, France, 2006. ACM Press.

[96] F. Jouault and I. Kurtev. Transforming Models with ATL. In *Proc. of Satellite Events at the MoDELS 2005 Conference*, Montego Bay, Jamaica, 2006. Springer Press.

[97] Lukasz Juszczyk, Jaroslaw Lazowski, and Schahram Dustdar. Web Service Discovery, Replication, and Synchronization in Ad-Hoc Networks. In *Proc. of the First International Conference on Availability, Reliability and Security (ARES'06)*, pages 847–854, Washington, DC, USA, 2006. IEEE Computer Society.

[98] K. Baïna and B. Benatallah and F. Casati and F. Toumani. Model-Driven Web Service Development. In *Proc. 16th International Conference on Advanced Information Systems Engineering (CAiSE)*, pages 290–306, Riga, Latvia, June 2004.

[99] A. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison Wesley, 2003.

[100] A. Knapp, N. Koch, G. Zhang, and H.M. Hassler. Modeling Business Processes in Web Applications with ArgoUWE. In *Proc. 7th International Conference on the Unified Modeling Language (UML2004)*, pages 69–83, Lisbon, Portugal, October 2004. Springer Verlag.

[101] N. Koch. Transformations Techniques in the Model-Driven Development Process of UWE. In *Proc. of 2nd Model-Driven Web Engineering Workshop*, Palo Alto, USA, July 2006.

[102] H. Koziolek and V. Firus. Empirical Evaluation of Model-Based Performance Prediction Methods in Software Development. In *Proc. of the First International Conference on the Quality of Software Architectures*, pages 188–202, Erfurt, Germany, 2005. Springer.

[103] G. Kramler, E. Kapsammer, W. Retschitzegger, and G Kappel. Towards Using UML 2 for Modelling Web Service Collaboration Protocols. In *Proc. First Interoperability of Enterprise Software and Applications (INTEROP-ESA2005)*, Geneva, Switzerland, February 2005.

[104] I Kurtev, J. Bézivin, and M Aksit. Technological Spaces: An Initial Appraisal. In *CoopIS, DOA'2002 Federated Conferences, Industrial track*, Irvine, CA, USA, 2002.

[105] I. Kurtev, J. Bézivin, F. Jouault, and P. Valduriez. Model-Based DSL Frameworks. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 602–616, New York, NY, USA, 2006. ACM Press.

[106] Christian F.J. Lange and Michel R.V. Chaudron. Managing Model Quality in UML-Based Software Development. *STEP*, 0:7–16, 2005.

[107] D. Leroux, M. Nally, and K. Hussey. UML 2: A Model-Driven Development Tool. *IBM Syst. J.*, 45(3):555–568, 2006.

[108] David Liu, Kincho H. Law, and Gio Wiederhold. Analysis of Integration Models for Service Composition. In *Proc. 3rd international workshop on Software and Performance*, pages 158–165, New York, NY, USA, 2002. ACM Press.

[109] D. Martin, M. Burstein, O. Lassila, M. Paolucci, T. Payne, and S. McIlraith. Describing Web Services using OWL-S and WSDL. DAML-S Coalition working document., 2003. `http://www.daml.org/services/owl-s/1.0/owl-s-wsdl.html`.

[110] D. Martin, M. Paolucci, S. McIlraith, M. Burstein, D. McDermott, D. McGuinness, T. Payne B. Parsia, M. Sabou, M. Solanki, N. Srinivasan, and K. Sycara. Bringing Semantics to Web Services: The OWL-S Approach. In *Proc. First International Workshop on Semantic Web Services and Web Process Composition (SWSWPC 2004)*, San Diego, California, USA, 2004.

[111] Martin Matula. NetBeans Metadata Repository, 2003. `http://mdr.netbeans.org/MDR-whitepaper.pdf`.

[112] M.C. Jaeger and L. Engel and K. Geihs. A Methodology for Developing OWL-S Descriptions. In *Proc. First International Conference on Interoperability of Enterprise Software and Applications Workshop on Web Services and Interoperability*, Geneva, Switzerland, 2005.

[113] D.L. McGuinness and F. van Harmelen. OWL Web Ontology Language Overview. W3C Recommendation 10 February 2004., 2004. `http://www.w3.org/TR/owl-features/`.

[114] N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, January 2000.

[115] S. Meliá, A. Kraus, and N. Koch. MDA Transformations Applied to Web Application Development. In *Proc. 5th International Conference on Web Engineering (ICWE 2005)*, pages 465–471, Sydney, Australia, 2005. Springer Verlag.

[116] D. A. Menascé. QoS Issues in Web Services. *IEEE Internet Computing*, 6(6):72–75, November–December 2002.

[117] M. Mernik, J. Heering, and A. M. Sloane. When and How to Develop Domain-Specific Languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.

[118] R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999.

[119] N. Minar. Distributed Systems Topologies, 2001. `http://www.openp2p.com/pub/a/p2p/2001/12/14/topologies_one.html`.

[120] D. Misic. Authoring UML profiles using Rational Software Architect and Rational Software Modeler, 2005. `http://www.ibm.com/developerworks/rational/library/05/0906_dusko/`.

[121] P.V. Mockapetris and K.J. Dunlap. Development of the domain name system. In *Proc. ACM SIGCOMM*, pages 123–133, CA, USA, 1998.

[122] R. Monson-Haefel, B. Burke, and S. Labourey. *Enterprise JavaBeans, 4th Edition*. O'Reilly Media, 2004.

[123] N. Moreno, P. Fraternali, and A. Vallecillo. A UML 2.0 Profile for WebML Modelling. In *Proc. of the 2nd Model-Driven Web Engineering Workshop (MDWE)*, Palo Alto, California, July 2006. ACM Digital Library.

[124] N. Aizenbud-Reshef and B. T. Nolan and J. Rubin and Y. Shaham-Gafni. Model Traceability. *IBM Syst. J.*, 45(3):515–526, 2006.

[125] G. Zhang N. Koch and M.J. Escalona. Model Transformations from Requirements to Web System Design. In *Proc. 6th International Conference on Web Engineering (ICWE 2006)*, pages 281–288, Palo Alto, USA, July 2006. ACM.

[126] Mangala Gowri Nanda, Satish Chandra, and Vivek Sarkar. Decentralizing Execution of Composite Web Services. In *Proc. 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 170–187, New York, NY, USA, 2004. ACM Press.

[127] OASIS. UDDI Version 3.0.2, 2004. `http://uddi.org/pubs/uddi_v3.htm`.

[128] OASIS. Web Services Business Process Execution Language Version 2.0 (Draft), 2006. `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-specification-draft.pdf`.

[129] OMG. Common Object Request Broker Architecture (CORBA/IIOP). Technical report, OMG, 2002.

[130] OMG. UML Profile for CORBA, v 1.0. Technical report, OMG, 2002.

[131] OMG. UML Profile for Schedulability, Performance, and Time Specification. Technical report, OMG, 2003.

[132] OMG. UML Profile for enterprise distributed Object Computing (EDOC). Technical report, OMG, 2004.

[133] OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification Final Adopted Specification. Technical report, OMG, 2005.

[134] OMG. Meta Object Facility (MOF) 2.0 XMI Mapping Specification, v2.1. Technical report, OMG, 2005.

[135] OMG. UML Profile for QoS and Fault Tolerance. Technical report, OMG, 2005.

[136] OMG. Business Process Modeling Notation (BPMN) 1.0, 2006. `http://www.bpmn.org/Documents/BPMN%20V1-0%20May%203%202004.pdf`.

[137] OMG. Meta Object Facility Core Specification version 2.0. Technical report, OMG, 2006.

[138] OMG. Object Constraint Language Specification, version 2.0. Technical report, OMG, 2006.

[139] OMG. UML Profile for Modeling Quality of Service and Fault Tolerance. Technical report, OMG, 2006.

[140] OMG. Unified Modeling Language (UML) 2.1.2 Superstructure Specification. Technical report, OMG, 2007.

[141] Oracle. Oracle BPEL Process Manager, 2007. `http://www.oracle.com/technology/products/ias/bpel/index.html`.

[142] P. Bengtsson and N. Lassing and J. Bosch and H. van Vliet. Architecture-Level Modifiability Analysis (ALMA). *Journal of Systems and Software*, 69(1-2):129–147, 2004.

[143] C. Pahl, S. Giesecke, and W. Hasselbring. An Ontology-based Approach for Modelling Architectural Styles. In *Proc. 1st European Conference on Software Architecture (ECSA 2007)*, Madrid, Spain, September 2007.

[144] C. Peltz. Web Service Orchestration and Choreography: A Look at WSCI and BPEL4WS. *Web Service Journal*, July 2003.

[145] C. Peltz. Web Services Orchestration and Choreography. *IEEE Computer*, 36, 2003.

[146] K. Pfadenhauer, S. Dustdar, and B. Kittl. Challenges and Solutions for Model Driven Web Service Composition. In *Proc. of 3rd International Workshop on Distributed and Mobile collaboration (DMC)*, pages 126–131, Linköping, Sweden, 2005. IEEE Press.

[147] D. S. Platt. *Introducing Microsoft .Net, Third Edition.* Microsoft Press, 2003.

[148] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical report, Dept. of Computer Science, Univ. of Aarhus, 1981.

[149] Remko Popma. JET Tutorial Part 1 (Introduction to JET), 2004. `http://www.eclipse.org/articles/Article-JET/jet_tutorial1.html`.

[150] S. Powers. *Practical RDF.* O'Reilly, 2003.

[151] Will Provost. UML for Web Services, 2003. `http://webservices.xml.com/pub/a/ws/2003/08/05/uml.html?page=1`.

[152] R. Grønmo and I. Solheim. Towards Modeling Web Service Composition in UML. In *Proc. 2nd International Workshop on Web Services: Modeling, Architecture and Infrastructure (WSMAI-2004)*, pages 72–86, Porto, Portugal, April 2004.

[153] R. Grønmo and J. Oldevik. An Empirical Study of the UML Model Transformation Tool (UMT). In *Proc. First Interoperability of Enterprise Software and Applications (INTEROP-ESA2005)*, Geneva, Switzerland, February 2005.

[154] R. Grønmo and M. Jaeger. Model-Driven Semantic Web Service Composition. In *Proc. of the 12th Asia-Pacific Software Engineering Conference (APSEC 2005)*, pages 79–86, Taipei, Taiwan, 2005. IEEE.

[155] R. Lucchi and M. Mazzara. A Pi-calculus Based Semantics for WS-BPEL. *Journal of Logic and Algebraic Programming (JLAP)*, 70:96–118, 2005.

[156] F. Ranno, S. Wheater, and S. Shrivastava. A System for Specifying and Co-ordinating the Execution of Reliable Distributed Applications. In *Proc. Distributed Applications and Interoperable Systems (DAIS'97)*, Cottbus, Germany, 1997.

[157] M. Ripeanu. Peer-to-Peer Architecture Case Study: Gnutella Network. In *First International Conference on Peer-to-Peer Computing*, pages 99–100, Linköpings, Sweden, Auguest 2001.

[158] Richard Torbjørn Sanders, Humberto Nicolás Castejón, Frank Alexander Kraemer, and Rolv Bræk. Using UML 2.0 Collaborations for Compositional Service Specification. In Lionel Briand and Clay Williams, editors, *Proc. ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, volume 3713 of *LNCS*, pages 460–475, Montego Bay, Jamaica, October 2005. Springer-Verlag.

[159] D. A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* William C. Brown Publishers, Dubuque, IA, USA, 1986.

[160] B. Selic. Model-Driven Development: Its Essence and Opportunities. In *Proc. IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2006)*, pages 313–319, Gyeongju, Korea, 2006. IEEE.

[161] B. Selic. UML 2: A Model-Driven Development Tool. *IBM Syst. J.*, 45(3):607–620, 2006.

[162] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice-Hall, 1996.

[163] Q. Z. Sheng, B. Benatallah, and M. Dumas. SELF-SERV: A Platform for Rapid Composition of Web Services in a Peer-to-Peer Environment. In *Proc. 28th International Conference on Very Large Data Bases*, pages 1051–1054, Hong Kong, China, August 2002.

[164] S. Shrivastava and S. Wheater. Architectural Support for Dynamic Reconfiguration of Large Scale Distributed Applications. In *Proc. International Conference on Configurable Distributed Systems (CDS'98)*, Maryland, USA, 1998.

[165] E. Sirin, J. Hendler, and B. Parsia. Semi-automatic Composition of Web Services using Semantic Descriptions. In *Proc. Workshop on Web Services: Modeling, Architecture and Infrastructure (WSMAI)*, pages 17–24, Angers, France, 2003. ICEIS.

[166] P. Sobe. Fault-Tolerant Web Services on a Computing Cluster. In *Proc. Third European Dependable Computing Conference (EDCC-3)*, Prague, Czech Republic, September 1999.

[167] IEEE Computer Society. IEEE Recommended Practice for Architectural Description of Software-Intensive Systems. Technical report, IEEE Std 1472000, 2000.

[168] P. Sriplakich, X. Blanc, and M. Gervais. Supporting Transparent Model Update in Distributed CASE Tool Integration. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1759–1766, New York, NY, USA, 2006. ACM Press.

[169] Stanford Medical Informatics. Protégé Ontology Editor and Knowledge-Base Framework, 2006. `http://protege.stanford.edu/`.

[170] S.Thöne and R.Depke and G.Engels. Process-Oriented, Flexible Composition of Web Services with UML. In *Proc. Joint Workshop on Conceptual Modeling Approaches for e-Business (eCOMO 2002)*, Tampere, Finland, October 2002.

[171] Sun. Java Metadata Interface (JMI), 2006. `http://java.sun.com/products/jmi/`.

[172] Sun. Java Server Pages, 2006. `http://java.sun.com/products/jsp/reference/api/index.html`.

[173] C. Szyperski. *Component Software - Beyond Object-Oriented Programming Second Edition*. Addison-Wesley, 2002.

[174] A. S. Tanenbaum and M. van Steen. *Distributed Systems - Principles and Paradigms*. Prentice-Hall, 2002.

[175] Information Society Technologies. The Software and Services Challenge, 2006. `ftp://ftp.cordis.lu/pub/ist/docs/directorate_d/st-ds/fp7-report_en.pdf`.

[176] France Telecom. SmartQVT, 2006. `http://smartqvt.elibel.tm.fr/`.

[177] The Python Software Foundation (PSF). The Python Programming Language, 2006. `http://www.python.org/`.

[178] Tigris.org. Argouml, 2006. `http://argouml.tigris.org/`.

[179] W. M. P. van der Aalst, M. Dumas, and A. H. M. ter Hofstede. Web Service Composition Languages: Old Wine in New Bottles? In *Proc. 29th EUROMICRO Conference*, pages 298–307, Belek-Antalya, Turkey, September 2003.

[180] W.M.P. van der Aalst, B. Kiepuszewski A.H.M. ter Hofstede, and A.P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14:5–51, 2003.

[181] M. Vasko and S. Duskar. An Analysis of Web Services Flow Patterns in Collaxa. In L.J. Zhang and M. Jeckle, editors, *Proc. European Conference on Web Services ECOWS 2004*, pages 1–14. Springer-Verlag, LNCS 3250, 2004.

[182] M. Voelter and T. Stahl. *Model-Driven Software Development.* Wiley, 2006.

[183] W3C. XSL Transformations (XSLT) Version 1.0, 1999. `http://www.w3.org/TR/xslt`.

[184] W3C. Web Services Description Language (WSDL) 1.1, 2001. `http://www.w3.org/TR/wsdl`.

[185] W3C. Web Services Architecture, 2002. `http://www.w3.org/TR/2002/WD-ws-arch-20021114/`.

[186] W3C. SOAP Version 1.2, 2003. `http://www.w3.org/TR/soap/`.

[187] W3C. Web Services Choreography Description Language Version 1.0, 2004. `http://www.w3.org/TR/2004/WD-ws-cdl-10-20041217/`.

[188] W3C. Extensible Markup Language (XML) 1.0 (Fourth Edition), 2006. `http://www.w3.org/TR/REC-xml/`.

[189] Qing Wang, Yang Yuan, Junmei Zhou, and Aoying Zhou. Peer-Serv: A Framework of Web Services in Peer-to-Peer Environment. In *Advances in Web-Age Information Management*, pages 298–305. Springer-Verlag, LNCS 2762, August 2003.

[190] J. Warmer and A. Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA, Second Edition.* Addison-Wesley Professional, 2003.

[191] WebRatio. Webratio, 2006. `http://www.webratio.com/`.

[192] Niklaus Wirth. What Can We Do About The Unnecessary Diversity of Notation for Syntactic Definitions? *Communications of the ACM*, 20(11):822–823, 1977.

[193] S. J. Woodman, D. J. Palmer, S. K. Shrivastava, and S. M. Wheater. A System for Distributed Enactment of Composite Web Services. In *Work in progress report, Int. Conf. on Service Oriented Computing*, Trento, Italy, December 2003.

[194] S. J. Woodman, D. J. Palmer, S. K. Shrivastava, and S. M. Wheater. Notations for the Specification and Verification of Composite Web Services. In *Proc. 8th IEEE International Enterprise Distributed Object Computing Conference (EDOC '04)*, pages 35–46, Monterey, California, September 2004.

[195] WSMO Working Group. Web Service Modeling Language, 2006. `http://www.wsmo.org/wsml/index.html`.

[196] L. Zeng, B. Benatallah, M. Dumas, J. Kalagnanam, and Q.Z. Sheng. Quality Driven Web Services Composition. In *Proc. of the 12th International Conference on the World Wide Web*, Budapest, Hungary, 2003. ACM Press.

[197] J. Zhang, J. Chung, and C. K. Chang. Migration to Web Services Oriented rchitecture: A Case Study. In *Proc. 2004 ACM symposium on Applied Computing (SAC'04)*, pages 1624–1628, New York, NY, USA, 2004. ACM Press.

[198] X. Zhou, W. T. Tsai, X. Wei, Y. Chen, and B. Xiao. Pi4SOA: A Policy Infrastructure for Verification and Control of Service Collaboration. *icebe*, 0:307–314, 2006.

Appendix A

ATL Helper Definitions

```
--Get the order value applied to the UML ControlFlow going into a UML CallBehaviorAction
helper def : getCBAOrder(cba : UML!CallBehaviorAction) : Integer =
  --We assume only one ControlFlow enters a CBA, this is a reasonable assumption
  cba.getIncomings()->select(e|e.oclType()= UML!ControlFlow).at(1).getValue(cba.getIncomings()
  ->select(e|e.oclType()= UML!ControlFlow).at(1).getAppliedStereotypes().first(),'order');

--Retrieve the number of UML CallBehaviourActions there are in a UML Activity
helper def : getCBACount() : Integer =
  UML!CallBehaviorAction.allInstances().size();

--Based on a UML CallBehaviorAction's related UML ControlFlow order value, retrieve the next
--UML CallBehaviorAction in the chain
helper def : getNextCBA(cba : UML!CallBehaviorAction) : String =
  --ensure there is another CBA in the chain with an order greater than this one
   if thisModule.getCBAOrder(cba) < thisModule.getCBACount()
   then
    --Get the name of the next CBA in the chain
    UML!CallBehaviorAction.allInstances()->select(e|e.getIncomings()->select(e|e.oclType()
    = UML!ControlFlow).at(1).getValue(cba.getIncomings()->select(e|e.oclType()
    = UML!ControlFlow).at(1).getAppliedStereotypes().first(),'order')
    = thisModule.getCBAOrder(cba)+1).first().name
   else
    --there are no more CBAs in the chain so return an empty string
    ''
   endif;

--Get a UML Pin based on a UML ObjectFlow
helper def : getPin(of : UML!ObjectFlow, dir : String) : UML!Pin =
   if(dir = 'source')
   then
    --Check to see if the ObjectFlow comes from the activity start node, if it does
    --we handle it differently i.e. Return the name of the UML Pin it targets
    if of.getSource().oclType() = UML!InitialNode
    then
     --return the name of the UML Pin to which the ObjectFlow terminates
      of.getTarget()
    else
     --return the name of the UML Pin from which the ObjectFlow originates
      of.getSource()
    endif
   else
    --Check to see if the ObjectFlow goes to the activity end node, if it does
    --we handle it differently i.e. Return the name of the UML Pin it comes from
    if of.getTarget().oclType() = UML!ActivityFinalNode
    then
     --return the UML Pin from which the ObjectFlow originates
      of.getSource()
    else
     --return the UML Pin to which the ObjectFlow terminates
      of.getTarget()
    endif
   endif;
```

Figure A.1: ATL helper definitions for UML to DPL, part 1.

```
--Retrieve the UML Type as a string, without the model name prefix
helper def : removeMMPrefix(type : UML!Type) : String =
  type.toString().split('!').at(2);

--Retrieve all the UML Activity Partitions as a sequence
helper def : getActivityPartitions() : Sequence(UML!ActivityPartition) =
  UML!ActivityPartition.allInstances();

--Correlation variables only exist on certain Distribution Patterns, check what pattern
--is being applied before creating the correlation variables. Correlation variables are
--used to persist data accross certain patterns
--**Pattern Specific**
helper def : getCorrelationVariables() : DPL!CorrelationVariables =
  --note sequences start at 1!
  if UML!Activity.allInstances()->collect(e | e.getValue(e.getAppliedStereotypes()
    .first(),'distribution-pattern')).at(1).toString() = 'hub-and-spoke'
  then
    --Hub & Spoke DPs do not have correlation variables so return an empty set
    Set{}
  else
    --These DPs do have correlation variables so retrieve all the correlation
    --variables as set on the model
    UML!Pin.allInstances()->select(e | e.getValue(e.getAppliedStereotypes()
     .first(),'is_collaboration_variable'))->collect(e|thisModule.PinToCorrelationVariable(e))
  endif;

--Construct the name of the DPL Mapping source based on UML CallBehaviorAction the UML ObjectFlow
--originates from
helper def : getMapping(of : UML!ObjectFlow, dir : String) : String =
  if(dir = 'source')
  then
    --Check to see if the source of the ObjectFlow is the start node, if it is then we
    --return a special case value to signify this
    if of.getSource().oclType() = UML!InitialNode
    then
      'InitialNode'
    else
      --return the name of the CBA which this ObjectFlow originates along with a constant string
      of getSource().getInPartitions().first().name + of.getSource().getOwner().name+'Response'
    endif
  else
    --Check to see if the target of the ObjectFlow is the end node, if it is then we
    --return a special case value to signify this
    if of.getTarget().oclType() = UML!ActivityFinalNode
    then
      'FinalNode'
    else
      --return the name of the CBA which this ObjectFlow terminates at along with a constant string
      of.getTarget().getInPartitions().first().name + of.getTarget().getOwner().name+'Request'
    endif
  endif;
```

Figure A.2: ATL helper definitions for UML to DPL, part 2.

```
--Retrieve the base-namespace string value from the DPL pattern-definition type
helper def: getBaseNamespace() : String =
  DPL!"pattern-definition".allInstances()->collect(e|e."base-namespace".text).first().toString();

--Retrieve the service-name string value from the DPL pattern-definition type
helper def: getServiceName() : String =
  DPL!"pattern-definition".allInstances()->collect(e|e."service-name".text).first().toString();

--Retrieve the operation-name string value from the DPL pattern-definition type. This value
--is used to identify the operation name for the composition
helper def: getOperationName() : String =
  DPL!"pattern-definition".allInstances()->collect(e|e."operation-name".text).first().toString();

--Retrieve the namespace-prefix string value from the DPL pattern-definition type
helper def: getNamespacePrefix() : String =
  DPL!"pattern-definition".allInstances()->collect(e|e."namespace-prefix".text).first().toString();

--Retrieve the first DPL Operation type based upon it's order value
helper def: getFirstOperation() : DPL!Operation =
  DPL!Operations.allInstances()->collect(e|e.operation)->flatten()->sortedBy(e|e.order).first();

--Retrieve the DPL Node which contains the first DPL Operation type based upon it's order value
helper def: getNodeContainingFirstOperation() : DPL!Node =
  thisModule.getFirstOperation().eContainer().eContainer();
```

Figure A.3: ATL helper definitions for DPL to BPEL.

```
--Retrieve the service-name string value from the DPL pattern-definition type
helper def: getServiceName() : String =
  DPL!"pattern-definition".allInstances()->collect(e|e."service-name".text).first().toString();

--Retrieve the distribution-pattern string value from the DPL pattern-definition type
helper def: getDistributionPattern() : String =
  DPL!"pattern-definition".allInstances()->collect(e|e."distribution-pattern".text).first().toString();

--Retrieve the base-namespace string value from the DPL pattern-definition type
helper def: getBaseNamespace() : String =
  DPL!"pattern-definition".allInstances()->collect(e|e."base-namespace".text).first().toString();

--Retrieve the operation-name string value from the DPL pattern-definition type
helper def: getOperationName() : String =
  DPL!"pattern-definition".allInstances()->collect(e|e."operation-name".text).first().toString();

--Retrieve the namespace-prefix string value from the DPL pattern-definition type
helper def: getNamespacePrefix() : String =
  DPL!"pattern-definition".allInstances()->collect(e|e."namespace-prefix".text).first().toString();

--Change the Ecore type identifier passed into to a WSDL type and return
helper def: convertETypeToWSDLType(type : String) : String =

   if type = 'EString'
   then
     'xsd:string'
   else
     if type = 'EInt'
     then
       'xsd:int'
     else
       if type = 'EBoolean'
       then
         'xsd:boolean'
       else
         'xsd:error'
       endif
     endif
   endif;
```

Figure A.4: ATL helper definitions for DPL to WSDL.

```
--Retrieve the service-name string value from the DPL pattern-definition type
helper def: getServiceName() : String =
  DPL!"pattern-definition".allInstances()->collect(e|e."service-name".text).first().toString();

--Retrieve the namespace-prefix string value from the DPL pattern-definition type
helper def: getNamespacePrefix() : String =
  DPL!"pattern-definition".allInstances()->collect(e|e."namespace-prefix".text).first().toString();

--Retrieve the base-namespace string value from the DPL pattern-definition type
helper def: getBaseNamespace() : String =
  DPL!"pattern-definition".allInstances()->collect(e|e."base-namespace".text).first().toString();
```

Figure A.5: ATL helper definitions for DPL to PDD.

```
--Retrieve the service-name string value from the DPL pattern-definition type
helper def: getServiceName() : String =
  DPL!"pattern-definition".allInstances()->collect(e|e."service-name".text).first().toString();

--Retrieve all the DPL Nodes as a sequence
helper def : getAllNodes() : Sequence(UML!Node) =
  DPL!Node.allInstances();
```

Figure A.6: ATL helper definitions for DPL to WSDLCatalog.

```
--Convert true to 'yes' and false to 'no'
helper def : convertBooleanToString(boolean : Boolean) : String =
  if boolean = true
  then
    --
     'yes'
  else
    --
     'no'
  endif;

--Retrieve all the BPEL Namespaces and return as a sequence
helper def : getAllNamespaces() : Sequence(BPEL!Namespace) =
  BPEL!Namespace.allInstances();

--Get the BPEL activity type by parsing the OCL type, this is necessary
--as there is no explicit "type" attibute on activities in the meta-model
--For example the OCL type might be BPEL!Invoke, then this will return invoke
helper def : getActivityType(type : String) : String =
  --convert the OCL type to a string, split it on the !, get the 2nd part and make
  --it lower case then return
  type.toString().split('!').at(2).toLower();
```

Figure A.7: ATL helper definitions for BPEL to XML.

```
--Retrieve all the PDD PartnerLinks and return as a sequence
helper def : getAllPartnerLinks() : Sequence(BPEL!PartnerLink) =
  PDD!PartnerLink.allInstances();

--Retrieve all the PDD WSDL references and return as a sequence
helper def : getAllWSDLReferences() : Sequence(BPEL!WSDL) =
  PDD!WSDL.allInstances();
```

Figure A.8: ATL helper definitions for PDD to XML.

```
--Retrieve all the WSDLCatalog's WSDLEntries as a sequence
helper def : getAllWSDLEntries() : Sequence(WSDLCatalog!WSDLEntry) =
  WSDLCatalog!WSDLEntry.allInstances();
```

Figure A.9: ATL helper definitions for WSDLCatalog to XML.

Appendix B

Additional ATL Definitions

```
module XMLtoUML; -- Module Template
create OUT : UML from IN : XML, IN2 : XML, IN3 : XML;

--Check to see if a construct (self) is unique across all input models
--return true if the construct is unique, otherwise false
helper context XML!Root def: isUnique : Boolean =
    self.uniqueValue = self;

helper context XML!Root def: key : OclAny =
    self.name;

helper def: rootByKey : Map(OclAny, XML!Root) =
   XML!Root.allInstances()->iterate(e; acc : Map(OclAny, XML!Root) = Map {} |
     acc.including(e.key.debug('key:'), e.debug('value:'))
   );

--Retrieve a construct from a Map
helper context XML!Root def: uniqueValue : XML!Root =
    thisModule.rootByKey.get(self.key);

helper def: getMessages() : XML!Element=
  XML!Element.allInstances()->select(e|e.name='wsdl:message').debug('x');

rule ProcessToRoot{
  from
    r: XML!Root (
      r.isUnique
    )
  using{
      --merge all the input models
      a:Set(XML!Root) = XML!Root.allInstancesFrom('IN')->union(XML!Root.allInstancesFrom('IN2'))
      ->union(XML!Root.allInstancesFrom('IN3'));
    }
  to
    rt : UML!Model(
      packagedElement <- a->collect(e|e.children->select(c|c.name='wsdl:portType')
      ->collect(e|thisModule.PortTypeToInterface(e)))
    )
}

lazy rule PortTypeToInterface{
  from
    pt: XML!Element(
      pt.name = 'wsdl:portType'
    )
  to
    inf : UML!Interface(
      name <- pt.children->select(e|c.name='name')->first().value,
      ownedOperation <- pt.children->select(c|c.name='wsdl:operation')
      ->collect(e|thisModule.OperationToOperation(e))
    )
}

lazy rule OperationToOperation{
  from
    opx: XML!Element
  to
    opu : UML!Operation(
      name <- opx.children->select(c|c.name='name')->first().value
    )
}

lazy rule PartToParameter{
  from
    ip : XML!Element
  to
    para : UML!Parameter(
    )
}
```

Figure B.1: ATL definition for transfroming XML to a UML 2.0 Class diagram.

```
module ClassToActivity; -- Module Template
create OUT : ACTIVITY from IN : CLASS;

--Transform a UML Class Diagram Model to UML Activity diagram Activity
rule ModelToActivity {
  from
    m : CLASS!Model
  to
    act : ACTIVITY!Activity(
      --assign all the interfaces to the activity

      group <- CLASS!Interface.allInstances(),
      node <- CLASS!CallBehaviorAction.allInstances(),
      node <- CLASS!Pin.allInstances()
    )
}

--Transform a UML Class Diagram Interface to UML Activity Diagram ActivityPartition
rule InterfaceToActivityPartition {
  from
    i : CLASS!Interface
  to
    part : ACTIVITY!ActivityPartition(
      name <- i.name,
      node <- i.getOwnedOperations()
    )
}

--Transform a UML Class Diagram Operation to UML Activity Diagram CallBehaviorAction
rule OperationToCallBehaviorAction {
  from
    o : CLASS!Operation
  to
    cba : ACTIVITY!CallBehaviorAction(
      name <- o.name,
      inPartition <- cba.getInPartition(o.getInterface().name),

      --fix emun shouldn't be using toStirng()
      argument <- o.getOwnedParameters()->select(e|e.getDirection().toString()
      = 'in' or e.getDirection() = #inout)->collect(e|thisModule.ParameterToInputPin(e)),

      --result can only create an outputpin
      result <- o.getOwnedParameters()->select(e|e.getDirection().debug('s')
      = #out or e.getDirection() = #inout)->collect(e|thisModule.ParameterToOutputPin(e))
    )
}

--Transform a UML Class Diagram Parameter to UML Activity Diagram OutputPin
lazy rule ParameterToInputPin {
  from
    p : CLASS!Parameter
  to
    pin : ACTIVITY!InputPin(
      name <- p.name

      --need to model type
      --inPartition
    )
}

--Transform a UML Class Diagram Parameter to UML Activity Diagram OutputPin
lazy rule ParameterToOutputPin {
  from
    p : CLASS!Parameter
  to
    pin : ACTIVITY!OutputPin(
      name <- p.name

      --need to model type
      --inPartition
    )
}
```

Figure B.2: ATL definition for transfroming a UML 2.0 Class diagram to a UML 2.0 Activity diagram.

```
module DPLValidation; -- Module Template
create OUT : DPL from IN : DPL;

--Return the number of node operations that are not compatible with the pattern
helper def : checkPatternNodeOperationConsistency(pd : DPL!"pattern-definition") : Integer =
  if pd."distribution-pattern".text = 'hub-and-spoke'
  then
    --check to ensure the node operation roles are compatible with this pattern type
    --i.e. all roles are either hub or spokes
    DPL!Operation.allInstances()->select(e|e.role<>'hub' and e.role<>'spoke')->size()
  else
    if pd."distribution-pattern".text = 'peer-to-peer'
    then
      --check to ensure the node operation roles are compatible with this pattern type
      --i.e. all roles are either hub or spokes
      DPL!Operation.allInstances()->select(e|e.role<>'peer')->size()
    else
      --if no pattern was matched return -1 to indicate an error
      -1
    endif
  endif;

 --Return the number of nodes
  helper def : checkNodeOperationCount(pd : DPL!"pattern-definition") : Integer =
    DPL!Operation.allInstances()->size();

rule PatternDefinitionToProcess {
  from
    pd : DPL!"pattern-definition"
  do
  {
   --run the validators
    'This is the TOPMAN validator:'.println();
    'Number of node operations: '.concat(thisModule.checkNodeOperationCount(pd).toString()).println();
    'Number of incompatible node operations: '
     concat(thisModule.checkPatternNodeOperationConsistency(pd).toString()).println();
  }

}
```

Figure B.3: ATL script for validating DPL model.

Appendix C

ANT Task Definitions

```
<!--convert all the wsdl documents to ecore representations using the AM3 injector-->
<target name="IJ_WSDLToXML">
  <am3.loadModel modelHandler="EMF" name="XML" metamodel="MOF" path="MetaModels/XML.ecore"/>
  <am3.loadModel name="out" metamodel="XML" path="Input/CoreBanking.wsdl">
    <injector name="xml" />
  </am3.loadModel>
  <am3.saveModel model="out" path="Output/CoreBanking.ecore"/>
  <am3.loadModel name="out" metamodel="XML" path="Input/RiskManagement.wsdl">
    <injector name="xml" />
  </am3.loadModel>
  <am3.saveModel model="out" path="Output/RiskManagement.ecore"/>
  <am3.loadModel name="out" metamodel="XML" path="Input/CreditCard.wsdl">
    <injector name="xml" />
  </am3.loadModel>
  <am3.saveModel model="out" path="Output/CreditCard.ecore"/>
</target>
```

Figure C.1: ANT script to convert WSDL interfaces to XML based ECore model.

```
<!--load all the ecore representations of the WSDLs and supporting metamodels-->
<target name="LM_WSDLToUMLClass">
  <am3.loadModel modelHandler="EMF" name="UML2" metamodel="MOF"
    nsURI="http://www.eclipse.org/uml2/2.0.0/UML"/>
  <am3.loadModel modelHandler="EMF" name="XML" metamodel="MOF" path="MetaModels/XML.ecore"/>
  <am3.loadModel modelHandler="EMF" name="wsdl1" metamodel="XML" path="Output/CoreBanking.ecore"/>
  <am3.loadModel modelHandler="EMF" name="wsdl2" metamodel="XML" path="Output/RiskManagement.ecore"/>
  <am3.loadModel modelHandler="EMF" name="wsdl3" metamodel="XML" path="Output/CreditCard.ecore"/>
</target>

<!--convert all the ecore representations of the WSDLs to a single Class diagram-->
<target name="TF_WSDLToUMLClass" description="Perform WSDLToUMLClass transform"
  depends="LM_WSDLToUMLClass">
  <am3.atl path="Transforms/XMLtoUML.atl">
    <inModel name="XML" model="XML"/>
    <inModel name="IN" model="wsdl1"/>
    <inModel name="IN2" model="wsdl2"/>
    <inModel name="IN3" model="wsdl3"/>
    <inModel name="UML" model="UML2"/>
    <outModel name="OUT" model="outModel" metamodel="UML2" path="Models/Service_Interfaces.uml"/>
  </am3.atl>
  <am3.saveModel model="outModel" path="Models/Service_Interfaces.uml"/>
</target>
```

Figure C.2: ANT script to convert XML based model to a UML 2.0 Class diagram model.

```
<!--load all the UML Class Diagram and the UML2 metamodel-->
<target name="LM_UMLClassToUMLActivity">
  <am3.loadModel modelHandler="EMF" name="UML2" metamodel="MOF"
    nsURI="http://www.eclipse.org/uml2/2.0.0/UML"/>
  <am3.loadModel modelHandler="EMF" name="inModel" metamodel="UML2" path="Models/Service_Interfaces.uml"/>
</target>

<!--convert the UML Class diagram to a UML Activity diagram-->
<target name="TF_UMLClassToUMLActivity" description="Perform ClassToActivity transform"
  depends="LM_UMLClassToUMLActivity">
  <am3.atl path="Transforms/ClassToActivity.atl">
    <inModel name="IN" model="inModel"/>
    <inModel name="CLASS" model="UML2"/>
    <inModel name="ACTIVITY" model="UML2"/>
    <outModel name="OUT" model="outModel" metamodel="UML2" path="Models/UMLActivityDiagram.uml"/>
  </am3.atl>
  <am3.saveModel model="outModel" path="Models/UMLActivityDiagram.uml"/>
</target>
```

Figure C.3: ANT script to convert a UML 2.0 Class diagram model to a UML 2.0 Activity diagram model.

```
<target name="LM_UMLActivityToDPL">
  <am3.loadModel modelHandler="EMF" name="UML2" metamodel="MOF"
    nsURI="http://www.eclipse.org/uml2/2.0.0/UML"/>
  <am3.loadModel modelHandler="EMF" name="DPL" metamodel="MOF" path="MetaModels/DPL.ecore"/>
  <am3.loadModel modelHandler="EMF" name="inModel" metamodel="UML2" path="UMLActivityDiagram.uml"/>
</target>

<target name="TF_UMLActivityToDPL" description="Perform UMLActivityToDPL transform"
  depends="LM_UMLActivityToDPL">
  <am3.atl path="Transforms/UMLActivityToDPL.atl">
    <inModel name="UML" model="UML2"/>
    <inModel name="IN" model="inModel"/>
    <inModel name="DPL" model="DPL"/>
    <outModel name="OUT" model="outModel" metamodel="DPL" path="Models/DPL_Instance.ecore"/>
  </am3.atl>
  <am3.saveModel model="outModel" path="Models/DPL_Instance.ecore"/>
</target>
```

Figure C.4: ANT script to convert a UML Activity diagram model to a DPL based model.

```
<!--load the DPL metamodel and the input model-->
<target name="LM_ValidateDPL">
  <am3.loadModel modelHandler="EMF" name="DPL" metamodel="MOF" path="MetaModels/DPL.ecore"/>
  <am3.loadModel modelHandler="EMF" name="inModel" metamodel="DPL" path="Output/DPL_Instance.ecore"/>
</target>

<!--perform the validation of the DPL model -->
<target name="TF_ValidateDPL" description="Perform DPL validation" depends="LM_ValidateDPL">
  <am3.atl path="Transforms/DPLValidation.atl">
    <inModel name="IN" model="inModel"/>
    <inModel name="DPL" model="DPL"/>
  </am3.atl>
</target>
```

Figure C.5: ANT script for validating a DPL model.

```
<target name="LM_DPLToBPEL">
  <am3.loadModel modelHandler="EMF" name="DPL" metamodel="MOF" path="MetaModels/DPL.ecore"/>
  <!--BPEL meta-model references the WSDL meta-model so we must load this too-->
  <am3.loadModel modelHandler="EMF" name="WSDL" metamodel="MOF" path="MetaModels/WSDL.ecore"/>
  <am3.loadModel modelHandler="EMF" name="BPEL" metamodel="MOF" path="MetaModels/BPEL.ecore"/>
  <am3.loadModel modelHandler="EMF" name="inModel" metamodel="DPL" path="Models/DPL_Instance.ecore"/>
</target>

<target name="TF_DPLToBPEL" description="Perform DPLToBPEL transform" depends="LM_DPLToBPEL">
  <am3.atl path="Transforms/DPLToBPEL.atl">
    <inModel name="DPL" model="DPL"/>
    <inModel name="IN" model="inModel"/>
    <inModel name="WSDL" model="WSDL"/>
    <inModel name="BPEL" model="BPEL"/>
    <outModel name="OUT" model="outModel" metamodel="BPEL" path="Models/BPEL_Instance.ecore"/>
  </am3.atl>
  <am3.saveModel model="outModel" path="Models/BPEL_Instance.ecore"/>
</target>
```

Figure C.6: ANT script to convert a DPL based model to a WS-BPEL based model.

```
<target name="LM_DPLToWSDL">
  <am3.loadModel modelHandler="EMF" name="DPL" metamodel="MOF" path="MetaModels/DPL.ecore"/>
  <am3.loadModel modelHandler="EMF" name="WSDL" metamodel="MOF" path="MetaModels/WSDL.ecore"/>
  <am3.loadModel modelHandler="EMF" name="inModel" metamodel="DPL" path="Models/DPL_Instance.ecore"/>
</target>

<target name="TF_DPLToWSDL" description="Perform DPLToWSDL transform" depends="LM_DPLToWSDL">
  <am3.atl path="Transforms/DPLToWSDL.atl">
    <inModel name="DPL" model="DPL"/>
    <inModel name="IN" model="inModel"/>
    <inModel name="WSDL" model="WSDL"/>
    <outModel name="OUT" model="outModel" metamodel="WSDL" path="Models/WSDL_Instance.ecore"/>
  </am3.atl>
  <am3 saveModel model="outModel" path="Models/WSDL_Instance.ecore"/>
</target>
```

Figure C.7: ANT script to convert a DPL based model to a WSDL based model.

```
<target name="LM_DPLToPDD">
  <am3.loadModel modelHandler="EMF" name="DPL" metamodel="MOF" path="MetaModels/DPL.ecore"/>
  <am3.loadModel modelHandler="EMF" name="PDD" metamodel="MOF" path="MetaModels/PDD.ecore"/>
  <am3.loadModel modelHandler="EMF" name="inModel" metamodel="DPL" path="Models/DPL_Instance.ecore"/>
</target>

<target name="TF_DPLToPDD" description="Perform DPLToPDD transform" depends="LM_DPLToPDD">
  <am3.atl path="Transforms/DPLToPDD.atl">
    <inModel name="DPL" model="DPL"/>
    <inModel name="IN" model="inModel"/>
    <inModel name="PDD" model="PDD"/>
    <outModel name="OUT" model="outModel" metamodel="PDD" path="Models/PDD_Instance.ecore"/>
  </am3.atl>
  <am3.saveModel model="outModel" path="Models/PDD_Instance.ecore"/>
</target>
```

Figure C.8: ANT script to convert a DPL based model to a PDD based model.

```
<target name="LM_DPLToWSDLCatalog">
  <am3.loadModel modelHandler="EMF" name="DPL" metamodel="MOF" path="MetaModels/DPL.ecore"/>
  <am3.loadModel modelHandler="EMF" name="WSDLCatalog" metamodel="MOF" path="MetaModels
     /WSDLCatalog.ecore"/>
  <am3.loadModel modelHandler="EMF" name="inModel" metamodel="DPL" path="Models/DPL_Instance.ecore"/>
</target>

<target name="TF_DPLToWSDLCatalog" description="Perform DPLToWSDLCatalog transform"
  depends="LM_DPLToWSDLCatalog">
  <am3.atl path="Transforms/DPLToWSDLCatalog.atl">
    <inModel name="DPL" model="DPL"/>
    <inModel name="IN" model="inModel"/>
    <inModel name="WSDLCatalog" model="WSDLCatalog"/>
    <outModel name="OUT" model="outModel" metamodel="WSDLCatalog" path="Models
       /WSDLCatalog_Instance.ecore"/>
  </am3.atl>
  <am3.saveModel model="outModel" path="Models/WSDLCatalog_Instance.ecore"/>
</target>
```

Figure C.9: ANT script to convert a DPL based model to a WSDLCatalog based model.

```
<target name="TF_BPELXMIToXML" description="Transform the Models from XMI representations to XML">
  <am3.loadModel modelHandler="EMF" name="XML" metamodel="MOF" path="MetaModels/XML.ecore"/>
  <am3.loadModel modelHandler="EMF" name="BPEL" metamodel="MOF" path="MetaModels/BPEL.ecore"/>
  <am3.loadModel modelHandler="EMF" name="inModel" metamodel="BPEL" path="Models/BPEL_Instance.ecore"/>
  <am3.atl path="Transforms/BPELToXML.atl">
    <inModel name="IN" model="inModel"/>
    <inModel name="BPEL" model="BPEL"/>
    <inModel name="XML" model="XML"/>
    <outModel name="OUT" model="outModel" metamodel="XML"/>
  </am3.atl>
  <am3.saveModel model="outModel" path="Models/BPEL_As_XML_Instance.ecore"/>
</target>
```

Figure C.10: ANT script to convert a WS-BPEL based model to an XML based model.

```
<target name="TF_WSDLXMIToXML" description="Transform the Models from XMI representations to XML">
  <am3.loadModel modelHandler="EMF" name="XML" metamodel="MOF" path="MetaModels/XML.ecore"/>
  <am3.loadModel modelHandler="EMF" name="WSDL" metamodel="MOF" path="MetaModels/WSDL.ecore"/>
  <am3.loadModel modelHandler="EMF" name="inModel" metamodel="WSDL" path="Models/WSDL_Instance.ecore"/>
  <am3.atl path="Transforms/WSDLToXML.atl">
    <inModel name="WSDL" model="WSDL"/>
    <inModel name="IN" model="inModel"/>
    <inModel name="XML" model="XML"/>
    <outModel name="OUT" model="outModel" metamodel="XML"/>
  </am3.atl>
  <am3.saveModel model="outModel" path="Models/WSDL_As_XML_Instance.ecore"/>
</target>
```

Figure C.11: ANT script to convert a WSDL based model to an XML based model.

```
<target name="TF_PDDXMIToXML" description="Transform the Models from XMI representations to XML">
  <am3.loadModel modelHandler="EMF" name="XML" metamodel="MOF" path="MetaModels/XML.ecore"/>
  <am3.loadModel modelHandler="EMF" name="PDD" metamodel="MOF" path="MetaModels/PDD.ecore"/>
  <am3.loadModel modelHandler="EMF" name="inModel" metamodel="PDD" path="Models/PDD_Instance.ecore"/>
  <am3.atl path="Transforms/PDDToXML.atl">
    <inModel name="PDD" model="PDD"/>
    <inModel name="IN" model="inModel"/>
    <inModel name="XML" model="XML"/>
    <outModel name="OUT" model="outModel" metamodel="XML"/>
  </am3.atl>
  <am3.saveModel model="outModel" path="Models/PDD_As_XML_Instance.ecore"/>
</target>
```

Figure C.12: ANT script to convert a PDD based model to an XML based model.

```
<target name="TF_WSDLCatalogXMIToXML" description="Transform the Models from XMI representations to XML">
  <am3.loadModel modelHandler="EMF" name="XML" metamodel="MOF" path="MetaModels/XML.ecore"/>
  <am3.loadModel modelHandler="EMF" name="WSDLCatalog" metamodel="MOF" path="MetaModels
    /WSDLCatalog.ecore"/>
  <am3.loadModel modelHandler="EMF" name="inModel" metamodel="WSDLCatalog" path="Models
    /WSDLCatalog_Instance.ecore"/>
  <am3.atl path="Transforms/WSDLCatalogToXML.atl">
    <inModel name="WSDLCatalog" model="WSDLCatalog"/>
    <inModel name="IN" model="inModel"/>
    <inModel name="XML" model="XML"/>
    <outModel name="OUT" model="outModel" metamodel="XML"/>
  </am3.atl>
  <am3.saveModel model="outModel" path="Models/WSDLCatalog_As_XML_Instance.ecore"/>
</target>
```

Figure C.13: ANT script to convert a WSDLCatalog based model to an XML based model.

```
<target name="IJ_XMLToText" description="Convert the XML based Models to Text">
  <am3.loadModel modelHandler="EMF" name="XML" metamodel="MOF" path="MetaModels/XML.ecore"/>

  <am3.loadModel name="BPEL" metamodel="XML" path="Models/BPEL_As_XML_Instance.ecore"/>
  <am3.saveModel model="BPEL" path="Generated/BPEL_Instance.bpel">
    <extractor name="xml" />
  </am3.saveModel>

  <am3.loadModel name="WSDL" metamodel="XML" path="Models/WSDL_As_XML_Instance.ecore"/>
  <am3.saveModel model="WSDL" path="Generated/WSDL_Instance.wsdl">
    <extractor name="xml" />
  </am3.saveModel>

  <am3.loadModel name="PDD" metamodel="XML" path="Models/PDD_As_XML_Instance.ecore"/>
  <am3.saveModel model="PDD" path="Generated/PDD_Instance.pdd">
    <extractor name="xml" />
  </am3.saveModel>

  <am3.loadModel name="WSDLCatalog" metamodel="XML" path="Models/WSDLCatalog_As_XML_Instance.ecore"/>
  <am3.saveModel model="WSDLCatalog" path="Generated/WSDLCatalog_Instance.xml">
    <extractor name="xml" />
  </am3.saveModel>
</target>
```

Figure C.14: ANT script to convert XML based models to XML text.

Appendix D

Acronyms

Term	*Explanation*	*PageReference*
ADL	Architecture Description Languages	37
ADT	ATL Development Tools	28
ALMA	Architecture Level Modifiability Analysis	234
ATL	ATLAS Transformation Language	28
BNF	BackusNaur form	91
BPEL	Business Process Execution Language	15
BPMN	Business Process Modeling Notation	22
CBSD	Component Based Software Development	9
CIM	Computation Independent Model	23
CWM	Common Warehouse Metamodel	22
DECS	(a run time framework)	44
DPL	Distribution Pattern Language	54
DPLProfile	Distribution Pattern Language UML profile	52
DSL	Domain Specific Language	21
EBNF	Extended BNF	91, 113
ECL	Extended Constraint Language	43
ECORE	(an EMF compatible language)	206
EMF	Eclipse Modeling Framework	25
EMOF	Essential MOF	25, 91
GP	Generative Programming	18
GPL	General Purpose Language	86
IDE	Integrated Development Environment	199
ILDP	Integration Logic Document Processor	255
JET	Java Emitter Template	26
JSP	Java Server Pages	26
MDA	Model Driven Architecture	22
MDD	Model Driven Development	22
MDSD	Model Driven Software Development	22

Term	*Explanation*	*PageReference*
Model Bus	Tool integration suite	24
MOF	Meta Object Facility	18
MTF	Model Transformation Framework	26
MTL	Model Transformation Language	27
OCL	Object Constraint Language	19
OMG	Object Management Group	18
OOP	Object Oriented Programming	12
OWL	Web Ontology Language	16
OWL-S	Based on OWL for web services	17
PDD	Process Deployment Description	54, 139
PIM	Platform Independent Model	23
PSM	Platform Specific Model	23
QoS	Quality of Service	2
QVT-P	Query/View/Transformation Partners	27
RDF	Resource Description Framework	16
RDL	Relations Definition Language	27
RSA	Rational Software Architect (an IBM tool)	211
Smart QVT	A model transformation tool	24
SOA	Service Oriented Architecture	1
SOAP	Simple Object Access Protocol	12
SQL	Simple Query Language	87
UDDI	Universal Description, Discovery and Integration	12
UML	Unified Modeling Language	19
UMT	UML Model Transformation Tool	29
Web-ML	Web hypertext modeling notation	35, 42
WS-BPEL	Web Services Business Process Execution Language	15
WS-CDL	Web Services Choreography Description Language	15
WSCI	Web Service Choreography Interface	-
WSDL	Web Service Description Language	12
WSML	Web Service Modeling Language	17
XMI	XML Metadata Interchange Format	25
XML	Extensible Markup Language	1
XSLT	Extensible Stylesheet Language for Transformations	25