Formalisation of Simple MethOd Declaration Language (SMODL) by DPF

Kalakata Suneetha

Master's Thesis in Informatics – Program Development

Department of Informatics
University of Bergen

HØGSKOLEN I BERGEN
Department of Computer Engineering
Bergen University College

October 2012

Contents

List of Figures

Preface

Foreword

This master's thesis submitted for conclusion of my Master's Degree programme in Informatics – Program Development, at the University of Bergen and Bergen University College.

The development of this thesis was done as a sub project of the DPF project, I was introduced by skilled and interesting people and exposed to many new technologies. The DPF project gave the chance to learn many new things. At the starting of this thesis it was completely confusing about the model-driven engineering, but once involved, it was so interested and motivated to do thesis in challenging way. During bi-weekly meeting we had a chance to discuss about progress of the work done, the technologies and problems related to thesis and DPF project.

This project gave me the chance to formalise a web service with modelling and model transformation concepts of model-driven engineering in DPF project.

Acknowledgements

This thesis has been carried by me with invaluable support from my supervisor Yngve Lamo with great patience and valuable suggestions. A special thanks to Florian Mantz and Xiaoliang Wang for providing feedback on theoretical and technological aspects. I also would like to thank the DPF project team members: Anders Sandven and Sidra Nadeem. It would not be possible to finish master's thesis without support of my husband Chandra Sekhar and my daughter Gnapika. Thank you.

Bergen, 08 October 2012

Chapter 1

Introduction

1.1 Motivation

As decades are passing, there is an outcome of new generation of programming languages in software development. First generations of programming languages were machine level programming languages. Second generation of programming languages are assembly languages were introduced nn early 1950's . They need to translate to machine code in order to run the program. In between 1950's to 1970's third generation of programming languages were introduced. They are higher level programming languages. They are machine independent and programmer friendly languages such as structured programming. The examples of third generation programming languages are FORTRAN, COBOL, Java, Ada, C, C++ and Algol.

The problem criteria's are changing from simple to complex, small to large and I/O-intensive to computation-intensive, especially for the commercial business software development. It became difficult to maintain the high quality of the software with general purpose programming languages those required to produce huge amount of lines of code. The questions were raised how we can produce complex structures from simple parts and how to reuse or integrate with other parts of the system [20].

In order to reduce the time, cost and effort in developing large complex systems for a particular domain, developers are in need of new generation of languages. Fourth generation programming languages such as SAS (Statistical Analysis System), SQL (Structured Query Language) were introduced in between 1970's to 1990's. They are based on structured query languages and designed to reduce the time and effort on developing software systems. But they are lack of concepts for solving problems related to a particular domain. Modelling languages has become more popular to overcome those issues. The developer could then get the same functionality by representing in graphical notation instead of several as earlier [21]. Modelling languages supports graphical and textual representation of the system.

The software development methodology was moved from code-centric to model-centric. Creating a model for a particular domain either in graphical or textual representation increases the productivity and quality of the software. Model driven approach focuses on higher level of abstraction of a particular domain problem by specifying important aspects of the system in models as first class entities and automation activities such as code generation and model transformation. This way of approaching for software development is referred to as Model Driven Engineering.

Model driven engineering was originated from the Computer-Aided Software Engineering (CASE) tools in early 1980's. CASE tools uses graphical representation in modelling to enable developers to express their design in structured diagrams or data flow diagrams. The main purpose of CASE tools are to produce high quality and maintainable software component by automating the activities in the life cycle of the software development process. The CASE tools often has issues like quality, security, to handle complexity in broad range of application domains and fault tolerance. To overcome the issues with CASE tools, especially to address the platform complexity and inability in expressing complex domain concepts effectively, MDE requires new software technologies [34].

Object Management Group (OMG) proposed a set of standard rules for a software design called Model Driven Architecture (MDA) [24] to overcome the problems with CASE tools. MDA provides a framework for software development that uses the models to describe the system to be built. Those models are defined in Unified modeling language (UML), which was proposed by OMG. UML uses graphical representation for syntax and the constraints are expressed in Object constraint language (OCL) [26]. MDA supports 4-layer of architecture for metamodelling [24]. MDA does not support the multileveled metamodelling.

In order to overcome issues with MDA, Diagram Predicate Framework (DPF) provides a formal specification approach for modelling based on category theory and graph transformation [6]. DPF is supported by the DPF Workbench tool [23]. It is a diagrammatic tool for a domain specific modelling, which supports the development of metamodelling hierarchies with an arbitrary number of metalevels and also checks the conformance of models to their metamodels by validating both typing and diagrammatic constraints [23].

Service Oriented Architecture (SOA) is an architecture not a framework with a set of disciplines for designing and developing software, in the form of services referred to as service-oriented entities. Services are self contained functions build as a reusable software components. SOA approach promises the reusability, loose coupling and interoperability of services. The main goals of SOA are integration of different business/IT systems under different managements. Service-Oriented modelling provides solution to a particular business service by modelling them with particular modelling disciplines and language concerned to that service [3]. Modelling of services in DPF Workbench [22] provides loose coupling and specification formalisation to a well defined business services especially such as web-based applications.

The first goal of the thesis is to specify the metamodel for the Simple Method Declaration Language (SMODL) [30] in DPF Workbench. SMODL metamodel was specified in Relax-NG schema language by RUnit Software. Formalisation of SMODL by modelling and making it as reusable component for the DPF Workbench. Models specified by the metamodel of SMODL will be used to generate code for implementation of SMODL web service with other technologies developed by RUnit Software. DPF Workbench provides the support to specify the metamodel in a higher level of abstraction. So, this thesis defines the metamodel of SMODL by specifying the important concept of SMODL services by means of Node and Arrow of DFP metamodelling language. Modelling of SMODL in DPF Workbench requires to define constraints specific to SMODL model.

Second goal of the thesis is to perform the model transformation, the DPF representation of SMODL models can be used by other plug-ins to generate the SMODL web services. It supports the bidirectional model transformation, i.e. the DPF SMODL model which is specified by the DPF SMODL metamodel with the DPF SMODL modelling language will be transferred to a textual model of XML SMODL specified by the modelling languages of Relax-NG[30]. It also provides the implementation for reverse transformation from textual SMODL model to diagrammatic DPF SMODL model. DPF Workbench tool provides the support with Code Generation facility based on metamodel [32] for transformation from DPF SMODL model to textual SMODL model as a transformation engine. The reverse transformation of textual SMODL to diagram DPF SMODL model with support of XML parser [40] to parse XML SMODL and with some graphical representation for visualization. Figure 1.1 shows an overview of this process.

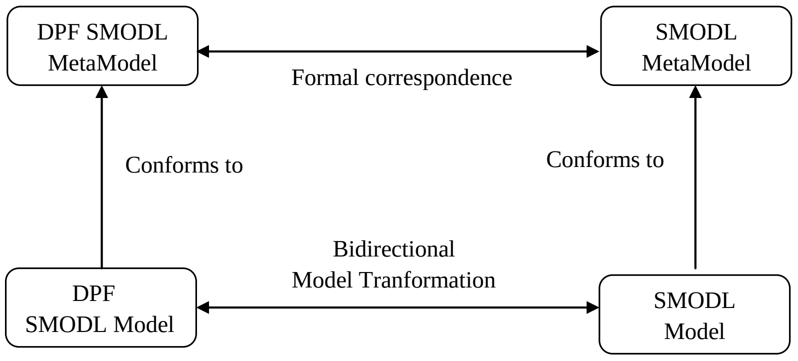


Figure 1.1: Overview of Thesis

1.2 Structure of Thesis

The structure of the thesis:

Chapter 2 – Background
This chapter gives the background information about the software development methodologies and technologies such as Model-driven Engineering, Service-oriented Architecture, Service-Oriented Modeling, SMODL specification to define SMODL model for SMODL web services, Diagram Predicate Framework, DPF Workbench tool and Metamodel based Code Generation in DPF Editor.

Chapter 3–Problem and requirement analysis
In this chapter we are explaining what research method we followed. What kind of development methodologies we followed and technologies we chosen for implementation of the solution.

Chapter 4 – Solution
This chapter provides the solution how to define SMODL metamodel and SMODL model in DPF Workbench. Implementation of bidirectional model transformation from diagram model to text model and reversing from textual model to diagram model.

Chapter 5 – Evaluation and Conclusion
This chapter gives a comparison between how the SMODL model is formalised in the DPF Workbench and traditional way of modelling SMODL models. Here we also summarizes how the problem is solved in DPF and give some suggestion for further work.

Chapter 2

Background

This chapter gives introduction about the Model-driven engineering (MDE) methodology, Service-oriented Architecture (SOA), Service-oriented Modelling (SOM). Simple Method Declaration Language (SMODL) model specification for generating code for SMODL web service. Diagram Predicate Framework (DPF), DPF Workbench tool to reference DPF, Code generation based on metamodel.

2.1 Model Driven Engineering

As mentioned earlier, the evolution of programming languages for solving the problems in enterprise systems, it became difficult to maintain the efficient documentation and specification of the system as the problem complexity increases. Modelling languages and usage of models are become popular to sort out these issues. Models are used for designing the system, analysing the system, specifying required functionality and creating documentation [21].

The Software development paradigm moved from code-driven development to model-driven development. Model-Driven Engineering (MDE) is a software development methodology which relies on models. In model-driven approach models should be specific to a particular domain and give an abstract representation of problem, which encapsulate the implementation details by concentrating more on domain concepts. MDE was driven using the models as their primary artefacts during the life cycle of the development process and improves the productivity by performing model transformation with automatic generators [31]. MDE promises productivity, quality, facilitate separation between business logic and application technologies by effective expressing of domain concepts [24][34]. MDE can be characterized by two relations; *representation* and *conformance* [13].

- Representation: A model represents a software artefact or real-world

domain.
- Conformance: A model satisfies the constraints of a metamodel.

A model can be prescriptive (a specification of the real system to be constructed, i.e. as a pattern for design) or descriptive (documented representation to explain the major aspects of the existing system). Models can represent the structure of the software system to be developed during the design phase. A model should have the following characteristics [13].

- Abstraction: A model should only describe the interesting properties of a system.
- Reflection: A model should represent some of the features of the system to be constructed or already exists.
- Understandability: A model should represents intuition of the system.
- Substitution: A model can used instead of the original system.

Modelling languages are needed for defining the syntax and semantics of the models. Syntax specifies the conceptual structure of the system, with a set of rules needed for specifying the model. Semantics specifies the meaning of every modelling concept has some meaning [21]. In general, any modelling language is represented by a metamodel to express any model that is an instance of that modelling language [14]. Formal specifications is a method for defining the semantics of any languages in terms of mathematics. There are several formal specification approaches for defining modelling languages such as Algebraic, Logic, Type theory, Category theory and Graph transformation.

Logic: It's a subfield of mathematics. It shows expressiveness power of a formal system. So, mathematical concepts are expressed in formal logical system such as propositional logic and first-order logic. Propositional logic provides the standard way for assigning meaning to its expression with limited expressive power [37]. First-order logic consist of well-formed formulas and formula-binding operations of quantification over individual elements [37].

Algebraic Specification: It's a collection of formal methods which uses the ideas from logic and mathematics to model, analyse, design, construct, and improve software [33]. It allows specifying the required behaviour of the system and making sure that correctness of the individual steps with proof. This ensures the correctness of the proposed system. Logical system or first-order logic called axioms are required for capturing the required behaviour of the system. Semantics of the logic system ensures whether the axiom is satisfied or not.

Type theory: Type theory provides the formalism in terms of formal system. Formal system is a well defined system of abstract thought based on model of mathematics [38]. Formal language is used to define the formal system which uses primitive symbols for constructing certain rules called Axioms. Axiom is a statement that can serves as a starting point, from that a new statement can be derived. Type theory is based on a simple typed lambda calculus, which is a formal system in terms of variable binding and substitution for computation. Lambda calculus provides the isomorphism between proposition and type, proposition refers to the meaning of full declarative sentence and type refers each value should be associated with a type system to be computed.

Category theory: Category theory is based on high level of abstraction that takes the following view point [39]

- There are objects of interest and directed relations (arrows) between these objects.
- There is no possibility to look inside the objects i.e., objects are considered as black boxes. The only interesting study about the objects is the relations going into this object and going out, respectively.

For example referred from [33]: Given a pair of sets A and B , a product of A and B is a set P together with two functions $\pi 1{:}P \rightarrow A$ and $\pi 2{:}P \rightarrow B$ such that for any set C with functions $f : C \rightarrow A$ and $g : C \rightarrow B$ there exists a unique function $h : C \rightarrow P$ such that $h; \pi 1 \equiv f$ and $h; \pi 2 \equiv g$.

The figure 2.1 shows the example of Category theory. According to this all interesting information about A and B is captured by P.

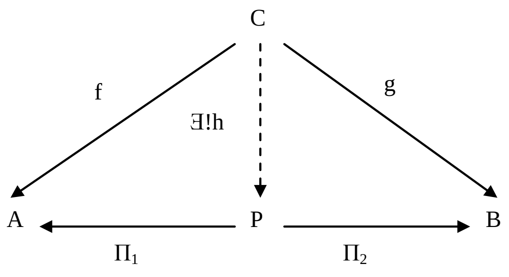

$$\begin{tikzcd}
 & C \arrow[dl, "f"'] \arrow[d, dashed, "\exists!h"'] \arrow[dr, "g"] & \\
A & P \arrow[l, "\Pi_1"] \arrow[r, "\Pi_2"'] & B
\end{tikzcd}$$

Figure 2.1: Example for Category Theory

Graph transformation: Graphs plays a key role in Graph transformations. Graphs are useful to explain complex situations at a intuitive level [12]. A graph is given by a collection of nodes N, a collection of edges E, a map $S : E \rightarrow N$, assigning each edge to its source node and a map $T : E \rightarrow N$ assigning each edge to its target node. The purpose of Graph transformation is the rule base modification of Graphs, as shown in figure 2.2. The core rule $P = (L, R)$ is a pair of left hand side graph L and right hand side graph R. Once the rule P is applied then L in the source graph is replaced by the R to get target graph.

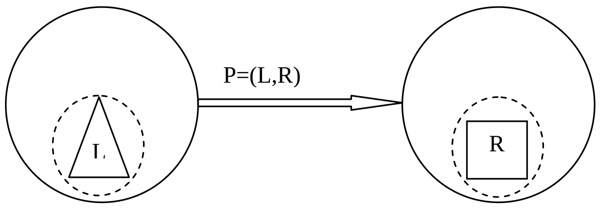


Figure 2.2: Overview of Graph Transformation

2.1.1 Model Driven Architecture

In 2001 OMG[24] proposed a software design namely, Model Driven Architecture (MDA) with a set of standards for structuring the specification of the system in terms of models. MDA was the first initiator to the MDE approach of development methodology. MDA defines the three viewpoints for analysing the system. They are

- Computational-Independent Model (CIM): It is completely defined by the domain expert of the system, does not contain the details of the structure.
- Platform-Independent Model (PIM): Model should be independent of the computer specific platforms.
- Platform-Specific Model (PSM): It's a combination of PIM and the specification of the computer system.

The few standards of MDA for supporting the metamodelling are

Meta-Object Facility (MOF)

Meta-Object Facility (MOF)[25] was proposed by OMG in the 90's. It is self-defined (reflexive) language and framework for specifying, constructing and managing technological independent metamodels [13]. In the MOF 2.0 version there are two meta-metamodels. One was defined in the scope of Model driven architecture Complete MOF (CMOF), the other one Essential MOF (EMOF) was defined in the scope of Eclipse Modeling Framework (EMF), which is light weight version of CMOF. CMOF is more expressive and harder to implement than the EMOF.

Unified Modeling Language (UML)

UML[27] became a part of OMG's standard in 1997, is one of the most popular software modeling language. UML supports the abstraction and graphical representation of models. The main purpose of the UML is to specify, visualize and documentation of Software systems. As the complexity of the system increases, UML supports to define inter-related models and finally overall model can be viewed as composition inter-related models. Specific and critical problems of the system under construction are easier to express by graphical representation of models called UML diagrams. UML2.4.1 is the latest version released in August 2011. UML specification consists of mainly two kinds of UML diagrams; namely *structure diagrams* and *behaviour diagrams*.

- Structured diagrams: These diagrams shows the static structure of the system and relation between parts of the system at different level of abstraction and implementation.Some of the most used structured diagrams are *Class diagrams,Object diagrams* and *package diagrams*.
- Behaviour diagrams: These diagrams are used to show the dynamic behaviour of the objects in a system. Some of the behaviour diagrams are *use case diagrams* and *activity diagrams*.

2.2 Metamodelling

Metamodelling is one of the key activities in the MDE approach. Metamodelling can be seen as modelling of modelling languages. Definition of metamodelling can be defined as art and science of creating metamodels, which are qualified variants of models [14]. In general a metamodel is a model of models [31]. The main purpose of the metamodel is to define the syntax of the modeling language. The syntax specifies the set of rules consists of conceptual structure of a modelling language, their properties and connections to each other [21], as well as the rules for combining these concepts to specify valid models [31]. In metamodelling, a model is defined by a metamodel, which in turn can be used as a metamodel for another model on a lower level of abstraction. The process of defining one model from another model makes metamodelling recursive.

As explained the example in figure 2.3 Class and Reference are the metaclasses of the metamodel of type Class diagrams of UML. Service and Method are the instance of the metaclass Class and isMethodOf is instance of Reference. isMethodOf shows the relation between the Class instances Service and Method in model.

Metamodelling languages are required to specify the metamodels. Metamodels which are specified by the metamodelling languages can serve as a modelling language for defining another model in the next lower abstraction level. A model in a particular level conforms to the metamodel in the level above and can act as a modelling language for the next lower level. Figure 2.4 explains this generic pattern of metamodelling hierarchy [31].

OMG proposed 4-layer architecture for describing the relation between metamodelling languages and metamodels. The relation between the meta-

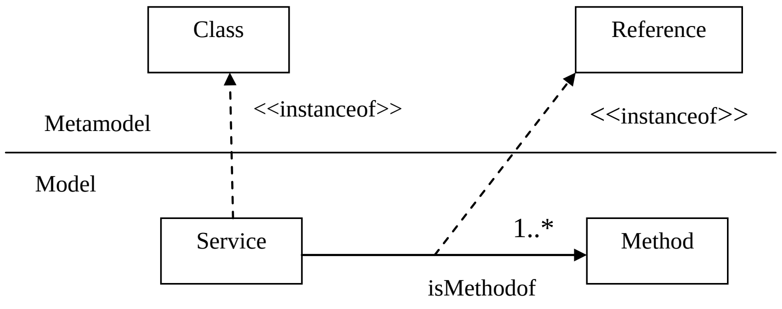


Figure 2.3: Example of Metamodelling

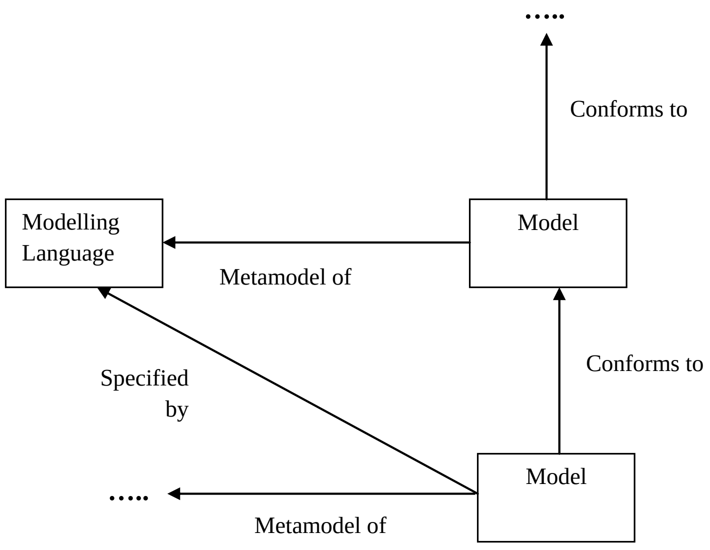


Figure 2.4: Generic pattern of metamodelling hierarchy

modelling languages, modelling languages and models are organised in 4-layers from Mo to M3.The topmost model in the hierarchy is the meta-metamodel known as MOF. It is a self-defined language and framework for specifying, constructing and managing technological independent meta-models [13]. MOF is used as a base for defining any modeling languages such as UML or MOF by itself and conforms to itself, that means that it is reflexive at the topmost layer of the metamodelling hierarchy. MOF contains the metadata repository for defining metamodels.

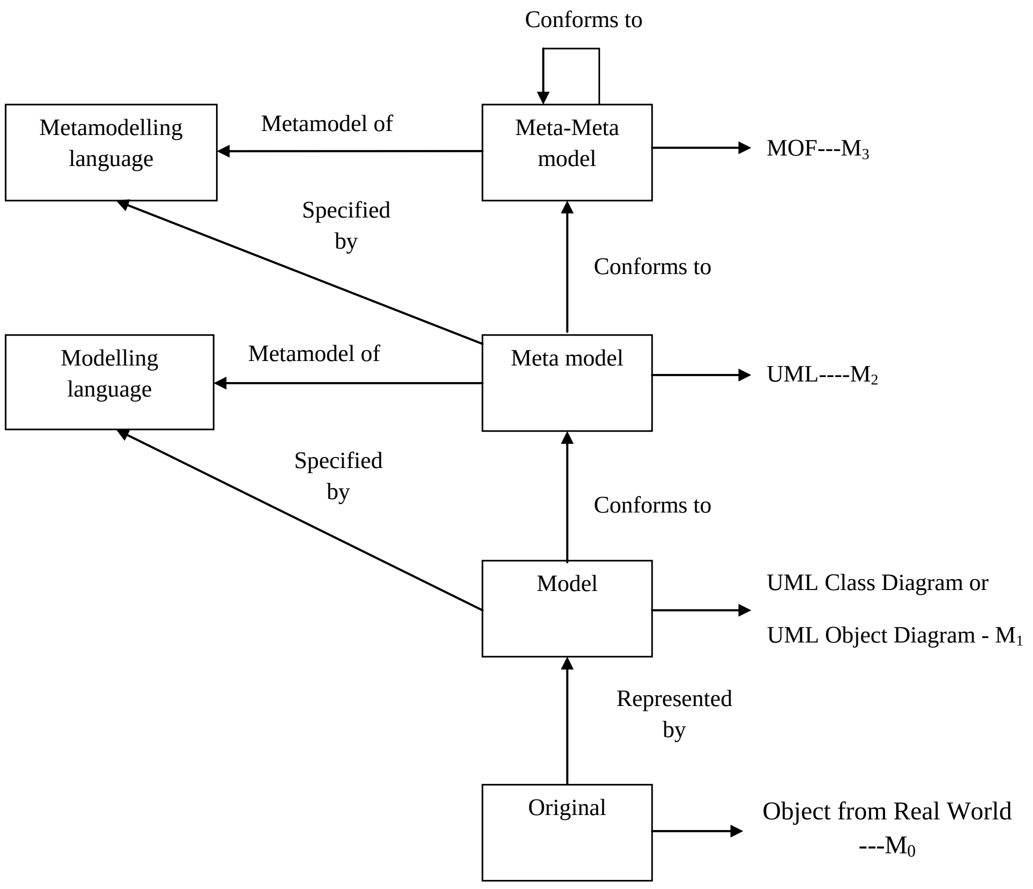


Figure 2.5: OMG's 4-layer architecture of metamodelling

The following summarises the OMG metamodelling hierarchy as shown in figure 2.5

- Level M3 contains the meta-metamodel MOF, which conforms to itself.
- Level M2 contains metamodels, e.g. the UML metamodel conforms MOF at level M3.

- Level M1 contains models, e.g. a UML class diagram or UML Object diagrams and conforms to UML metamodel at level M2
- LevelM0 contains originals, real object in the world, which represented by model at level M1

2.3 Model Transformation

Model transformation is another key activity in the MDE approach. The main purpose of model transformation is to avoid redundancy of code, to save the effort and reduce errors. Model transformation in MDE is used to automate several model related activities such as code generation, refactoring, optimisation and language translation [31].

In MDE model transformation transfers the abstract models to concrete models in other words the transformation of source models to target models. A transformation definition is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language. A transformation rule is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language [16].

To perform the transformation of models, they should be expressed in some modelling languages. Syntax and semantics of the modelling languages will be specified by metamodels of modelling languages as shown in the figure 2.6.

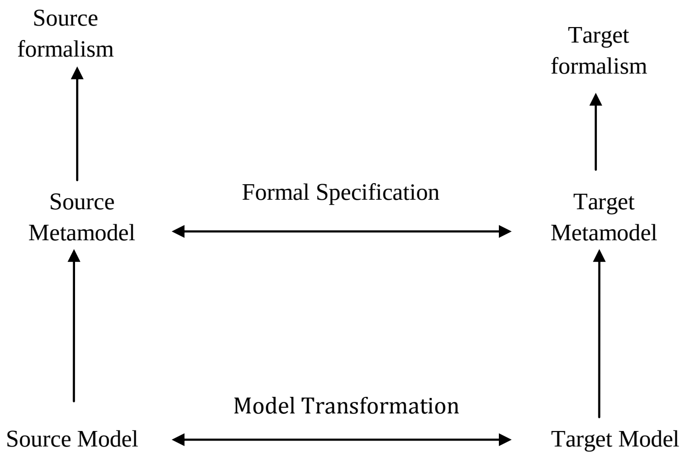


Figure 2.6: Model Transformation

Model transformations are carried out automatically by tools in the transformation processes. Each part of the transformation process is described by a transformation definition, which in turn is written in a transformation definition language. The tool which is used for the execution of model transformations is called a transformation engine [31].

Transformation of models is classified into several types.

- Homogeneous transformation: The source and target models are defined within the same modelling language.
- Heterogeneous transformation: The source and target models are defined in different modelling languages.
- Outplace transformation: The target models are created from scratch.
- Inplace transformation: The target model will be generated by modifying the source model.

Examples of Homogeneous transformations are:

- Refactoring: To improve the internal structure of the system, without modifying the behaviour of the system.
- Optimization: Improvement in the performance of the system, without changing the semantics of the system.

Examples of Heterogeneous transformations are:

- Reverse engineering: Is the inverse of synthesis and extracts a higher-level specification from a lower-level one [16].
- Migration: Changing of a software written in one programming language to another programming language.

There are several mechanisms available for model transformation implementation. They can be categorized in either declarative or operational approach. Declarative focuses on the what aspect, means what need to be transformed into what. Operational focuses on the how to aspect, means how the transformation itself need to be performed. Example of declarative mechanisms are functional programming, logic programming and graph transformations. They are well-founded and support bidirectional transformation. They offer a simpler semantic model since order of execution, traversal of source models, as well as generation of target models are implicit [31]. Operational mechanisms are supportive when the models are required to update incrementally. They provide sequences, selections and increments and thus control the order of execution. QVT is a lower level programming language specific to operational mechanism.

2.4 Service Oriented Architecture (SOA)

As business/Information technology's (IT) systems and processes are growing day by day and becoming more and more complex. IT flexibility, perfection, redundancy and maintainability has became key factors when individual system are integrated into large distributed systems. Another problem is business/IT gap, because different business and IT people have their own culture and professional languages. SOA is a paradigm for organizing and utilizing capabilities that may be under the control of different leadership domains. It provides uniform means to offer, discover, interact with and use capabilities to produce desired effects consistent with measurable preconditions and expectations. SOA is an architecture, not a framework or a development technology, its a way of thinking and approaching the concrete design of software architectures.

SOA is an approach that helps the system to remain scalable and flexible while growing and that also helps to bridge the business/IT gap [19]. The main purpose of SOA is to organise large distributed systems in individual components. Large distributed systems are heterogeneous programming languages, implemented in different programming platforms and infrastructures. The important concepts of SOA are:

- Technical characteristics of SOA are

  Service: It is a simple business functionality that concentrates on the business value of an interface. Services bridges business/IT gap. They sould be simple and provide single functionality. There should be a tight coupling inside a service.

  An Enterprise Service Bus (ESB): It is an infrastructure, which enables interoperability between services of distributed systems and makes it easier to distribute the process across heterogeneous platforms, languages and technologies by encapsulating implementation details of the services to consumers. In simple words, it delivers the requested services to the consumers from the service provider and makes sure that service is delivered to the consumer. There will be no interaction between consumer and provider, interaction thorough the ESB.

  Loose coupling: It reduces the system dependencies, all services communicates through ESB, which provides the communication between consumer and provider. Each service provides a simple business functionality with out depending on other dependencies. Loose coupling reduces the chances of modifying one service impact on other services. Loose coupling guarantee low dependencies between services.

- SOA defines the set of roles, policies and process for systems such as model-driven service development and distributed software development.
- Web services are the reference for the implementation of SOA.
- To success with SOA it’s important for finding the right governance and management such as finding the right people and balance in distributing the processes over different systems.

2.4.1 Services

A Service can be defined as a self contained functionality that corresponds to a real world activity [19]. The usage of the services will be decided on the basis of behaviour and semantics of the service through interfaces. They can be represented by signatures, but signatures are not sufficient to know the behaviour of the services. A *contract* provides the detailed specification of the service and the relation between a provider and consumers of the service.

There are several services, the differences between services will be identified by their behaviour and the purpose of the service providing like creating, reading, updating, deletion of data. Services are categorized based on their properties such as self-contained, stateless, reusable, composable and so on.

In general services are classified into three categories according to Nicolai M. Josuttis [19]

- Basic services
- Composed service
- Process services

For each category of services, there is a layer of services such as Basic layer, Composed layer, Process layer. There are three stages of expansions by expanding each layer of corresponding category of service.

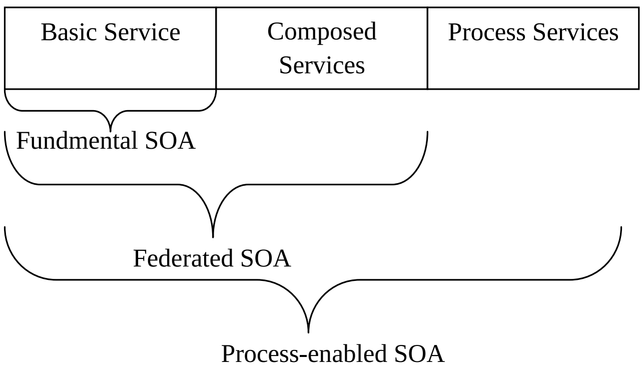


Figure 2.7: Service layers and stages of SOA expansion

Fundamental SOA: This stage consists of only basic layer with basic services. These services provides basic business functionality for a specific problem domain or a backend system to read and write data called as *basic data services*. Another type of this services are *basic logic services*, these process some input data and produces some results. These services are very short term running, do not required to maintain the state of the service. So, they are stateless and hides the implementation details.

Federated SOA: It is an expansion over the fundamental SOA stage. It allows the composition of basic services with other basic services or can be with other composed services. A composed service provides the access of multiple backend services. For example maintaining of one employee details in different departments of organisation. They are short term service and stateless services.

Process-enabled SOA: It includes process services in addition to the federated SOA. Process services are long-running flow of activities and need to maintain the state over the multiple session. So, they are stateful. Example of process services are shopping-cart service.

2.4.2 Loose Coupling

Loose coupling is the concept of SOA, to deal with the requirements of the scalability, flexibility and fault tolerance [19]. The kind and degree of loose coupling should employed will be based upon the circumstance of the system. In order to achieve loose coupling the service providers defines the data types used by the service, and the service consumers should accept these types or consumers can have a mapping layer between providers data type and their own data types in source code. This kind of coupling will avoid the conflict if one system data types modified with out effect the other system data types.

2.4.3 Enterprise Service Bus (ESB)

The ESB is an infrastructure, one of the key role in SOA. There are several responsibilities of ESB

- Interoperability: It is one of the key role of ESB is to provide the connectivity among different languages and different platforms.
- Routing: It provides the way to transfer the messages between consumers call and provides service.
- Low coupling.

2.5 Service oriented modelling (SOM)

Service oriented modelling (SOM) is a software development practice that employs modelling disciplines and language to provide strategic and tactical solutions to enterprise problems[3]. Modelling process is governed by the principles of SOA. Modelling in SOM provides the support to create models for the software entities in an organization. The models are referred to as service-oriented assets or services.

SOM is contributed mainly by two stakeholders namely; business and IT personnel of problem domain(business) and solution domain(IT) organizations. Business stake holders, the top level executives should have the knowledge about the service-oriented modelling languages to provide the valuable input to the modelling artefacts. IT personnel stakeholders involves in architectural modeling activities and analysis and design aspects of services. There should be a coordination in between IT personnel and business stakeholders and they jointly facilitate alignment between problem domain and solution domain.

2.5.1 Classification of modelling Services

Modelling services are categorized into three types based on their life cycle state and corresponding disciplines, they are:

Conceptual service: At the stage of proposing a informal solution to a problem, there should be a requirement of terminology of business and technological abstraction to provide the collaboration between different stakeholders. These abstractions are considered to be conceptual services.

Analysis services: During the modelling process, analysis services enables us to verify the capacity and validation of the business requirements of the solution. At this stage the proposed solutions are required to have a platform for verification.

Design service: Design services enables us to visualize and plan future service behaviour,structure and peer relationships in production environment by focusing on the message and information exchange capabilities of a services [3].

2.5.2 SOM disciplines

There are several modelling disciplines in SOM to enable us to focus mainly on modeling strategies rather than being concerned with source code and detailed programing algorithm [3]. In service oriented development activities of services, modelling disciplines offers a set of standards and policies. And offers best practices before construction and deployment of service into production environment. Modelling disciplines enables the business and IT personnel experts to identify the core process to produce design and architectural artifacts. There are six SOM disciplines [3]:.

Service-oriented conceptualization: It provides the methodologies and process for conceptual services by means of *attribute analysis* and *conceptual service identification*. Attribute analysis yields to a set of core attributes to identify the conceptual services by studying business requirements and problem domain statements.

Service-oriented Discovery and Analysis: In the process of finding the valuable solution to a business or technological problem, the service-oriented analysis and design process enables us to identify the services. The service-oriented discovery and analysis activities are *service typing and profiling*, *service analysis* and *service analysis modeling*

- Service typing allows to label a service based on business or technological context and its internal structure. This information can be used in the establishment of service profile, that can be used in the future activities.
- Service-oriented analysis is the act of pursuing the best possible solution by the identified service. The service analysis process validates service practicality in the solution proposition. It also explores reusability and loose coupling of the services.
- Service analysis modeling allows to model the service with the proposed language, by capturing the concept of the service formation and proposing the solution to the addressed problem.

Service-oriented discovery and analysis modelling concentrates on semantics of the service rather than internal structure and context of the service. This can be achieved by the specification of the service.

- The functionality of service in the specification should be trimmed of by reducing the scope of the operations.
- By narrowing the speciality of the service. Reducing the task load by having limits on it.
- By giving suitable name to the service. Name itself specifies what functionalities service is providing

Semantics of the service can be obtained by reducing the spectrum of functionality, speciality and name of the service.

Service-oriented Design: Describe the perspectives of the service in transaction context. A transaction context consists of service functionality, activities, message coordination and interaction [3].

Service-oriented Business Integration: It's a ongoing activity through out the service development life cycle. The business architecture is chosen to support the integration between business and technological initiatives by means of transaction context.

Conceptual Architecture: It provides the guidance for the future technological implementation process. Those can assist to find the architecture concepts for future development of the project.

Logical Architecture: It provides the guidance to address reusability, best practices for loose coupling, interoperability and consumption. Which assists for the collaboration of the service assets.

2.6 SMODL development suite for Web services

In this section we are going to explain SMODL declaration language to describe web services, which provides standard Create-Retrieve-Update-Delete (CRUD) functionality. SMODL stands for Simple Method Declaration Language, it is a simple XML-dialect for declaring method signatures and data structures. The main intuition of a SMODL model is to allow non-technical domain experts to participate in the Web Services development by allowing them to describe, develop and use Web Services. Once they modeled the web service in SMODL. SMODL runtime engine uses the SMODL model to generated the code for the implementation of web services.

RUnit Software developed SMODL development Suite by using number of tools in different steps and these utilities are bundled as a plug-in for Eclipse. It was developed in 2004 and initially used internally in RUnit Software in response to the lack of simple tools for working with Web Services.

From 2005 to 2011 SMODL model turned out to be a simple tool for trained programmers for quickly prototype and develop Web Services and generate the code for different types of clients. Another advantage of SMODL service is to provide secure and role-based access.

RUnit Software used Relax-NG schema language to specify the SMODL model. Relax-NG is a schema language for XML. It is simple and easy to learn and has both XML syntax and a compact non-XML syntax. Using the Relax-NG schema language SMODL metamodel was specified [29] as shown in the list 2.1. The default value for the name space of SMODL is *http://smodl.org/v1*. The grammar for Service in the list 2.1 represents as follows:

- *NameAttr, TargetNamespaceAttr* It should have NameAttr, TargetNamespaceAttr
- *Doc?* means zero or one DOC
- *Method+* should have one or more method
- *Struct** zero or more struct

SMODL metamodel defined with Relax-NG Schema language

```
default namespace smodl = "http://smodl.org/v1"
grammar {
  start = element service {
     NameAttr, TargetNamespaceAttr, Doc?,
     (Method+ & Struct* & Fault* & Typedef*)
  }
  Doc = element doc { text }
  Method = element method { NameAttr, Doc?, Arg*, Result }
  Struct = element struct {
    NameAttr,
    attribute base { NamePattern }?,
    Doc?, Field+
  }
Fault = element fault { NameAttr, CodeAttr, Doc? }

  Result = element result { TypedElemCommon  }
  Typedef = element typedef { NamedTypedElem }
  Arg = element arg { NamedTypedElem }
  Field = element field { NamedTypedElem }

  TypedElemCommon = TypeRefAttr, Doc?, Constrain*, NullableAttr?
  NamedTypedElem = NameAttr, TypedElemCommon

  Constrain = MinLength | MaxLength | Pattern |
              MinExclusive | MaxExclusive |
              MinInclusive | MaxInclusive

  MinLength = element minLength { attribute value { xsd:unsignedInt }, Doc? }
  MaxLength = element maxLength { attribute value { xsd:unsignedInt }, Doc? }
  Pattern = element pattern { attribute value { string }, Doc? }
  MinInclusive = element minInclusive { attribute value { string }, Doc? }
  MaxInclusive = element maxInclusive { attribute value { string }, Doc? }
  MinExclusive = element minExclusive { attribute value { string }, Doc? }
  MaxExclusive = element maxExclusive { attribute value { string }, Doc? }

  TargetNamespaceAttr = attribute targetNamespace { xsd:anyURI }
  NameAttr = attribute name { NamePattern }
  TypeRefAttr = attribute type { TypePattern }
  BaseAttr = attribute base { TypePattern }
  NullableAttr = attribute nullable { "false" | "true" }
  CodeAttr = attribute code {
    xsd:integer { minInclusive = "-32768" maxInclusive = "-1000" }
  }

  NamePattern = xsd:string { pattern = "[A-Za-z][A-Za-z0-9_]*" }
  TypePattern = xsd:string { pattern = "[A-Za-z][A-Za-z0-9_]*(\[\])*" }
}
```

Listing 2.1: SMODL metamodel defined with Relax-NG Schema language

An example of SMODL model for a Web Service with CRUD functionality of the user defined with RelaxNG-Schema language is given in the given listing 2.2

```
  -<service name="repositoryService" xmlns="http://smodl.org/v1">
 -<method name="loginUser">
        <arg type="string" name="userName"/>
        <arg type="string" name="password"/>
```

```
       <result type="bool"/>
</method>
-<method name="logoffUser">
       <result type="bool"/>
</method>
-<method name="createNewUser">
       <arg type="Profile" name="userInfo"/>
       <result type="bool"/>
</method>
-<method name="getProfile">
       <result type="Profile"/>
</method>
-<method name="updateProfile">
       <arg type="Profile" name="userInfo"/>
       <result type="bool"/>
</method>
-<method name="getCurrentUsername">
       <result type="string"/>
</method>
-<struct name="Profile">
       <field type="string" name="userName"/>
       <field type="string" name="password"/>
       <field type="string" name="email"/>
       <field type="dateTime" name="birthdate" nullable="true"/>
</struct>
</service>
```

Listing 2.2: Example of SMODL model defined with Relax-NG Schema language

2.6.1 Model-driven service development (MDSD) in SMODL

As mentioned in section 2.4 SOA organises systems across heterogeneous languages and platforms. SMODL Development Suit (SDS) is collection of different utilities required for generating and run SMODL web service. These utilities are bundled into a plugin for Eclipse. SMODL service is basic service with standard CRUD functionality. Specific behaviour of the SMODL service based upon different methods with different attributes and different data types and structures. SMODL service can be developed in MDD approach by creating a SMODL model and uses model transformation for generating the code, which can be used by SMODL runtime engine to generate code to implement the SMODL service.

SMODL uses DPF for defining SMODL model and implementing model transformation specific to SMODL service. we can refer this as model-driven service development (MDSD) in DPF. Figure 2.8 shows MDSD approach for SMODL service.

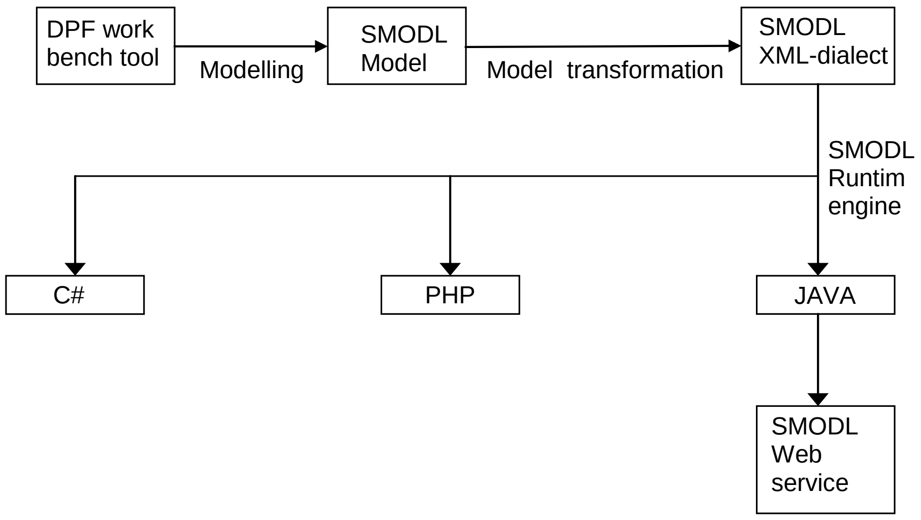


Figure 2.8: MDSD approach for SMODL

2.7 Diagram Predicate Framework

In early 2006, Bergen University College and the University of Bergen, Norway started a new research project Diagram Predicate Framework (DPF) to unfold the full potentials of MDE. The aim of DPF is to formalise the concepts of MDE such as metamodelling, model transformation, model management and version control based on category theory and graph transformations. DPF extended the formalism of generalized sketches developed by Zinovy Diskin and a group of researchers from Latvia [5].

DPF is a graph-based specification framework and provides diagrammatic metamodelling. In diagrammatic modelling models are formalised as *diagrammatic specifications*, which is a combination of a underlying graph and a set of diagrammatic constraints. So, metamodelling considers both typing and constraints.

The syntax (structure) of the specifications are defined by the graph and part of the graph marked with atomic constraints, that specifies this part of the graph should have special semantics. Here we present the definition of the Signature, Specification and Atomic constraints referred from [31]:

- A signature $\Sigma = (\Pi^{\Sigma}, \alpha^{\Sigma})$ consists of a collection of predicates Π^{Σ}, each having a symbol $\pi \in \Pi^{\Sigma}$, an arity (or shape graph) α^{Σ} , a visualisation and a semantic interpretation.
- A *specification* $\mathfrak{S} = \left(\mathrm{S}, C^{\mathfrak{S}} : \Sigma\right)$ consists of a underlying *graph* S to-

gether with a set $C^{\mathfrak{S}}$ of *atomic constraints* (π, δ) on S with $\pi \in \Pi^{\Sigma}$ for a given *signature* Σ .

- An Atomic constraint (p, δ) added to a graph S is given by a predicate symbol p and a graph homomorphism $\delta : \alpha^{\Sigma}(p) \to S$.

The kind of predicates, visualisation and semantic interpretation of predicates to be include into a signature will be based upon the specific modelling environment. The semantics of a predicate p is given by the set of its instances $\iota : O \to \alpha(p)$ where each ι is a graph homomorphism into the arity of the predicate [31]. Table 2.1 shows the generalised predicates used in service-oriented modelling.

The description for the predicates as shown in table 2.1

Multiplicity: The shape graph of the multiplicity predicate is two nodes and an arrow between them, the semantic interpretation are implemented in a java based validator and the symbol is [mult(m,n)]. The multiplicity of arrows between two nodes should be greater than or equal to m and less than or equal to n in the instance graph.

Irreflexive: The shape graph of the irreflexive predicate is one node and with a reflexive one arrow with the symbol is [irrf]. It forbids the morphism relation to the same node.

exclusive-or: The shape of the exclusive-or predicated with three nodes and two arrows and the symbol is [XOR]. If there is a [XOR] between arrows XY and XZ, then in instance graph either arrow of type XY or arrow of type XZ should be present. The [XOR] is not valid if both type of arrows exists or if both type of arrows does not exist.

The semantics of a specification is defined as follows[31]:

- The semantics of a specification is given by the set of its instances (I, ι)
- An Instance (I, ι) for a given *specification* $\mathfrak{S}$ is a *graph I* together with a *graph homomorphism* $\iota : I \to S$ which satisfies the *atomic constraints* $C^{\mathfrak{S}}$.

In diagrammatic metamodelling, DPF defines the relation between models at any two adjacent levels of the meatmodelling hierarchy by *conformance*. The conformance relation is defined by two properties namely; *typed by* and *conforms to*.

Typed by: If there exist a typing homomorphism between two underlying graphs $\iota[i] : S[i] \to S[i+1]$ of the corresponding specifications $\mathfrak{S}_i$ and $\mathfrak{S}_{i+1}$, then $\mathfrak{S}_i$ at level i is said to be typed by $\mathfrak{S}_{i+1}$ at $i+1$ level of the metamodelling hierarchy.

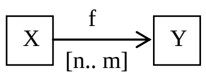
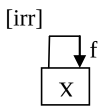
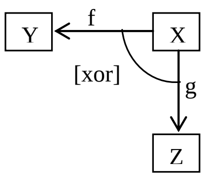
p	$\alpha^{\Sigma}_{2}(p)$	Proposed Vis.	Semantic Interpretation
[mult (n, m)]	$\begin{tikzcd} 1 \arrow[r, "f"] & 2 \end{tikzcd}$		$\forall x \in X : m \leq \lvert f(x) \rvert \leq n$, *with* $0 \leq m \leq n$ *and* $n \geq 1$
[irreflexive]	$\begin{tikzcd} 1 \arrow[loop right, "f"] \end{tikzcd}$		$\forall x \in X : x \notin f(x)$
[exclusive-or]	$\begin{tikzcd} 1 & 2 \arrow[l, "f"'] \arrow[d, "g"] \\ & 3 \end{tikzcd}$		$\forall x \in X$: $(f(x) = \emptyset \vee g(x) = \emptyset)$ and $(f(x) \neq \emptyset \vee g(x) \neq \emptyset)$

Table 2.1: Semantics of predicates

Conforms to: If there exist a typing morphism between two underlying graphs $\iota[i] : S[i] \rightarrow S[i+1]$, then $\mathfrak{S}_i$ is said to be conform to $\mathfrak{S}_{i+1}$ at $i+1$ level, if $(S[i], \iota[i])$ is an instance of $\mathfrak{S}_{i+1}$; i.e. $\iota[i]$ satisfies all atomic constraints[31].

2.8 DPF Workbench

DPF Workbench is a tool which implements concepts of DPF such as graph based formalisation of (meta)modelling and code generation based on metamodels. There has been a lot of effort in developing DPF tool since 2004 [17] but none of them were considered as a efficient tool for continuing development. In 2011, the first version of DPF editor was developed by Øyvind Bech [1] and Dag Viggo Lokøen using the technologies EMF and GEF with diagram specification editor with predefined signatures. With further extensions it was renamed to DPF Workbench. Which consists of a specification editor and signature editor and offers fully diagrammatic specification of domain-specific modelling languages [23] with code generation facility [32]. In DPF Workbench, the specification editor consists Node and Arrow of DPF metamodelling language.

The features of DPF Workbench are

- One can define an arbitrary number of metalevels in a metamodelling hierarchy
- Modelling in DPF Workbench provides graph-based formalisation for metamodelling
- Typing morphisms and constraint validators checks the conformance relation between metamodels and models in the metamodelling hierarchy
- Signature editor provides the facility to define new predicates with shape graph and graphical icon for domain specific modelling
- Signature editor allows to define arbitrary signatures
- Validators in signature editor facilitate semantics of the predicate to provide either in Java or in Object Constraint Language (OCL)
- By storing the DPF specifications in EMF data storage as XML metadata interchange (XMI), guarantees the interoperability with frameworks and tools
- Code Generator provides the facility to define transformation rules for model transformation from diagram model to textual representation

The figure 2.9 shows the DPF metamodel in DPF specification editor. At M_n level of metamodelling hierarchy DPF metamodel consists of Node to represent any object and Arrow to shows the relation between two nodes.

The figure 2.10 shows a specification of the metamodel for classes with attributes used to specify the SMODLmetametamodel on high level defined

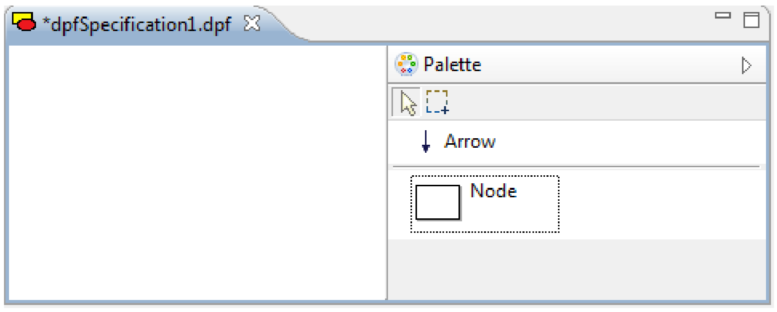


Figure 2.9: DPF Metamodel

at M_{n-1} with the DPF modelling language. The metamodel conforms to DPF metamodel.

- We defined Class and Datatype of type Node
- We defined Reference and Attribute of type Arrow

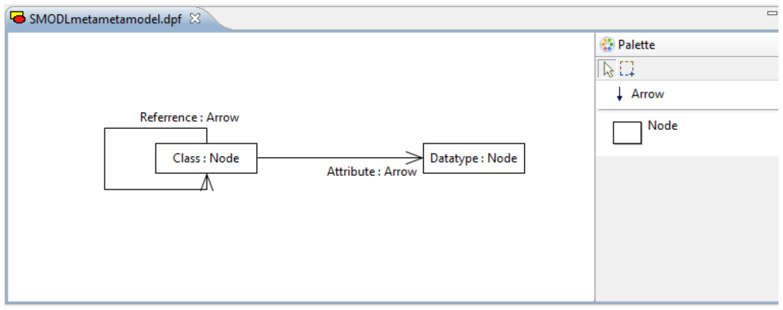


Figure 2.10: An example of metamodel specified by DPF metamodel

The figure 2.11 shows a subset of the SMODLmetamodel M_{n-2} which conforms to SMODLmetametamodel in the next lower level of metamodelling hierarchy. We defined Service, Method of type Class and isMethodOf of type Reference to represent the morphism between Service and method. The [mult(1,*)] predicate on the arrow isMethodOf to represent there should be at least one morphism between types of Service and Method in instance graph of SMODL metamodel in the next lower level of metamodelling. In DPF Workbench tool the default constraint is [mult(0,*)], if there is no predicate on any arrow.

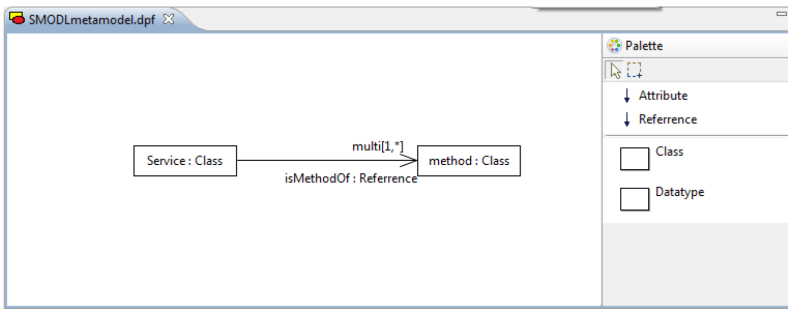


Figure 2.11: An example of SMODLmetamodel specified by SMODLmetametamodel

Figure 2.12 shows the selection of Signature editor from DPF workbench wizard. The figure 2.13 explains defining a new predicate [XOR4] in the signature editor used for SMODL modelling. According to documentation of SMODL model the Method can have result from any one of built-in datatype, array of built-in datatype, reference to a struct type and array of struct type. It is specified with a exclusive or constraint between four inputs. Once [XOR4] predicate is defined with visualisation and semantic interpretation, in OCL. To use this constraint in the next lower level of metamodelling we have to include the defined signature in the specification editor.

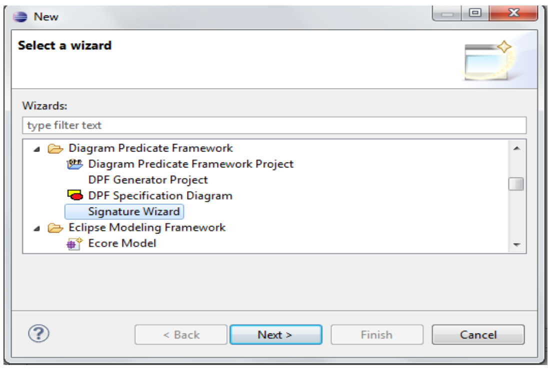
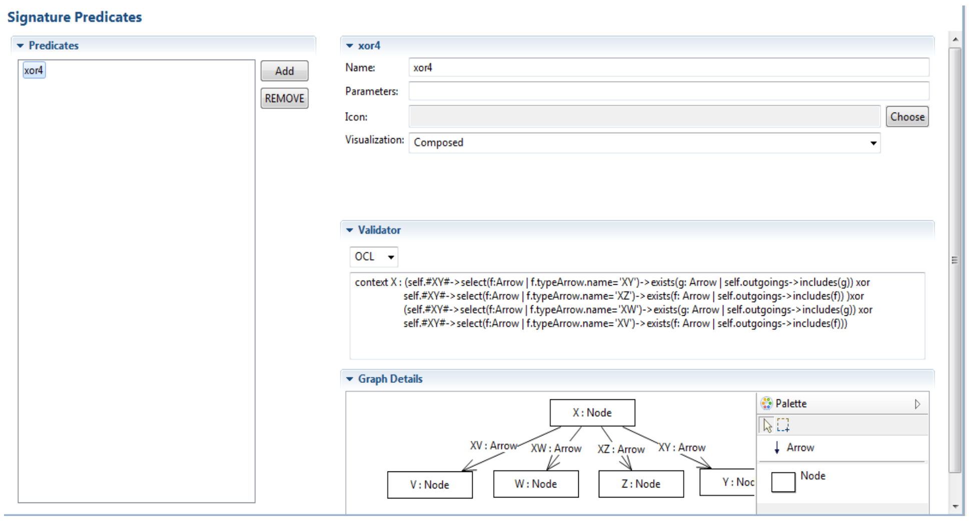


Figure 2.12: Signature wizard in DPF Workbench



Name: xor4
Parameters:
Icon:
Visualization: Composed

OCL

```
context X : (self.#XY#->select(f:Arrow | f.typeArrow.name='XY')->exists(g: Arrow | self.outgoings->includes(g)) xor
            self.#XY#->select(f:Arrow | f.typeArrow.name='XZ')->exists(f: Arrow | self.outgoings->includes(f)) )xor
            (self.#XY#->select(f:Arrow | f.typeArrow.name='XW')->exists(g: Arrow | self.outgoings->includes(g)) xor
            self.#XY#->select(f:Arrow | f.typeArrow.name='XV')->exists(f: Arrow | self.outgoings->includes(f)))
```

Figure 2.13: Signature editor in DPF Workbench

2.9 Metamodel based Code Generation

In 2012, Master's student Anders Sandven extended the functionality of DPF Editor (currently DPF Workbench), with the code generation based on metamodel using the Xpand SDK (1.1.1) [41] as a plug-in inside the Eclipse Indigo (3.7) [10]. The Code generation facility support model-to-text transformation. It provides a generalised solution for generating code to any domain specific modelling language (DSML). The features of Code generators supports clear expression of domain concepts, integration with Eclipse and standalone generators [32].

The features of the Code generators are:

Xpand metamodel: The Xpand metamodel is a core functionality of Code generators. It provides the mapping between DPF model types and custom Xpand types.

Type system: The type system enables Xpand to understand the modelling constructs such as DSML specific getter and setter or the functionality defined in the DPF Ecore metamodel.

Workflow integration: It allows to integrate the DPF metamodel with different components offered by the Xpand such as Modelling Workflow Engine (MWE).

Eclipse integration: Eclipse integration provides the support of using editors such as template editor for Xpand, extension editing with Xtend and constraints checking with Check.

Project environment: It facilitates a wizard for the generation of Code generation Project structure.

The figures 2.14 and 2.15shows how to create a Code generator project in Code generator wizard. We created a new project `no.hib.dpf.examples.smodl` for model transformation from diagram model to textual model. By default it creates template files for templ.xpt and workflow.mwe as shown in figure 2.16.

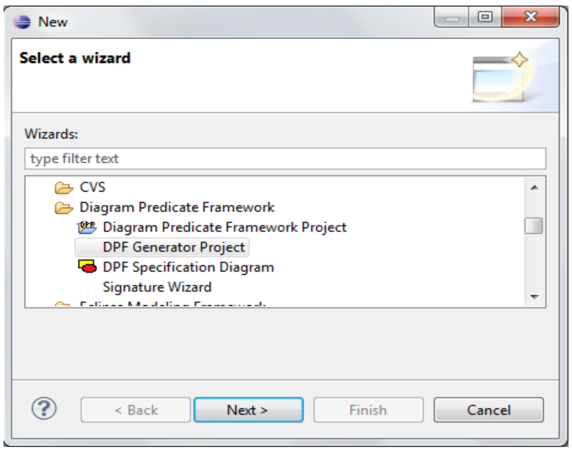
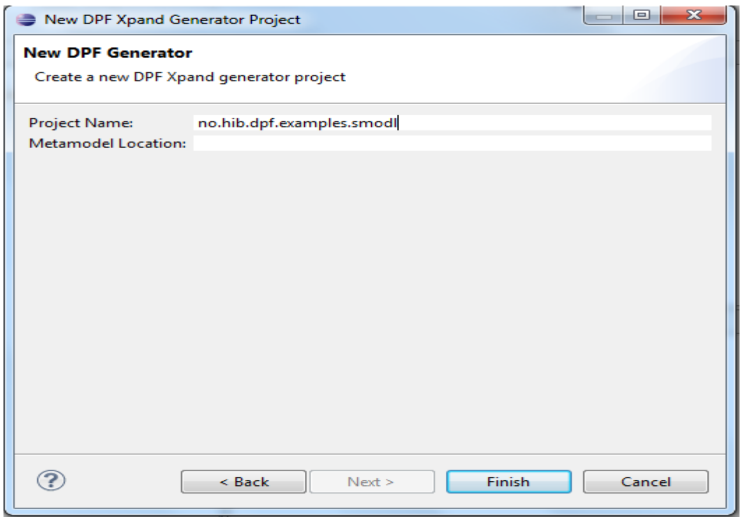


Figure 2.14: Creation of Code generator project



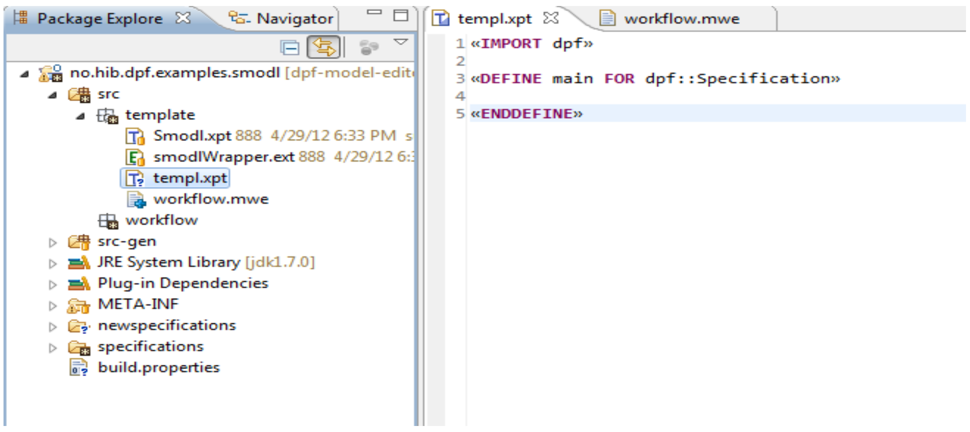
New DPF Generator

Create a new DPF Xpand generator project

Project Name: no.hib.dpf.examples.smodl

Metamodel Location:

Figure 2.15: Creation of Code generator project



```
no.hib.dpf.examples.smodl [dpf-model-edit
  src
    template
      Smodl.xpt 888 4/29/12 6:33 PM s
      smodlWrapper.ext 888 4/29/12 6:
      templ.xpt
      workflow.mwe
    workflow
  src-gen
  JRE System Library [jdk1.7.0]
  Plug-in Dependencies
  META-INF
  newspecifications
  specifications
  build.properties
```

```
«IMPORT dpf»

«DEFINE main FOR dpf::Specification»

«ENDDEFINE»
```

Figure 2.16: Creation of Code generator project with defalut templ.xpt and Workflow.mwe

Chapter 3

Problem and requirement analysis

This chapter explains about what type of research method and development methodologies that are used during problem analysis and requirement phase of this thesis. We also describe technologies that are used for the implementation of the solution.

3.1 Problem Anlysis

The first goal of the thesis is to define the metamodel for SMODL in the DPF Workbench. This involves a case study of SMODL model, which was developed by RUnit software.

According to Gary Thomas [36] Case studies are analyses of persons, events, decisions, periods, projects, policies, institutions, or other systems that are studied holistically by one or more methods. The case that is the subject of the inquiry will be an instance of a class of phenomena that provides an analytical frame—an object—within which the study is conducted and which the case illuminates and explicates.

DPF provides the formalisation of SMODL metamodel in a higher level of abstraction specified with the DPF formalisation. We introduce a formalisation by multilayered metamodelling hierarchy for services. The innovation of the first part of the thesis is to define metamodel for SMODL with required constraints. Evaluation of this part is done by comparing the defined metamodel in DPF Workbench with other approaches.

The second goal of the thesis is to implement a bidirectional model transformation from DPF SMODL model to textual SMODL model and reverse transformation from XML SMODL model to DPF SMODL model. It's a development of a new artifact for model transformation. We consider the research of this part as technological research, which is defined by *Ida Solheim* and *Ketil Stølen* [35] as follows.

Technology research is a research for the purpose of producing new and better artifacts. The technology researcher seeks principles and ideas for manufacturing of new and better artifacts, which may be materials, automates, medicines, oil production methods, computer programmes, etc. The basic question is: How to produce a new/improved artifact?

The innovation of this part is to develop an artifact for transferring DPF SMODL model to textual SMODL model with help of the Code generation tool as a transformation engine. Another artifact is to develop a plug-in for transformation of textual SMODL model to DPF SMODL model.

3.2 Development Process

This section describes what development methodologies are applied during development life cycle of this project. And also about the coding standards and technological tools used during development.

3.2.1 Development methodology

During this project time we chosen to follow the manifesto of *Agile development methodologies*, which was introduced in 2001 by a group of experts referred as Agile [4]. We adopted one of the well know methods *Extreme Programming* (XP) [2] as development method for this project. Also we now describe the the manifesto of Agile and how it was applied during the project as follows:

Individuals and interaction over process and tools: The team members communicate with others during the bi weekly DPF project meetings. Each DPF project team member have a chance to review about what they have done and what they need to do for the next meeting. Bi weekly meeting gave a chance to improve the technological knowledge by discussing with team members and suggestions from supervisor.

Working software over comprehensive documentation: The Source code of the project was never treated as a documentation. We always use to maintain the documentation about the code in simple and salient manner about the functionality. The next developer who is going to work on this project can easily understand whats happening in the code.

Customer collaboration over contract negotiation: RUnit software manager specified the contract in terms of requirements what they need us to develop. And provided the collaboration during the case study of SMODL service.

Responding to change over following a plan: We always use to have short term plans in development of software, if any changes are made by client, development process is easily adaptable to changes in development of software.

As said earlier, followed the XP as development methodology since it was not possible to practice all principles of XP. This project practiced some of the principles of XP in the development process such as:

Customer team member: Customer use to present whenever there is a requirement for discussion about the project.

Short cycles: As said early, we use to have a bi weekly project review meetings. When ever project comes to a demonstration stage, we use to have meeting with customer.

Continuous integration: We used Subversion source control as source code repository that is available at Bergen University College. Once the assigned task is finished we use to check in the code into subversion and check out the updated version of the DPF Workbench, whenever changes are made. The master student will leave the project on completion of their thesis, but the subversion repository will keep the back up of the source code to hand over to the next member of the project.

Pair programming: One of the key principle of XP is pair programming is not really practiced in this project. As this thesis is aimed on a single master student thesis, there is no chance of participate in pair programming.

3.2.2 Coding standards

One of the important practice of XP is Coding. At the starting stage of the DPF project, the team members has fixed a set of rules for coding standards. The code looks familiar and is easy understandable by just looking at name of the variable or a function. Coding standard practices supports the collective ownership and is easily understandable for a new entry developer.

Coding standards provides the coding consistency from developer to developer and avoid the conflict for naming conventions. Coding standards used in this project are Eclipse Naming Conventions [9]. According to the eclipse naming conventions, project should avoid company or person names in source code. Eclipse Naming Conventions have different naming standards for different elements in the project.

The naming conventions are used in this projects are

Projects: no.hib.dpf.examples.smodl

Package: no.hib.dpf.m2m.transformation

Class: SmodltoDPFTransformation

Method: protected void createTheDPFFiles

Variable: DSpecification newSpec

3.2.3 Technological tools

This section explains about the technologies used for this project. For providing the specification formalism of SMODL service, this project has been developed by using the DPF Workbench tool. During the implementation of textual to model transformation we used XML parser for parsing textual SMODL model and Eclipse Indigo (3.7) for developing a plugin for transformation from textual model to a DPF model.

Model transformation from DPF SMODL to textual SMODL model transformation (M2T)[7] there are several template engines like Java Emitter Templates, Acceleo, Xpand [7]. But as mentioned in section 2.9 Code generation facility of DPF Workbench is a model transformation engine based on Xpand SDK (1.1.1). Code generation facility allows to read the DPF metamodel with Xpand types. So, implementation of transformation from DPF SMODL model to textual model has been chosen to use Code generation tool as model transformation engine, Xpand and Xtend textual languages of XPand framework for writing code generation templates and Modeling Workflow Engine (MWE) [8] to control the execution of generators.

3.3 Xpand Framework

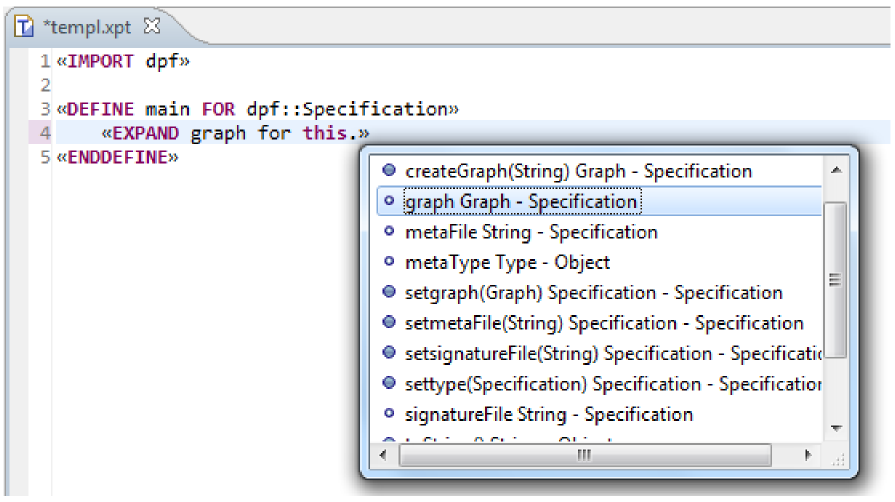
The section describes about the Xpand and Xtend textual languages of Xpand framework used for creating code generation templates. Modeling Workflow Engine (MWE) to control the execution of generators. Examples are given in this section for Xpand and Xtend textual language are from the implementation part of this thesis.

Xpand is included as a part of the Eclipse Model to Text (M2T) project and maintained by Itemis [18], one of the active strategic member for developing Eclipse products. It was originally developed by openArchitectureWare before it become a part of the Eclipse M2T [28].

Xpand is a generator framework with textual languages like *Xpand, Xtend, Check*. The execution of the generators will be controlled by Modeling Workflow Engine (MWE), which is under the project Eclipse Modeling Framework Technology (EMFT). The textual languages are constructed based upon the common expression language and type system, which reduces the time and effort to learn the each textual language. Type System provides the access to the registered metamodel implementations with their types. Expression language provides the concrete syntax for executing the expression by using the type system. The delimiter :: is used for the name space fragments. The fully qualified name looks as given below

- dpf::Specification
- dpf::Service

Figure 3.1 shows the types of the DPF SMODL metamodel registered for code generation.



```
«IMPORT dpf»

«DEFINE main FOR dpf::Specification»
    «EXPAND graph for this.»
«ENDDEFINE»
```

```
createGraph(String) Graph - Specification
graph Graph - Specification
metaFile String - Specification
metaType Type - Object
setgraph(Graph) Specification - Specification
setmetaFile(String) Specification - Specification
setsignatureFile(String) Specification - Specificati
settype(Specification) Specification - Specificatio
signatureFile String - Specification
```

Figure 3.1: DPF SMODL metmodel type system

3.3.1 Xpand

The Xpand language is used to control the format of generating output. The template file extensions are .xpt. The *guillemets* (« and ») are supported by the eclipse editor with key board short cuts Ctrl+< and Ctrl+>.

(1) of list 3.1 refers to the general structure of the Xpand template file and description of the statements in (1) are explained as follows:

IMPORT : There can be any number of IMPORT statements in a template file, followed by any number of EXTENSIONS and one or more DEFINE blocks. IMPORT states allows to include the namespace, which contains the fully qualified names of types and definitions.

EXTENSION: This statement allows to include utility extensions and components of *org.eclipse.xpand.util.stdlib* package. It also points to the *extend* files with .ext extensions. Extend files provides the additional features to the metamodel classes.

> *NOTE: In order to use org.eclipse.xpand.util.stdlib package, one need to add a dependency to the plug-in. To import Xtend file with .ext extension, it should be in the same class path of template file*

DEFINE: These are called as templates or definitions. Its a identifiable unit with template name and metamodel class name to which template is defined. (2) of list 3.1 refers to the example of DEFINE block *main* is the template name and the specification is the metamodel class. The DEFINITION body contains a sequence of statements including some text.

FILE: The target file name is expressed in the FILE statement followed by the file extension with concatenation operator. The body will contain any other statements. In the above listing of code, RepositoryService.xml file is created into file system with the name of the service class with file extension .xml.

EXPAND: It works exactly as the subroutine call. It expands another definition and output will be inserted at the place where it is defined and continues with another statement.

«EXPAND graph FOR this.graph» In that sentence *graph* is the unqualified name for the fully qualified type *this.graph*.

> *NOTE: The expanded definition should be in the same template file , otherwise it should be specified along with the template file name, like* «EXPAND Templatefile::graph FOR this.graph»

FOR vs. FOREACH: In the expand statement if FOR is specified then the definition is executed for the result of target expression. If FOREACH specified the definition is executed for the collection type of target expression. For example

«EXPAND graph » is equal to «EXPAND graph FOR this»

> *NOTE: FOR this* is the default one if none of them is specified.

```
(1)      «IMPORT dpf»

        «EXTENSION org::eclipse::xtend::util::stdlib::io»
        «EXTENSION template::smodlWrapper»
        «DEFINE service FOR dpf::Service»

        «FILE this.name+".xml"»
        «EXPAND graph FOR this.graph»

        «ENDFILE»
        «ENDDEFINE»

(2)      «DEFINE service FOR dpf::Service»
        «FILE this.name+".xml"»
        «EXPAND graph FOR this.graph»
        «ENDFILE»
        «ENDDEFINE»
```

Listing 3.1: Example for Xpand template

3.3.2 Xtend

Its a primarily object oriented language. Syntactically and semantically Xtend is similar to Java programming language with additional features such as Type inference, Recursions, Cached and Private extensions [11]. The main purpose of creating extensions is to support the model transformations. Xtend provides the facility to define the libraries for independent operations and non-invasive independent metamodel extensions in either java methods or Xtend expressions [41]. The Xtend file extension is *.ext*. Example of Xtend file is given in listing 3.2

```
import dpf;

extension org::eclipse::xtend::util::stdlib::io;

setQuotes(String strName):
        "\""+strName+"\"";

setNextArg(dpf::nextArg this):
"<arg␣type=␣"+setQuotes(setArgType(this.target))+"␣name="+setQuotes(this.target.name)+"/>";

String nextArgRecursion(dpf::nextArg this): this.target.getANextArgs().size>0 ?
setNextArg(this)+"␣"+nextArgRecursion(this.target.getANextArgs().first()):
setNextArg(this) ;
```

Listing 3.2: Example for Xtend template

The description about the statements in the Xtend file are as follows:

Import: To import the name spaces.

Extension Import: To import libraries and also another extension files.

Extension methods: These methods allows to add new feature to the existing types without modifying them.

The lines of code in (1) of list 3.3 shows the extension method to add the double quotes for given type name *strName*. These extension can be invoked as *setQuotes(strName)*.

Extensions are by default public, but if we want to hide the extensions need to add keyword *private* in front of the extension as shown in (2) of list 3.3.

If we are invoking the same extension very frequently, we can add the keyword *cached* to the extension to increase the performance by caching the result as shown in (3) of list 3.3.

Type Inference: For Extensions are not required to specify the return type, return type will be derived from the extension expression. (4) of list 3.3 shows how the the return type will be derived as *String* from the given expression.

> *NOTE : But for the Recursive extensions return type is not inferred, need to specify it explicitly*

(5) of list 3.3 shows the return type of recursive *nextArgRecursion* extension *String* is specified explicitly in the extension.

```
(1)     setQuotes(String strName):
        "\""+strName+"\"";

(2)     private setQuotes(String strName):
        "\""+strName+"\"";

(3) cached setQuotes(String strName):
        "\""+strName+"\"";

(4)     setQuotes(String strName):
        "\""+strName+"\"";

(5)     String nextArgRecursion(dpf::nextArg this):
        this.target.getANextArgs().size>0 ? setNextArg(this)+"
␣␣␣␣␣␣␣␣"+nextArgRecursion(this.target.getANextArgs().first()):
        setNextArg(this) ;
```

Listing 3.3: Example for Xtend extensions

3.3.3 Modeling Workflow Engine (MWE)

MWE is XML-based configuration file, which controls the execution of modelling workflow components in sequence order. Modelling components can be of model parsers, model validators, model transformers and code generators. In general MWE workflow consists of reader components and generator component. In the context of this thesis the components are code generator component and DPF metamodel reader component.

General content of the MWE file is explained as follows .

Properties: The properties can be declared in the same worlflow file or in another properties file. The name of the properties file will be included in the properties declaration as shown in (1) of list 3.4.

Components: Reader Component : The reader component will read the input dpf metamodel and stores the metamodel in a model slot named as *dpf* as given in lines of code (3) of 3.4. The metamodel to be read by the component should be instantiated before the reader component as shown in (2) of 3.4.

Generator Component: In generator first we have to specify the metamodel reference and then reference to the Xpand template file. Next is the path reference of the source folder where to place the generated files with the beautifier specification for the generated code as shown in (4) of list 3.4.

Beautifiers: To control the format of the generated output file. Xpand workflow component configured with two beautifiers. They are

<postprocessor class="org.eclipse.xpand2.output.JavaBeautifier"/>

<postprocessor class="org.eclipse.xtend.typesystem.xsd.XMLBeautifier">

Bt implementing *postprocessor* interface, we can define customized beautifiers.

```
(1)     For a general declaration of a property
        called as simple property
        <property name="dpf_model" value=""/>

         To include the property file
        <property file= "my.properties"/>

(2)     <bean class="no.hib.dpf.codegen.xpand.metamodel.DpfMetamodel" id="mm_dpf"/>

(3)     <component class="no.hib.dpf.codegen.xpand.metamodel.workflow.DpfReader">
                <dpfMetaModel value="${dpf_metamodel}"/>
                <dpfModel value="${dpf_model}"/>
                <metaModel idRef="mm_dpf"/>
                <modelSlot value="dpf"/>
        </component>

(4) <component class="org.eclipse.xpand2.Generator">
        <metaModel idRef="mm_dpf"/>
        <expand value="template::Smodl::main_FOR_dpf"/>
        <outlet path="${src-gen}$">

        <postprocessor class="org.eclipse.xtend.typesystem.xsd.XMLBeautifier">
                <maxLineWidth value="120" />
                <formatComments value="true" />
                <fileExtensions value=".xml"/>
        </postprocessor>

        </outlet>
        </component>
```

Listing 3.4: Workflow controls

As shown in figure 3.2 the overall view of the MWE controls the execution of generator components. Once the metamodel and model of the SMODL specified by the DPF workbench tool, DPF SMODLmetamodel should be instantiated in the workflow. Registering of DPF metamodel in reader component, allows to convert the DPF metamodel types to Xpand types and stores in the modelslot, this transformation is done though Code generator facility. Generator component controls the execution of Xpand and Xtend template files.

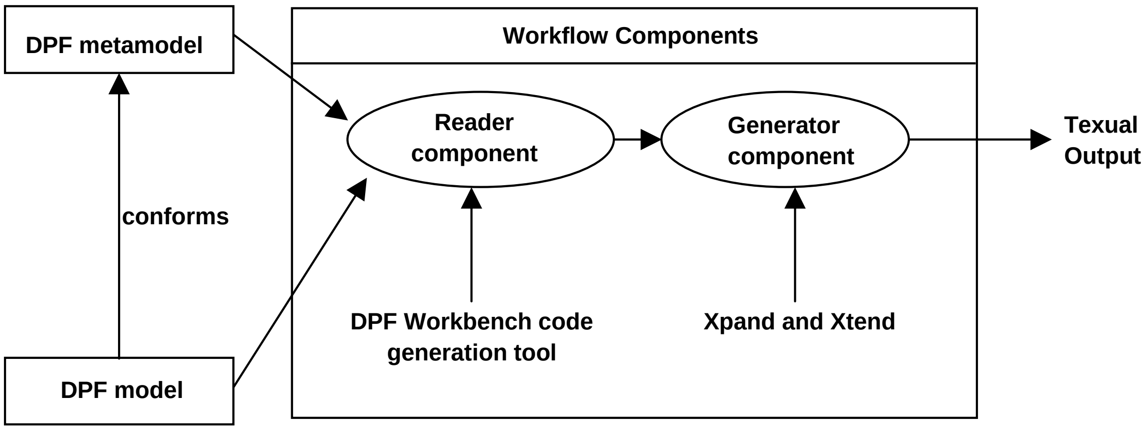


Figure 3.2: Workflow control

3.4 eXtensible Markup Language (XML)

For implementing the model transformation from textual SMODL model to DPF SMODL model we used a XML [40] parser for parsing the SMODL XML-dialect. The first working specification of the XML was released in 1998 by the contribution of the World Wide Web Consortium (WWW). The main purpose of XML is to transport and store data. In order to do transformation of a textual SMODL model to the DPF SMODL model we need to traverse the textual SMODl model.

In XML documents the data is stored in a tree structure with root and child elements. In order to traverse the tree structure XML parsers are required. XML parsers reads the XML documents and provides some mechanisms, that can be used by the programs. There are several types of parsers, some of the well known are Simple API for XML (SAX), Document Object Model (DOM).

SAX: It reads the each XML document and creates a event for each unit. So the calling program can avoid the unnecessary parts. Even it runs very fast. SAX does not provide much functionalities, the user need to provide their own functions for creating the data structures

DOM: It reads the XML document and stores the internal structure in the form of tree in the memory. DOM is a memory intensive and allows to read the document repeatedly. But it takes more time if the document is too large.

In the context of this thesis DOM parser has been chosen for traversing the XML document, as it is providing more functionalities to read the elements of the XML documents.

Chapter 4

Solution

This chapter explains how the modelling of SMODL is defined in DPF and presents a bidirectional model transformation of DPF SMODL model to textual SMODL model. We also show how this is implemented in DPF Workbench.

4.1 Defining models and modelling languages for SMODL web Services in the DPF Workbench

Before translating a diagrammatic DPF specification to a textual SMODL model, which in turn can be used for implementing web services. For this we need to define a modelling languages for SMODL in DPF.

According to Cesar Gonzalez-Perez and Brian Henderson-Sellers, a model is a statement about a given subject under study (SUS), expressed in a given language [15]. In the context of this thesis the SUS is SMODL services. While defining a DPF SMODL model in DPF, the structure of DPF SMODL model should coincide with the structure of the SMODL service. Both should be related by a structure, referred to as *homomorphic* [14]. Operations and properties of the entities in a model should provide the same functionality of the entities in SUS, i.e. entities in the model are substitutes for the entities in the SMODL service. If the DPF SMODL models are not homomorphic to SMODL service, it means that we are violating the defining properties of the models. Figure 4.1 shows a DPF SMODL model homomorphic to SMODL service and the relation between them.

The connection between model and SUS is described in two ways:

A forward-looking models: A model is created to specify the SUS, that doesn't exists.

A backward-looking models: A model is created, to represent an existing system in the absence of SUS.

According to Cesar Gonzalez-Perez and Brian Henderson-Sellers, there are three kinds of interpretive mapping between model and SUS [14]. They are

- Isotypical: Its a one-one relationship between entities of model and SUS.
- Prototypical: One entity of model can be mapped to more than one entity in SUS.
- Metatypical: One entity of model can be mapped to a set of entities in SUS declaratively.

The connection between a DPF SMODL model and a SMODL service is defined as *forward-looking models* representation, as shown in the figure 4.1 DPF SMODL model looking forward to create the SMODL service. All DPF SMODL entities should be connected to SMODL service entities but the reverse need not be satisfied. Figure 4.1 shows the general interpretive mapping between DPF SMODL model and SMODL service.

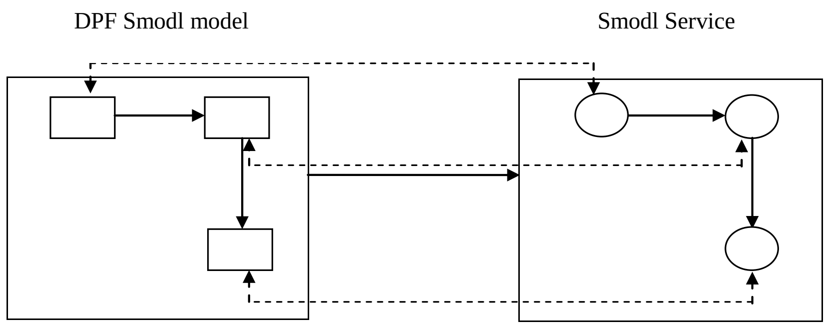


Figure 4.1: Homomorphic and Interpretive mapping of models

Before defining what kind of interpretive mapping that exists between the DPF SMODL model and the SMODL service, we will first define the modelling language for modelling DPF SMODL models and then we will continue with the mapping between them.

4.1.1 Modelling language for the DPF SMODL metmeta-model

A modelling language is defined as an organized collection of model unit kinds that focus on a particular modelling perspective [14]. A model kind is a specific kind of model, characterized by its focus, purpose and level of abstraction. In our context model kind is DPF and the kind of the SMODL model is given by the collection of DPF model unit kinds such as *Node* and *Arrow* not by the nature of the SMODL service. Collection of DPF model unit kinds are referred to as DPFmetamodel.

In the reference document of SMODL service [30] we can see that each element of the SMODL is a collection of other SMODL elements with some properties. So, we define the abstract SMODL metametamodel as shown in figure 2.10 in DPF at higher level and it consists of:

- Class and DataType of model unit kind of Node.
- Reference and Attribute of model unit kind of Arrow.

The figure 4.2 shows the relationship between DPF modelling language, DPF metamodel and SMODLMetaMetaModel at a higher level of abstraction of modelling at level M_n.

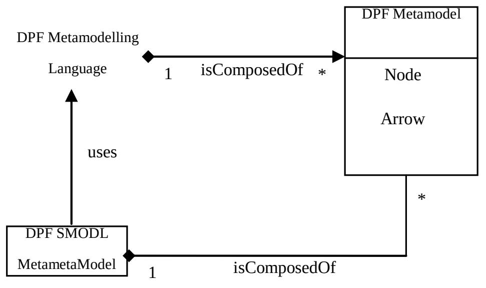


Figure 4.2: Relation between SMODLmetametamodel and DPF metamodel

At the level M_n, we can say the interpretive mapping between the DPF SMODL metametamodel and the SUS SMODL service is *prototypical* because the entity Class is mapped to a set of entities of SMODL service like Service, Method and so on. The DataType is mapping to a set of built in types such as int, long, string and so on. The Reference represents the association between two entities and Attribute represents the attribute type of the entities of SMODL service. Figure 4.3 shows the prototypical mapping between SMODL metametamodel and SMODL service at level M_n.

Here we are describing how the conformance relation are satisfied.

- The DPF SMODL metametamodel is specified by the DPF metamodelling language and is typed by the DPF metamodel.

DPF Workbench tool supports multi-layer modelling, we continued defining new modelling layers for DPF SMODL models until DPF SMODL model can be able to use by RUnit Software for further implementation of the web

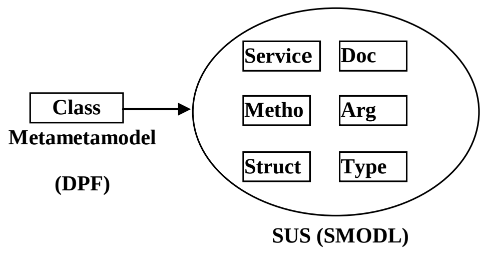


Figure 4.3: Prototyping mapping of DPF SMODL model and SMODL service

service. In the following section we will give the details how to define the modelling language for the next level of modelling.

4.1.2 Defining the modelling language for SMODLMeta-Model

All SMODL XML belongs to a namespace*http://smodl.org/v1*. In the specification of SMODL XML-dialect [30], the root element is Service. The elements of the SMODL specifications are given below:

Service: Service is collection of

- Zero or one doc element
- One or more method elements
- Zero or more struct elements
- Zero or more typedef elements

The attributes of the Service are name and targetnamespace that is a URI.

Method: Method is a sequence collection of:

- Zero or one doc element,
- A sequence of one or more arg elements, the sequence specifies the ordering of the of the method
- The Method should have a unique result

Doc: It specifies the semantics or other documentation of a particular SMODL element

Arg: It specifies the name and the type of the argument in a service method

Result: To specify the result of the method, it can be a built in data types or it can be struct data type

Struct: Its a structure or complex type with a collection of type fields. Optionally, struct can extends another struct. Its a collection of:

- Zero or more doc element
- One or more field elements

Field: It specifies the name and type of the field in a struct of the service, type can be a built in data type.

Typedef: Its a alias name of a particular type along with some constraints, the type need to fulfil

The DPF SMODL metamodel is an instance of DPF SMODL metameta-model with model unit kind Class, DataType, Reference and Attribute. At modelling level M_{n-1} the interpretive mapping between SMODL metameta model and SMODL service is a Isotypical and Prototypical mapping as shown in figure 4.4.

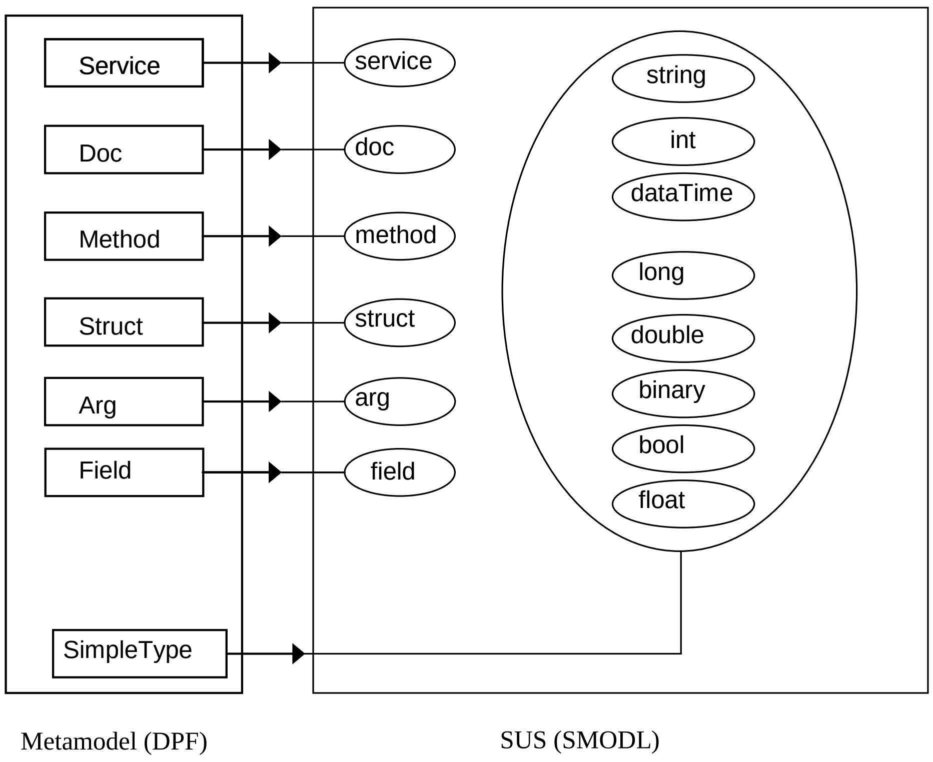


Figure 4.4: Isotypical and prototypical mapping between DPF SMODL metamodel and SMODL service

Service, Doc, Method, Struct, Arg and Field have entities in the DPF SMODL metamodel to provide the same functionality of entities as SMODL service. It is a one to one correspondence between DPF SMODL metamodel and SMODL service. So, its a Isotypical mapping. SimpleType can be used to map built-in types string, int, long, float, double, bool, binary and dateTime. So, its a one to many correspondence referred to as prototypical mapping.

The figure 4.5 shows the relationship between SMODLmetmetamodel, SMODLmetametamodelling language and SMODLmetamodel.

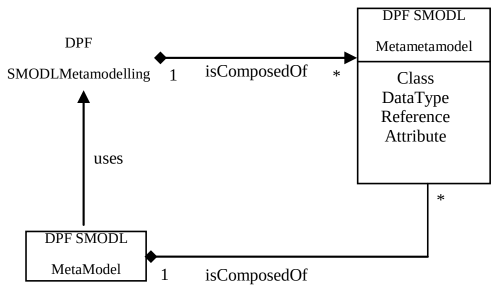


Figure 4.5: Relation between SMODLmetametamodel and SMODL metamodel

SMODLmetamodel is a concrete metamodel to define any SMODL model with all required constraint. SMODLmetamodelling language consists of Service, Method, Doc, Struct, Arg, SimpleType to refer built-in datatype, and references between nodes to show the morphisms. SMODLmetamodel model unit kinds can be used to define the model unit of DPF SMODL model and the figure 4.6 shows the relation between DPF SMODLmodel, DPF SMODLmetamodel and DPF SMODLmetamodelling language.

4.1.3 Metamodelling hierarchy of SMODL

While modelling SMODL service, at the level M_n we defined SMODLmetametamodel with the DPF modelling formalism and this model typed by the DPF metamodel. In the next modelling level at M_{n-1} DPF SMODL metamodel is defined with SMODL metametamodel and this model conforms to and typed by the DPF SMODL metametamodel. At each level we are defining the modelling language and model which can be used as metamodel for the next lower level modelling.

Figure 4.7 shows the multileveled modelling hierarchy of SMODL in DPF Workbench and the modelling hierarchy of SMODL in traditional approach from M_{n-1} to M_{n-3} .

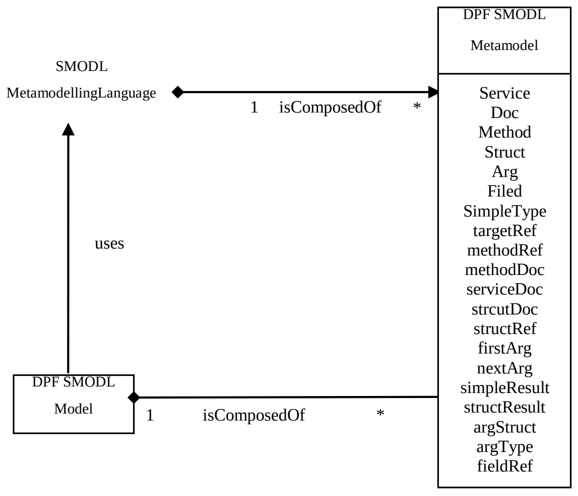


Figure 4.6: Relation between SMODLmetamodel and SMODL model

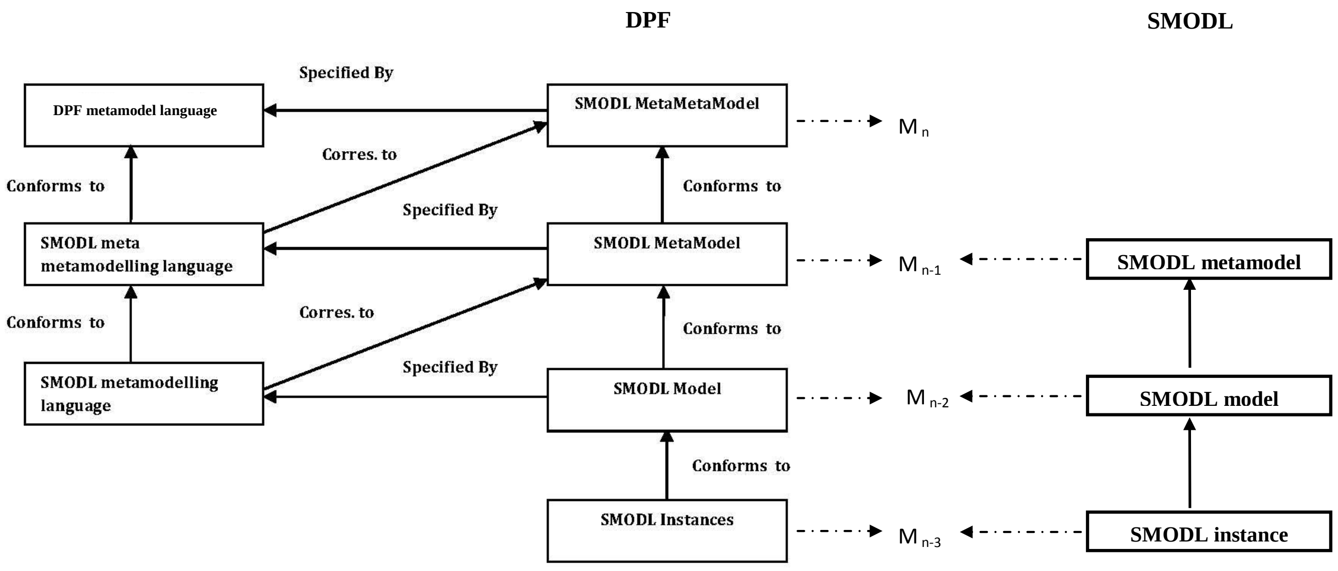


Figure 4.7: Multileveled modelling hierarchy of SMODL service in DPF editor

4.2 Demonstration of SMODL modelling in DPF workbench

This section demonstrate the modelling of a SMODL model in the DPF Workbench. We have defined a abstract SMODL metametamodel in DPF Workbench as shown in the figure 4.8.

- SMODL metametamodel consists of Class and DataType of type Node and Attribute and Reference of type Arrow.
- Class can be used to define any type of the SMODL entities and Datatype to define the attribute values of entities.
- References are used to define morphisms between Class types and Attributes that defines what type of attributes a Class type can have.

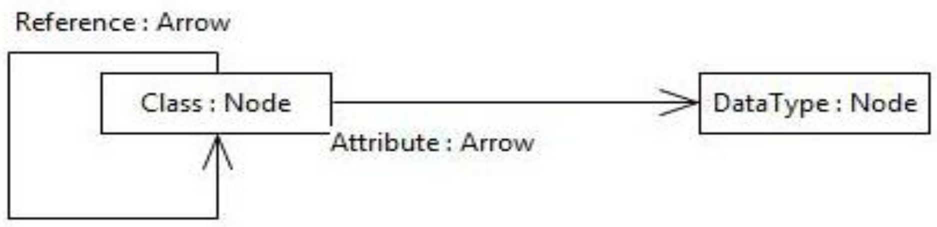


Figure 4.8: SMODLmetametamodel of SMODL service in DPF Workbench

The DPF SMODL metamodel in the next lower level can be used to define any SMODL model required for the implementation of SMODL web service. For defining a concrete SMODL metamodel we need to apply the atomic constraint in order to satisfy the constraints specified in the SMODL specification. In the DPF workbench multiplicity constraints has default value [mult(0,*)]. The following list shows, what constrains are applied in modelling of SMODL metamodel as shown in the figure 4.9.

- Service should have one targetnamespace, specified by [mult(1,1)].
- Service, method and struct can have zero or one doc element, arg can have reference to zero or one of next arg to follow the ordered of the arguments, these constraints are specified by the [mult(0,1)] constraint.
- Service can have one or more methods specified by [mult(1,*)].
- result type of method and arg can be any one of simple data type or can refer to name of the struct name, specified by the [XOR].

- A Struct can be extended to another struct but should not refer to itself, specified by [irrf].

Relation between model unit kinds of DPF SMODL metamodel and DPF SMODL metamodel as given below:

- Service, Doc, Method, Struct, Arg and SimpleType are of model unit kind of Class.
- target is of model unit kind of DataType.
- serviceDoc, methodRef, methodDoc, structRef, extends, structResult, simpleResult, firstArg, argStruct, argType, nextArg and structDoc are of model unit kind of Reference.
- targetRefis of model unit kind of Attribute.

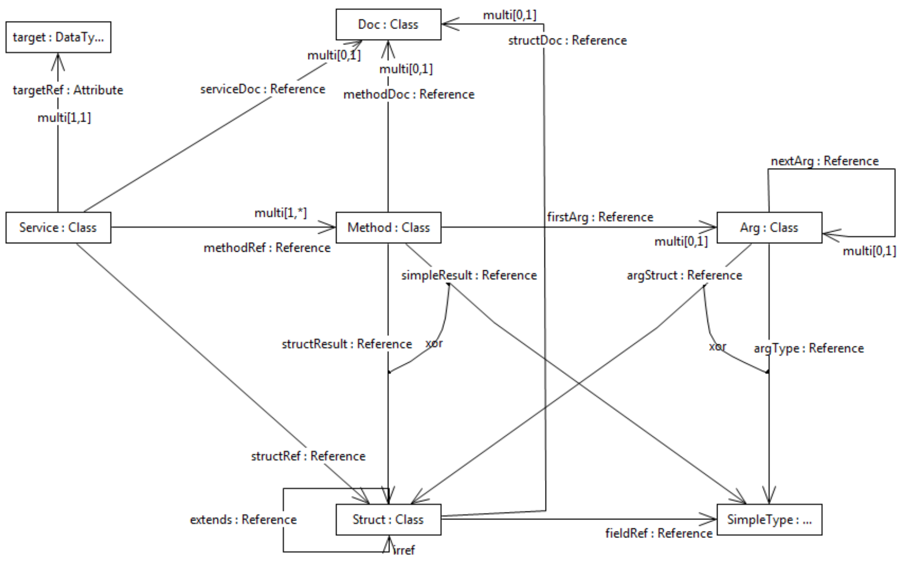


Figure 4.9: SMODLmetamodel of SMODL service in DPF Workbench

Finally we demonstrate the DPF SMODL model with a small example, which is created by RUnit Software to provide the service of a user to login, create or update and logoff their profile through the web service.

The figure 4.10 explains how SMODLmodel for a given example in list 2.2 is defined with SMODL metamodel and modelling languages and which conforms to the DPF SMDOL metamodel

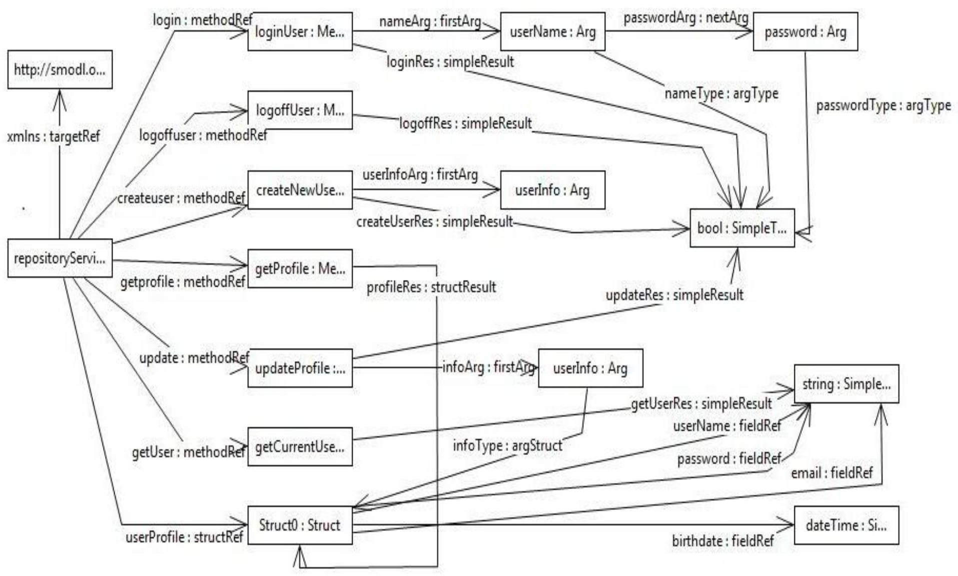


Figure 4.10: SMODLmodel of SMODL service in DPF Workbench

4.3 Bidirectional model transformation

The concrete DPF SMODL metamodel is defined in the DPF workbench. Now, its possible to define a DPF SMODL models by using DPF SMODL metamodelling language in similar way defined in the figure 4.10. But DPF SMODL model need to be transformed to the SMODL XML for further generation of SMODL service by RUnit Software. We also need to implement the reverse transformation from SMODL XML representation to DPF SMODL model. Because it is more convenient to make changes to the model in visual representation than in textual representation. More over the DPF SMODL metamodel ensures whether model validates all constraints or not.

As mention in section 2.3, a transformation is the automatic generation of a target model from a source model, according to a transformation definition. A transformation definition is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language. A transformation rule is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language. In order to specify the transformation definition, we need to clarify some aspects related to transformation:

- The primary artifacts of DPF are model, so we are performing a model transformation. The DPF SMODL source model is defined with DPF SMODL modelling language and the SMODL model in XML with the Relax-NG Schema language. As source and target models of SMODL are defined in different modelling languages, we used to create the *Heterogeneous transformation*.

- The main characteristics of this SMODL model transformation are
  - Fully automated.
  - Complexity of the transformation is medium, as we are using Code generator of DPF workbench tool as transformation engine, Xpand languages for output generation and MWE for controlling the execution of the workflow.
  - The behaviour of the SMODL model is preserved even though the source and target of SMODL models are defined in different modelling languages.

4.3.1 Transformation rules for DPF SMODL to textual SMODL model

The following rules are the transformation rules for transforming the DPF SMODL model to a textual SMODL model:

- The target textual SMODL model is created from the DPF SMODL source model by preserving the behaviour of the entities of SMODL service.
- The define and expand blocks of the Xpand template defines the pattern of which key elements of the source model are considered for the transformation.
- The generated target SMODL model should conforms to the meta-model defined in Relax-NG Schema.
- The transformed SMODL model should preserve the predefined entity structure of the textual SMODL model. i.e. The root element is a Service and is collection of:
  - Zero or one doc element
  - One or more method elements
  - Zero or more struct elements
  - Zero or more typedef elements

4.3.2 Transformation rules for textual SMODL to DPF SMODL model

We are parsing textual SMODL XML models to DPF SMODL in diagrammatic representation. For this we have chosen DOM parser for parsing the XML document, which stores the document in a tree structure and provides more functions to navigate in the tree elements of the document. Generating the source model from target model is a *backward-looking model* representation. The transformation rules for textual SMODL model to DPF SMODL model are as follows:

- The transforming generated DPF SMODL model should conform to its DPF SMODLmetamodel.
- The model units of the transformed Source model should consists model unit kinds of the DPF SMODL metamodel.
- The generated DPF SMODL model from the textual SMODL model should be a valid instance of the DPF SMODLmetamodel, i.e. it is satisfying all the atomic constraints.

4.4 Demonstration of the bidirectional model transformation

In this section we are explaining, how we implemented the model transformation from DPF SMODL model to textual SMODL model and transformation of a textual SMODL model to the DPF SMODL model.

4.4.1 Model transformation from DPF SMODL to textual SMODL

For implementing the model transformation from DPF SMODL to textual SMODL model, we created a new project by following the same naming convention as in DPF workbench tool. The project name is started with no.hib.dpf.examples, because we are modelling and implementing model transformation of a web service model as an example for model-driven service development in the DPF workbench tool. The web service SMODL was chosen as sub project. As shown in the figure 2.15 no.hib.dpf.example.smodl project is created in Code generator wizard. Because model transformation of diagrammatic to textual model in the DPF workbench is based on the Code generator metamodel. The nature of the Code generator is the Xpand framework project.

We need to specify the metamodel location in no.hib.dpf.codegen.xpand.ui.prefs as shown in the list 4.1

```
eclipse.preferences.version=1
metamodel.location=platform:/resource/no.hib.dpf.examples.smodl
                                   /specifications/SmodlMetaModel.xmi
```

Listing 4.1: SMODL Metamodel location in preferences

To run the Xpand template we have to define the workflow, which controls the execution steps of the components.

Defining the workflow

In the workflow file first we defined the location of the metamodel and model of the SMODL into the attributes of the properties in dpf_metamodel and dpf_model, can be referred in the workflow components. The workflow properties are:

- The location of the SMODL model is inserted in the value attribute by the name dpf_model in the properties.
- The location of the SMODL metamodel is inserted in the value attribute by the name dpf_metamodel in the properties.

The order of the components in the workflow are instantiation of the metamodel, registering the DPF reader component and then generator component. The generator component contains the Xpand Template information that need to expand against the instantiated metamodel and saved in the value of modelSlot in DPF reader component.

The given list of code in 4.2 gives the details how the MWE controls the workflow execution

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?><workflow>
        <!-- workflow properties -->
        <property name="dpf_model" value="platform:/resource
        /no.hib.dpf.examples.smodl/specifications/RepositoryService.xmi"/>
        <property name="dpf_metamodel" value="platform:/resource
        /no.hib.dpf.examples.smodl/specifications/SmodlMetaModel.xmi"/>

        <property name="src-gen" value="src-gen"/>

        <!-- set up EMF, only needed when using URI’s -->
        <bean class="org.eclipse.emf.mwe.utils.StandaloneSetup">
                <platformUri value=".."/>
        </bean>

        <!-- instantiate metamodel-->
        <bean class="no.hib.dpf.codegen.xpand.metamodel.DpfMetamodel" id="mm_dpf"/>

        <!-- DPF component -->
        <component class="no.hib.dpf.codegen.xpand.metamodel.workflow.DpfReader">
                <dpfMetaModel value="${dpf_metamodel}"/>
                <dpfModel value="${dpf_model}"/>

                <metaModel idRef="mm_dpf"/>
                <modelSlot value="dpf"/>
        </component>

        <!--  generate code -->
        <component class="org.eclipse.xpand2.Generator">
                <metaModel idRef="mm_dpf"/>
                <expand value="template::Smodl::main FOR dpf"/>
                <outlet path="${src-gen}">
                <postprocessor class="org.eclipse.xtend.typesystem.xsd.XMLBeautifier">
                        <maxLineWidth value="120" />
                        <formatComments value="true" />
                        <fileExtensions value=".xml"/>
                </postprocessor>
                </outlet>
        </component>
</workflow>
```

Listing 4.2: SMODL Workflow

Defining the Xpand template

We are using the Xpand template for generating the textual SMODL model. The template defines the format of the textual SMODL model in XML. We use Xtend to define extensions as sub routines, recursive extensions reduces the repeated complexity of the template and increase the reusability of the extension.

- At the start of the template we imported the namespace DPF, which is a name of the modelslot.
- And *extension import* to import smodlWrapper as name of the extensions file.
- We defined main as a definition for DPF specification, which contains the specification of metamodel. The root element of the Specification is Graph. So graph is expanded inside the main.
- The root element of the target SMODL model is Service. From graph we defined and expanded the Service element for doc, method and struct.
- The method blocks are defined and expanded for the method arguments and for the result of the method. The method arguments should be in the sequence order, for this we defined the extensions.
- We defined struct block and expanded for the fields.

The given list 4.3 shows the code for Smodl Xpand template

```
«IMPORT dpf»

«EXTENSION org::eclipse::xtend::util::stdlib::io»
«EXTENSION template::smodlWrapper»

«DEFINE main FOR dpf::Specification»
        «EXPAND graph FOR this.graph»
«ENDDEFINE»

«DEFINE graph FOR dpf::Graph»
        «EXPAND service FOREACH this.getServices()»
```

```
«ENDDEFINE»

«DEFINE service FOR dpf::Service»
        «FILE this.name+".xml"»
        <?xml version="1.0" encoding="UTF-8"?>
        <service name=«setQuotes(this.name)»
        xmlns=«setQuotes(this.getATargetRefs().first().target.name)»>

        «EXPAND servicedoc FOREACH  this.getAServiceDocs()»
        «EXPAND methodref FOREACH this.getAMethodRefs()»
        «EXPAND struct FOREACH  this.getAStructRefs()»
        </service>
        «ENDFILE»
        «syserr(this.name)»
«ENDDEFINE»

«DEFINE servicedoc FOR dpf::serviceDoc»
        <doc>«this.target.name»</doc>
«ENDDEFINE»

«DEFINE methodref FOR dpf::methodRef»
        <method name=«setQuotes(this.target.name)»>
        «EXPAND methodDoc FOREACH  this.target.getAMethodDocs()»
        «EXPAND firstargref FOREACH  this.target.getAFirstArgs()»
        <result type=«setQuotes((setMethodResult(this.target)))»/>
         </method>
«ENDDEFINE»

«DEFINE methodDoc FOR dpf::methodDoc»
        <doc>«this.target.name»</doc>
«ENDDEFINE»

«DEFINE firstargref FOR dpf::firstArg»
        <arg type=«setQuotes(setArgType(this.target))»
                name=«setQuotes(this.target.name)»/>
        «EXPAND nextArg FOREACH  this.target.getANextArgs()»

«ENDDEFINE»

«DEFINE nextArg FOR dpf::nextArg»
        «nextArgRecursion(this)»
        «REM»«setNextArg(this)»
        <arg name= «setQuotes(this.target.name)»
        type=«setQuotes(setArgType(this.target))»/>«ENDREM»

«ENDDEFINE»

«DEFINE struct FOR dpf::structRef»

        <struct name=«setQuotes(this.target.name)»>
        «EXPAND structDoc FOREACH  this.target.getAStructDocs()»
        «EXPAND field FOREACH this.target.getAFieldRefs()»
        </struct>
«ENDDEFINE»

«DEFINE structDoc FOR dpf::structDoc»
        <doc>«this.target.name»</doc>
«ENDDEFINE»

«DEFINE field FOR dpf::fieldRef»
        <field type=«setQuotes(this.target.name)»
        name=«setQuotes(this.name)» />
«ENDDEFINE»
```

Listing 4.3: SMODL template

Creating Xtend extensions

The *smodlWrapper* is created to include extensions required for creating the XPand templates. The advantage of the Xtend extensions are type inference and we can define recursive extension.

In SMODL XML-dialect, attributes values should be enclosed in double quotes. While defining the template instead of adding the double quotes every time while assigning values to the element attributes of SMODL, we defined the extension as shown in (1) of list 4.4

For setting the built-in type or name of the struct for the type of the argument in a method, we defined the extension as shown in (2) of list 4.4.

The arguments should be in order as defined in the method signature, they should be in sequence order. For this we defined a recursive extension, which checks recursively whether expanded method have any more arguments for that method signature. The lines of code in (3) 4.4 shows the recursive extension.

Methods should have one result. To define the result type of the method, we defined the extension as shown in (4) of list 4.4. This extension will check whether the result type is a bulit-in datatype of a name of the struct type.

```
(1)     setQuotes(String strName):
        "\""+strName+"\"";

(2)     setArgType(dpf::Arg this):
        this.getAArgStructs().size>0
        ?this.getAArgStructs().first().target.name:
        (this.getAArgTypes().size>0
        ?this.getAArgTypes().first().target.name:"");

(3)     String nextArgRecursion(dpf::nextArg this):
        this.target.getANextArgs().size>0 ?
```

```
         setNextArg(this)+"␣"+nextArgRecursion
         (this.target.getANextArgs().first()):
         setNextArg(this) ;

(4)     setMethodResult(dpf::Method this):
        this.getAStructResults().size>0?
        this.getAStructResults().first().target.name:
        (this.getASimpleResults().size>0?
        this.getASimpleResults().first().target.name:"");
```

Listing 4.4: SMODL Xtend extensions

4.4.2 Model transformation from textual SMODL to DPF SMODL

We created a plug-in project for the model transformation from the textual SMODL model to the DPF SMODL model. The given name of the project is no.hib.dpf.examples.smodl.transformation. The packages starts with no.hib.dpf.examples denotes its under examples of DPF project for implementing the model-driven service development. The sub-project name smodl.transformation denotes its implementation project for textual to model transformation. The following dependencies are added to the project:

- org.eclipse.jdt.core
- org.eclipse.core.resources
- org.eclipse.core.expressions
- no.hib.dpf.core
- no.hib.dpf.diagram
- org.eclipse.emf.ecore.xmi

In the plug-in properties file we changed singleton to true to allow for the creation of the extension as shown in (1) of list 4.5.

We added the extensions points for org.eclipse.ui.menus and org.eclipse.ui.commands in the plugin.xml. The lines of code shows of (2) in list 4.5 shows how to call the model transform plugin on the selection of .xml file.

```
(1)     Bundle-SymbolicName:
        no.hib.dpf.examples.smodl.transformation;singleton:=true

(2) <extension
     point="org.eclipse.ui.menus">
     <menuContribution
        locationURI="popup:org.eclipse.jdt.ui.PackageExplorer" >
           <command
              commandId="model.parser" label="Model␣To␣Model" style="push" >

              <visibleWhen
```

```
                 checkEnabled="false">
                   <with
                         variable="activeMenuSelection">
                         <iterate ifEmpty="false" operator="or">
                         <adapt
                               type="org.eclipse.core.resources.IFile">
                         </adapt>
                         </iterate>
                   </with>
                 </visibleWhen>
           </command>
     </menuContribution>
   </extension>
```

Listing 4.5: Model to Model menu extension

The command id for this extension is model.parser. The label appears on the selection of .xml file is *Model To Model*, here actually we are performing the text to model transformation, but the name given to the label as *Model To Model* is to mention both are model converting from textual SMODL model to DPF SMODL model. And the default handler for this is no.hib.dpf.m2m.transformation.SmodltoDPFTransformation, this class is the handler to take the action on selecting the label *Model To Model* on a .xml file. If the selected file is not .xml then a message dialogue will popup with message *Please select a SMODL xml file*. If we select a SMODL.xml file its creates the SMODL.dpf and SMDL.xmi files. Manually we need to store the location of the DPF SMODL metamodel in the variable META_DIAGRAM_FILE as shown in (1) of list 4.6.

NOTE:Assigning location of the metamodel should be changed in further development

The generated DPF SMODL model should conform to the DPF SMODL-metamodel. To obtain that first we are loading the resource set with the SMODLmetamodel for getting the type graph. And then creating a new specification for generating the source SMODL model. The new specification type graph will be filled with the graph of SMODL metamodel specification as shown in (2) of list 4.6.

The function call *XmlParser.SmodlXMLParser(file)* parses the XML document and stores the internal structure as a tree in memory. The summarized details for implementing the textual SMODL model to DPF SMODL model are given below:

- The root element of the document is Service.
- First get the root element Service and then get the attribute name and value of it as shown in (3) of list 4.6.
- For visualization of nodes we are starting from point(10,200) as the starting point of the location to place the nodes in the diagrammatic

editor specification. Changing the location according to the type of the nodes.

> *NOTE for further development: Location of the Nodes to the specification should be added dynamically*

- For adding the content of a Service to the specification, we need to check whether the Service type is exists in DPF SMODL metamodel.
- If it exists in the metamodel, then the attribute name and the value of it will be added to the graph of the source SMODL specification.
- Iterate through the child elements of the Service, and then first add the child element details of Struct to the specification if it exist. Because method results and argument type may refer to the name of the Struct and then add the child elements Method.
- Once Struct or Method element is added to the graph, then immediately we are adding particular element's child elements to the graph of the specification as shown in (4) of list 4.6.
- Every time when adding the specification graph nodes, we are checking whether that particular elements name exists in the type graph or not as shown in (5) of 4.6.
- For adding arrows to show the association between two nodes in the graph, we are taking into consideration of two document elements names and checking with the type graph arrow's source and target name by iterating the arrows list. Once if it matches, then we are adding the arrow of that particular typearrow to the specification graph as shown in (6) of list 4.6.

```
(1)     META\_DIAGRAM\_FILE =
        "D:\\runtime-EclipseApplication\\no.hib.dpf.examples.smodl
␣␣␣␣␣␣␣␣\\specifications\\SmodlMetaModel.dpf";

(2)     public static void dpfFileTempleteToGenerate(String dpfFile, IFile file){
                ResourceSetImpl resourceSet = DPFDiagramUtil.getResourceSet();
                newSpec = DPFDiagramUtil.loadSpecWithMetaModel(resourceSet);
                DPFDiagramUtil.loadSMODLModel(resourceSet,newSpec,dpfFile,file);
                XmlParser.SmodlXMLParser(file);
                DPFDiagramUtil.saveDSpecification(resourceSet,dpfFile);
        }

(3)     DPFTemplates.SpecificationContent(nNode.getNodeName(),
        getNodeValue(nNode,"name"),null,null,p.getCopy());

(4)     public static void getMethodAndStructdetails
                                (String childName, Node nNode,Point p){
         NodeList nServiceChildNodes = nNode.getChildNodes();
         Point nextMethodOrStruct = new Point(p);
        for(int count=0; count<nServiceChildNodes.getLength(); count++){
   Node nServiceChild = nServiceChildNodes.item(count);

   if(nServiceChild.getNodeType() == Node.ELEMENT_NODE) {
      if(nServiceChild.getNodeName().equalsIgnoreCase(childName)){

   DPFTemplates.SpecificationContent(nServiceChild.getNodeName(),
                 getNodeValue(nServiceChild,"name"),
                 nNode.getNodeName(), getNodeValue(nNode,"name"),
                  p.getCopy());
            getChildNodes(nServiceChild,p);
            p.setLocation(nextMethodOrStruct);
            p.translate(0, 100);
            nextMethodOrStruct.setLocation(p);

      }
      }
   }
   }

(5)   DNode node = DiagramFactory.eINSTANCE.createDefaultDNode();
      for(DNode typeNode : newSpec.getDType().getDGraph().getDNodes()){

      if(targetnodeName.equalsIgnoreCase(typeNode.getName().toString())){
      node.setDType(typeNode);
      node.getNode().setName(targetnodeValue);
      node.setLocation(p);
      if(targetnodeName.equalsIgnoreCase("SimpleType")
              ||targetnodeName.equalsIgnoreCase("Struct")){
      for(DNode existNode:newSpec.getDGraph().getDNodes()){
      if(targetnodeValue.equalsIgnoreCase(existNode.getName().toString())){
              existNodeInGrp = true;
              node = existNode;
              break;
              }
              }
              }
              if(!existNodeInGrp)
              newSpec.getDGraph().addDNode(node);
              break;
              }
              }

(6)   if(sourceNodeName!= null){
      DArrow arrow = DiagramFactory.eINSTANCE.createDefaultDArrow();
      for(DArrow typeArrow :newSpec.getDType().getDGraph().getDArrows()){
      if(typeArrow.getDSource().getName().equalsIgnoreCase(sourceNodeName) &&
              typeArrow.getDTarget().getName().equalsIgnoreCase(targetnodeName)){
      arrow.setDType(typeArrow);
      arrow.getArrow().setName(targetnodeValue);
      arrow.setDTarget(node);

      for(DNode sourceNode:newSpec.getDGraph().getDNodes()){
              if(sourceNodeValue.equalsIgnoreCase(sourceNode.getName().toString())){
                      arrow.setDSource(sourceNode);
```

```
        }
}
newSpec.getDGraph().addDArrow(arrow);
break;
}

}
}
```

Listing 4.6: Code for SMODL Model to DPF SMODL Model transformation

Finally when all the elements of the XML documents are added to the specification of the SMODL model. We check that the DPF SMODL model is a instance of the DPF SMODL metamodel by satisfying the all atomic constraint. If the XML document is not correct then we can easily identify the error in diagrammatic specification editor with error message for violated constraints. Invoking of textual model to diagram model by selecting on repositoryservice.xml on execution of model transformation plugin project is shown in figure 4.11.

The repositoryservice.dpf and repositoryservice.xmi files are generated in the same source folder for the corresponding repositoryservice.xml document. The location of the nodes are placed by calculating the location positions. Graphical representation is not clear as shown in the figure 4.12, it need to be improved to present in nicer way.

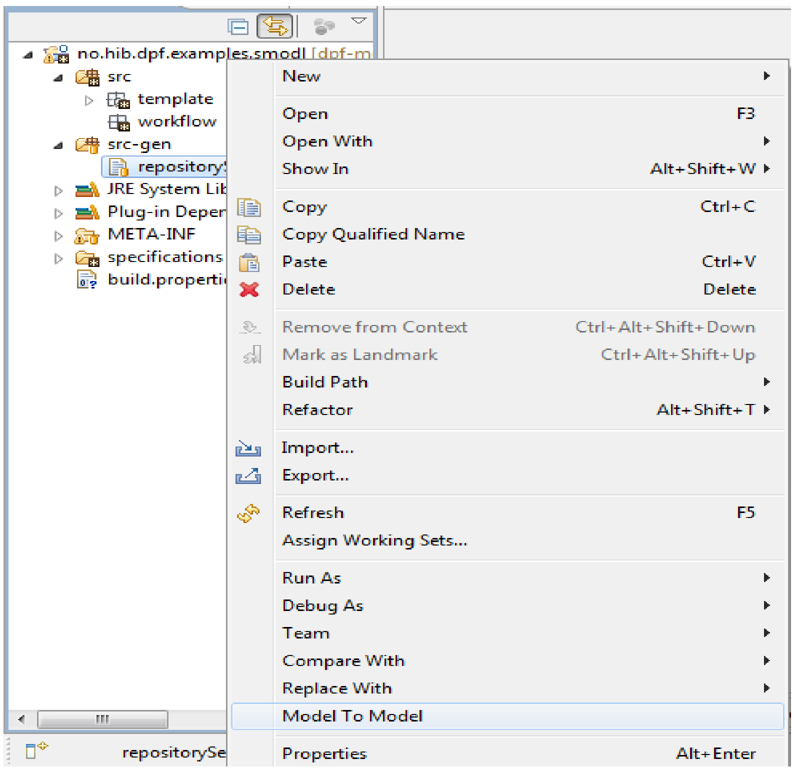


Figure 4.11: Model transformation from textual SMODL to DPF SMODL

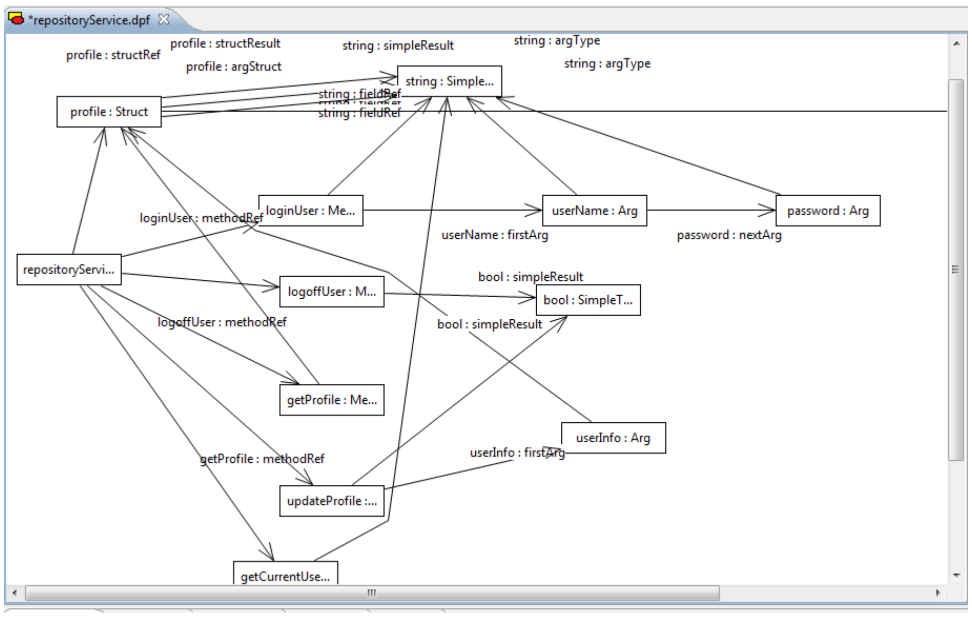


Figure 4.12: Generated DPF SMODL from SMODL XML-dialect

Chapter 5

Evaluation and Conclusion

5.1 Evaluation

This section shows the comparison of modelling and model transformation of SMODL service by use of DPF technologies with traditional approach by using the Smodl Development Suite (SDS)[30].

5.1.1 Traditional approach

We start by describing SDS, it automatically generates the code for web services from Smodl models. SMODL model is used to automatically generate the java classes. Further Java classes are used for the generation of web services. Some major characteristics of the SDS approach are:

- SMODL model is defined with the Relax-NG schema language, which is similar to XML.
- The generated SMODL model from the Relax-NG schema language is a XML document.
- SDS uses a textual representation of the modelling language and models are as represented in lists 2.1 and in list 2.2.
- To be able to use SDS approach the developer should have knowledge about the Relax-NG schema language and also XML.
- Its difficult to find errors in textual representation of of SMODL model.
- SDS will not show any error message, if any constraint is violated
- Modelling in SDS is following a 2-layerd metamodelling hierarchy, it doesn't support a multi layered metamodelling hierarchy. Figure 5.1 gives the overview of the metamodelling in the SDS approach
- SDS supports model transformation from SMODL model to Java code and reverse transformation from java code to XML SMODL model, again its a tedious work to find out the errors in XML SMODL as it is textual representation.

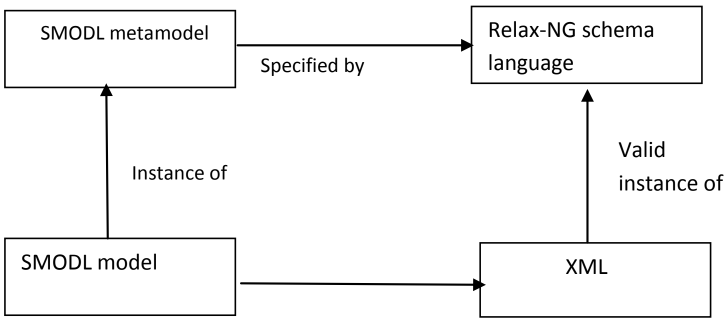


Figure 5.1: SMODL modelling hierarchy

5.1.2 Formal approach in DPF

DPF provides multilayered metamodelling hierarchy. In this thesis the formalisation of SMODL service with metamodelling and model transformation by using, adapting and extending the DPF Workbench tool. In chapter 4 we showed how we can define metamodelling and model transformations for SMODL service with formal approach. In the thesis, it has been proved that DPF can be used for model-driven service development by the formalisation of SMODL model. The summarised modelling details of DPF SMODL in the DPF Workbench tool are as follows:

- The representation of SMODL model in the DPF workbench is defined by a 4-layered metamodelling hierarchy from M_n toM_{n-3} as shown in figure 4.7.
- Relatively it is easy for the developer to use thr DPF modelling language. Because the DPF metamodelling language consists of only Node and Arrow as model unit kinds.
- The DPF workbench tool provides a signature editor, in which developers can define their own customised predicates such as [XOR4] for SMODL model as shown in figure 2.13.
- The DPF SMODL model is specified with DPF SMODL metamodelling language in, DPF SMODL model conforms to metamodel by satisfying all the atomic constraints. The DPF workbench tool shows if there is any constraint violation automatically. This do not required to check the constraints manually.
- In DPF workbench tool we defined the abstract SMODL metamodel from which we can define any SMODL model required for the implementation of web services.
- The bidirectional model transformation of SMODL in the DPF workbench is fully automated for transformation from a DPF SMODL model to a textual SMODL model and reverse transformation from a textual SMODL model to a DPF SMODL model.
- During the transformation of a DPF SMODL to a textual SMODL model,

the developer required to specify the location of the model and metamodel of SMODL.

- A well formed source DPF SMODL model is transformed to a textual SMODL model with transformation definition. We can ensure that the target SMODL model is also well formed and a valid instance of SMODL metamodel defined with Relax-NG schema language.
- We defined the reverse transformation from a textual SMODL model to a DPF SMODL model with a set transformation rules. So, the generated source DPF SMODL model is a valid instance of DPF SMODL metamodel. If there is any constraint is violated, DPF workbench tool shows which constraint is violated and gives error message. The developer can easily find out the errors in the generated DPF SMODL model in DPF workbench tool.
- The developers need some training to use DPF Workbench tool to get knowledge about modelling language and predicate semantics, even though their visualization represents their semantics in the default signature editor.
- We can conclude a model can be transformed from DPF SMODL to textual SMODL and again to DPF SMODL model as a valid instance.
- Finally, we have proved that DPF workbench tool can be used for model-driven service development by given a formalisation of the SMODL SDS with use of metamodelling and model transformation.

5.2 Conclusion

This section provides an overview of this thesis. How we have used metamodelling and model transformation to represent SDS SMODL model for generating code for the implementation of a web service in DPF Workbench tool. Though this thesis, we can conclude that the DPF work bench tool can be used for model-driven service development by the formalisation of a services such as SMODL service. We also proposes some further actions need to be taken in the direction of model-driven service development.

5.2.1 Summary

In this section we summarize, how we defined the modelling and achieved the bidirectional transformation between models SDS SMODL to DPF SMODL by extending the DPF workbench tool. We can define the metmodelling of SMODL service with modelling language types Node and Arrow in diagrammatic editor of DPF workbench tool. Using the Code generator tool, we can able to develop a tool to perform transformation of DPF SMODL model to textual SMODL model by using the Code generator as a transformation engine. We have also developed a pugin tool as sub-project of DPF workbench tool for transformation of textual SMODL model to diagrammatic SMODL model.

Through this thesis we achieved the following:

Modelling of SMODL

Metamodelling of SMODL in the DPF workbench is of concept for defining formal modelling of web services. Its a case study for model-driven service development. Following this approach we can a define a abstract metamodel for any specific domain and then continue to refine the model, until one can able to create the real object of the model. In the DPF workbench we defined the metamodelling for SMODL in 4-layered hierarchy as shown in figure 4.7. We defined a SMODL metametamodel at higher level of abstraction, which only consists of Class and DataType of Type Node and Reference and Attribute of type Arrow. This DPF SMODL metametamodel is a abstract model to define modelling of any web service. At each level of the modelling hierarchy, we are defining modelling language and a model for SMODL. In the next lower level of modelling hierarchy we defined the DPF SMODL metamodel which conforms to the SMODL metametamodel and specified by DPF SMODL metametamodelling languages. DPF SMODL metamodel is a generalised metamodel to define any DPF SMDOL model for further development of web service by SDS. By this formal approach we can a define modelling in a multi layered metamodelling hierarchy for a services.

Model transformation from DPF SMODL to textual SMODL model

In this thesis, we can conclude that, the DPF Workbench can used for model transformation of web services.We created a project $no.hib.dpf.examples.smodl$ for model transformation of DPF SMODL model to textual SMODL model. This project generates a well formed target SMODL model with the well defined transformation rules in transformation definition as shown in the subsection 4.3.1 and we have used DPF's code generator as transformation engine. We created the project in the Code generator wizard which was build on the Xpand framework. We used XPand textual languages to create a template and defined the well formed structure for generating target SMODL model. We have used extensions to define the routines required for generating well formed XML SMODL. We used MWE for controlling the execution of components.

Model transformation from textual SMODL to DPF SMODL model

We have developed a plugin project as a sub-project under DPF workbench tool. This project proves, we are able to perform the transformation from textual SMODL model to diagrammatic DPF SMODL model. We implemented this solution with a set of transformation rules in transformation definition. XML Dom parser for parsing the XML SMODL. By parsing the XML SMODL tree structure we created a .dpf and .xmi specification files for the DPF SMODL model. We calculated positions to locate nodes on the diagrammatic specification editor for the visualization representation of the specification. This project generates a DPF SMODL model, which is a valid instance of DPF SMODL metamodel by satisfying the all atomic constraints. Bu using this solution we get an an easier way for detecting the error in a XML SMODL by transforming into a DPF SMODL model in diagrammatic specification and using the DPF workbench for conformance check.

We can concluded this thesis formalises a web service SMODL by defining metamodelling and bidirectional model transformation in the DPF workbench tool, which are key concepts of Model driven development. Finally, we can say DPF workbench tool provides the formalisation web services.

5.3 Further actions need to be taken

In this section, we propose the some further work in modelling and model transformation for SMODL service in the DPF Workbench tool. We also suggestion some further actions need to be taken in the direction of model-driven service development.

5.3.1 Modelling and model transformation of SMODL

While modelling SMODL model in the DPF workbench tool, we haven't defined all the constraints required for modelling of SMODL. The solution need to be improved with all required constraints.

While implementing model transformation from DPF SMODL model to textual SMDOL model, we haven't implemented the *typedef* and array type of SMODL service. Because at the time of implementing this solution, Code generation tool was not supporting the Xpand types for predicates defined in the DPF SMODL metamodel specification. It has to be implemented in the future, when Code Generation tool supports Xpand types for predicates.

5.3.2 Code Generation

Code Generation in DPF workbench tool is a key tool for model transformations. Without code generation tool its not possible to perform model transformation from diagrammatic model to textual model. Code generation provides the support by transforming DPF modelling types to Xpand types. Using the Xpand types we can create the code generation templates for generating the output. But shortage in the Code generation tool is it does not provides the support to get the constraint details of the predicates defined between nodes or on arrow labels. It is not possible without the constraint types to satisfy all requirements needed for model transformation in model driven development. The Code generation tool need to be improved to implementation of model-driven service development in the DPF Workbench tool.

5.3.3 Graphical representation of diagrams in DPF workbench tool

In the DPF workbench tool, DPF metamodel is more convenient for modelling with Node and Arrow. Developers requires no training to do modelling with Node and Arrow. But if we refer to the figure 4.10 the presentation of the specification is so messy with many nodes and arrows for a small XML as shown in the list 2.2. The diagrammatic representation of the transformed DPF SMODL model from XML SMODL should be refined to place the nodes in the diagrammatic editor in nicer view. Model transformation from textual SMODL to DPF SMODL was not providing the optimal solution for calculating positions to locate nodes in editor. The visualization of diagrams in the DPF workbench tool could be nicer if classes and their attributes are in single diagram representation.So, we propose to change the concrete graphical representation of the models in nicer view of presentation to allow customisable visualization in the DPF Workbench tool.

Bibliography

[1] Øyvind Bech. DPF Editor – A Multi-Layer Modelling Environment for Diagram Predicate Framework in Eclipse. Master's thesis, Department of Informatics, University of Bergen, Norway, May 2011.

[2] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004.

[3] Michael Bell. *Service-oriented modeling:service analysis, design, and architecture*. Wiley, 2008.

[4] Robert C.Martin. *Agile Software Development principles, patterns and practices*. Pearson, 2012.

[5] Zinovy Diskin and Boris Kadish. Variable set semantics for keyed generalized sketches: formal semantics for object identity and abstract syntax for conceptual modeling. *Data Knowl. Eng.*, 47(1):1–59, 2003.

[6] DPF: Diagram Predicate Framework. *Project Web Site*. `http://dpf.hib.no/`.

[7] Eclipse M2T Project. Project Web Site. `http://www.eclipse.org/modeling/m2t/`.

[8] Eclipse Modeling Framework Technology. *Project Web Site*. `http://www.eclipse.org/modeling/emft`.

[9] Eclipse Naming Conventions. *Project Web Site*. `http://wiki.eclipse.org/Naming_Conventions`.

[10] Eclipse Platform. *Project Web Site*. `http://www.eclipse.org`.

[11] Eclipse Xtend. *Project Web Site*. `http://www.eclipse.org/xtend/documentation.html`.

[12] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation* . Springer, 2006.

[13] Dragan Gašević, Dragan Djurić, and Vladan Devedžić. *Model driven engineering*. Springer, 2009.

[14] Cesar Gonzalez-perez and Brain Henderson-Sellers. *Metamodelling for Software Engineering*. Wiley, 2008.

[15] Cesar Gonzalez-Perez and Brian Henderson-Sellers. Modelling software development methodologies:A conceptual foundation. *Journal of Systems and Softwares*, 2007.

[16] Pieter Van Gorp, Tom Mens, and Krzysztof Czarnecki. A Taxonomy of Model Transformations, 2005.

[17] Ørjan Hatland. Sketcher .NET – A drawing tool for generalized sketches. Master's thesis, Department of Informatics, University of Bergen, Norway, June 2006.

[18] Itemis. Company Web Site. `http://www.itemis.com`.

[19] Nicolai M. Josuttis. *SOA in practice:the art of distributed system design*. O'Reilly, 2011.

[20] Kaisler and Stephen H. *Software Paradigms*. Wiley, 2005.

[21] Steven Kelly and Juha-Pekka Tolvanen. *Domain-specific modeling:enabling full code generation*. Wiley, 2008.

[22] Yngve Lamo and Adrian Rutle. A metamodel approach to model driven service development, 2012.

[23] Yngve Lamo, Xiaoliang Wang, Florian Mantz, Wendy MacCaull, and Adrian Rutle. DPF Workbench: A Diagrammatic Multi-Layer Domain Specific (Meta-)Modelling Environment . In *Studies in Computational Intelligence*. Springer, 2012.

[24] Object Management Group. *Web site*. `http://www.omg.org`.

[25] Object Management Group. *Meta-Object Facility Specification*, January 2006. `http://www.omg.org/cgi-bin/doc?formal/2006-01-01`.

[26] Object Management Group. *Unified Modeling Language Specification*, May 2010. `http://www.omg.org/spec/UML/2.3/`.

[27] Object Management Group. *Unified Modeling Language Specification*, May 2010. `http://www.omg.org/spec/UML/2.3/`.

[28] openarchitectureware. Company Web Site. `http://www.openarchitectureware.org`.

[29] RUnit Software. *SMODL – Formal Specification*. `http://www.smodl.org/smodl-formal-specification.html`.

[30] RUnit Software. *SMODL – Simple MethOd Declaration Language*. `http://www.smodl.org/`.

[31] Adrian Rutle. *Diagram Predicate Framework: A Formal Approach to MDE*. PhD dissertation, Department of Informatics, University of Bergen, Norway, 2010.

[32] Anders Sandven. Metamodel based Code Generation in DPF Editor. Master's thesis, Department of Informatics, University of Bergen, Department of Computing, Mathematics and Physics Bergen University College, Norway, March 2012.

[33] Donald Sannella and Andrzej Tarlecki. *Foundation of Algebraic Specification and Formal Software Development*. Springer, 2012.

[34] Douglas C. Schmidt. Model-Driven Engineering. *IEEE Computer*, 2006.

[35] Ida Solheim and Ketil Stølen. Technology research explained. Technical Report A313, SINTEF ICT, Oslo, Norway, March 2007.

[36] Gary Thomas. A Typology for the Case Study in Social Science Following a Review of Definition, Discourse, and Structure. *Qualitative Inquiry*, 2011.

[37] Michal Walicki. *Introduction To Mathematical Logic*. World Scientific, 2011.

[38] Wikipedia. Modeling languages.

[39] Uwe Wolter. Category Theory and Diagrammatic modelling, 2011.

[40] XML. *Project Web Site*. `http://www.w3schools.com/xml`.

[41] Xpand. *Project Web Site*. `http://wiki.eclipse.org/Xpand`.