
A Meta-Modelling Language Definition for Specific Domain

Ph.D. Thesis



Zhihong Liang

Software Technology Research Laboratory

De Montfort University

2008

To *my wife*, Xing Wei,
my daughter, Jiarui Liang and *my parents*
for their love and support

Declaration

I declare that the work described in this thesis was originally carried out by me during the period of registration for the degree of Doctor of Philosophy at De Montfort University, U.K., from December 2003 to December 2008. It is submitted for the degree of Doctor of Philosophy at De Montfort University. Apart from the degree that this thesis is currently applying for, no other academic degree or award was applied for by me based on this work.

Acknowledgements

I would like to express my gratitude to the many people who helped me in different ways with research for this thesis.

My deepest gratitude goes to my supervisor, Professor Hongji Yang, for his inspiring supervision, constant support and encouragement during my study. He also provided me with many useful comments and suggestions for improving the thesis.

I would like to express my deep appreciation to my second supervisor Professor Hongzhi Liao, for his invaluable advice and support. I am grateful to my colleagues at Yunnan University in China, Professor Hua Zhou, Professor Tong Li and the other members of the Information Technology Institute.

I am very happy to have had the chance to study software engineering in the Software Technology Research Laboratory of De Montfort University. The laboratory provides such a friendly working atmosphere. A great many thanks go to Professor Hussein Zeden and the rest of the staff of the laboratory.

Finally, I am particularly grateful to my wife, Xing Wei, my daughter, Jiarui Liang and to my my parents for their love, encouragement and support over the years. This thesis is dedicated to them.

Abstract

Model Driven software development has been considered to be a further software construction technology following object-oriented software development methods and with the potential to bring new breakthroughs in the research of software development. With deepening research, a growing number of Model Driven software development methods have been proposed. The model is now widely used in all aspects of software development. One key element determining progress in Model Driven software development research is how to better express and describe the models required for various software components. From a study of current Model Driven development technologies and methods, Domain-Specific Modelling is suggested in the thesis as a Model Driven method to better realise the potential of Model-Driven Software Development.

Domain-specific modelling methods can be successfully applied to actual software development projects, which need a flexible and easy to extend, meta-modelling language to provide support. There is a particular requirement for modelling languages based on domain-specific modelling methods in Meta-modelling as most general modelling languages are not suitable. The thesis focuses on implementation of domain-specific modelling methods. The "domain" is stressed as a keystone of software design and development and this is what most differentiates the approach from general software development process and methods. Concerning the design of meta-modelling languages, the meta-modelling language based on XML is defined including its abstract syntax, concrete syntax and semantics. It can support description and construction of the domain meta-model and the domain application model. It can effectively realise visual descriptions, domain objects descriptions, relationships descriptions and rules relationships of domain model. In the area of supporting

tools, a meta-meta model is given. The meta-meta model provides a group of general basic component meta-model elements together with the relationships between elements for the construction of the domain meta-model. It can support multi-view, multi-level description of the domain model. Developers or domain experts can complete the design and construction of the domain-specific meta-model and the domain application model in the integrated modelling environment. The thesis has laid the foundation necessary for research in descriptive languages through further study in key technologies of meta-modelling languages based on Model Driven development.

List of Figures and Tables

Figure 2. 1 Modelling Pattern Series [11]	16
Figure 2. 2 Framework of MDA [76]	21
Figure 2. 3 Hierarchical Model in MDA	23
Figure 2. 4 Platform Independent Model Is Mapped To the Platform Specific Model.....	28
Figure 2. 5 Bridging the Abstraction Gap of An Idea in Domain Terms and Its Implementation [48]	31
Figure 2. 6 Architecture of DSM.....	35
Figure 3. 1 Workflow of Generative Programming.....	51
Figure 3. 2 Creative Workflow for Intentional Software [69].....	55
Figure 4. 1 Hall Three Dimensional Structure from Systems Engineering.....	68
Figure 4. 2 Hall-Management Matrix.....	72
Figure 4. 3 Relationships between Roles and Knowledge Structure in DSM....	74
Figure 4. 4 A General Modelling Framework based on Development of DSMLs	81
Figure 5. 1 Three Dimensions of Design Goals of XMML	92
Figure 5. 2 Layer Architecture Model of UML.....	98
Figure 5. 3 Layer Architecture Model of XMML	99
Figure 5. 4 XMML from the View of Model and Modelling.....	103
Figure 5. 5 Relationship Model between Syntax and Semantics of Modelling Language.....	106
Figure 5. 6 Abstract Syntax Model of XMML	109
Figure 5. 7 Example of Visual Primitive Definition.....	112
Figure 5. 8 Example: Creating Domain Rules by onElementCreate Event	115
Figure 5. 9 Concrete Syntax Schema of XMML.....	119
Figure 5. 10 Definition of ModelsType Schema	121
Figure 5. 11 PropertiesType Schema	122
Figure 5. 12 EventsType Schema	123
Figure 5. 13 SpecificationType Schema	124
Figure 5. 14 CodeGeneratorsType Schema	124
Figure 5. 15 RefEntitiesType Schema	124
Figure 5. 16 EntityType Schema	125
Figure 5. 17 RelationshipsType Schema	128
Figure 5. 18 RoleElement Schema	129
Figure 5. 19 Diagram Schema	130
Figure 5. 20 Diagram Form of Model Meta-Type.....	138
Figure 5. 21 Diagram Form of Meta-Typed Entity	139
Figure 5. 22 Diagram Form of Relationship of Meta-Type.....	140

List of Figures and Table

Figure 5. 23 Diagram Form of Referenced Entity of Meta-Type.....	140
Figure 5. 24 Diagram Form of Attribute Meta-Type.....	141
Figure 5. 25 Diagram Form of Constraints Meta-Type.....	142
Figure 5. 26 Diagram Form of Containment Meta-Type.....	142
Figure 5. 27 Diagram Form of ProvCont Meta-Type.....	143
Figure 5. 28 Diagram Form of Attachment Meta-Type.....	144
Figure 5. 29 Diagram Form of Self-Attached.....	144
Figure 5. 30 Diagram Form of Forming Attached Loop.....	144
Figure 5. 31 Diagram Form of Attachment Path.....	145
Figure 5. 32 Diagram Form of Meta-Type EntiCont.....	146
Figure 5. 33 Diagram Form of Reference Meta-Type.....	147
Figure 5. 34 Diagram Form of AssginRela Meta-Type.....	149
Figure 5. 35 Relationship between Text Concrete Syntax and that of Graphic	151
Figure 5. 36 Structure Described by Diagram.....	153
Figure 6. 1 Meta-Modelling Based on General Modelling Tool.....	161
Figure 6. 2 Meta-Modelling Based on Modelling Tool Generator.....	161
Figure 6. 3 Meta-Modelling Infrastructure Based on XMML.....	164
Figure 6. 4 Definition Model of ADL Meta-Model Element.....	179
Figure 6. 5 Definition Model of Attached Relationships among ADL Meta-Entity Elements.....	180
Figure 6. 6 Definition Model of Refinement Relationship for ADL Meta-Entity Element.....	181
Figure 6. 7 Definition Model of Refinement Relationship for ADL Meta-Entity Element.....	181
Figure 7. 1 Meta-Model Element of MetaEdit+.....	190
Figure 7. 2 Meta-Model Legend of MetaEdit+.....	191
Figure 7. 3 Architecture Style of Archware.....	194
Figure 7. 4 Sub-architecture of Archware View Components.....	195
Figure 7. 5 Sub-architecture of Archware Model Component.....	198
Figure 7. 6 Sub-architecture of Archware Controller.....	199
Figure 7. 7 Realisation of Domain Meta-Modelling in Archware.....	202
Figure 7. 8 Realisation of Domain Modelling in Archware.....	202
Figure 7. 9 Six Definition Models involved in Meta-Model Design of Archware	203
Figure 7. 10 Attribute Form Designer for Archware Modelling Element.....	205
Figure 7. 11 Modelling Element Attributes Editor Form Interpretive executed by Archware.....	206
Figure 7. 12 Code Editor for Primitive Physical Appearance.....	207
Figure 7. 13 Preview Window of Primitive Physical Appearances.....	208
Figure 7. 14 Script Designer of Primitive Events.....	208
Figure 8. 1 Model of ATM Transaction Processing System.....	213
Figure 8. 2 Definition Model of Meta-Model Element.....	216

List of Figures and Table

Figure 8. 3 Role Definition Model of Meta-Association Element	218
Figure 8. 4 Definition Model of Meta-Model Diagram	221
Figure 8. 5 Definition Model of Meta-Model Element	228
Figure 8. 6 Role Definition Model of Meta-Association Element	229
Figure 8. 7 Definition Model of Meta-Model Diagram	230
Figure 8. 8 Design of Target System	231
Table 5. 1 EntityType Schema Element.....	126
Table 6. 1 Refinement Relationship of ADL Meta-Entity Element	182
Table 8. 1 Meta-modelling Entity Element of ATM Transaction Processing System.....	214
Table 8. 2 Meta-modelling Associated Element of ATM Transaction Processing System.....	215

Chapter 1

Introduction

1.1 Motivation and Targets of Research

After 40 years of development, the software industry has become an important pillar of the modern information society. Today, productivity in software development struggles to keep up with the growing demand for software. As software becomes increasing large and complex, achieving quality is an ever growing challenge. Software complexity, diversity and volatility have become the very real problems that today's developers have to face.

How to make software development more efficient while maintaining quality has always being the focus of the software industry. However during software development the only never changing theme is change itself. Constantly changing user requirements, together with constantly changing implementation technologies, systems architecture and platforms are all factors making software development more difficult but they do not lie at the heart of the problem. The key problem is that traditional software development methods and development tools cannot adapt to these inevitable changes.

Almost 50 years of history, software development has progressed in its level of abstraction from the use of machine languages and assembly languages through to advanced languages. Now, Model Driven development is becoming a new research focus to further develop the level of abstraction so that software developers can more easily focus on the real nature of the tasks to be faced [11].

As abstraction levels are raised and Model Driven development methods are applied, thought processes are moving on from code-centric to model-centric

approaches. In essence this serves to further improve the abstraction level of development, so that developers can be freed from onerous and error prone code compiling tasks to concentrate on core domain problems of software systems.

Seeking to promote Model Driven development, the software industry is constantly exploring new methods, technologies and tools, such as Model Driven Architecture (MDA) [77], Language-Oriented Programming (LOP) [21], Language Workbenches, Generative Programming (GP) [124], Intention Programming (IP) [105], Software Factories and Domain-Specific Modelling (DSM) [49]. All these address the same question: how to implement the description of the system model of target domain? This is also the problem to be solved first in order to fully realise model driven development.

Of these, MDA from the OMG (Object Management Group) puts the emphasis on using Unified Modelling Language (UML) to establish a domain system model. The others advocate the use of DSLs designed according to target domain or DSMLs to establish models of the target domain software system. This suggests that combining domain-specific development with Model Driven development is an important direction for research and practice.

MDA is one of the most representative standardisation systems of Model Driven development. Recently, it has become the main focus of research in Model Driven development [122]. However after several years of research and pilot projects, the results produced by MDA in Model Driven development have not been as good as expected. At present, there have been few successful commercial applications of MDA. One of main reasons is that the modelling language used, namely UML has some inherent deficiencies in its power of expression of the characteristics of domain and in its ability to improve the abstract level of modelling. After all, UML is based on an object-oriented paradigm. Indeed it had unprecedented success in object-oriented modelling. But it lacks the necessary flexibility and extensibility when it is used to model the domain system. It is the case that OMG provides MOF (Meta Object

Facility) as UML's meta-modelling language and this does permit an extension of UML by modifying the MOF. However, the grammatical structure of the MOF and modelling elements are very complex. So, for the general developers and development organisations, if they want to further develop modelling tools by modifying the MOF this will be a very difficult task.

Compared with the unification and standardisation emphasised by MDA, DSM pays more attention to a simple, practical, economic and agile approach. It is a Model Driven development method system that is more suitable for typical development teams and organisations. From the perspective of meta-modelling, MOF can be described as a kind of "heavy-meta-modelling language" which suits MDA. While the research purpose of this thesis is to put forward a "light-meta-modelling language" which is suitable for DSM. Goals include providing adequate extensibility and facilitating tool support by keeping the construction of the language as tidy as possible.

Consequently, the research emphasis of the thesis addresses:

- (1) Syntax and semantics of meta-modelling language and its formal definition.
- (2) Hierarchical structural design of meta-modelling language.
- (3) Infrastructure design of meta-modelling language.
- (4) Visualisation of modelling elements and scalable definition mechanisms of meta-modelling language.
- (5) Support of the meta-modelling language through the architecture design of the integrated modelling environment.

The intention of this thesis is to seek a new solution for the spread and application of Model Driven development methods through research and exploration based on DSM methods and its meta-modelling language, and to lay a foundation for further study of Model Driven development methods.

1.2 Scope of Thesis

Using research based on meta-modelling theory and Model Driven software development technology to investigate the design of a visual meta-modelling language based on a Model Driven approach, and to use Domain Specific Modelling (DSM) to implement Model Driven software development. The research scope of this thesis is mainly focused on:

(1) An analysis and understanding of the differences and connections between the MDSD method and traditional software development methods, to include both the principles of MDSD and its application.

(2) An investigation of the inherent complexity and systematic characteristics of modelling activities and software system models from the perspective of system science and the research methods of system science to provide an insight into the implementation framework of DSM.

(3) A study of current research results relevant to modelling language and methods of Model Driven software development, and to an analysis of the similarities and differences between DSMLs and other Model Driven development methods, especially the architecture of DSM methods and its core constituent elements.

(4) A study of an infrastructure needed for the application of a DSM approach to Model Driven development and the corresponding implementation scheme and application framework.

(5) The combination of concepts and principles of domain-specific software architecture to the study and design of meta-modelling language suitable for DSM.

(6) A study of the provision of instantiation of meta-modelling language and modelling activities for a software system via the necessary tools and environmental support.

1.3 Original Contributions

By discussing and studying the above problems, the original work in this thesis:

(1) Puts forward an implementation scheme and application framework based on DSM and offers a Model Driven development process model based on DSM methods and a series of guiding principles. The method emphasises domain analysis and modelling as the core, and achieves agility, economy and efficient Model Driven development through rapid design and development of DSMLs (Domain-Specific Modelling Languages).

(2) Defines a visual meta-modelling language XMML, which is suitable for DSM and supports both the development and design of DSMLs and domain application systems. This includes a description of the design concepts of XMML and a formal description of its core elements.

(3) Provides a design for a meta-modelling infrastructure based on XMML, including meta-meta modelling which supports the design of DSM languages and a model reflection interface. Detailed modelling elements and design model of meta-meta modelling are given.

(4) Gives an implementation framework of visual integrated environment Archware based on XMML and according to the instantiation activities of the meta-modelling language. An analysis and explanation is given for the supporting tools and implementation schema provided by the integrated environment for making the design of the main members of its architecture.

1.4 Success Criteria

For Model Driven Development, a basic standard which can be used to judge the success of a modelling language is whether or not it can effectively improve productivity in software development together with the quality of the software produced. In order to focus on these criteria, the thesis mainly looks at the

following aspects in order to study and judge a modelling language and quality of its supporting tools.

(1) Can it effectively improve the abstract level of system development?

The modelling language must be able to achieve system modelling according to domain concepts. Only in this way can it really embody advantages brought by Model Driven development.

(2) Does it provide meta-modelling extensibility? In different

domain-specific software developments it is inevitable that there will be different requirements of the modelling language applied in the domain. Therefore, it is necessary to be able to customise and extend the modelling language so that it can be suitable for domain specific meta-modelling.

(3) Does it support system model descriptions of multi-view and

multi-level? Usually, a complex system model should be described from different angles. Only in this way, can a complete system requirement be comprehensively characterised. However, it is necessary to be able to take a step by step approach to disassembling and refining at different levels within similar models to clearly reveal the essential characteristics of a system.

(4) Can it provide a formal definition mechanism for the semantics of the

modelling elements? Formal semantic definitions of modelling elements and descriptions are necessary conditions for the conversion, iteration, refinement and testing of a model.

(5) Does it have a flexible reflection mechanism? During Model Driven

development, the boundary between modelling and development is becoming fuzzier. So, it is required that modelling language not only statically describes the target system, but is also capable of introducing the dynamic characteristics of an advanced programming language.

(6) Does the modelling language support visual definition? Visualisation is

an absolutely necessary and key characteristic of any modern modelling

language. It is essential that modelling elements should support visualisation, customisation and extension of interactive interfaces for design elements in different domains and views.

1.5 Organisation of Thesis

The rest of this thesis is organised as follows:

Chapter 2 gives an overview of Model Driven software development and domain specific modelling. Current challenges facing software development are discussed together with the basic connotations of a Model Driven approach. Related contents and progress with MDA which is the most representative of the Model Driven development approaches are also discussed. Finally, a method system for Model Driven development, DSM as studied in the thesis is given.

Chapter 3 discusses related work ranging from the perspective of domain specific development, current methods and technologies related to Model Driven development.

Chapter 4 investigates a general implementation framework supporting DSM methods. Model Driven development methods together with the core values and main features of DSM method are discussed. Ways of combining the organisation of traditional software engineering with basic theories of management technology, systematic engineering methods, and software architecture etc are explored. A general implementation framework based on DSM methods is put forward. An instructive implementation framework at engineering application level is given to include engineering methods applied to the implementation of DSM methods, role definition for developers, development frameworks, development environments and modelling languages.

Chapter 5 discusses the architecture of the visual modelling language XMML. The design characteristics of a domain-specific modelling language, as well as design goals and concepts of XMML are discussed. The design model of

XMML is given. The basics of the syntax and semantics of XMML are discussed. Special attention is given to a description of the abstract syntax of XMML, together with the concrete language and formal definition of its various modelling elements. The visual definition schema of the visual modelling language XMML is also given.

Chapter 6 offers a discussion of meta-modelling infrastructure based on XMML. The basic principles and implementation framework are discussed. Particular attention is paid to the architecture of meta-modelling infrastructure. According to the design of XMML given in Chapter 5, a meta-meta model based on XMML as well as a definition of meta-modelling elements and the composition of the model reflection interface are given. Finally, an example using the XMML meta-meta-model to describe ADL is also given.

Chapter 7 covers the design of the general integrated modelling environment. Development of current modelling tools is discussed. The characteristics of several general modelling environments are analysed. The architecture of an integrated modelling environment supporting XMML Archware is given together with supporting mechanisms for extensibility.

Chapter 8 provides two case studies. A modelling process for using XMML to implement domain application in integrated environments is given. A group with characteristics of domain application examples is used to show domain meta-modelling and modelling examples.

Chapter 9 contains the conclusions and overview. Research conclusion for a meta-modelling language based on the Model Driven development described in the thesis are summarised together with prospects for further research.

Chapter 2

Background

2.1 Challenge for Software Development

As software development increases in scale and difficulty, the cost of software development is increasing and reliability of software is harder to ensure. “Software crisis” [19] is a difficult problem that faces the software development industry. Decades have now past since software engineering came into being.

Many researchers have made contributions to the progress of software development methods. From the very early days of structured methods to the widely used object-oriented development methods used at present, these software development methods make for a software industry that is ceaselessly developing and progressing. However, the “software crisis” problem has not been solved. Software development, beginning with requirement analysis and proceeding through to final code implementation is still a long process. In 1986 Frederick Brooks could assert that “in ten years, there has been no single software engineering process bringing a major productivity improvement” [10]. The judgment is widely called the “silver bullet law” [10].

As a rule, user is inclined to believe that software development is easy. Developers can better understand domain problems by communication among themselves and with domain users. According to the information gathered they can design corresponding solutions and finally deploy the developed systems to their customers. But in this seemingly simple process, there are always some problems to make software development full of challenges [6].

Often, developers do not immediately gain a full understanding of all the problems of a particular domain. Meanwhile, the users of the domain will likely

have a comprehension of the problems that is limited to just that part with which they themselves are familiar. Besides, they look at the problems from different angles and are interested in different methods and processes. Their priorities for dealing with problems are different and different points of view appear. Some of the viewpoints conflict with each other. Therefore, in the actual development process, it takes system developers time to analyse the various data obtained from domain users and finally to synthesise from these a system requirement specification that covers all the domain problems.

When designing a system solution, it is necessary to consider many limiting factors. These can include the time required to realise a system, balancing difficulties affecting realisation, integration with current applications, systems or technologies and coordinating the use of components developed by several different development teams working separately on the overall system.

In the development stage, various changes will be required. For example: errors of system design may be discovered with a need to adjust the original design solution; user requirements may change; the planned development priority of system modules may change; the original implementation code may be redrawn and so on. Such changes are hard to accurately estimate in advance. Change is a problem that faces any software development process. The developer must take a lot of time to understand and analyse these changes. Consideration must be given to what effects will there be on system development. The necessary plans and solutions to deal with change must be put in place and care taken to ensure that the measures are appropriate and effective.

A large software engineering project usually needs various specified technically competent staff to take part in the project. Their work must be coordinated and this coordination must continue for a long time after development has been completed and the system has been put into use. The result is that we take into account many project management technologies. These can include process management, resource management, risk

management, ROI analysis, document management and supporting services.

Development technology for software systems is becoming increasingly difficult as the related application domains become more and more widely spread. Developers must be able to apply many different technologies, including the popular object-oriented technology XML, script language, interface definition language, procedure definition language, database definition and query language. In the actual projects, the developer should have a deep understanding and grasp of many technologies, architectures, protocols and tools. Only in this way can we see a smooth transformation from the requirements of a problem domain into corresponding solutions.

Core issues for software engineering research include: the challenges that can be found anywhere during software development; shortening the development cycle: improving development efficiency and software quality; and adapting to changing requirements.

Some domains besides software development, such as software architecture, electronic products and car manufacturing have successfully incorporated design for automation or semi-automation into their production process. In this way they have effectively achieved production at low-cost and with high reliability and high efficiency. It is reasonable to ask, is it possible to use the same thinking and principles to construct software systems and so substantially improve software development efficiency and software quality? In the light of the experience of these project domains, in recent years many software development methods and process control strategies have been put forward. These include software architecture, web services [44], MDA [76], RUP [29], agile development and CMM. They help to decrease software development costs and to improve software development process and software quality.

For the software development process, we need to have better forecasts, visibility and reliability for the whole software life cycle. For software

development methods, we need better and more efficient automation development methods for coding. Driven by these requirements, MDD becomes a new research hotspot of the software industry and it has been predicted that it will be the most important software development methodology over the next several years [92].

2.2 Model Driven Development

2.2.1 Introduction of Model Driven Development

Model Driven development (MDD) [62] is a new software engineering method developed from object-oriented development methods. It is a new approach that can be suitable for many methods of software development. At its core is a system model design providing best practice in the construction of the software system model. The model guides various stage such as requirement analysis, system design, code design, system-testing and system maintenance. MDD involves such technologies and methods as model description, modelling methods, model transformation and code generation.

MDD does not completely abandon earlier software development methods and technologies to develop its own new methods. Instead, it is one further step in an ongoing process based on sound software development methods. A study of MDD can serve to explore and consolidate methods for improving software production, extending these step by step. In order to make the successful application of MDD practicable, further progress is necessary in current software development methods and technologies. Let us now consider some of the main technologies and methods supporting and accelerating the development of MDD.

(1) The development and progress of software programming languages progressed from the original assemblers to the current general advanced programming languages. This has led to significant productivity improvements

in software development. During the development process, the level of abstraction in program descriptions of software systems constantly improved and developers now use higher level language tools for coding.

(2) Component-Based Development (CBD) [3] extracts experience gained in industrial production process and applies this to create flexible components to be assembled in creating applications. Such components mean it is not necessary to re-invent the same solution in different application programs. So productivity is improved and development costs are decreased.

(3) The use of design patterns [32] makes a big contribution to the improvement of software development efficiency and quality. The concepts of design patterns are key points for industrial production flows. With these for guidance, developers can reuse general design patterns already created and proven in use by others.

(4) Middleware [5] is a step in improving abstraction of computer platforms beyond the operating system layer. Middleware makes application developers focus more on business logic rather than on functions when considering how best to provide message mechanism, business control and security.

(5) Declarative Specification [80] refers to compiling systems by configuring attribute values. Meanwhile, Imperative Specification refers to writing systems by giving instructions that follow a prescribed order of execution. Using Declarative specification developers replace relatively complex programming code with relatively simple declarations.

(6) Application Framework technology [27] constructs a basic implementation framework for the full application system. Layer architecture and the technique of splitting the focus point when organising a complex system framework is helpful as it allows system change to be limited within one part of the system. At the same time, having an application framework means developers can effectively make necessary modifications in response to system

change.

(7) The concept of Design by Contract [67] is a method for building reliable software. In a visual and formal way, it places the emphasis on specifying contract relationships among software components. This is of benefit in driving high-quality software engineering.

(8) Object-oriented method. Object-oriented development technology brought a big leap forward in software development technology and had an important role in facilitating the emergence of large scale software. The development and application of OOP, OOD and OOA led in turn to the Object-oriented software development method.

MDD is a development method that integrates a series of new technologies. It provides a new solution in improving software development productivity, enhancing software quality and in the system maintainability of existing software engineering. The application of MDD can bring increased automation to the design process for software systems and enhance design consistency and system maintainability within each phase of the software system development process.

2.2.2 Model and Modelling

Model is the main software artefact of MDD and code and other software artefacts can be generated by the model. The model provides an abstract description of the target system. It can help system developers look beyond less important details and focus on more important parts of the system. Many projects depend on the model in order to understand complex, real world systems. Model's many uses include: predicting system quality when some aspects of the system are changed; understanding specified attributes; and helping stakeholders to communication effectively about key system characteristics.

The model can be developed as an outcome of the implementation of the software system or to model an existing or developing system as an aid to understanding system behaviours [92]. By modelling the target system to an abstract system, key problems are revealed and the problem domain and its solution domain can be better described. Modelling objects and relationships allow us to record and describe the mapping of relationships from problem domains to solution domains and to resolve problems at the model stage. Advantages include helping not only developers but also users of the target system better understand, analyse and estimate system design accuracy and reliability and to timely discover the potential problems of system design by building a system model before the full system is realised. It means developers can examine every aspect of the model to deeply consider and analyse the system to ensure a correct understanding of the system.

Modelling has a long tradition. For thousands of years, engineers, artists and craftsman have built models to test design solutions before putting their projects into effect. Software systems are no exception to the benefits of modelling. In software development, various aspects of the model reflect the actual system. These include the logic model, maths model, architecture model and the mixed model. In MDD, it is necessary to integrate several types of model to replace a final system realisation model written by hand. In recent years, many researchers have done much work in the field with the most visible result being the appearance of a large number of modelling tools based on object-oriented development technology.

Most recent reforms pay attention to symbols and tools. These tools allow users to easily map the model into a specific operating system and so be able to express a valuable system view to assist developers with architecture and programming language code. The current state of practice is that UML provides the preferred modelling symbols [8]. UML allows the development team to get key characteristics of various aspects of corresponding models. Transformations

among these models are carried out by hand. UML modelling tools typically support the tracking of requirements and dependent relationships among modelling elements. In the context of a large scale development they provide guidance on best practice using synchronous models through supporting documents and complementary consultation information [1].

As software engineering continues to develop, various system modelling patterns are put forward to further enhance the proportions and status of models and modelling in software system development activities as shown in Figure 2.1. This shows a series of modelling methods used by today’s software developers [11]. Each can help the software developer create application code to be run on specific run-time platforms and relationships between models and code.

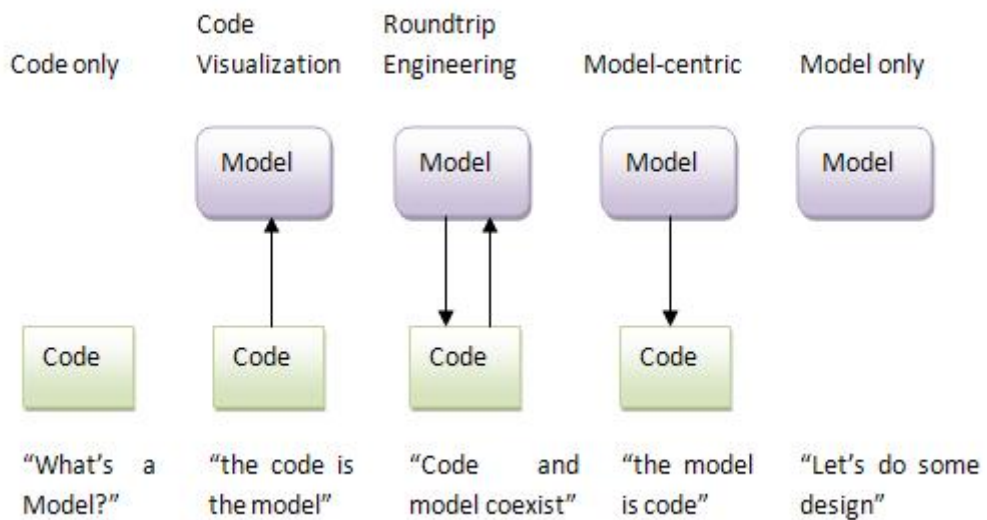


Figure 2. 1 Modelling Pattern Series [11]

In the software world, source code is the most accurate model used to describe a real system. Today, many software developers still use a pure code method and do not use other detached definition models. This is the most direct means of development. System creation fully depends on the code they write, and they are always in an integrated development environment (such as IBM WebSphere studio, Eclipse or Microsoft VisualStudio) and using a 3rd generation programming language such Java, C++, or C# to directly express the

models that they are building. Any “modelling” they do is carried out in an abstract form of programming that is embedded in code (package, module and interface, etc.). These are managed by a program library and the mechanism of object hierarchy. Any design model separately showing the architecture is not normally produced and if so will be based on instinct and will reside on a white board, in a PowerPoint presentation or just in the developers’ head. However, this kind of method may be sufficient for the individual developer or small development team. This approach makes realisation of the detail of business logic very hard to understand particularly in respect of the key features of the system. What’s more, such an approach is not effective in managing system evolution with increasing system scope and complexity or where members of the original design teams can not directly communicate with those maintaining the systems.

One improvement is to provide code visualisation through some appropriate modelling symbol. When developers create or analyse an application, they usually prefer to produce some sort of code visualisation to help their understanding of the code or graphical symbols to help in understanding behaviour. For optional editing of text code, making use of graphical symbols is possible. This visual description can be viewed as a direct representation of code. This kind of description is called code model or implementation model. In those tools that allow painting (such as IBM WebSphere Studio and Borland Together / J), a code view and a model view can be displayed at the same time. When developers operate one view, the other view will be synchronised at once. In this way, the graphics are tightly connected with code, and provide a view at code level so offering an optional means of editing.

The core benefits of modelling come through Round Trip Engineering (RTE). RTE provides architecture to describe the system or a round trip exchange mechanism between the design and model of the code. In typical cases, the developer first produces the system design to a certain level of detail, then the

first round implementation is realised by transforming between model and code with this being completed by hand. For example, a team working on a high level design may provide a design model to a team working at the implementation level. This may be via a simple printed graphical model or by providing files including a model to the implementation team. The realization of team transform of abstract, high level design becomes a set of detail design model and realization of programming language. Repetition of expressions will appear as errors and these errors will be corrected in the design model or in the implementation model. Good discipline is required to ensure that the abstract model and the implementation model proceed at a synchronised pace. Tools allow initial transformation to be carried out automatically. This helps the models keep pace with each other as the design model and implementation model evolve. The typical application case is that tools can generate framework of code from the design model but these must then be further refined. Modification of the code must be made consistent with the original model at some point (so we have the term “Round Trip Engineering” or RTE). To ensure effective implementation, there is a need to adopt some method to identify generated and user-defined code. One method is to place tags in codes. There are some tools for realising this, such as IBM Rational Rose, which can provide several transformation services for the round trip between the model and different implement languages.

A method which takes the model as the core means of modelling the system must have enough detail to generate a realisation of the whole system from the model. To ensure this can be done the model may include information such as persistent data and non-persistent data, business logic and representations for elements of the presentation layer. If there is any integration between legacy data and services, modelling with interfaces of those elements is necessary. Then code generation process use a series of patterns to transform the model into code. Usually, developers can choose among various applied patterns such

as a choice of different deployment topologies. This method often uses standard or private application framework and run-time services. These application frameworks and run-time services can make the task of code generation easier by limiting the generation of applied types. Therefore, tools using the method typically focus on the generation of specific application types (such as IBM Rational Rose Technical Developer which is used for real time embedded systems and IBM Rational Rapid developer for enterprise IT systems). However, in each case the model is the main product created and operated by the developers.

In a modelling pattern using a single model, developers treat the model purely as a means of aiding comprehension, as a solution domain, or as playing a supporting role in analysing the proposed solution architecture. The model is often used as a vehicle for discussion and communication within a single organisation or as a basis for analysis in a project crossing over several different organisations. These models often appear in proposals for new work or are displayed on the office wall and in the software laboratory as a method for facilitating understanding of some complex domains. They are also helpful in building a vocabulary and set of concepts to be shared across different teams. As a matter of fact, implementation of a system must be separated from models which start from draft or update an existing solution. The design of the model absolutely does not take realisation into account, it only relates to system business, and in building a domain business system model from the angle of pure domain application.

At present, the keystone of MDD research focuses on modelling patterns in which the model is taken as the core of the research process. Using a modelling pattern taking the model as the core can make developers pay more attention to the architecture of the system model, and the realisation code for the actual system is automatically completed by a code generator. With adjustments and maintenance performed by the design model, it is not necessary for developers

to maintain a process of generated code. The emphasis on a single modelling pattern is helpful to make domain modelling grow naturally in MDD to enhance realisation of domain modelling. In software development, many of those factors that can make software development more difficult relate to domain issues. General domain modelling can effectively find domain related problems that may appear during the modelling process and solve these by adjusting the domain model.

2.3 Model Driven Development in MDA

2.3.1 Concepts of MDA

Model Driven Architecture (MDA) was introduced by the Object Management Group (OMG). It represents a fundamental shift in the approach to software development from object-oriented design to model driven development [68]. It provides an open, neutral approach to deal with changes of operation and technique. The basic concepts of MDA involve: model, abstraction, refinement, view, zoom and platform.

(1) Model is the description of a part of the structure, function or behaviour of the system.

(2) Abstraction: An infinite set of details can be extracted from an objective system. Any specification of a system only describes the system at a level matching a specific perspective. This is the abstraction for the objective system.

(3) View: a particular point of view or level of abstraction.

(4) Refinement is the actualisation.

(5) Shrink/Release: Developing from an object in the abstract model (relationship) to a number of objects in the refinement model (relations) is called the releasing process and the converse is called the shrinkage process.

(6) Platform: A new type of developmental environment, which is independent of the hardware and software environment. The model will be divided into the platform independent model (PIM) and platform specific model (PSM) by the platform. It implements the functional description of the system and systems separated from the achievement in a specific platform. PIM describes the function of the system and the abstraction of the structure. PSM is the process of refining the function of the system in line with the specific goals of the platform. However, PSM it is not the same thing as the concrete and platform specific computer language, which generates the model language.

2.3.2 Model Framework of MDA

In MDA, the model is not only the description of the system and a tool of auxiliary communication, but is also at the core of software development and major work. Figure 2.2 shows the architecture of MDA software development as issued by OMG in July 2001.

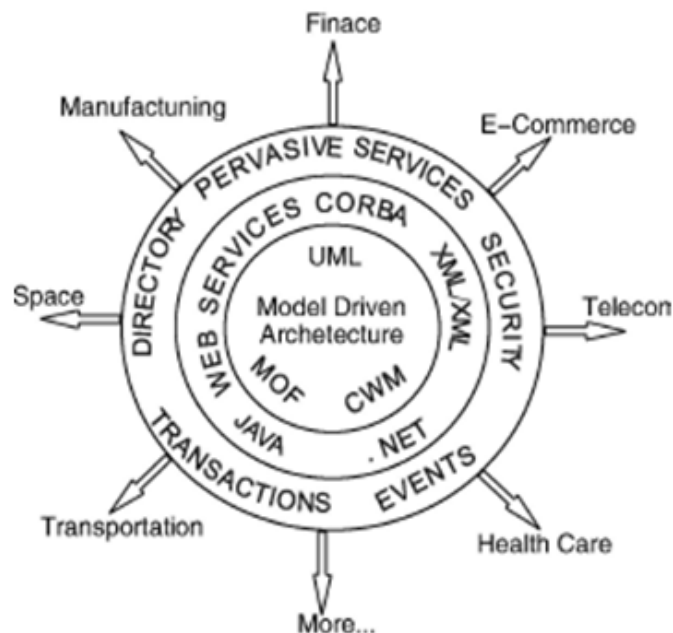


Figure 2.2 Framework of MDA [76]

OMG puts forward MDA in order to deal with the changing needs of software development and technology, and to protect investment in IT. As a framework for software development, MDA provides a new way to design a software system. It can be seen as a new methodology for software development. The core idea of MDA is to abstract the platform-independent model separate from the technology and provide a complete description of the operational functions. Next is the platform-specific model which is concerned with the concrete implementation technology through the specific mapping rules. Finally, we have automatic conversion into code through a series of auxiliary tools for mapping rules.

The framework of MDA raises the level at which problems are addressed. It achieves this by separating analysis of the design of the operational function of the system from the concrete implementation of the system. Software system modelling is divided into platform-independent models (PIM) and platform-specific models (PSM). PIM describes the structure and processes of the system, including considerations of transaction process, information security, data persistence and other technical issues. However, PIM has nothing to do with the concrete implementation technology, which is used to address such problems as imprecise definition of user requirements and imprecise abstraction of enterprise business. PSM corresponds with specific platform technologies, for example the persistence layer corresponds to the database, and the middleware platform corresponds to J2EE and so on. Using model rules, the two models can be converted into each other. PSM can automatically generate code and deploy description files [51].

MDA divides the model and meta-model into four layers, as shown in Figure 2.3.

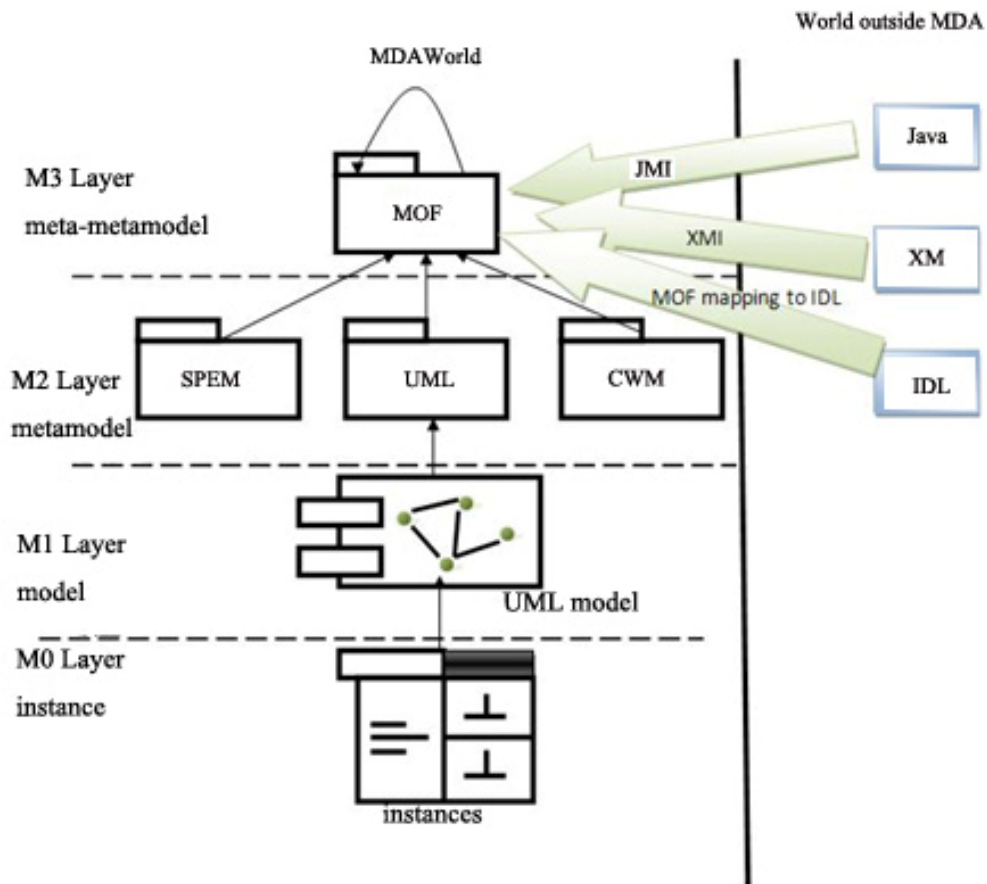


Figure 2.3 Hierarchical Model in MDA

Among them:

(1) The M0 layer is an instance level. It is an example of the model in M1 level. For example, the UML model corresponds to a specific program.

(2) The M1 layer is a model level. It is a model usually faced by the modelling people, such as the UML model in the figure for analysis and design.

(3) The M2 layer is called the meta-model level, which corresponds to the meta-model of the M1 layer, such as UML and SPEM, and so on. The M2 layer extracts abstract concepts and relative structure of different areas in M2's meta-model. It also provides modelling symbols for the modelling language of the M1 layer. Namely, the M2 layer provides corresponding domain-specific

modelling language for different areas.

(4) The M3 layer is meta-meta model level. MOF is located on this level. MOF provides a more abstract level of modelling support for defining the needs of M2's meta-model. MOF is a meta-model for all meta-models in the M2 layer. At the same time, it is self-describing for MOF can describe the meta-model of MOF itself. We should note that in the framework of MDA, there is only a model of MOF in the M3 layer. It is at the very core of MDA and provides a unified semantic basis for all models / meta-models in the framework of MDA. MOF makes it possible to unify all model operations [122].

2.3.3 Main Core Technology of MDA

At the core of MDA lie modelling and the techniques of model mapping. These are Meta-Object Facility (MOF), Unified Modelling Language (UML), and Common Warehouse Meta-model (CWM).

(1) MOF (Meta Object Facility) [72] defines the modelling language and also provides the concept and tools for visualising the modelling language. MOF is the core technology of MDA.

(2) UML (Unified Modelling Language) [73] is used as the standard modelling language for MOF definition of meta-models. It can be applied to almost all areas of application and platforms. UML is the basis for the existence of MDA. Meanwhile, MDA technology creates its programs based on the standardised, platform independent UML model. UML has been used to describe a variety of models and it does not exist for MDA alone. However, currently as the most popular Modelling Language, UML occupies a 90% market share among the world's modelling languages and has become the de facto standard for modelling languages. It is not only the basis for MDA but is also its most powerful weapon.

(3) XMI (XML metadata Interchange) [74] facilitates the exchange between

the data and metadata of UML modelling tools and provides the mechanism for data storage in the multi-tier distributed environment. XMI is based on the meta-data exchange in XML. Through the standardised XML document format and DTD (Document Type Definitions), it can define all models for the format of data exchange based on XML. It makes a model of the final products and transfers using all kinds of tools and ensures that MDA will not subsequently re-introduce a new layer of restrictions. The specifications of XMI support any data exchange of meta-data (including model and meta-model) which can be expressed by MOF. At the same time, the specification supports the conversion of a complete model or a fragment of a model to XML.

(4) CWM (Common Warehouse Metamodel) [75] provides a means to transform the format of data so that it is possible for it to be the common data model of the transformation engine. MDA appeared in order to promote improvements in the efficiency of software development, to enhance the portability, inter-operability and ease of maintenance of software. The object-oriented technology sector has predicted that CWM will be the most important software development methodology over the next few years. However, since its introduction by OMG in 2001 to the present day, MDA has had only lukewarm success. It has neither achieved market domination nor has it been abandoned.

(5) JMI (Java metadata Interface) makes it possible to achieve the infrastructure of meta-data management. This infrastructure greatly facilitates the integration of applicable programs, tools and services. In the past, in the absence of a standard way to express their own unique characteristics, it was difficult to achieve full interoperability and integration among the systems. JMI provides the framework for such meta-data to capture these semantics. EJB hides in the complexity of computer platforms and allows developers to avoid having to deal directly with affairs, security, resources and a range of other low-level programming tasks and has been proved to be very efficient. Similarly,

JMI allows developers to hide complexity through the creation of a specific technology and high level models of the business field [69].

(6) QVT (Query/View/Transformation) [34] is one of the new OMG development standards. It is used mainly to solve problems relating to the achievement of transformation in models. QVT is used for the definition of MOF and is a part of MOF.

2.3.4 Research Situation of MDA

At present, key research relating to MDA is focused on: the supporting technology of MDA, model language, model conversion, setting up the model, running the model, application of the model and so forth. The areas can be summarised as follows:

(1) Standardisation: In order to ensure that all the components can understand the shared metadata, MDA-based systems need to be further standardised in the following areas [85]: Firstly, the use of formal language (including syntax and semantic) to express Metadata. Secondly, the use of an exchangeable format to exchange and disseminate data. Thirdly, the use of some kind of programming model for the metadata to visit and find, which must include a universal programming capability to deal with the metadata which has the performance of location. Fourthly, an optional form of meta-data service, which can be used to release the stored metadata. Finally, expanding the four mechanisms above.

(2) Research into model semantics and the improvement of model language. Nowadays, the focus is on designing and establishing accurate and effective model definition language [7]. This leads on further to a study of the methodology [35] and development of the Object Constraint Language (OCL). At present, the RFP of UML2.0 has seen much improvement and advancement in the model semantics [76]. However, the study of this aspect is still ongoing.

The defects of UML1.x have been much improved in UML2.0, but there are still too many cross-dependencies in the meta-model elements. Meanwhile, the use of graphical methods among the tools in the standard method of exchange has not yet been fully resolved.

Object Constraint Language (OCL) may provide a guarantee in respect of the refinement definition and the conversion of the model. The company involved, Klasse Objecten has released Oetopus in the Netherlands. This supports the development environment of Eclipse control in OCL2.0; however, it is not perfect for OCL when using only its own tools. UML can also fail to provide the meta-model for OCL when a number of specific tools are collaborating in the MDA environment, which does not currently have the necessary extensibility. Therefore, improvement of the UML language and its OCL is a focus of research.

(3) The establishment of sub-modelling languages in the business area. Relevant applications include: establishing a large number of MOF-based meta-models, defining and establishing a sub-domain model language for this area [115] and defining the abstract syntax of specification. These meta-models should include the special rules and semantics in the field of application. This is necessary not only to achieve accuracy and to avoid ambiguity but also to facilitate proficiency in the field for non-expert users. Because the meta-models must understand each other, a variety of different languages also must be based on the MOF meta-model to be studied in this area.

(4) The study of Mapping Models in the platform. MDA is a means of solving interoperability problems in the system-level model. It separates the specific platform from the implementation technology. Meanwhile, it generates a variety of implementation models depending on the mapping relationships of different specific platforms. Then it maps to code, such as Java, XMI or SOAP as shown in Figure 2.4. At present, there have been various mappings such as from MOF to CORBA, from MOF to XML and from MOF to Java. Research

includes mapping of WSDL and SOAP and from MOF to Web Services.

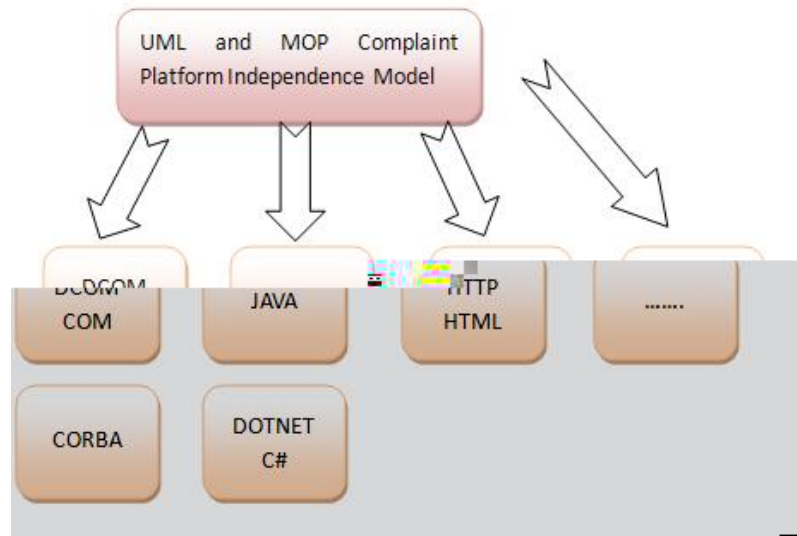


Figure 2.4 Platform Independent Model is Mapped To the Platform Specific Model

(5) The organisation and study of the knowledge warehouse in MDA tools. Here the key content of implementation in MDA tools is the effective organisation, development and modelling of the knowledge base for software development. For example, modelling has proven to be a successful experience for previous practitioners whose models have involved algorithms, design models, models of modelling and where the knowledge has been organised in the form of the model or code. Thus we have seen technology based on the pattern [32], the aspect [33], the contract [31], the object-orientation, and so on and the independent development of platform architecture with a variety of architecture together with integration of oriented MDA at a level above the technology and architecture. These are all of considerable value for implementation with the development tools of MDA. Eventually, there is a need to create a system that raises the majority of software development knowledge and field knowledge to a higher level of abstraction. Such a system will understand how best to extract and operate information and will further support the process of model development.

(6) The further development of compiler technology. There is a need to compile at a single or multi-levels or to work together with a virtual machine from model through to executable code. At present, even though compiler technology has matured it is still a new challenge to compile for MDA at all levels. This can be described as mapping technology.

(7) The research and development of the Super Virtual Machine and specific CPU. The virtual machine is actually the development of the running platform. It can learn and improve from the experience of Java Virtual Machine so that the model can directly run in virtual machines [63]. There is also the concept of the development of specific CPU referring to JavaCPU [89] (the CPU for running Java directly). The development of a direct running high-level model for MDA remains a challenge.

(8) Applications. Modelling the work flow in the field of business under the guidance of a meta-model in MDA. An example of this would be modelling the CRM system for communication carriers or devising architecture for the framework [114]. The integration of different systems in the application of a project may be achieved through conversion to a high level model. In addition, designing the meta-model of the same level conversion in a variety of data models [77] under the guidance of CWM is a current focus of the major companies.

The appearance of MDA is playing a good role in improving software development efficiency and in enhancing portability, interoperability and maintainability of software. So in the object-oriented technology industry MDA is predicted as likely to be the software development methodology of choice in future years. However as previously mentioned, since its introduction by OMG in 2001 to the present day, MDA has had only lukewarm success.

2.4 Domain Specific Development

2.4.1 Domain Specific Modelling

Steven Kelly and Juha-Pekka Tolvanen tell us what Domain-Specific Modelling (DSM) is about. “Domain-Specific Modelling mainly aims to do two things. First, raise the level of abstraction beyond programming by specifying the solution in a language that directly uses concepts and rules from a specific problem domain. Second, generate final products in a chosen programming language or other form from these high level specifications. Usually the code generation is further supported by framework code that provides the common atomic implementations for the applications within the domain. The more extensive automation of application development is possible because the modelling language, code generator, and framework code need fit the requirements of a narrow application domain. In other words, they are domain specific and are fully under the control of their users.”

(1) Higher Levels of Abstraction

Abstractions are extremely relevant to software development. Throughout the history of software development, raising the level of abstraction has led to the greatest leaps forward in developer productivity. The most recent example was the move from Assembler to Third Generation Languages (3GLs), which happened decades ago. As we all know, 3GLs such as FORTRAN and C gave developers much more expressive power than Assembler and did so in format that was much easier to understand, yet compilers could automatically translate them into Assembler.

According to Capers Jones’ Software Productivity Research, 3GLs increased developer productivity by an astonishing 450%. In contrast, the later introduction of object-oriented languages did not raise the abstraction level much further. For example, the same research suggests that Java allows

developers to be only 20% more productive than BASIC. Since the figures for C++ and C# do not differ much from Java, the use of newer programming languages can hardly be justified by claims of improved productivity.

The great leap from compiler to 3GL benefits from improvement of abstract level. Each statement in C++, BASIC or JAVA is equal to several compiler instructions. The important is that these languages can automatically translate and compile instructions. This means that a line of manual code is equal to five lines of machine code from the angle of productivity,

If raising the level of abstraction reduces complexity, then we need to ask ourselves how we can raise it even further. Figure 2.5 shows how developers at different times have bridged the abstraction gap between an idea in domain terms and its implementation [48].

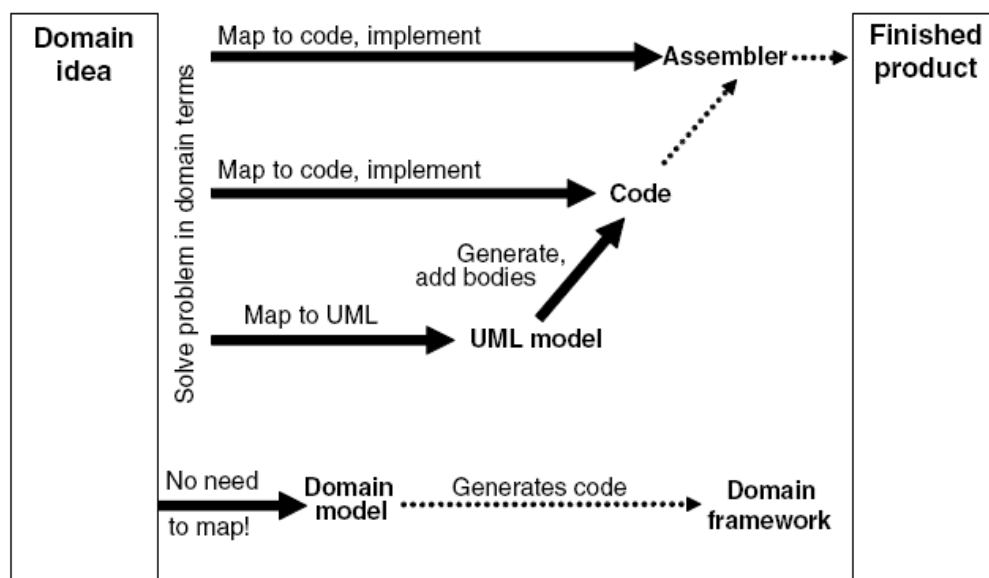


Figure 2.5 Bridging the Abstraction Gap of An Idea in Domain Terms and Its Implementation [48]

The first step in developing any software is always to think of a solution in terms that relate to the problem domain. This will be a solution at a high abstraction level. An example here would be deciding whether we should first

ask for a person's name or payment method during registration for a conference.

Having found this solution, we would then move on to the second step and map that to a specification in some language. Here, with traditional programming, the developers map domain concepts to coding concepts: “wait for choice” maps to a “while loop” in code. With UML or other general-purpose modelling languages, developers map the problem domain solution to the specification with the modelling language in the same way as “wait for choice” triggers an action in an activity diagram.

The third step then sees the full solution implemented giving the right conditions and code content for the loop code. However, if general-purpose modelling languages are used, there is an extra stage of mapping from model to code. It is most remarkable that developers still have to perform the first step without any tool support, especially when we know that mistakes in this phase of development are the most costly ones to resolve. Most of us will also argue that finding the right solution on this level is exactly what has been the most complex task.

A traditional general modelling language like UML cannot further improve the abstraction level. This is because the abstraction level that is provided by their core model and programming language are same. When designing with UML, we still have to work directly with objects, their attributes, return values etc. A day spent on modelling work using a modelling tool supporting UML is still equal to a day spent on coding.

Of course, UML has its own advantages e.g. the visual expression is easy to read and so we can get a holistic description. However, looking at the actual application of UML, we find that many developers feel that they have aspects remaining that they cannot express in the model and that they must take further steps to deal with these. There are many people who believe that UML is too complex, and hope that it can be reduced to its core elements.

By trying to do too much, UML fails to improve abstraction level.

(2) Domain-Specific Modelling used to Improve Abstract Level

How can we provide a higher abstraction level than 3GL? Perhaps we should stop trying to use a group of general programming language concepts for every type of application program. Instead we can use product specific concepts that can create unique visual representations.

DSML tags together with the rules for using them and the relationships among them come directly from the problem domain, the target environment of the system. They provide a much higher abstract level than UML. The result is a design language that is rich in expression, with a clear boundary and specialising in defining the system run in the problem domain.

For example, a DSM language used for developing a mobile phone system can use concepts like “Soft Button”, “menu”, “Send SMS” and “notification”. These concepts can be used in the mobile development domain but it is hard to apply them within the web system, ERP or commercial intelligent software. It is much easier to define a special domain-specific modelling language than to define a general language like UML or Java. It only takes a little modelling work to fully define a system.

Because it contains so much functionality a domain-specific modelling language can automatically generate full code. It gets a graphical model from the core part of the development work then automatically generates all the codes and documents required.

The model can provide both design and documentation. It can introduction a higher level of abstraction to the product and system and provide a direct source of code realisation. Documentation and reliability run through the whole life cycle of the system and are driven by the model.

(3) Automation with Generators

While making a design before starting implementation makes a lot of sense, most companies want more from the models than just throwaway specification or documentation that often does not reflect what is actually built. UML and other code-level modelling languages often just add an extra stepping stone on the way to the finished product. Automatically generating code from the UML designs (automating the third step) would remove duplication of work, but this is where UML generally falls short. In practice, it is possible to generate only very little usable code from UML models.

Who can easily write compile code by hand then keep it in synchronisation with their C++ code?

A code generator is needed to generate full code from DSML. The mapping between model and code is defined in the generator. DSML needs a code generator to meet the requirements of the problem domain. That is to say, we need a means of ensuring that we can generate full code from DSM language as required while having full freedom to define how the language is mapped into code.

In DSM, the generated code is functional, readable and efficient. Ideally it looks like code handwritten by the experienced developer who defined the generator. Here DSM differs from earlier CASE and UML tools. The generator is written by a company's own expert developer who has written several applications in that domain. The code is thus just like the best in-house code at that particular company rather than the one-size-fits-all code produced by a generator supplied by a modelling tool vendor.

At this point, we need to emphasise that code generation is not restricted to any particular programming language or paradigm. The generation target can be for instance, an object-oriented, structural or functional programming language. It can be in the form of a traditional programming language, a scripting language, data definitions or a configuration file.

(4) DSM Solution Evolves

Changes to the DSM language and generators are more of the norm than an exception. A DSM solution should never be considered ready for use unless all the applications for that domain are already known. The DSM solution needs to be changed because the domain itself and related requirements change over time. Usually this leads to changes in the modelling language and related generators. If a change occurs only on the implementation side, like a new version of the programming language to be generated or the use of a new library, merely changing code generators can be adequate. This keeps the design models untouched and hides implementation details from developers using DSM.

2.4.2 Architecture of Domain-Specific Model

To get the DSM benefits of improved productivity, quality, and complexity hiding, we need to specify how the automation from high level models to running systems should work. For this task DSM proposes a three-level architecture on top of the target environment, as illustrated in Figure 2.6:

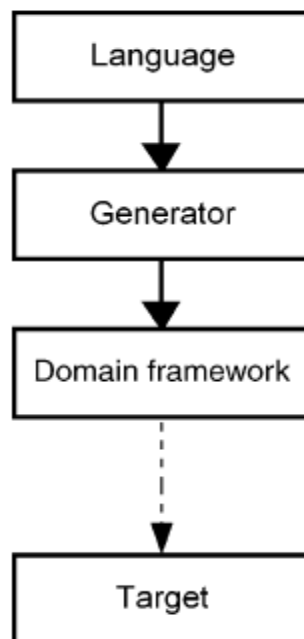


Figure 2.6 Architecture of DSM

(1) Language

A domain-specific language provides an abstraction mechanism to deal with complexity in a given domain. This is done by providing concepts and rules within a language that represent things in the application domain, rather than concepts of a given programming language. Generally, the major domain concepts map to modelling language objects, while others will be captured as object properties, connections, submodels, or links to models in other languages. Thus, the language allows developers to perceive themselves as working directly with domain concepts. The language is defined as a metamodel with related notation and tool support.

Language provides the abstraction for development and as such is the most visible part for developers. In DSM, it is used to make the specifications that manual programmers would treat as source code. If the language is formed correctly, it should apply terms and concepts of a particular problem domain. This means that a domain-specific language is most likely useless in other problem domains.

Generally the major domain concepts map to the main modelling concepts, while others will be captured as object properties, connections, submodels or links to models in other languages. This allows users of DSM to perceive themselves as working directly with domain concepts. The focus for the narrow domain is provided through language properties such as its modelling concepts, underlying model of computation, and notational symbols.

General definitions adopted are also suitable for the domain-specific language. It is general recognised that the modelling language includes syntax and semantic. We further abstract syntax into abstract syntax and concrete syntax. The former represents language structure and grammatical rules. The latter deals with symbols used by the language and representation. Usually it is necessary to extend language and semantics to improve the abstract level of the

design and generate more concrete code.

(2) Generator

A generator specifies how information is extracted from the models and transformed into code. In the simplest cases, each modelling symbol produces certain fixed code, including the values entered into the symbol as arguments. The generator can also generate different code depending on the values in the symbol, the relationships it has with other symbols, or other information in the model. This code will be linked with the framework and compiled to a finished executable code. While creating a working DSM solution the objective is that after generation, additional manual effort to modify or extend the generated code is not needed. The generated code is thus simply an intermediate by-product on the way to the finished product, like .o files in C compilation.

In DSM, code generators transform the model into codes which are interpreted or compiled into executable code. Code generator is helpful in realising the productivity claimed for the DSM method and in ensuring quality is achieved. It does this by making the necessary changes automatically. From the viewpoint of the modeller, generated code is complete. It means generated code is full, executable and that quality is ensured. That is to say, there is no need for the manual rewriting of code or for additional manual operations on the codes after code generation. This is possible because both generator and modelling language are constructed to meet the requirements of a small domain, such as is used within a company.

We must emphasise that this does not mean that all the codes used are automatically generated. This is also the reason why the domain framework and target environment exist. They may be generated from different models or manually written as is common today. The generator itself, like the domain framework and target environment is largely invisible to developers. This invisibility is similar to that of the black box or compiler, which are also unseen

by the developers.

Code generators are be classified in different ways. They can be divided into declarative and operational types and hybrid (mixed) versions are also in use. This kind of classification is based on methods used to specify the generator. The declarative type describes a mapping between source (meta-model) and target programming language. An example of the operational type would use graph transformation rules to define the necessary steps to generate target code from a given source model.

(3) Domain Framework

A domain framework provides the interface between the generated code and the underlying platform. In some cases, no extra framework code is needed and the generated code can directly call the platform components, whose existing services are enough. Often, though, it is good practice to define some extra utility code or components to make the generated code simpler. This framework code can range in size from components down to individual groups of programming language statements that occur commonly in code in the selected domain. Such components may already exist from earlier development efforts and products.

In general, the generated code is not executed alone but rather together with additional code in some target environment. This target comes with platform code, the code that is already available with the target.

2.4.3 Domain Specific Language

The area of modelling language is not strange to most of us. The reason is that UMLs are so popular. However, the use of domain-specific modelling language is still novel. The idea of domain-specific languages has existed since the first computer languages were designed [60] in fact it probably contributed to the early proliferation of programming languages. Many years later, the idea that

drives the development of today's languages remains nearly identical to that which emerged with the first languages. This idea is that an improved abstraction of the problem allows for the rapid creation and maintenance of a complex application [98]. A domain-specific language (DSL) is a language designed to provide a notation tailored toward an application domain, and is based only on the relevant concepts and features of that domain. As such, a DSL is a means of describing and generating members of a program family within a given domain, without the need for knowledge about general programming. By providing notations tailored to the application domain, a DSL offers substantial gains in productivity and even enables end-user programming [55].

Domain-specific languages (DSLs) are being increasingly used as a realistic approach to address a program family. That is, a set of programs that shares enough commonalities to be considered as a whole. These programs may already exist or their development is anticipated. In this situation, in principle, software development can benefit from introducing a DSL in that (1) it offers concise and specific notations to express a member of the program family, and (2) it enables the development of safe code thanks to its restricted semantics and / or requirements for additional information.

Domain-specific languages are special languages defined for developers to resolve domain-specific problems. Martin Fowler believes [28] that DSLs are not a new idea and that the early "little language" of Unix, the use of lex and yacc to generate program code together with languages defined in LISP are all examples of application of DSLs technology. Karl Frank believes that [29] DSLs can refer to any domain-specific language, such as UML, XML and that even C# and Java may be considered as a domain-specific languages because they are aimed at special purposes (software development) and used in special situations though this a rather broader view. However, in terms of software development domains, C# and Java can be applied to various types of software development so we usually regard them as general-purpose languages.

Here we prefer to use a definition [16] that Steven Cook gives us, namely “domain-specific development”. DSLs are computer programming languages to resolve domain-specific problems. They provide fixed abstract concepts and symbols to fit to the domain. DSLs are usually small and focus on limiting the number of rules or instructions. However, ability of expression is limited compared with General Purpose Languages (GPLs) and such DSLs cannot operate complex data structure. So domain-specific language is called application domain language, “little” language or macro language and closed with script language. For example SQL, Unix, shells and makefiles with which we are familiar can all be considered as domain-specific languages [102, 104, 88]. At present, application of domain-specific language has been introduced across various domains, such as graphics, financial products, phone switching systems, protocols, device driven programs, network routers and Robot Languages. Due to DSLs’ higher level of abstractness of domain, making use of DSL facilitates programming, validation and brings benefits of improved productivity, reliability and portability of products together with realisation of system level reuse [103].

DSL presentations can be in the form of text only but can also use graphical symbols. Text has the advantages that it is easy for the computer to handle various useful operations such as search or replace string. Text also allows for a contrast of differences in text content, mergers and so on. Meanwhile, graphical presentations are easy to understand and can offer a master profile of the whole model as well as illustrating relationships between and among elements.

Since DSLs are languages intended to deal with domain-specific problems, domain experts must be included among the necessary staff. Their skills are required even to define a suit of relatively simple syntax for domain-specific use that end users will later be able to modify themselves. One hope for the continuing use of a DSL is that it can evolve into a domain logic which the end users can modify unaided. Meanwhile the program developer can mainly focus

on development of the DSL's supporting tools rather than on ever-changing domain requirements. With DSLs, many of the simpler software requirement or permissions can be controlled by end users who can adjust the software themselves. The more that can be controlled by DSLs, the lower the cost. Ideally, program developers can be freed to focus their efforts on more valuable work. But to say the least, even if only the developers use DSLs this is a big help in improving productivity in software development.

2.5 Summary

Software is becoming increasing large, complex and difficult to produce. Costs are increasing and software reliability is becoming more difficult to ensure. Software development brings many challenges. Core problems of software engineering research include how to shorten the development cycle, improve development efficiency and software quality and respond effectively to change-on demand. Driven by these requirements, MDD is becoming a new research hotspot for the software industry. It is predicted that it will be the most important software development methodology over the next several years

At present, the research focus with MDD is on model-oriented modelling pattern and that of a single model. Seeking how best to realise MDD, the software industry is constantly exploring new methods, techniques and tools. MDA, which has been proposed by the Object Management Group (OMG), places the emphasis on using unified modelling language UML to build a domain system model. Other approaches advocate using DSLs designed according to target domain or DSMLs to establish models of target domain software systems. This shows that a combination of domain-specific development and model driven development is an important direction for research and practice. This chapter explains the relevant content and progress of MDA, which is the most representative of the model driven developments. The method for domain-specific development studied by the thesis is the model

driven development method. This is described from the viewpoint of domain-specific modelling, architecture of domain-specific modelling and domain-specific languages.

Chapter 3

Related Work

3.1 Introduction to Domain Specific Development

Domain-Specific Development is not new. In 1976, David Parnas introduced the concept of families of programs in his paper “On the Design and Development of Program Families” [81]. He also drew attention to the possibility of using a program generator to create the family members. In 1986, Jon Bentley in his column in the journal *Communications of the ACM* pointed out that much of what we do as programmers is the invention of “little languages” [4] that solve particular problems. Later in 1994, the popular and seminal book *DesignPatterns: Elements of Reusable Object-Oriented Software*, by Gamma, Helm, Johnson and Vlissides (also known as the “Gang of Four” book), introduced the Interpreter pattern. According to the authors, the intent of this pattern is: “Given a language, (to) define a representation of its grammar along with an interpreter that uses the representation to interpret sentences in the language.” But it is only relatively recently that Domain-Specific Development has begun to gain widespread acceptance in the IT industry. Domain-Specific Development is closely related to many emerging initiatives from other authors and organisations, of which the following is a partial list.

3.2 Model Driven Development

The majority of MDA research deals with PIM, PSM and transformations between these models. However, the scientific community has not expressed much interest in computational independent models and few proposals defining CIMs exist [70, 78, 79].

Regarding the model driven development and language family supporting environment, literature has been put forward to describe [61] PKUMoDEL (Peking University Model Driven Development Environment for Languages Family). This is a modelling environment based on UML 2.0 with an integrated body of meta-modelling environment based on MOF that can be seamlessly integrated into the blue bird component library management platform and other middleware platforms. Other literature [117] gives an analysis of problems that must be faced currently in the management of software development and puts forward a business-oriented software integrated platform (BOSIP) as the basis of the problem solving process. This literature also describes the model driven implementation techniques of the BOSIP system together with a study of model driven principles and expresses a point of view on the run-time driven model. Literature [12] describing research into embedded systems gives a design method for a model driven embedded system. In the development of WEB applications, literature [43] starts from the requirements of model transformations and puts forward a WEB application development method for combining software architecture with MDD as validated by the J2EE platform. In key algorithms used by enterprises in Heterogeneous Data Integration and the realisation of technology based on semantics and models, literature [116] puts forward a proto system of semantic model driven HDSI. Other literature [119] puts forward an automated method for executing distribution testing of model driven, designed and realised distributed test executable models. This is achieved through scheduling and deployment models leading to the building of a test executable distribution-oriented framework. With the constant increase in the existing number of web services, study is now turning to address the issue of how to make use of current Web services to create new and more complex Web services. Literature [44] puts forward a combined Web service development method driven by transforming the MDA model, according to the static modelling aspects of Web composition. Here, a static-structure method is put forward to build both platform-independent and platform-dependent Web

services and transformation rules between these are given.

Meanwhile as a standardisation method for conversion from PIM to PSM has not yet been completed, IBM, Borland, Oracle and other large software developers remain cautious on tool support for MDA. Though they are following each other in providing development tools that provide some of the functions necessary for MDA they do not comply fully with the definition of MDA norms as defined by OMG. With IBM adding MDA functions in Rational, and in the open-source projects of Eclipse together with EMF (Eclipse Modelling Framework), which is an innovative system project of MDA code generation, we can see that IBM is also poised for further explorations in developing MDA technology.

In contrast to the caution of the large software developers in the industry, a number of small and medium-sized companies are particularly active. These include such well known products as Arc-Styler from Interactive Objects, OptimalJ from Compuware Corporation and AndroMDA. These companies have already integrated MDA technology into their own enterprise-class solution software. This software has already been widely used and has achieved remarkable results [87].

MDA research is more active in the modelling of simulation systems. Successful MDA simulation modelling product applications have appeared with SMP2 developed by the European Space Agency. Here, the main goal is to use open standards to improve portability of the models in different simulation environments and on different platforms together with improvements in model reuse and development efficiency. SMP2 has been successfully applied to simulation tools in such projects as the Galileo satellite positioning system, a general project test-bed and the Rosetta spacecraft simulator [99]. In addition, SM plicity [86, 82, 83] developed by the Australian company Calytrix, is a simulation component used to assistant developers. This is based on a MDA design method for the rapid development of a HLA simulation integrated

development environment. It supports the full processes of design, development, deployment and management of HLA simulation projects.

Most other relevant areas of research are based on MDA applications of MOF. Literature here [126] puts forward a GIS application development model based on MDA. According to a CAPP (Computer Aided Process Planning) data model and different areas of business needs together with various CAPP process chart based solutions, Literature [41] puts forward an XML model driven solution to CAPP customised information release. Further literature [100] presents a model driven intelligent form of design method, which is e-government-oriented. This is based on supporting the business object library by making an analysis of traditional forms of development techniques and the current main forms of intelligent products. Addressing the difficulties faced during the design process in current disaster recover systems, Literature [113] puts forward a design for a disaster recovery system based on model driven decision-making support. Literature [42, 50] constructs a meta-model based on a meta object facility using SACRED (Subject, Action, Data, Constrain, Event, Relation) to define the model and a further-developed development tool PureX supporting MDA based on the SACRED meta-model. The tool supports development from model creation through to code generation, and even the generation of the final executable system process. It also supports a simulation of the implementation of the model.

3.3 Language Oriented Programming

The cycle of software development has been lengthening resulting in the need to expend a great deal of manpower, material and financial resources. This has to some extent hindered the development of the software industry. Sergey Dimitriev, founder and CEO of the JetBrains company, in his article "Language-Oriented Programming: The Next Programming Paradigm",

describes methods of "language-oriented programming" that is the creation of a domain-specific language to solve a particular programming problem. This is in order to cut down the length of the software development cycle for some domain-specific problems.

At present, the major research companies are competing with LOP [21] (Language Oriented Programming) system platforms. Examples include MDA from IBM, software factories from Microsoft and MPS from JetBrains. These tools are in an initial stage of development. Although they differ in form, the basic principles of each involve converting a domain-specific model or language into a general programming language (such as Java, C++) and to compile to generate executable program [101].

Meta Programming System (MPS) is a new programming environment which makes it easy for the developer to define new specialised languages that can be used as required together with any other language. Such new languages also have full IDE support with code completion, navigation, refactoring and more. Specialised support (such as special editors) can be added if necessary. Existing languages can be extended with new features [45]. MPS eliminates the programmer's dependence on languages and environments giving more freedom and power to the programmer. It makes programming easier, more fun and more productive. MPS is an implementation of Language Oriented Programming whose goal is to make defining languages as natural and easy as defining classes and methods is today

The ideas underlying LOP and MPS are not new and have actually been around for more than 20 years [54, 20]. The term Language Oriented Programming itself has been around for at least 10 years [118]. Martin Fowler gives us the traditions of language oriented programming such as Unix Little Languages, Lisp, Active Data Models, Adaptive Object Models, XML Configuration Files and GUI Builders [28].

A program in LOP is not a set of instructions. Instead a program is any unambiguous solution to a problem. Or, more exactly: A program is any precisely defined model of a solution to some problem in some domain, expressed using domain concepts.

In LOP, a language is defined by three main things: Structure, editor and semantics. Its structure defines its abstract syntax, what concepts are supported and how they can be arranged. Its editor defines its concrete syntax, how it should be rendered and edited. Its semantics define its behaviour, how it should be interpreted and / or how it should be transformed into executable code. Of course, languages can also have other aspects, such as constraints and type systems [21].

Martin Fowler divides LOP into two broader styles: External DSLs and Internal DSLs. External DSLs are written in a different language to the main (host) language of the application and are transformed into it using some form of compiler or interpreter. The Unix little languages, active data models and XML configuration files all fall into this category. Internal DSLs morph the host language itself into a DSL. The Lisp tradition is the best example of this.

The biggest advantage of External DSLs is that any form can be used freely. The biggest shortcoming is the lack of symbolic integration, namely DSL is not really connected into the host language. The host language environment cannot check code written by external DSLs. Now that the programming environment is becoming more complex, this is becoming an increasing serious problem.

For many people, one of the advantages of External DSLs is that they can be dynamically handled while they are being run. This facilitates modification as amendments can be put into effect with no need for recompilation. This is also one important reason why XML configuration files are so popular in the Java world. Although calculating external expressions at run-time is a major

problem for static compiled languages there are many other languages that can do so easily so the above-mentioned problem is becoming less and less important. At present, there are many people interested in languages that bring together compile-time and run-time, for example IronPython in .NET. So IronPython as an Internal DSL can be dynamically evaluated in the context of the system most developed by C#. This is a very common technology in the UNIX world together with C / C ++ and script language.

The merits and demerits of Internal DSLs are the opposite of those of External DSLs eliminating obstacles to integration with the symbol languages. They can also make full use of the strengths of host languages and their related supporting tools. Lisp and adaptive object model are examples of DSLs. The internal DSL forms feature Lisp or Smalltalk rather than Java or C#. In fact, the advocators of dynamic language believe that it is one of main advantages of the dynamic language. But internal DSLs are limited by the syntax and structure of the host language. Because internal DSLs are close to their programming languages, when something to be expressed is not well mapped to the programming language itself, this may be a difficulty. For example, there are layer concepts in enterprise application software. To a large extent, these layers can be defined by package constructs of the programming language. However, the dependence rules among these layers are hard to define.

3.4 Domain Specific Modelling

Steven Kelly and Juha-Pekka Tolvanen have elaborated what is meant by domain-specific modelling [49]. The main ideas of DSM are first to improve the abstract level beyond programming while specifying the language for the solution and secondly to generate target code by using selected programming languages or other methods generated from high-level specifications. It is believed that the main benefits brought by the application of DSM, such as improvements in productivity and quality as well as the ability of the whole

development team to share specialised knowledge are hard to achieve by other development methods. Kelly and Tolvanen also introduce DSM tools such as:

(1) The domain-oriented modelling environment (DOME) researched and created by Honeywell laboratory and used exclusively for their own projects.

(2) MetaEdit the forerunner of MetaEdit+ was developed by Smolander in 1991. It is composed of general modelling tools and the support of its modelling language is provided by binary metafiles. Its meta-modelling language is based on OPRR (Smolander, 1991) and is an example of the reuse of existing work. OPRR was originally developed by Welke (Welke, 1988) and is applied in QuickSpec (Meta Systems, 1989).

(3) The framework of the TBK (Tool Builders Kit) and ToolBuilder metaCASE systems were initially reported in Alderson framework and later commercialised by IPSYS.

Domain-specific modelling has been successfully applied in many industrial domains with the productivity of these domains being increased by 5-10 times. This is an area in which many companies seem worried about revealing the basis of their competitive advantage. In the open reference literature, only a few have openly given examples and results of case studies. One example is the Nokia Series 60 / Python [22] which gives an in-depth explanation of an example of DSM used in mobile applications as well as examples of microcontroller programming [23]. It shows how the DSM is applied in an embedded system with limited resources. The language menus of the family of automation systems used an 8 bit microcontroller.

3.5 Generative Programming

“Generative programming: method, tools and application” written by Krzysztof Czarnecki and Ulrich W. Eisenecker [17] discusses how to automatically generate an application program, especially in domain engineering and feature

modelling and also provides a detailed discussion on different program generation technologies. A regular meeting on the theme “generative programming and component engineering” (GPCE) is held committed to the topics of this research.

Generative programming (Figure 3.1) is such a technology. The characteristics of the technology are not concerned with the final program, but with the generator program. Its input is in domain code and the final program is output in object code. The domain code is expressed in DSL [18, 93].

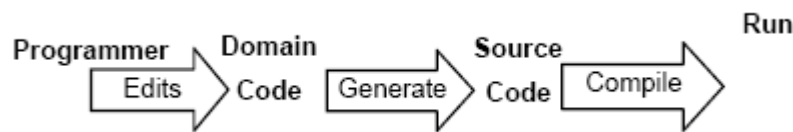


Figure 3. 1 Workflow of Generative Programming

The different applications of generative technologies depend on how DSLs is defined, how editing will be carried out and how the generator itself will access the language. Krzysztof Czarnecki and Ulrich W. Eisenecker summarise the possibilities [17].

Charles Simonyi gives other methods using generative technology to help resolve the complexity of software development. Code generation is the most common way used in the generative method. For example, a template library, like STL [71] uses code generation technology. CASE from the 1980’s can generate standard COBOL or C applications as well as a fixed generator from specialised diagrams. If the components of the generator library cannot satisfy user requirements, then users have to maintain the generated COBOL or C code. More specifically, if the generator library of components available to resolve user problems is satisfactory then code generation can be effective. Reference [15] gives a current and comprehensive list of existing code generators.

L. Robert Varney and D. Stott Parker present a new generative interface

oriented-programming method and suggest that transformations between source code and output code play a complementary role in interface-oriented programming design [110].

With the increasing scale of software, complexity is growing ever higher and higher. How to efficiently develop software of high-quality and how to effectively maintain and update software are key issues for current software methodology. To achieve these goals, various effective methods and technologies have appeared one after another. Krzysztof Czarnecki and U. Eisenecker seek to harmonise some advanced methods and technologies by presenting a new software engineering paradigm in the form of generative programming design. It is based on software system modelling to develop a given requirement specification and the use of configuration knowledge to achieve automatic configuration of basic and reusable components as required to generate customised, optimal software products. The basis of generative programming design is a system-oriented generative domain model. This model includes three basic components: the problem domain, the solution domain and configuration knowledge connecting these two domains. The generative program design includes various separate development cycles. These are the design and realisation of the generative domain model, providing the support necessary for reusable development, the use of the generative domain model to generate the actual software system and finally supporting development applied in reuse [26].

Neither can the current generative programming method be effectively applied in the development of complex software systems (such as information systems), nor can the software be reused at the analysis and design levels. The literature [125] presents a generative programming method based on refinement characteristics. Here a feature model is used to describe the basic concepts and characteristics of the domain. Next, a method of feature refinement is used to refine the model into a set of basic characteristics together with the relationships

between the basic features. This is used to explain how the features are to be realised. Finally the basic features are mapped into components that are used to construct the system as a whole according to a system feature model for assembling the components. Problems that object-oriented theory finds hard to resolve are analysed in the literature [124] and a way of thinking of using generative programming to construct general domain models and low-coupling modules according to such issues is presented. This takes Aspect Oriented Programming (AOP) as an example, and lists its main methods together with an analysis of their merits and demerits and goes on to compare pattern realisation by traditional OO methods and pattern implementation with Observer.

3.6 Intentional Software

The purpose of Intentional Software is to develop an environment in which all of the program design is based on a specific domain. Domain workbench technology is used to express the program and the model as data and to provide a wide range of channels for the use of domain specific text and graphics syntax to render programs and models and to interact with them. Enterprises have invested considerable time and money to develop such software, but the knowledge and insights obtained in the development have then disappeared in the details of the code or even in the best case scenario they only exist in the document which has a weak link back to the source code. There is potential value to be found in the intention behind the software. This is why this method is called intentional software. Intentional Software is a software company created by Charles Simonyi. It focuses on the development of software tools that can deliver functional control to the users [108]. The approach also embodies the principles of intentional programming that feature in the current programming software movement [94]. Its goal is to "to accelerate innovation by allowing experts from the business domain to participate in the generating process."

Intentional software captures tremendous value that often disappears in the process of design and development and makes it become part of the software. Using intentional software, domain knowledge is obtained rather than lost. All of the software stakeholders including programmers, domain experts and others are able to clearly express their own design intentions in the code. This increases the quality and value of the software and is mainly realised by making software development, maintenance and modification easier.

Intentional software provides a software method that can separate enterprise knowledge from software engineering. Business specialists directly use domain knowledge and symbols with which they are familiar. Capgemini is developing a new Pension Workbench. Here, old-age insurance experts use Pension Domain Language (PDL) to express their knowledge. PDL follows a format and terminology of their domains with which they are already familiar. The corresponding applications are generated from generators created by programmers who do their best to create a simple, reusable and reliable program. Innovation is accelerated in a creative team where everyone has an effective way to express their intentions.

A WYSIWYG editor can simplify the process of document creation by separating document content from document layout. Automated re-use of existing layouts can facilitate changing document content. In the same way, intentional software simplifies software creation. Domain-dependent software content can be separated from the software itself so that when the content changes the software can be automatically regenerated. In this way, domain experts and programmers can work in parallel in their respective areas of expertise while repeated instances of software completion can be automated. Intentional software is provided through the tool of the domain workbench. During the process of software creation, several domains can be defined, created, edited, transformed and integrated by the tool. Its key features include many domain-related unified representations, which can simply access program

generators through many projection-domain editable symbols.

Charles Simonyi, Magnus Christerson and Shane Clifford give a creative workflow for intentional software as shown in Figure 3.2.

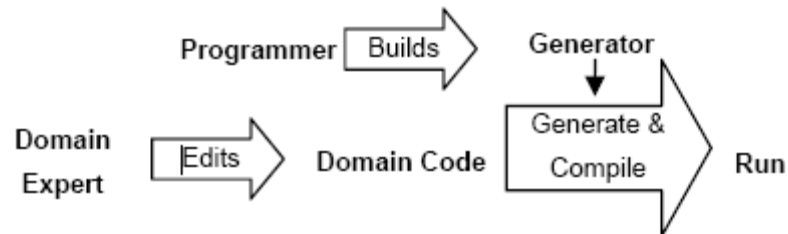


Figure 3. 2 Creative Workflow for Intentional Software [69]

Effective evolution and maintenance of the software needs adequate documents to be provided in software development. However, continuous evolution of the software means that documents and software code may not keep pace. Intentional software has put forward a document technology to address this problem. The creation of such a view is not a trivial task, Tom Tourwe, Johan Brichau and others have proposed using a learning algorithm to create the viewpoint of intentional software. This algorithm comes from an extensional software viewpoint, which is easier to build. The method combines the advantages of the viewpoint of intentional software and the characteristics of a more easily constructed extensional view [105].

3.7 Software Factory

Software factories are described in “Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools” written by Jack Greenfield, Keith Short, Steve Cook and Stuart Kent [36].

Software Factory is an implementation solution given by Microsoft for the development of model driven research. It combines passive content such as pattern, model, DSLs, components and help documentation with dynamic

content such as custom tools, customised processes, templates, guides and testing. All these are integrated into Visual Studio, the use of which produces a particular type of solution. Software Factory is also a Microsoft strategy for initiative in the field with DSL featuring as an important part of the initiative. A software factory is a production line, which configures unitised content and guidance for extendable development tools, such as Microsoft's VSTS (Visual Studio Team System). These are carefully designed to build a variety of specific applications. Software Factory contains three main ideas: a software factory model, a software factory template and an extendible development environment. If the software factory model is compared to a recipe, then the software factory model is just like a bag of groceries containing the various components listed in the recipe. What's more, the extensible development environment such as VSTS is like the kitchen used for cooking the food [36].

The literature [81] presents a belief that a product family provides an environment in which many problems are very common to members of the family and can be resolved collectively. Based on the above thinking, in order to construct the software product line, the software factory provides a full solution by understanding the environment and managing changes among software products [14, 37].

Instead of waiting for perhaps unlikely special circumstances in which the software can be reusable, software factories systematically capture data relating to members of a specific product family group in the form of assets, such as patterns, frameworks, models and tools. These assets are then systematically applied in the automated development of new family members. This reduces costs, reduces product time to market and improves product quality compared with one-off development.

Of course, development of the software factory necessary to achieve the above objectives still involves development costs. In other words, the software factory embodies visible trade-offs of cost-effectiveness to be found in the

product line as a whole. Competitive advantage is not achieved by generating multiple copies; instead it comes from generating many related but unique products or product variants, such as the document for a series of case studies with costs being spread across the product line. We believe that the key to industrialisation is effective cost control in the construction and operation of software factories. Jack Greenfield and Keith Short give us an example of an operational software factory.

The Software Factory's core ideology is focused on a software product line, component-based development and model driven development. Its innovation lies in the integration of these into a cohesive framework supporting new tools and new practices. By combining model driven technology and the principal product lines, software factory has ushered in a new application development model. Its development tools provide high levels of extensibility. Development tools in the model bring fast, inexpensively configured, domain-specific development.

3.8 Summary

In order to better realise model driven development, the software industry is constantly exploring new methods, technologies and tools. These include: Model Driven Architecture (MDA), Language-Oriented Programming (LOP), Language Workbenches, Generative Programming (GP), Intention Programming (IP), Software Factories and Domain-Specific Modelling (DSM). Based on a review of the above methods and technologies, this chapter argues that all of these are involved in addressing the same problem: how to implement a description of the system model of the target domain? This is the problem that must be properly resolved before the goal of realising model driven development can be fully achieved. This thesis aims to find a new solution for the promotion and application of the model driven development method by researching and exploring the DSM-based method and its meta-modelling

languages and go on to lay a foundation for the further development of the model driven method.

Chapter 4

A General Implementation Framework Supporting DSM Methods

4.1 Overview

DSM is a member of a large family of model driven development methods. It provides us with new research ideas and trends in the use of model driven development in creating software systems. Unlike other Model Driven development methods, there is no formal guidance framework for the implementation and application of DSM. One problem that needs to be further researched and explored is how to make best use of DSM methods and hence improve technical practice in software engineering projects.

In this chapter, a general implementation framework is put forward according to the characteristics of DSM methods. The framework also takes account of traditional methods of organising software engineering, management technologies, systematic engineering methods, software architecture, etc. This instructive implementation framework is given at the engineering application level and includes a consideration of engineering methods for implementing DSM methods, division of developer's roles, development architecture, development environment and modelling language.

Any software engineering methodology must have an appropriate application of core values, namely values that practitioners of the methodology accept and conform to. This is a fundamental body of knowledge and ideas that forms the cornerstone of the methodology. Practice without such values is just a wild potpourri of activities. At the BOF meeting of OOPSLA 2003, experts

defined a group of core values for Model Driven software development [111].

- ✓ Verification of software under development is more important than verification of software requirements.
- ✓ Level of domain-related assets, including model, component, framework, generator, language and technology were identified.
- ✓ Automatic output of software constructed according to the domain model, while at the same time, consciously differentiating between software factory construction and software application.
- ✓ It is believed that the software development supply chain will come to the top, which leads to domain-related specialisation as well as to mass customisation.

The establishment of a DSM implementation framework actually provides a practical method of embodying the above values. The core values that DSM methods have mainly embodied in practice are as follows.

(1) Reduce the gap between system requirements and system realisation

There is a large gap between requirement description and software realisation. Requirement description is a specification description with a high-level of abstraction, while coding is carried out at a low level in the description of system implementation. It is a long process to turn an abstract software requirement into an actual software system using traditional software development methods. During the process, because there is deviation between requirements as expressed by users and requirements as understood by developers, the software system is unlikely to turn out be exactly as the users expected. The corresponding domain application system model generated by modelling can effectively build a bridge for better communication between users and developers. Therefore, it is requirement that the level of abstraction of the model should be neither too high nor too low otherwise it will be very

difficult to achieve the desired effect. Although current UML modelling technology is widely used, the results have not been as good as expected. One of the main reasons is that there is big gap between the UseCase diagram used to capture user requirements and the Class diagram that supports system realisation. A UseCase diagram can only describe system requirements informally and at a very abstract level. Meanwhile, a Class diagram gives a solution at a level that is very close to implementation code and is a visual representation of the code layer. At the same time, system users usually cannot easily understand the information expressed in class diagrams. Therefore, even use of UML modelling technology cannot effectively guarantee good communication. Unlike modelling methods using a modelling language, UML design is based on an object-oriented paradigm. DSM emphasises domain-specific modelling languages (DSMLs) to create an application model. The design philosophy of the domain-specific modelling language resides at a level above that of the implementation layer. It also uses formal modelling elements that are closer to the target domain to directly describe the system. This means that the system modeller can work directly at the domain layer to build a system modelling solution. It also not only allows users within the domain to understand and analyse the domain model using domain knowledge they are familiar with, but also enables final implementation code to be generated from the domain model using the code generator supporting the domain model. It can effectively narrow the gap between system requirements and system realisation and so effectively enhance the usefulness of the model.

(2) Full reuse of domain-related assets.

As the demand for software increases, the requirement for application software is also growing and the software industry continues to pursue the goal of finding better ways to develop high quality software systems. There is considerable evidence to support the view that software reuse is an effective method. Research into a component-based approach to development methods

based on an application platform framework is now moving towards seeking to increase the amount of reusable software and practical applications have been achieving good results. However, these methods are limited in the range of general software development methods used to study software reuse. For example, current developments based on components or application frameworks concentrate on the design and development of the components or frameworks. The emphasis is on universal properties and adaptability and on trying to use these to resolve all software problems. In the end, this just leads to the components or frameworks becoming huge and more and more complex. Studying how to use them can be more difficult than learning a programming language. How best to use them to realise the dream of reuse is a dilemma facing many developers. Any development organisation that has engaged in domain-specific development for a long time will have accumulated many domain related assets such as, domain system platforms, frameworks, components and technologies. One main target of DSM research is to efficiently integrate these assets to achieve better reuse. In DSM methodology, system development is carried out at the domain-related asset layer. It can encapsulate the common features of the domain application system into the platforms, frameworks, components etc of the domain application infrastructure. Meanwhile, those differences that are domain-specific are extracted and analysed to form the modelling elements of the domain model. The domain model enables the description and reuse of domain-specific knowledge that cannot be easily captured in reusable components. This underlying technology facilitates automated code generation and without the need for further input from the domain-application modeller can handle many aspects relating to the platform, framework and components. So, the domain-application modellers are free to pay attention to the organisation and construction of related domain application business without the need to master reuse implementation technology. Therefore, compared with traditional reuse technology, DSM methodology puts forward a new means of implementation and software reuse.

Full reuse of domain knowledge can be achieved through comprehensive use of its basic framework, modelling language and modelling tools.

(3) Development of the meta-model allows for the automated creation of the domain application from the domain application model.

Auto-creation of the application system is one of the targets of many Model Driven development methods. They try to find a general technology which can automatically generate various software systems. Unfortunately, as far as current development technology is concerned, this target has proven to be difficult to achieve. However, the early successful use of the CASE tool, which is used to generate systems with fixed applications, shows that if the implementation range is limited to one particular domain, difficulties are effectively reduced and the feasibility of automatically generated software can be greatly improved. However, many restricting factors make it difficult to expand the use of CASE tools. One of the main problems is that these CASE tools lack sufficient flexibility so users cannot modify and customise them according to their own requirements. So, the final automatically generated application system can barely satisfy the users' actual application requirements. At the same time, current technical and cost restraints mean it is not practicable to allow the developers of each domain to design and develop their own CASE tools to suit that specific domain. Therefore, the essential issue is not that CASE tools are not good but that they are neither available in large numbers nor sufficiently flexible and powerful. Similarly, the same questions also exist in Model Driven development. How can we rapidly design and develop a suitable modelling language and realise auto-creation of application systems by the modelling language? In addressing these questions, solutions given by DSM methods realise the development and construction of modelling tools by introducing meta-modelling and the use of modelling tools to build a domain model of the application system and so realise auto-creation. DSM methods emphasise meta-modelling infrastructure as well as the importance of

meta-modelling activities. Here, the main value lies in reducing the implementation threshold for developing a domain-specific modelling language and providing a low-cost means for rapidly developing and constructing a suitable modelling language together with the modelling tools required by the developer. So, it is helpful to enrich domain-specific modelling languages and to promote the range of applications for auto-creation software.

DSM as a special Model Driven development approach still has some main features that are different from other Model Driven development methods. If we want to get a general implementation framework for DSM methods, we need to analyse and research its characteristics as well as its emphasis in project implementation. Only in this way can we propose a general implementation framework suited to DSM methods.

The importance of using and implementing DSM methods is embodied in two terms: “domain-specific” and “modelling”:

“Domain-specific” refers to both functional and non-functional requirements and features that reside within an area covered by a group of application systems with similar software requirements. During the process of DSM implementation, it is necessary to analyse the domain-specific system in order to recognise both the common and the different characteristics of the target application system. These characteristics can be structural, functional, or non-functional, or they can belong to the mechanisms that regulate the business process. They are further selected and abstracted so that the domain concepts of the target modelling system can be obtained and to provide the reusable specification, design and architecture information necessary for “modelling”. The difference compared with other Model Driven development methods is that with DSM methods the implementation focus is mainly on carrying out modelling development based on identifying variations among the domain systems, rather than on fully constructing a software system by modelling everything over again from beginning to end.

During software development, “modelling” usually shows establishment activities at an abstract level of the software. “Modelling” in DSM includes two main activities: domain meta-modelling and application system modelling. These modelling activities are in different phases and are at two different levels of abstraction. Of these, domain meta-modelling is a basic modelling activity of the DSM approach and represents the keynote of the entire process of DSM project implementation. This is also an important aspect in which DSM methods differ from other Model Driven development methods. For example, building an application model by using the general modelling language UML is given greater emphasis in MDA. Although it provides MOF as the meta-modelling infrastructure, meta-modelling is not a keystone modelling activity of MDA methodology. UML used as a main modelling tool cannot effectively embody domain characteristics. But in DSM, meta-modelling is a very important means of realising domain-specific modelling. The relevant domain knowledge can be utilised in common specifications and design for domain application systems by meta-modelling. This extends the range of usable information to high-level abstractions in the analysis and design phases. In this way, the cost of domain application modelling is reduced and the domain application system can be developed more efficiently. Essentially, DSM is a software development method based on domain-specific analysis that works through domain meta-modelling and domain application modelling. Therefore, a “domain-specific” approach and “modelling” are core elements of any DSM general implementation framework.

4.2 Thinking of System Engineering

4.2.1 Characteristics of Systems Engineering

The late computer scientist Edsger Wybe Dijkstra once said, "The art of programming is to deal with the complexity of the arts." In software engineering,

the activities involved in developing software include identification, decomposition and isolation and these must deal with the various complex situations that may be encountered. The same activities are also key aspects of a variety of systems engineering methods. Applying such engineering methods can help us to control the complexity of the project, and to resolve or reduce the impact of those complexities which may bring risks to a software project. Such methods can also be an appropriate means of reducing costs and shortening development time.

DSM is a development method based on domain-analysis. It defines the modelling and solving of the target domain within a certain range in order to seek specific solutions according to the characteristics of that domain. This helps in the design and development of the target software system, as to a certain extent, this can simplify the complexity that exists in the process of domain-specific software development. At the same time, the complexity of a software engineering project is not only determined by the software itself for it is also generated by the environment in which the software must be created and implemented and through the influence of the various stakeholders in the software creation process. From the perspective of the system, strong coupling effects exist in the environment, in the software and in the people. They are generally in the form of a special kind of complex system which is emergent, instable, nonlinear, in-definable and unpredictable etc. Examples include: there is no strictly linear relationship between system complexity and the number of functional modules that comprise the system; there is no inevitable positive correlation relationship between system development efficiency and the number of people involved in system development; each individual in the system development team is likely to understand and grasp only that system information that is local to them; and no one is well aware of the entire system plus each segment of the development process. System requirements, running platform, changes in and evolution of implementation technologies are not

completely predictable during system planning and design stages. System science research offers some effective and important theories and methodologies to refer to when dealing with such problems. When we use the DSM approach in specific software projects, introducing and borrowing ideas from some of the theories and methods of systems engineering can offer significant guidance in building a framework for the implementation of DSM methods.

4.2.2 Engineering Thinking Based on Hall Three Dimensional Structures

In software engineering, characteristics such as description, logic, norms and artistry are intertwined and constitute the unique thought processes of software engineering together with its theoretical basis, basic procedures and process flows. In the practice of software engineering based on Model Driven development, DSM is a solution which focuses on technology to ensure a complex software engineering project can be successfully carried out and implemented. Besides general technical methods we must also consider engineering implementation methods and procedures together with staffing and other factors.

In the research and application of systems engineering, a variety of scientific working methods and procedures have been gradually explored, accumulated and summarised. In 1969, American systems engineer, A.D. Hall put forward a method with universal significance in the application of systems engineering. This is the "Hall-three-dimensional structure." Its appearance provided a unified way of thinking for resolving issues in the planning, organisation and management of large complex systems. We can learn from it to specify a unified engineering way of thinking for a general implementation framework for the DSM method. This is in the form of a spatial structure composed of a time dimension, a logic dimension and a knowledge dimension. Figure 4.1.

illustrates such a structure showing each step of a project using the DSM method together with the relevant scope of knowledge:

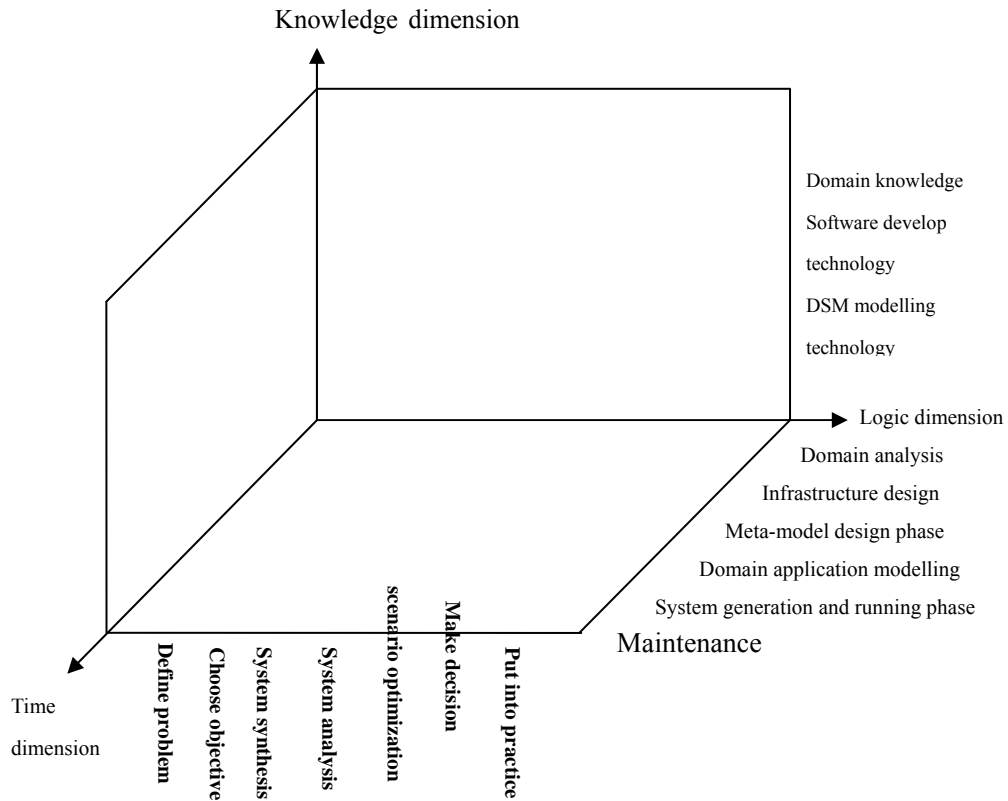


Figure 4. 1 Hall Three Dimensional Structure from Systems Engineering

(1) Time Dimension

The time dimension refers to the chronology of the implementation process from planning through to updating. In software engineering, it is performed as an iterative cycle for the engineering project. The implementation process can be divided into six stages (shown a-f below) by integrating this into software development with DSM.

(a) Domain analysis phase

This involves investigating and researching the domain-specific system to be developed, defining the domain characteristics of the target system and based on

this, producing a domain definition together with a scope definition. Domain analysis differs from requirement analysis, whose main objectives are to establish basic concepts, to identify the scope of the domain model, to gain an understanding of requirements that are common to the various systems in the domain, to reveal the commonalities and differences among the application systems within the domain and to produce a reference architecture model that can be adapted to all applications of the domain .

(b) Infrastructure design phase

The objectives of this phase include gathering the reusable components, basis library, specifications, etc. for the domain-specific system architecture (DSSA). DSSA describes a domain-specific solution for domain requirements. It is not a representation of a single system but a high-level design adapted to the requirements of all the various systems of the domain [84]. Reuse of the infrastructure is organised according to the domain model and DSSA. So, the infrastructure specification is obtained at the same time by including DSSA in the phase.

(c) Meta-model design phase

The main activity of this phase is to complete meta-modelling of the target domain application system. Meta-modelling is an activity carried out according to the nature of a series of concepts (objects, terminology, etc.) of a specific domain. A model is often used to create an abstract representation of some things or phenomena in the real world but the meta-model involves a further level of abstraction for its emphasis is on an abstraction of a model that is itself abstract. In a DSM project, this phase will see the completion of the design and development of the modelling language for a specific domain.

(d) Domain application modelling phase

This phase will use the results of meta-modelling i.e. the domain specific modelling language to model the domain application and obtain a system model

of the domain application. In the previous meta-modelling phase, what we got is a structure common to the domain system. Now, in the domain application modelling phase, we will confer value assignment specific attributes and behaviours on these general domain structures according to actual system requirements of the domain applications. This is a process of further refinement of the domain model and what we get in this phase is a high level abstract model of the target system.

(e) Domain application system generation and running phase

The main objective of this phase is the operational software system, which is usually generated by the DSM code generator. This directly transforms the domain application model obtained in the previous phase into the code of the computer programming language and so produces a software system that can be deployed and run with the assistance of the necessary compilers, linkers, etc.

(f) Maintenance phase

At this stage, we mainly modify and adjust the already-running system to ensure failure-free operation. When we use the methods of systems engineering, we must note that the field of software engineering is different to other areas. The changeability of requirements for the software product is a main characteristic that differs from other types of product engineering projects. It can be said that in the software engineering project, the only constant theme is "change." Long-term research and practice indicate that the "iterative" process is an effective tool to address the problems of constantly changing system requirements. A way forward can be found in the face of change as iterative development allows each iteration cycle to provide a solid foundation for the following development plan. A revised working version of the system can be produced after completion of each iteration cycle. Successive revisions gradually make all the required system functions possible. As these revisions are made, even though full final functionality may not yet have been achieved, all

the functions must be faithful to the final system requirements and each must be fully integrated and tested. Therefore, the time dimension is not the full life cycle of the software product, instead it represents each iterative cycle in the development process.

(2) Logic Dimension

In the three-dimensional structure, the logic dimension refers to steps carried out at each stage. This is a normal procedure that should be followed when using a systems engineering approach to consider, analyse and solve problems and involves:

(a) Identifying problems and gathering together as much information as possible in order to understand these problems. This step includes user research, requirement analysis and market analysis.

(b) Choosing objectives for resolving the problems and developing standards to measure whether or not these have been achieved.

(c) System synthesis i.e. collecting and integrating solutions which achieve the desired goals and give necessary explanations for each solution.

(d) System analysis i.e. using systems engineering methods and techniques to systematically compare and analyse the various solutions by integration when necessary and also by building mathematical models to carry out simulations or by theoretical deduction.

(e) Solution optimisation by evaluating the results given by the mathematical models etc and selecting the best solution to meet the required objectives.

(f) Decision-making to determine the best option.

(g) Implementing the solution to complete the various steps at each stage.

(3) Knowledge Dimension

The knowledge dimension of the three-dimensional structure refers to a variety of professional knowledge, management expertise and systems development knowledge, etc. required for completion of the above steps. For system development using DSM the knowledge needed to develop the project can be divided into three broad categories: domain knowledge, software development expertise and DSM modelling knowledge. Using a knowledge of systems engineering to combine the six time phases and seven steps a so-called Hall-Management Matrix can be constructed [120] as shown in Figure 4.2.

Logic dimension Time dimension	Define problems	Choose objects	System synthesis	System analysis	enhance design	Make decisions	Put into practice
Domain analysis phase	a11	a12	a13	a14	a15	a16	a17
Infrastructure design phase	a21	a22	a23	a24	a25	a26	a27
Meta-model design phase	a31	a32	a33	a34	a35	a36	a37
Domain application modelling phase	a41	a42	a43	a44	a45	a46	a47
System generation and running phase	a51	a52	a53	a54	a55	a56	a57
Maintenance phase	a61	a62	a63	a64	a65	a66	a67

Figure 4. 2 Hall-Management Matrix

Each stage of the time dimension in the matrix and the point corresponding to each step of the logic dimension represent a specific management activity. There are different key points for implementation and management at the various steps and stages as well as different knowledge requirements. The activities of the matrix affect each other and are closely related. The activities of the various stages of the steps must be repeated to achieve optimal overall results.

The introduction of an engineering approach based on the Hall three-dimension structure to the DSM implementation framework can help the

software practitioner better understand each stage of the DSM method. It can also assist in identifying the tasks that are necessary at each stage in order to reduce mistakes in decision-making and implementation difficulties.

4.3 Team Organisational Structure for the Implementation Model

Work undertaken through systems engineering puts the emphasis on peoples' creativity and initiative. People have always played the leading role. Systems engineering procedures, principles, viewpoints and methods can make a well organised team work better and do so in a shorter time. However, they will not make an awful team achieve the same effect. The author of «*The Mythical Man-Month*» believed that the qualities of the project developer and personnel organisational management are factors that are more important for the success of a project than the tools or technology used.

The headcount needed for communication and coordination in a software engineering project affects development cost. The main constituent parts of the cost come from mutual communication and exchanges and costs are increased wherever there are failures in communication. The target of team organisation is to reduce the number of unnecessary communications and unproductive cooperation so good team organisation is the key measure in solving such problems in coordinating and communicating.

The organisational structure of the development team is very closely related to the development methods adopted. Practitioners will be allocated to various roles according to the characteristics of the methods used by the development team. Staff in each role should have the corresponding knowledge and skill. When this is the case and when these practitioners cooperate efficiently and naturally, the true effectiveness of the chosen methods can be realised. Therefore, a well thought through division of roles is not only a precondition for

ensuring a team can work efficiently but is also a safeguard necessary to ensure the methods used are carried out properly.

In our implementation framework based on the characteristics of DSM methods, implementation is divided into the work of individuals in four different roles. These are: infrastructure developer, domain expert, meta-model developer and application system modeller. The relationship between these roles and the development knowledge that the project should have is shown in Figure 4.3.

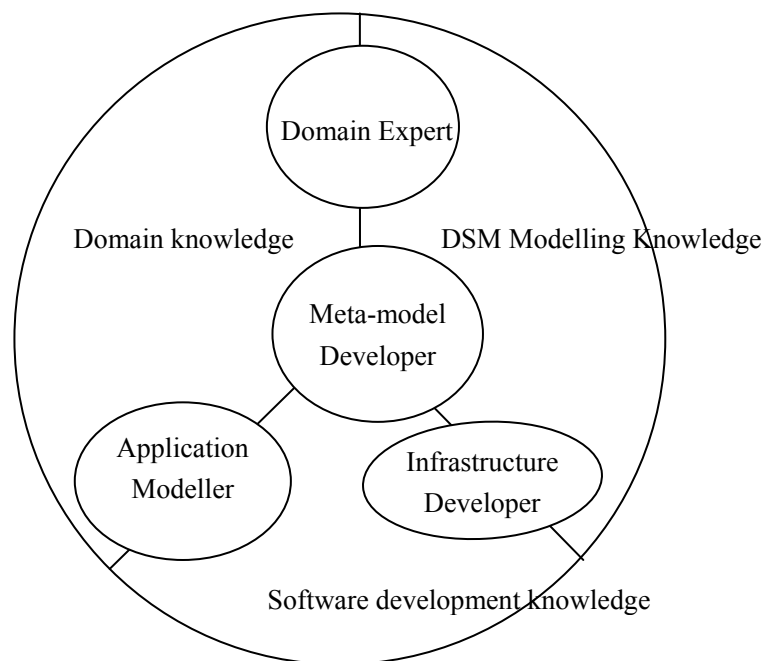


Figure 4. 3 Relationships Between Roles and Knowledge Structure in DSM

(1) Domain Experts

These include experienced users of the application system of the domain and the experienced software engineers who engage in requirement analysis, design, implementation of the application system in the domain, etc. Their main tasks include: the provision of knowledge for the requirement specification, implementation of domain application systems, helping organise a normative and consistent domain vocabulary, helping choose a sample system as the foundation for domain analysis, reviewing the domain model, domain analysis

for DSSA, and so on [58]. Domain experts should know about the overall system design thinking of the domain, restrictions on software and hardware, future user requirements and trends in technology, etc. They must also be able to compare software applications and to build an application domain model using DSM modelling tools.

(2) Infrastructure Developers

Experienced programmers are required for this position. Main tasks include: developing an understanding of the software architecture of the domain system, developing new reusable components from start, extracting reusable components from existing system by re-engineering, verifying reusable connectors and building the relationship between the DSSA and the reusable components. The infrastructure developer should be familiar with software reuse, low layer technical realisation together with re-engineering technology and programming and should have software development experience relevant to the domain.

(3) Application System Modeller

Main tasks include: controlling the entire domain application system design process, building the DSM domain application system according to domain application requirements with existing DSM modelling tools, verifying the veracity and consistency of models and building the relationship between the domain model and the software system.

Meanwhile, domain designers should be familiar with domain applications and DSM modelling as well as methods for software design. They must also have good skills in domain modelling together with relevant experience of domain software development in order to analyse domain problems and interact with domain experts.

(4) Meta-model Developer

This role requires a systems analyst with excellent development experience and a background in software engineering. Main tasks include: controlling the analysis process for the entire domain, capturing knowledge, organising that knowledge into a domain meta-model, verifying the veracity and consistency of domain meta-models according to the existing system, standards & specification and finally, maintaining the domain meta-model. They should be familiar with software reuse and domain analysis methods together with the technologies, languages and tools used to capture knowledge and knowledge representation. They also require experience of the domain in order to analyse domain problems and interact with domain experts together with the ability to handle high-level abstractions, associations and analogies. Compared with other practitioners, they need a higher level of ability in software development and to be better able to interact and cooperate with others.

Experience with most large programming systems shows that a method of development that “throws too many people at the problem” will be high-cost, low- speed and inefficient. What’s more, the products that are developed this way are unlikely to meet user requirements.

From the above viewpoint, the allocation of staff roles is actually one application of an organisation model that may be described as “the surgical team”. In the team, the role of the meta-model developer is at the core and the position must be occupied by a particularly able individual within the development organisation. This individual should be not only well versed in software development but also have a very rich experience in domain-specific development together with an especially strong ability to comprehend domain concepts and business rules. Meta-model developers are mainly responsible for the design of the domain meta-model and the development of the code generators, which are the most difficult and vital parts of the entire development project. Therefore, they must grasp the most valuable part of the design and development of the system. Meanwhile others working at lower levels of

difficulty in the development process must give them the support necessary to enhance efficiency and productivity. In order to achieve a good division of roles, based on individual knowledge and skills, we may use the knowledge dimension of a Hall three dimensional structure to help us assign and coordinate tasks. This is obviously different to some development processes where the division of roles within the development process just emphasises that those in different roles should complete given tasks according to different engineering phases. During a single phase such as programming, each team member is allocated a part of the model of the system to develop. Here, team members with a variety of different skills develop and cooperate at the same abstract level on tasks that are logically allocated.

4.4 Hierarchical Development Architecture

Any software system aims to solve related problems within a domain. However, in a great deal of software development activities, developers concentrate on onerous and error prone code compiling tasks of software systems. This fact can easily lead us to confuse domain solutions with general software technology solutions when we develop the software system and may eventually lead us to deviate from our development focus and so introduce considerable difficulties to the maintenance and evolution of the system.

As mentioned earlier in this chapter, the “domain” is a core element of the general implementation framework of the DSM method and the domain is the most valuable part of the software system. We need to detach consideration of the domain from other functions of the system to avoid confusing domain concepts with other concepts related to software technology or even losing control of the domain in the overall structure of the system.

It goes without saying that any software system has its own software architecture and that any development method should have development

architecture accordingly. In the design of software development architecture, a "hierarchy" metaphor borrowed from the construction industry has been widely used. A hierarchical structure becomes an effective way to split the functional modules of a software system and has been widely used by a majority of developers. The basic principle of hierarchy is that all the elements of the layer can only rely on other elements of the same layer or directly depend on the lower elements. The significance is that each level is responsible for a particular area of software system functions. The division based on functions and responsibilities can make all aspects of the design more cohesive and make these designs easier to understand and organise. However, how should we divide each layer? Specific hierarchical principles for the development of different methods are not unified. So, how can we best define layers in a way that is most favourable for development based on DSM methods?

In our general implementation framework, the final target of the hierarchy is to achieve separation of domain focus points. So, it is vital to separate out the domain-related parts of the software system. We have adopted the following four-layer structure as the hierarchical development structure solution to our general implementation framework. It is a variant of the usual three-tier structure:

Presentation Layer	An interface layer between the software system and the outside world. Responsible for collecting and displaying data for external users and analysing user commands. External users may sometimes be other systems.
Application layer	To define tasks that can be completed by software and produce domain objects with the necessary depth of knowledge. The tasks charged to this layer are of far-reaching impact on the business. Interaction with the application layers of other

	<p>systems is necessary to maintain this level.</p> <p>It does not include business rules or knowledge and merely coordinates tasks with and entrusts work to domain objects cooperating with each other on the next level. This level does not reflect the state of domain business operations, but instead reflects progress in undertaking tasks or the state of the user's tasks or programs. With DSM methods, realisation of the layer is mainly by application modelling.</p>
<p>Domain layer</p>	<p>Responsible for description and realisation of the business concept, information on the state of the business and business rules. Though the preservation of these technical details is completed by the infrastructure layer, statements reflecting the business are controlled and used by this layer. It is at the core of business software. In the DSM method, implementation of this level is completed at the meta-modelling stage.</p> <p>By using meta-modelling activities, the domain concepts (objects) used in the upper layers and domain rules can be abstracted and represented as corresponding meta-model elements to be used in modelling by the higher applications.</p>
<p>Infrastructure layer</p>	<p>Provides general technological capability to the upper layers. E.g. sending application messages, domain persistence and network communication as well as providing components using interface elements. Besides, this is usually a layer supporting the realisation of interactive protocols among the four</p>

	layers.
--	---------

Compared with the traditional three-tier structure, we separate domain in particular and all the codes related to the domain model are concentrated in a single layer that is separated not only from the user interface layer, but also from the application layer and the infrastructure layer code. The domain objects will be able to focus on expression of the domain model as they do not need to deal with the display, storage and management of application tasks, etc. This will enable the model to be powerful and clear enough to grasp the nature of business knowledge and to realise effective solutions. At the same time, it will be very easy to make the hierarchy correspond with the structure of the roles in the development team and facilitate a clear separation of the functions carried out by the various individuals so making more effectively use of their knowledge to complete the tasks within the corresponding problem areas.

4.4 Development Environment and Modelling Language

DSM stresses that "modelling" is a main activity throughout the entire development process. General models are used to drive the design and development of the software system, therefore the development environment and modelling language are important parts of the DSM method. At the same time, the design of the modelling language must also be closely integrated with the modelling environment to provide modelling development with a flexible, efficient and easily expandable basis for implementation in the environment through effective integration.

A general modelling environment supporting development of DSMLs in the general implementation framework for the DSM method is proposed. It is made up of a domain-specific meta-modelling language, XXML and an integrated modelling environment as shown in Figure 4.4.

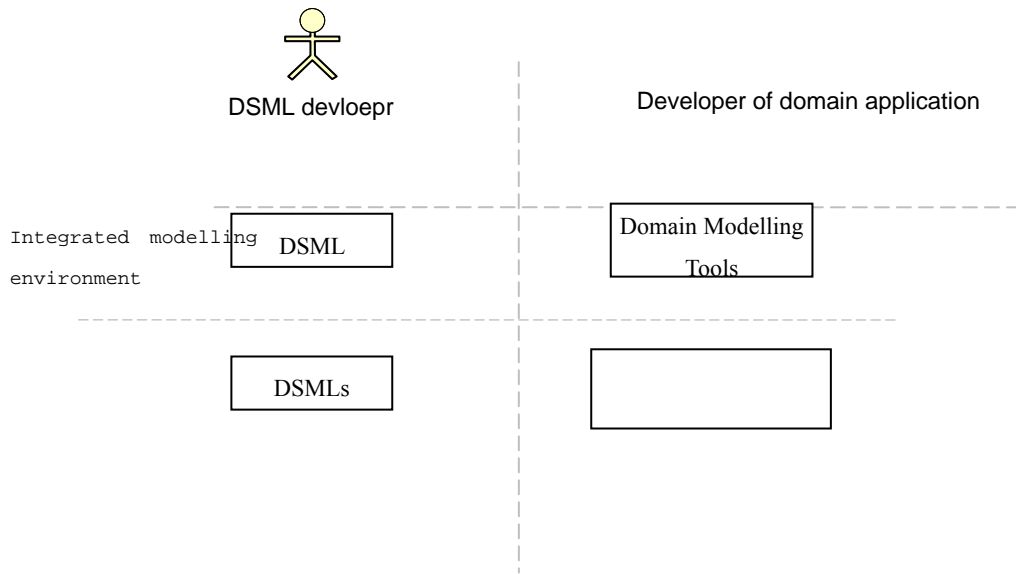


Figure 4. 4 A General Modelling Framework Based on Development of DSMLs

In the framework, the modelling process for applying DSM methods in domain-specific software development (DSSD) [90] is divided into the two main activities of meta-modelling and modelling. The main task of meta-modelling activities is to model the syntax of the domain-specific modelling language together with its semantics and rules, etc. This is achieved by using the meta-modelling language to define the DSML model [13]. Then a model reflection mechanism is used to generate and construct the corresponding domain application modelling environment according to a DSMLs model derived from meta-modelling. DSMLs and domain application models are respectively generated by the two modelling activities and they are defined and described by the same meta-modelling language, XMML.

XMML is the core modelling language for the general modelling framework. As such, it is not only able to formally describe the abstract syntax of DSMLs, concrete syntax and semantics, but can also provide basic support at meta-language level for functional expansion of the framework. Tool-oriented [40, 64] principles were used to design and construct XMML in order to make the integrated modelling environment of the framework capable of flexibly

constructing a domain-specific modelling environment through the extensibility provided by XMML and the formal modelling ability of the domain rules.

4.5 Summary

DSM is a new Model Driven software development method. It organically combines ideas of domain-specific development with software modelling methods and offers a new practical direction for facilitating Model Driven software development in software engineering applications. In this chapter, the thesis analyses the core values that are both emphasised and necessary in DSM methods:

- (1) Reduce the gap between system requirements and realisation of a description of the system.
- (2) Fully reuse domain-related assets.
- (3) Develop a meta-model to achieve automated creation from the domain application model to the domain application.

These core values are both keystones of and targets for actual software project practices using DSM methods. They reflect the main characteristics in which DSM methods differ from other Model Driven software development methods.

The application and implementation of DSM methods focus on “domain-specific” areas and on “modelling”. In order to analyse the target system of a domain and identify the common characteristics together with the volatile characteristics of the target application system, the common characteristics will be selected and abstracted to provide the necessary reusable specifications, design and architecture. Meanwhile, modelling will mainly focus on the volatile characteristics of the domain system. Selective use of domain elements and concepts in modelling the application system, rather than fully

modelling all aspects of the system, is a key to success in the use of DSM methods. The modelling of the domain application system depends on domain meta-modelling. So, the analysis and design of the meta-model is an important and necessary activity in the implement process of the DSM method. Therefore, DSM “modelling” includes domain meta-modelling and application system modelling. These model creation activities are carried out at different levels of abstraction. Compared with other Model Driven software development methods, DSM pays more attention to the significance and necessity of meta-modelling. Using some existing general modelling languages such as UML it is hard to support and realise the characteristics required by DSM. Therefore, it is necessary to provide a corresponding modelling language for the DSM method in order to better support its modelling activities.

The DSM method is a solution that places the emphasis on technology to ensure that complex software projects can be successfully developed and carried out. Not only general technical methods, but also implementation methods, procedures, staff organisation and other factors relating to the project must be considered. Through an analysis and summary of DSM methods, a general implementation framework based on DSM methods is given in the chapter.

In the framework, a “Hall three dimensional structure” is used to specify a unified engineering approach to implementation using the DSM method. This has a spatial structure that is made up of a time dimension, a logic dimension and a knowledge dimension. These dimensions show the steps in each stage of the process as well as the scope of the knowledge necessary to implement a DSM project. Introducing engineering principles and methods based on the “Hall three dimensions structure” leads the practitioner of the DSM method to clarify not only the stages in the process but also the tasks within each stage. This serves to reduce errors in decision-making and implementation difficulties.

In the implementation framework, an organisational structure applicable to a development team using the DSM method is given. According to the

characteristics of the DSM method, development team members occupy four different roles: infrastructure developers, domain experts, meta-model developers and application modellers. The division into these roles is a separate issue to the knowledge and skill requirements that the developers need to master the method. Among them, the meta-model developer is at the core of the whole team. This has clear differences compared with alternative approaches to the division of roles based on the development process.

The layered architecture becomes an effective means used by the majority of developers to split functional modules of the software system. In the general implementation framework, the ultimate goal of layering is to realise separation of the domain focus. Therefore, it is vital to separate out the domain related parts of the software system. In this thesis, the four-layer architecture is used as the layer development architecture solution for the general implementation framework, the infrastructure layer, the domain layer, the application layer and the presentation layer. In practice, compared to a traditional three-layer structure, this approach keeps aspects of the domain separate and integrates all code related to the domain model into a single layer. Besides, it separates it from the user interface layer, application layer and from infrastructure layer code. Domain objects can be focused on the expression of the domain model without concern for their own display, storage and application task management, etc. This will enable the development of a model powerful enough and clear enough to grasp the essence of business knowledge and put it into practice.

At the end of the chapter, a general description of the general modelling environment supporting the development of DSMLs is given. This is achieved through the composition of a domain-specific meta-modelling language, XMML together with an integrated modelling environment. In later chapters of this thesis we will respectively analyse and explain these two important aspects.

Chapter 5

Domain-Specific Meta-Modelling Language XMML

XMML (XML-Based Meta-Modelling Language) is a domain-specific Meta-Modelling Language designed according to the methods, systems and concepts of DSM. It is used to provide a domain meta-modelling language and domain application modelling with descriptive language support in the implementation framework. It supports the description and construction of the domain meta-model and the domain application model.

5.1 Overview

5.1.1 Characteristics of Domain-Specific Modelling

Languages

Domain-Specific Modelling Languages (DSMLs) are one form of Domain-Specific Languages (DSLs). DSLs should be differentiated from General Purpose Languages (GPLs) such as C, Java, C#, etc. DSM itself has been in use for time. Many computer languages are domain-specific, examples include: SQL, HTML, macro language for word processing software, etc. Earlier languages came with a variety of generic names e.g. application-oriented language, special purpose language, specialised language, task-specific language, application language, 4GLs, etc.

In essence, they all have common characteristics for they are computer languages trimmed down and designed according to a single specific target domain. They are aimed at making DSL users more accurate and more efficient

through using the languages to express solutions to domain-specific application. Compared with general programming languages, DSLs have two notable characteristics:

(1) Narrowing scope of application, and reducing complexity of language

A DSL is a computer language that is specially designed according to the problems of software development existing in one application domain and in “seeking the specific not the general” it does not require the scope to cover all software problems. Despite the drawback that DSLs are used at the expense of generality in the application of the language they improve the accuracy of description of domain-specific problems and the solutions to these problems while at the same time, reducing the complexity of the language itself.

Compared with generally applicable languages, the characteristics of DSLs bring two benefits: Firstly they make it easier for domain-specific users to study and master the language and for domain application developers to express their domain knowledge. Secondly, as they are more concise and accurate in their syntax and semantics, they make it easier to devise support tools so facilitating the development of compilers, interpreters and the supporting environment. Tool support is a very important characteristic of DSLs and is also a key area in which they differ from formal languages with a mathematical basis.

As a DSL is an application-oriented computer language, it must be able to be applied to the development process of the software system and not only apply to reasoning and validation processes. After all, the primary users of DSLs are programmers engaged in software development rather than mathematics experts. This requires that DSLs should not be too complex or exotic or it will be difficult to achieve tool support in computers. What is more, unless they are simple it will be impossible to spread their use in the industrial community and ultimately they would be unable to play their role of improving development efficiency in actual software projects.

Therefore, to reduce complexity of language makes DSLs rich as well as provide feasibility of security to extension of application method of DSLs.

(2) DSLs are high-level abstract, and they abstract from low-level implementation details and realisation platform of concrete matter

A DSL can better describe domain concepts, relationships between domain concepts and domain rules applying to domain objects.

Compared with general languages, domain-specific languages are at a higher level of abstraction and domain users can directly use built-in domain knowledge elements of the DSL to develop the target application system. They facilitate the construction of components or program codes in line with domain concepts rather than from the basic class or object. So they can effectively improve development efficiency of the system. For most domain application development projects the focus should be on the domain and the domain logic. The complexity of many systems does not lie in the technology but in the domain itself e.g. due to the nature of the business activities or in the complexity of its domain rules. If we fail to get a deep understanding of the domain when it is designed, then no matter how advanced or how powerful the platform and facilities we use, it will be difficult to ensure success of the project. Meanwhile, domain-specific languages are computer languages that are designed to suit the domain after a serious analysis of the domain. So, the DSL already includes corresponding domain elements as well as rules that must be obeyed when designing. This will help developers to develop directly at the domain level and effectively avoid some potential errors of understanding and so improve development quality for the system.

From the above two points we can see that the characteristics of DSLs differ from those of general languages. One does not need to be as sharp as a needle to see what is domain-specific. After all, the domain-specific concepts are so flexible that so far no one has give them a precise definition. In domain-specific

modelling, we are more concerned about how to establish domain models and how to resolve domain software development problems by domain modelling.

DSMLs belong within a DSM methodology. They represent a new type of modelling language with an emphasis on Model Driven development. This is where DSMLs differ in a general sense from DSLs. There are special design requirements for DSMLs used in DSM methods, so although DSMLs by their nature belong among the family of DSLs, there are some differences in actual design ideas and realisation approach between general DSLs and DSMLs.

(1) Emphasis on DSMLs supporting meta-modelling

Although research institutions and software developers have developed some DSLs, most of these are related to specific domains or platforms. What is worse is that these DSLs do not support adjustment through meta-modelling so users of these DSLs are subjected to constraints when they use them to design and develop. For example, SQL is a typical DSL but not a DSML. The users cannot adjust or extend SQL by meta-modelling because it does not provide a flexible meta-modelling supporting mechanism for itself.

If DSLs do not provide users with the ability to support meta-modelling then, although they can improve development efficiency for software systems to some extent, just like with CASE tools in back in the 1980's, as the users cannot extend and customise the tools, ultimately this will limit their practical application in both extent and scope. With DSM methods, modelling work is actually divided into two parts: the first is modelling according to domain concepts and any domain rules that may exist in the target application domain, namely establishing the domain meta-model; the second is to use the results of meta-modelling (DSMLs) to implement domain-application modelling of the target application system. During both parts of this work, supporting meta-modelling is the core task of the DSMLs.

Therefore, great emphasis is attached to supporting meta-modelling in DSM

methods, only in this way can the user get much more flexibility in the modelling language to adapt to the requirements of different domain applications.

(2) Emphasise on the ability of description not executability

In DSM architecture, the modelling language and code generator have different roles. Modelling language is mainly used as input for the code generator. Its focus is on a formal description of the domain model so as to enable the code generator to understand the model correctively and produce the final source code for the target system. This is different from the executive modelling language of other Model Driven development method systems (such as the executive UML in MDA). Therefore, DSMLs pay more attention to the ability to describe models. They belong among the DSLs and do not emphasise executability.

Stripping DSMLs of any requirements for the executability of the DSMLs can bring many benefits in design and application as follows:

(a) Reducing difficulty in designing DSMLs

There is no need to take implementation of their executable ability into account when designing the DSMLs. Therefore, there is no need to attach corresponding definitions of executable semantics when designing the modelling element. Clearly, this largely reduces difficulties of formal definition of DSMLs. At present, some DSLs start from existing advanced languages in a consideration of executability and define the corresponding DSL by clipping or secondary development. The largest drawback of this method is that it reduces the abstract level of the DSL. What is more, it makes users at the abstract level closed to high level programming language to use the DSL.

(b) Enhancing DSMLs' ability to describe domain models

In Model Driven development, the model plays two main important roles:

the first is to describe a solution to each domain problem given by the system developers by means of a highly abstract expression, that is the model describes how to realise system functions; the second is to describe the original intentions that is the model expresses user requirements and system specifications.

We can see that the characteristics and requirements of using a model to express user requirements are very important. There are considerable obstacles to the evolution and maintenance of a system in the absence of a high-abstract level model. This is because developers have to read a great deal of low level source code to refer back to the original user requirements and business rules. Therefore, the modelling language should be able to provide multi-view and multi-level descriptions making the system models carry more high-level description information helpful in understanding system requirements. This is something that is difficult for an executable modelling language to do. The reason is that to make it executable, the modelling language must have a strict formal definition for execution action semantics. However, it is hard to achieve complete formalisation of model user requirement information. Finally, in order to pursue executability, they have to give up some of the flexible mechanisms used to enrich model representation.

(c) Improve flexibility of realisation of code generator

Due to DSM separating code generation from the modelling language, this task is done by the code generator alone using modelling language as its input. There is no correlation between the modelling language and development language for the code generator. This means code generation can be more flexible and powerful. Conversely, a number of executable modelling languages adopt an integrated approach to bind model description and code generation together. Under this condition, the two functions must be realised by the same basic language, and this will greatly adversely affect the flexibility and extensibility of code generation.

Therefore, in the DSM method, DSMLs is a specification description modelling language, not a code representation modelling language. Its focus is on the constituent parts of the domain model such as, attribute, structure and relationship, etc. It is used to describe and express concrete domain concepts and business process, and it is closer to the domain layer rather than to the description of implementation layer. So, its characteristics differ from those of many DSLs.

(3) Emphasis on DSMLs improving level of abstraction in Model Driven development, rather than advocating bi-directional mapping between model and source code

The key application of DSMLs is to realise domain modelling, that is, the modelling language applied in the domain layer. In fact, the gap between the domain layer and the implementation layer or code layer is very large, and they belong to different abstract levels and bodies of knowledge. At present, there is no mature technology to realise automated bi-directional mapping between the domain model and the code model. In the DSM method system, mapping between the domain model and the code model is achieved by developing the meta-model and the code generator as developed by the designer of the meta-model. The modelling language supports bi-direction mapping between model and code, actually they work at the same implementation level and realise visualisation of coding, but do so at the cost of reducing the abstract level of the modelling language.

From a Model Driven perspective the key to system development has been transferred from coding to model design. The DSM method is a model-centric development method. With this method, the users of the modelling languages are not concerned with generating code, because they do not need to complete the generated code. Under these circumstances, the round trip between model and source code is not necessary. So, DSMLs emphasise that to improve abstract level of modelling while designing, the designer of the DSMLs mainly

focus on how to make them effectively represent domain related concepts and rules. They are not concerned with how to implement bi-direction mapping between elements of the modelling language and the code.

The above analysis helps us to understand and grasp the essential characteristics of DSMLs. It also provides us with help and guidance for design ideas and implementation of a design modelling language suitable for the DSM method system.

5.1.2 Design Goals and Concepts of XMML

XMML is a domain-specific meta-modelling language designed to make domain-specific modelling possible. It is used in the implementation framework of DSM to provide modelling language support to the meta-modelling language and domain application modelling. When designing XMML, we can confirm the design goals of XMML from the following three dimensions, descriptive, usable and verifiable, as shown in Figure 5.1.

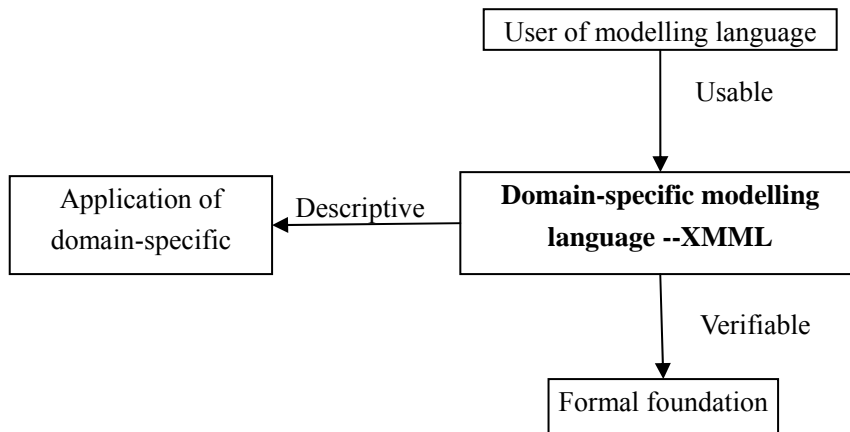


Figure 5. 1 Three Dimensions of Design Goals of XMML

(1) Descriptive

This is concerned with the ability of the modelling language to describe the contents of the target domain and is one of the essential properties any

modelling language should have. Meanwhile, it is also a broad concept, that is, it is hard to measure by a fixed standard. Different modelling languages differ in their understanding of a target, and will embody different characteristics in their requirements and manner of realisation of a description. In the design of XMML, we have identified the following requirements for a “descriptive target”:

- (a) To be able to define and extend domain concepts required in the domain model.

Different domains will have different domain concepts. One of the descriptive requirements of modelling language is to provide the modeller with a corresponding extension mechanism for the modelling language. This is in order to be able to map various domain concepts to the basic modelling elements of the modelling language.

- (b) To be able to describe attribute features of domain objects.

There are different domain objects in domain models and they have their own attribute features. How can we accurately describe attribute features of these domain objects? This is an important (descriptive) target to be embodied in the modelling language.

- (c) To be able to describe the life-span of domain objects.

It is a requirement that the modelling language can describe modelled domain objects at the time when they were created and destroyed as well as describe how concurrencies and parallel objects come into being. Therefore, the modelling language must not only be able to specify the structure of the model but also to describe time sequences.

- (d) To be able to make a multi- hierarchy, multi-view description of the model.

Complex software systems cannot be clearly and unambiguously expressed

by relying on a single layer and model with a fixed viewpoint. Usually for analysis and design, it is necessary to establish models having multi-views at different levels. Therefore, a modelling language should be capable of providing multi-level and multi-view description.

- (e) To be able to describe domain rules.

In domain application systems, business rules and constraining relationships among domain objects are keys aspects of domain modelling. They will be represented as domain rules in the domain models. Modelling languages should be able to provide a necessary formal description mechanism for these domain rules so that the code generator can correctly understand and identify and so deal with them correctly.

(2) Usable

This property is hard to define. It is related to the experience gained by the modellers when they use a modelling language to develop a domain system. It is also related to what the modelling language requires from the knowledge system of the modellers. We mainly work from the following aspects to measure and enhance usability when in the design of XMML.

- (a) Definition of simple syntax and clear semantics

It goes without saying that when completing some task, the simpler the syntax of the modelling language is the more helpful this is for the modeller. Complex syntax structure not only increases learning difficulty for users but also increases the probability of errors in the process. Similarly, clear semantics used in the definition of modelling elements will help users more accurately use the modelling language.

- (b) Provide a visual definition mechanism

In modern software development technology, visualisation technology is becoming more and more important. It can effectively speed development,

enhance intuition and increase interest in the development process. Visual modelling is also a key characteristic of popular modelling languages. Compared with general modelling languages, the domain-specific modelling language must be able to provide corresponding visual definition to various domain modelling elements so as to make the modelling language easier to use. It is helpful for domain users to be able to easily use and understand the domain model through their domain knowledge coupled with the use of graphical representation.

(c) Support tool implementation

To provide necessary tool support is an important means of enhancing the usability of a modelling language, especial for graphic modelling tools. It can hide complicated details of the modelling language. The users can operate and use the modelling language by using a simple and intuitive graphical interface and quickly construct the required domain application model. Therefore, we must consider how to effectively realise tool support when designing a modelling language. At the same time, we can consider providing further assistance by transferring some of the language features and tasks to tools such as checking the grammar of the modelling language, automatically checking the models, etc.

(3) Verifiable

It is essential that a modelling language should have a good formal foundation. In principle, the method is to use mathematical and logic methods to describe and verify software. It cannot be denied that the formal definition of modelling language is a challenging task. However, from the practice of software development, it is difficult for general software developers to accept the formal method at present. In the history of software development, the earlier software was used in numerical calculations with programming languages focussing on describing functions and on algorithms. Later, database

application and data structure became increasingly important. Today's software systems are more complex. Early attempts to achieve automation in the development of software systems by formal mathematical language modelling proved to be a hard target to achieve.

In an analysis of DSMLs characteristics with DSM methods, compared with some strict formal development methods based on mathematics and logic, the requirement for formalisation depends more on usability and practicality. Therefore, when designing a modelling language formalisation is not the final goal and we should locate this instead at the level of verification where it can better assist in the realisation of the modelling language.

In XMML, the target of formalisation relates to two aspects: first, the modelling language itself should be formal enough to ensure that the model described by the modelling language can be correctly parsed by the code generator, and also to support model checking, rule checking and model refinement, etc. Second, to provide users with a formal means of description to express the properties of modelled objects in the modelling language e.g. static constraints and dynamic constraints.

The formal description of the model can be produced in several ways. These include: based on logical, state machine, network, process algebra, algebra, a special programming language, as subset of programming language, etc. In order to achieve the benefits of convenient tool support, users of XMML can more easily accept taking a descriptive method of programming language into account. Meanwhile, we adopt the general programming language as the formal descriptive language of the modelling language. For example, the formal description of domain rules for the model is realised by the general programming language.

The formal infrastructure of XMML itself adopts XML and XML schema as formal basic languages. At present, formal description technology based on

XML and XML schema is growing mature and has been successfully applied in various formal description languages such as, xADL、ABC / ADL、ACME, etc. Its characteristics are strict formality; clear; easy to understand; easy to parse; easy to store and exchange; easy to expand; abundance of tool support, etc. This can provide basic language support to the formal definition of syntax structure of the modelling language and some semantic attributes. However, XML and XML schema cannot provide good support for complex semantic definitions. Therefore, a general programming language (such as JavaScript) is introduced to make up for the deficiency and to provide formal description of complex domain rules for the model.

5.2 Design Model of XMML

5.2.1 Layer Architecture

XMML is a domain modelling language with the capability of meta-modelling. The process of using XMML to complete the modelling of the target domain application system is a treatment process, which is a multi-level instantiation of XMML (from abstract to concrete). The design model of XMML is realised by adopting layer architecture. Regarding the concepts behind the architecture of the layer modelling language, the typical application is the layer definition structure of UML, shown in Figure 5.2.

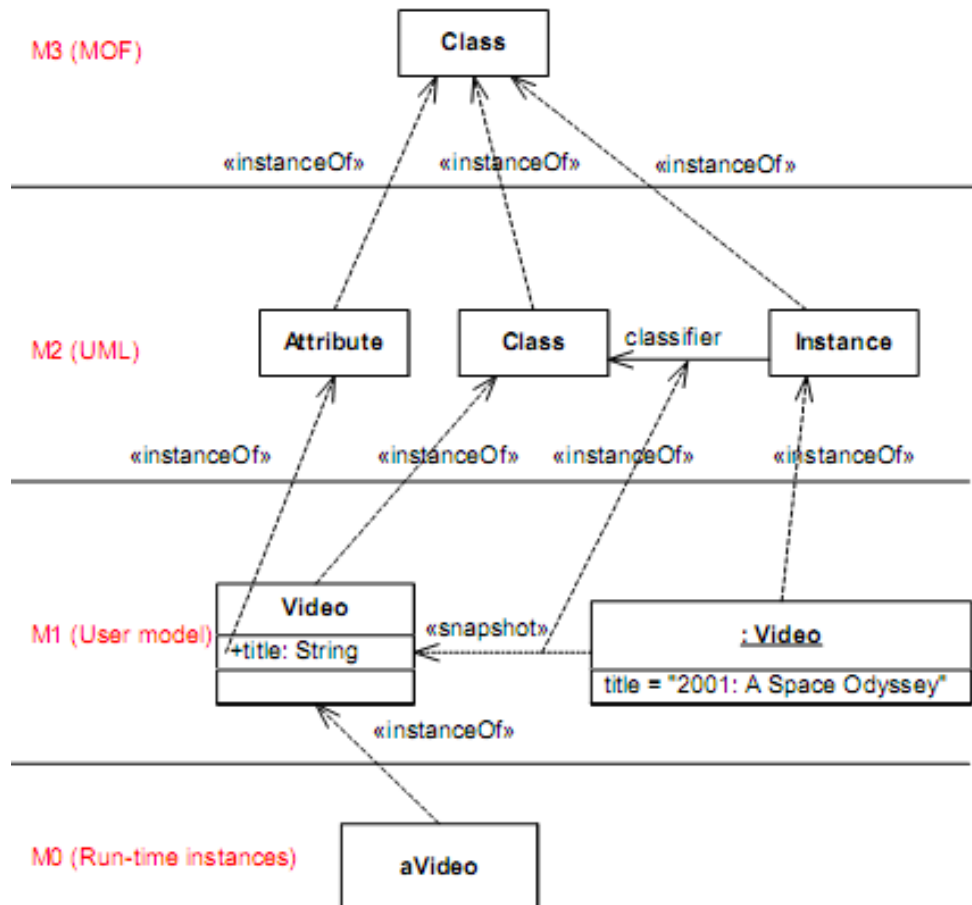


Figure 5. 2 Layer Architecture Model of UML

There are 4 layers of UML architecture. The M0 layer is an example of a target system application model; the M1 layer is a system model built by users; M2 is meta-model of UML, and its syntax is defined in the layer; M3 is a modelling layer, modelling the meta-model and mainly existing to define the specification of the meta-language. It is defined by MOF, the meta-object basis of UML.

The layer architecture of XMML adopts a similar structure and there are 4 layers according to the abstract level of the language, as shown in Figure 5.3.

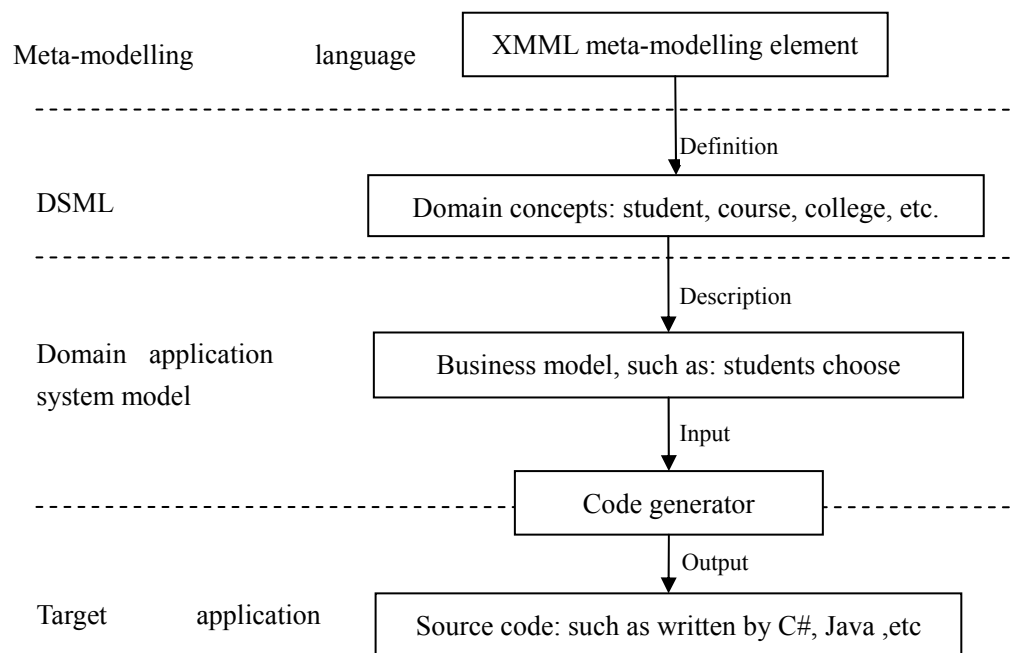


Figure 5. 3 Layer Architecture Model of XMML

In the layer model of XMML, we see XMML is at the meta-modelling language layer. This supplies the necessary meta-modelling elements for the development of the domain-specific modelling language. These meta-modelling elements do not have any semantic characteristics of the domain. The language elements of XMML are independent of the domain, and we can use them to define various domain-specific modelling language elements. These include: concepts, business, rules, etc, which are abstracted from the domain. Users of the domain model can use the well defined domain modelling elements to create corresponding domain objects, which are instantiations of domain modelling elements of the meta-model and to be used to draw together and construct the domain application model. Next, the domain business model obtained by domain modelling is input into the code generator. This can examine, check and parse as well as generate corresponding target source code according to business logic and rules designed by the meta-model developer.

In such a hierarchical structure, language concepts of the upper layers are

abstracted from lower layers. For example, the modelling elements of the DSMLs layer are abstracted from domain object concepts of the domain application system model layer, while its concepts are abstracted from concepts of the real world target application system. The semantics of the modelling language provide a bridge between defined concepts in the upper layer and model instance concepts of the lower layer. Across this bridge, upper layer concepts are endowed with practical significance, and the lower layer concepts are highly abstracted and pre-digested descriptions by upper layer [52]. The semantics of the modelling language represent the bridge between concepts in the two layers and map their relationships. By this mapping, the lower layer concepts give practical significant to the upper layer.

Use of the layered structure is helpful in the division of labour and cooperation in the development organisation. In the architecture, the design of the domain-specific modelling language and development of code generators are achieved by the meta-model developer and modelling of the domain application system is completed by the domain model designer. From this angle, domain model designers are users of the developer of the meta-model. Besides, another great advantage of using layer architecture is that it helps reduce the risks brought about by variations in requirements. With the hierarchical structure, such variations, which are likely to occur in any practical project, can be classified into three types:

(1) Variation from business model

This type is the most common variation of system requirements. A typical example in an office automation system would be the variation of a user document delivery processes. Another would be variation of credits in a student management system. Usually, this kind of variation is not related to a change in the nature of the concrete domain concepts but caused instead by a change in a combination of domain objects or in the distinguishing conditions of some domain object attributes. When these variations happen, they are only related to

an adjustment of the domain model layer and do not affect the domain meta-model. As a result, what we need to do is to adjust the original domain business model according to these practical demands and regenerate the code.

(2) Variation from domain concepts

This type of variation is the opposite of the former one. This kind of requirement variation is usually caused by changes in some concepts of the domain application. For example, an office automation system has to be updated to take account of a new type of document type or some departmental organisation. Here, the domain modelling elements of the original design of the meta-model will not correspond with the new domain application concepts or rules. In these circumstances, there is a need to bring meta-modelling into effect and adjust the treatment rules of the code generators according to these newly appeared domain concepts. If this kind of variation is not related to a change in the business model, we will still be able to reuse results of original domain application modelling. For example, if a new kind of student was added to the student management system, the original business model for student registration and choice of course could remain unchanged. Here, the original business model can be directly applied to the new domain concepts so that in practice much of the domain can be reused.

(3) Variation from implementation technology or platform

Such changes are usually due to technical changes in the software system, for example, the change of a programming language. Say, the original language is Java but is then changed to C#, or by transplanting the database system. Such changes will not give rise to changes at the domain level and the original meta-model and the domain model remain the same. However, what we need to do is to change the code generator and the underlying technical support to the framework.

Thus it can be seen that the hierarchical structure of the modelling language

is a suitable way of thinking for the characteristics of the DSM development method. It provides language-level support for successful application of the development method and has a fundamental effect in promoting sound definition of staff responsibilities of staff and in identifying concerns.

5.2.1 Meta Description Language of XMML

This thesis uses XML (extensible markup language) as the meta-language to define XMML. As a standard independent operating system and programming language, XML is not only a structured language, but is also used to define other language systems. It is usually used as a meta-language to define structured language organised with graded and strictly nested data objects.

Attaching the term “Meta” to the two concepts of “model” and “modelling language” can often cause misunderstanding and confusion. Therefore, it is necessary to explain which level XMML is in and this is put forward by the thesis from both these two angles.

As shown in the following figure, from the point of view of the model, XMML is a meta-meta model for the domain model and is used to characterise and define meta-model-DSMLs applied in domain modelling, while the relationships between DSMLs and Domain Model are those of Type and Instance. Meanwhile, viewed from the modelling language, XMML is a meta-modelling language based on XML, which is used to define and develop DSMLs, at the same time, it is meta-language used to define the domain model.

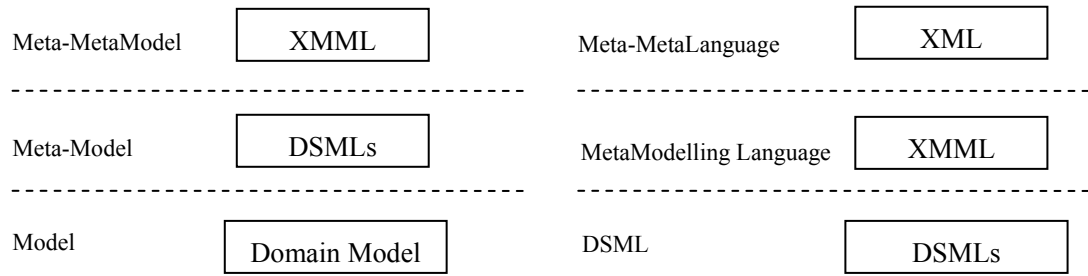


Figure 5. 4 XMML from the View of Model and Modelling

The use of XML to define XMML is mainly based on the following considerations:

Firstly, as a formal meta-modelling language, XMML must provide a clear definition for meta-modelling elements. In XMML, groups of XML Schema define the syntax structure of these modelling elements, which are shared by all DSMLs. XML tags defined by XMML also have the basic function of providing information to describe various domain-specific modelling languages so that there is interoperability among the domain-specific modelling languages developed by XMML.

Secondly, XML's tag name and its relationship can be freely defined and extended, just like the original meaning of XML as "XML is a description language by markup", and the hierarchical structure marked by XML tag names of can be defined by the user. That is, Tags set with special purposes for the users can be defined according to the syntax of XML, and this can form a new symbolic language. We can say that XML is "a language used to define language", namely a meta-language.

In the definition technology of XMML, XML Schema is used to define and describe the structure of XMML documents and content patterns. It is used to define what elements exist in XMML documents and the relationships among these elements, and it can also define element and data types of attributes. XML Schema is made up of components, such as type definitions and components of the element statement, and can be used to evaluate the validity of well formed

element and attribute information. XML Schema is a set of Schema components, these components are divided into three groups:

- (1) Basic components: Simple type definition, Complex type definition, Attribute declarations and Element declarations.
- (2) Components: Attribute group definitions, Identity-constraint definitions, Model group definitions and Notation declarations.
- (3) Helpful components: Annotations, Model groups, Particles, Wildcards and Attribute Uses.

Application of XML Schema can effectively overcome the many shortcomings of early DTD such as:

- (1) DTD is based on regular expressions, so descriptive capacity is limited.
- (2) DTD has no data type support, and lacks capacity in most application environments.
- (3) DTD has limited constraint definition capacity and it cannot make more detail semantics constraints on XML instance documents.
- (4) DTD is not sufficiently structured so the cost of reuse is relatively higher.
- (5) DTD does not use XML as its means of description and access is formed without a standard programming interface so DTD cannot be maintained by using standard programming.

While designing XML, Schema can address these shortcomings of DTD as XML Schema has the following advantages:

- (1) XML Schema itself is based on XML and has special additional language
- (2) XML can be parsed and dealt like other XML files

- (3) XML Schema supports a serious of data type (int, float, Boolean, date, etc)
- (4) XML Schema supports scalable date models
- (5) XML Schema supports comprehensive namespace
- (6) XML Schema supports Attribute group definitions

XML Schema is self-described by XML 1.0, and uses namespaces, has rich nested data type and extremely strong data structure definition functions. It completely changes and greatly expands DTD capacity (traditional mechanism of describing XML document structure and restriction of content) and becomes an official language type of the XML system. It is a solid foundation of the XML system together with XML specifications and namespace specifications.

5.2.3 Design for Syntax and Semantics

The design of a model language includes syntax design and semantics design. The syntax design includes both abstract syntax definition unrelated to concrete expression of the model language and concrete syntax definition related to concrete expression of the model language. The concrete syntax is usually classified through text syntax expressed by text and as graphical syntax expressed by graphics. The semantics used to express the meaning of concepts described by the abstract syntax of the model language. A good understanding and correct use of modelling concepts by the modeller is based on an accurate understanding of the semantics of the modelling concepts.

When designing a modelling language, the relationship among abstract syntax, concrete syntax and semantics can be partitioned as two mappings which do not cross over each other [56]. One mapping is between Modelling concepts and concrete syntax, the other is between modelling concepts (abstract syntax) and instances (semantic domain). The concrete syntax concepts are

concrete expressions of abstract syntax which can have several or multiple concrete syntaxes. The mapping between abstract syntax and concrete syntax is used to express the relationship between them.

The semantics of the modelling language is expressed as mapping between model concept elements and model instances objects. By the separation of the above two area abstract syntax, concrete syntax and semantics can be made relative independence to reduce coupling between syntax design and semantics and so improve efficiency of the design of the model language.

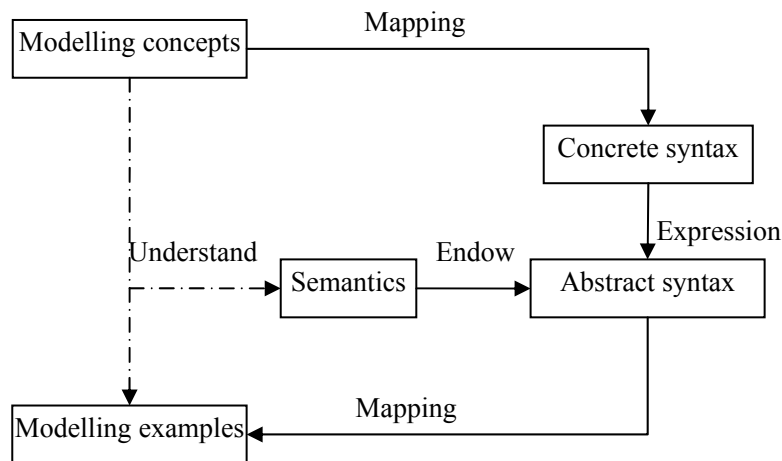


Figure 5. 5 Relationship Model between Syntax and Semantics of Modelling Language

The semantics of a modelling language describe the meaning of the concepts of the language. When using a modelling language, if we want to understand how to use a concept, we need to specify the meaning of that concept of the language. The understanding of a modelling language element that we select to use in modelling is a key issue for any specified aspect of the problem domain. It is necessary to give a formal method for describing the semantics in the design of the model language. But, for a model language applied in software engineering projects with the intention of replacing a programming language, the key method for describing semantics is by practical means.

There are many ways to describe semantics. One way is to use mathematical methods (such as the lambda calculus, Turing machine, π calculus, CSP, etc.) to describe the semantics. Many theoretical studies [106, 121, 123, 46, 30] have used this method to describe the semantics of a great many modelling languages. However, complex mathematical description is hard to understand and so restricts practical application. Another method is to express semantics by an external programming language and this is more practical. But, this method will lead the modelling language away from the original platform independence to become related to the external programming language to a certain extent. Besides, it also makes the designed environments for providing the model language induce a programming language together with a definition of the model language needed to interpose another language. This leads to a separation of the definition process. So, how can we overcome the problem?

The semantics of a modelling language are quite different from the abstract syntax model of a modelling language. The model of the abstract syntax defines the language structure and well formed relationships and is a prerequisite for the definition of semantics. Meanwhile, semantics have some rules and these rules prescribe whether the expression of a language is well formed or not and add a layer of meaning to concepts defined by the abstract syntax. Based on these principles, in the DSM method the concept of separation of semantics is used to solve the problem. That is, parts of the semantics are separated by the code generator and wholly put into the code generator and described by the external programming language in order to, as far as possible, maintain platform-independence of the modelling language and linguistic homogeneity. At the same time, the static semantics (such as the corresponding relationship between Class of UML and the object of the semantic domain) in the modelling language can still be retained and these static semantic rules can be used by modelling language tools, such as to check whether the model element type is consistent or not, and whether there will be connections among modelling

elements, etc.

5.3 Abstract Syntax Definition of XMML

5.3.1 Abstract Syntax Model of XMML

The Abstract syntax model describes the essential character of the Modelling Language. It is used to describe relationships between concepts of modelling language and concepts. In the design of a modelling language, in addition to recognition, concepts of the descriptive language and modelling of these concepts, the abstract syntax model is also needed to define and judge whether the model written by the language has legitimate rules or not, and if these rules are the rules of well-formed model language. XMML is a meta-modelling language that describes a concrete model of the DSML. So, a key area is defining the basic concepts needed in the design of a domain-specific language and the relationships among these concepts. Besides, the rules used to restrict whether the constructed model is legitimate or not, are also needed to describe such factors as whether we can build a connection between two domain modelling elements, etc.

The purpose of constructing an abstract syntax model of the modelling language is to describe relationships between basic modelling concepts of the modelling language and concepts. During the design of the modelling language, the concepts and relationship between these concepts are also basic to the design of related products (such as concrete syntax, semantic model) of all other languages. In the context of modelling language definition, a concept is anything that can express a word in the language. Abstract syntax as opposed to the specific expression (the concrete abstract) of concepts, is concerned with the abstract expression of concepts, rather than what is presented to the user or the meaning of its components.

The modelling objects of XMML are DSMLs. Like the domain-specific

application system model, DSMLs are also domain-specific models. We can obtain an abstract syntax model of XMML by analysing and identifying the basis for the composition of DSMLs as shown in Figure 5.6.

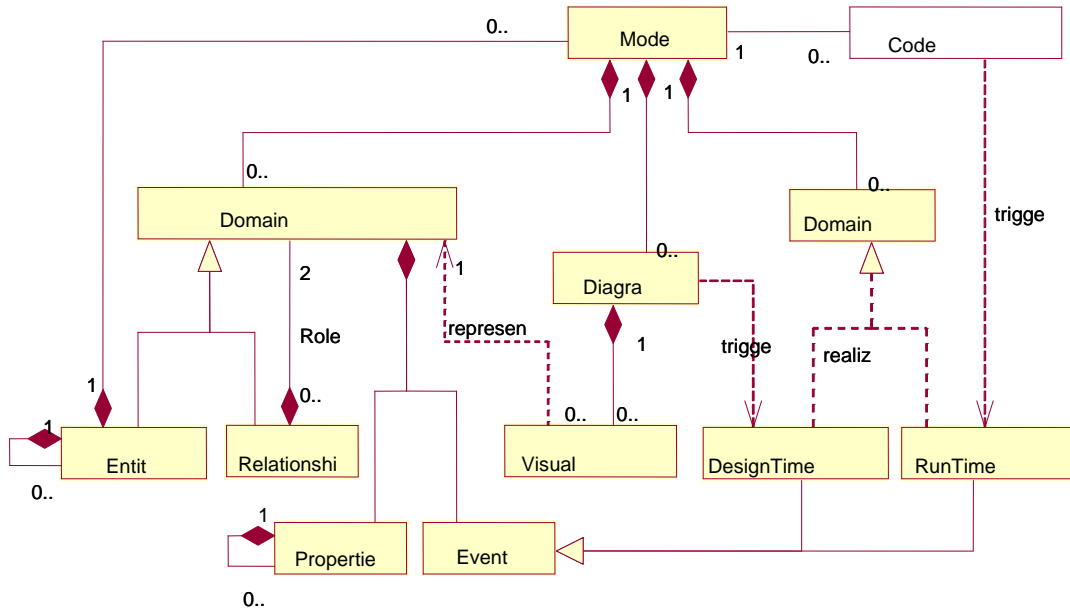


Figure 5. 6 Abstract Syntax Model of XMML

Abstract syntax model of XMML expressing the necessary modelling elements for building DSMLs as well as the relationship between these modelling elements.

5.3.2 Abstract Syntax Model Elements of XMML

(1) Model

In XMML, a modelling type or domain-specific solution is expressed as a model. A domain application system can be described by many models. Every model describes domain problems and solutions. At the same time, and we can build many different types of model for the same domain problem and describe it from various perspectives.

The key purpose of building a model is to express domain concepts and the relationship between them as well as their constraints and configurations. For some domain problems, it is also necessary to exhibit various aspects of a

model by different views. Therefore, in XMML, a model has three main constituent elements: domain elements, domain rules and diagrams. The relationship between model and them is one to many. In order to make a model produce different software artefacts (such as source code of various programming language or documents) it can have several code generators associated with it. These code generators can use the model as input to parse content described by the model. They can deal with the whole model or only parse a partial view of the model as decided by the designer of the code generator.

(2) Domain Element

The main key task of domain modelling is to map domain concepts in the domain as domain elements [9]. In XMML, a domain element is derived as an entity and a relationship. “Entity” is used to express the type of the various entity modelling elements of the domain and they are instantiated as various specific entity object during domain modelling. The type of relationship existing among domain entities is expressed by “Relationship” and these are instantiated as associations among various domain elements. The relationship itself in XMML is a binary, but we can achieve the purpose of expression of n-ary relations by combining entity and relationship even with association between relationships. The entity itself can nest many entities within it and in expressing requirements for one entity we can be including many sub entities. Besides, some complex entities may need further decompositions and use new models to express the structure after decomposing. Therefore, in XMML, we can define many sub-model types to be allowed when creating on entity to characterise a multi-level model structure.

(3) Diagram and Visual Element

The visual modelling language should be able to show each composition element of the model and the associations among them. Furthermore, the same

model should be able to express every aspect of the model from different views, or use many views to illustrate each part of the same model. In XMML, a diagram is used to express view information for the model, and many diagrams can be built in a single model. In a complex model, each diagram only expresses a part or an aspect of the model. The make up of the model itself is not related to its visual form. The relationships between diagrams and model are loosely coupling.

In XMML, the representation of a Domain Element in a Diagram is achieved by a Visual Element, which is used to define the visual design of each domain element. The same domain element may present a different graphical appearance in different views, therefore, the relationship between domain element and visual element is one to many.

For a visual modelling language, the appearance of its modelling elements should be able to directly express the corresponding domain concepts. At the same time, in the visual modelling environments, it is also necessary to make these graphic modelling elements able to respond to interactive actions by the users. So it is necessary that the meta-modelling language should be able to provide a flexible and valid extended mechanism to enable developers of DSMLs to define the appearance of various domain elements together with their interactive behaviours. Therefore, we designed a visual description language structure into XMML to enable it to define both the appearance of visual elements and interactive behaviours. The following XML fragment shows how to use the kind of visual description language structure to define a UseCase's appearance and handing information for interactive events.


```
<?xml version="1.0" encoding="UTF-8"?>
<VisualElement id="..." type="UseCase" elementId="" events="">
<div id="UseCase" style="background-image:RES(UseCase.emf);
border:none; " features="linkable:true; resizable:false; moveable:true;
editable:false; hostable: false; selectable:true"
events="ondbclick:fnOnDbClick();">
  <div id="UseCaseText" style="color:black; text-align:center;"
features="linkable:false; resizable:false; moveable:true; editable:true;
hostable: false; selectable:true">
    </div>
  </div>
<script>
  function fnOnDbClick(){
    ArchwareAPI.openPropertyWindow(ArchwareAPI.currentElement.Id);
  }
</script>
</VisualElement>
```

Figure 5. 7 Example of Visual Primitive Definition

Among them, the node `VisualElement` is used to define every part of the Visual Element, such as a background picture, editable text box, etc. At the same time, a variety of UI interactive event action calls can be specified for them to enhance the user's interactive experience of the process of graphical modelling. `HotAreas` are used to define the connectable regions where graphical elements connect to association lines and the regions where sub-element can be placed. An outline node is used to define the drag path for an accessory such as a port. Due to the need for this visual description, the structure of the language is designed using a method compatible with Document Object Model (DOM) [112] of W3C. This means that the developers can dynamically operate the visual structure of the modelling elements by using DOM API when modelling. For example this could be by using the event mechanism to call script to achieve dynamic addition of text or graphics, or by changing visibility and size, etc.

(4) Domain Rules

Domain Rules are used to characterise the essential key components of a domain model. They are related to business rules in the application domain and to domain knowledge. They usually represent some constraint or configuration information when they are mapped to the model together with domain knowledge. They will have an impact on the modelling process and the results of code generation. Therefore, the formal description of domain rules is a key characteristic of a modelling language supporting DSM development [39].

Domain rules can be divided into two types, fixed rules and conditional rules. The former mainly expresses some fixed constraints of the domain and are not affected by the properties of the modelling elements when they are applied in the model. However, they are affected by constraints arising from the type of modelling element. Such constraints usually manifest themselves as what modelling elements are allowed in the model and fixed constraints, such as what kind of associations are allowed among these modelling elements. Meanwhile, conditional rules relate to certain flexible domain constraints. Whether a conditional rule is applied or not is decided by some complex domain business logic. For example, during the modelling process a conditional rule might require that certain kinds of business rules should be applied according to the properties of some real world domain elements.

For the fixed rules, we can use visual modelling to complete a formal description when meta-modelling, but it is difficult to express conditional rules in a graphical way. For example, we can use a graphical means to define modelling elements in the model together with their organisational structure, but we cannot specify some special business logic to be applied among them. As conditional rules are a very complicated variable, it is necessary that some flexible mechanism should be provided to achieve a formal description for them.

Adopting pure mathematical or logical methods can provide compact and precise formal definition for domain rules and many theoretical and prototypal modelling language use formal description mechanisms. A descriptive method is more complex and difficult to use for most developers and it is difficult to achieve tool support. Some modelling languages and tools use special specification languages (such as: OCL, B Language) to achieve a formal description of domain rules. This also causes some difficulties for the developers who study and use these languages. In XMML, we use a general programming language (such as, JavaScript, VBScript, Pascal, etc.) to formally describe the model's domain rules. Due to the fact that they are executable, they are not only used to describe domain rules, but can also be used to dynamically and directly change some domain rules into mechanisms for the operation and control of the model with the support of modelling tools.

Domain rule modelling not only defines what the rule is it must also define what it is to be used for and when it is to be used. We use a domain rule modelling method based on events to achieve these requirements. Namely, according to their conditional character to classify the domain rules, then each type of these domain rules can be mapped to various events. The developer can use the general programming language of the corresponding event-handler to describe domain rules corresponding to the event together with a series of logical processes initiated by these rules.

For example, take a domain rule, “instance object of modelling entity type A can only appear once in the model” that applies to a certain model and is to achieve a Singleton Pattern, for the domain rule. Here, we can use an onElementCreate event for modelling entity type A and use JavaScript to write its process logic, as shown in Figure 5.8.

```
var arrE = XMMLAPI. GetElements (this.model," EntityA");  
  
if (arrE.length>0) {  
  
return false;  
  
}else{  
  
return true;  
  
}
```

Figure 5. 8 Example: Creating Domain Rules by onElementCreate Event

Among them, XMML API is a global object of the model reflection interface, and we can use it to access and operate elements of the model and objects. The return value of the event handling denotes whether the event triggered by the operation meets domain rules or not.

Compared with methods (such as, pre-condition, post-condition, assertion and Invariant) of modelling the declarative rules used by some modelling languages, a modelling method for domain rules that is based on events can provide a more flexible, clearer and more accurate means of classifying DSML development. At the same time, due to the use of an executable programming language to formally define domain rules, the functions of modelling tools related to domain rules can be defined and achieved by DSMLs. So, the DSML is not just a static declarative modelling language, it can better integrate with tools and provide functionality extension for modelling tools.

(5) Properties and Events

Properties and Events are attributes that every domain element should have.

A set of properties is used to describe and record various characteristic values of the domain element. In XMML, the form of name-value can be used to describe the data structure of simple attributes. We can provide a form designer for this in the modelling language's supporting tool to help the meta-model developer design forms applied in editing complex attributes.

Events are a language definition mechanism provided for DSMLs developers to use when they are modelling in accordance with complex domain rules and so achieve event-driven modelling to suit the various complex modelling requirements. We define various Event Types (such as OnElementCreate, OnElementDeleted, OnPropertyChanged, OnGenerate, etc.) in XMML and developers can write a corresponding event handler for each Event Type. This can achieve the application of, or a check on, the corresponding domain rules by triggering a modelling tool to execute corresponding processing logic.

The Events of XMML can be classified into two types: DesignTime Events and RunTime Events. The former refers to those Events that happened at the time of design, and which can provide the modeller with dynamic interaction and feedback during the process of modelling. For example, an entity is put into a diagram or an association is built between two entities that will trigger their OnElementCreate event. As programming progresses it can be judged whether or not the element has been legitimately built or whether it is necessary to initiate some implicit configuration actions. Meanwhile, the term RunTime Events refers to those events that happened during the model processing phase. For example, when code is generated it triggers OnGenerate events for each of the modelling elements. At the event processing program, each modelling element can examine and check some attributes and generates its code fragment of to achieve automatic code generation mechanism. To enhance the usability of XMML, developers can use any technology that supports ActiveX (such as JavaScript, VBScript, Perl, etc.) to write even handling scripts and there is no

need to study a new special programming language. The integrated environment can provide a model reflection COM calling interface, and scripts can use these APIs to read model information or to operate the model.

5.4 Concrete Syntax Definitions of XMML

5.4.1 Overview of XMML's Concrete Syntax

Although an abstract syntax can describe the potential words or syntax of a modelling language, it does not define how to express that abstract syntax to the modeller. Such details will be described by concrete syntax. The concrete syntax of the modelling language mainly includes text concrete syntax (such as text syntax of Java language) and graphic concrete syntax (such as graphical symbols of UML). These are called text syntax and graphic syntax. Some of the concrete syntax of the modelling languages uses graphic syntax and some uses text syntax and it is ok to use both of them. The concrete syntax provides the modeller with a concrete means of expressing the model. There are different views of abstract syntax, so the modelling language can have many kinds of concrete syntax or many different concrete syntaxes to express its own abstract syntax.

Disposal of the concrete syntax of a modelling language is divided into two phases. The first phase includes parsing the concrete syntax and ensuring it is legitimate; the second phase is to build an abstract syntax using the concrete syntax. Although the two concrete syntax forms built by the modeller are very different, the above two phases are also suited to text concrete syntax and graphic concrete syntax. The graphical model is usually built in an interactive way, therefore it is increasing. Meanwhile, the graphic syntax must be parsed synchronously with interaction with the users; while text syntax is batch parsed, users use this syntax to construct a complete model, then the model is transferred to the parser. Syntax is dealt with in much the same way by the

programming language's compilers or parsers.

XMML is a visual modelling language, but its concrete syntax both uses text concrete syntax and graphic concrete syntax, which appear as text concrete syntax, to provide a visual means of modelling. In the realisation of XMML modelling tools, unidirectional synchronisation is carried out. That is, while the modeller is using a modelling tool to gradually build the model, the background of the modelling tool will synchronously parse graphic syntax as an equivalent text syntax representation of the abstract syntax. On the other hand, if the modeller wanted to use a text model, due to being unable to get information on the size and location of the graphic, it would be impossible to generate the corresponding model diagram. However, the code generator's parse for the model does not depend on graphical syntax information, so the model represented by text and built directly with the use of text mode can still be input into the code generator to generate code output corresponding to the model.

The concrete syntax of XMML is defined by XML schema, shown in Figure 5.9.

5.4.2 Concrete Syntax Description Scheme for XMML

The above gives an abstract syntax model of the basic modelling elements necessary to create a domain-specific modelling language, model, entity, relationship and diagram, etc. but it does not give clear definitions for and description of the attributes and actions of these basic modelling elements. There is also a further need to describe the concrete structure of XMML from the definitions of the concrete syntax.

XMML is a meta-modelling language based on XML, the meta-model as built together with the specific application system model based on the meta-model use the same concrete syntax structure. The descriptions of the two models are finally persisted as a XML file, therefore, the concrete syntax definition of XMML is described by XML Schema. XML Schema is a syntax tool used to define syntax based on XML, include all the functions of DTD, and enhance the stylised character of linguistic elements. Consequently, the structure is more rigorous. At present, most new standards of W3C are described by XML Schema. Its position and role are similar to the BNF paradigm, which is used by the syntax of other programming languages [53]. The concrete definition of XMML uses Schema rather than the BNF paradigm. The reason is that Schema can be expressed by XML, and expediently form a SOM (Schema Object Model) to enable other applications to know about and parse its structure, and better provide tool support for grammar analysis and checking.

(1) Model

In XMML, a domain modelling target or a solution to a domain-specific problem is presented as a model and this is the basic unit that the code generator has to parse. It is made up of member objects for describing the model in detail, such as, domain entities, associations, diagrams, etc. Its concrete syntax structure is shown in Figure 5.10 (for the concrete Schema language description please refer to Appendix A):

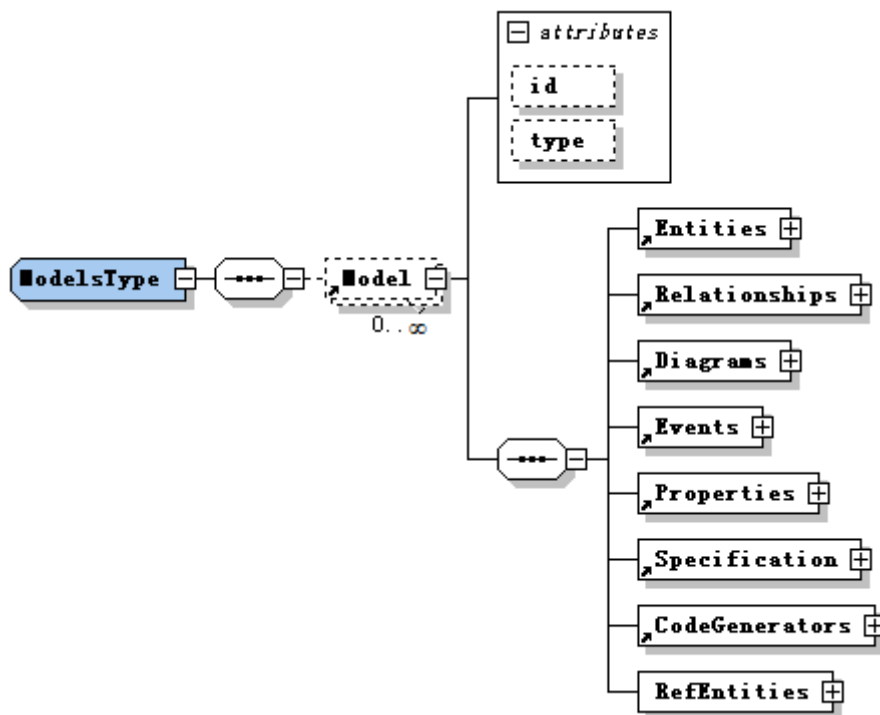


Figure 5.10 Definition of ModelsType Schema

In the XML Schema for defining syntax, the structure of the grammar of the Model is defined as a complexType. These complexly define the description of the structure of the Model by using other complexTypes. A description of the meaning of each composition element is shown following:

(a) Attributes

Describe basic attributes of the model object itself. Among them, id is a unique identifier of the model. Meanwhile type is used to describe the type of model.

(b) Entities

Each represents a set of various domain concepts of model objects. The elements of the set are described by a complexType EntityType.

(c) Relationships

Denote a set of associations among various entity model objects. The

elements of the set are described by a complexType RelationshipType.

(d) Diagrams

It is a representation using visual means to display a set of model objects in diagrammatic form. The elements of the set are described by a complexType DiagramType.

(e) Properties

It used to describe various items of attribute information in the model. Due to the use of XMML applied in meta-modelling, the attribute information of an element has extensibility. So, in XMML, the definitions of Properties elements adopts an open structure and be defined by a complexType as shown in Figure 5.11.

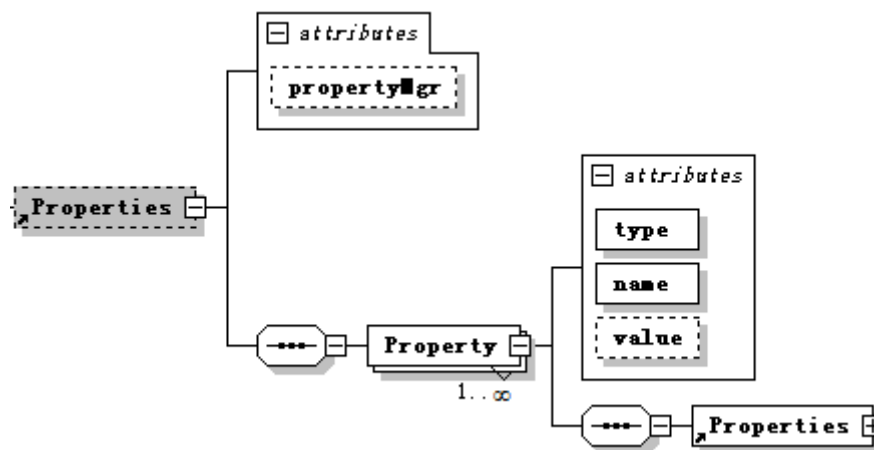


Figure 5. 11 PropertiesType Schema

As shown in the above figure, definitions of Properties are specified by meta-model development, so their infrastructure is regulated by the syntax of XMML. While PropertyMgr introduces definitions of Properties in charge of parsing and setting attributes in the actually modelling environment, PropertyMgr manages the special attributes of components. Its action is developed by the meta-model developer according to object attributes described by Properties to develop the corresponding data management interface and

processing logic and to provide query, rewritten, enumeration, and other operations with attribute information of external objects.

Properties are a reusable type of descriptions of structural definitions and are reused in Entities, Relationships and Diagrams, etc.

(f) Events:

Events describe certain constraints, rules and actions of model objects formed by events. In the modelling environment, corresponding judgements, verifying and processing actions can be triggered timely by the corresponding events. This uses complexType to describe related information as shown in the following Figure 5.12.

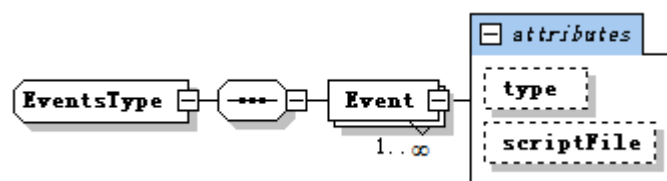


Figure 5. 12 EventsType Schema

The meta-model developer can define various event types in the event structure, and event handling can be described by corresponding scriptFile.

(g) Specification

Represents a set of some semantic features of the model and is used to describe model specification information such as, model functions, modelling intentions, required constraints and so on. The way in which they are described can be formal or informal and the meta-model developer will decide how to extend and use the structure. It is a complexType, as shown in the following Figure 5.13.

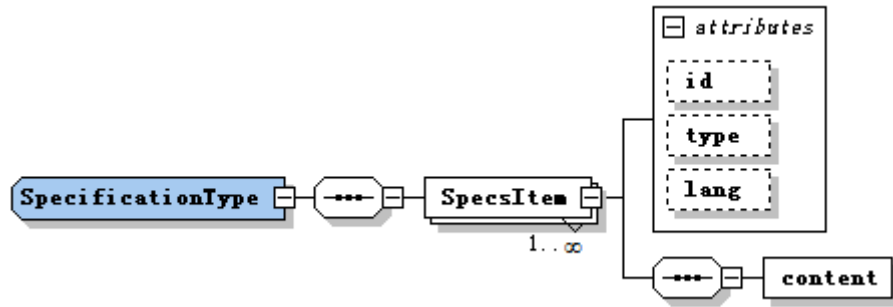


Figure 5.13 SpecificationType Schema

(h) CodeGenerators

Express how a set of code generators can be accepted as input to the model. A model can be associated with many CodeGenerators to produce different outputs. Its structure is shown in the following Figure 5.14.

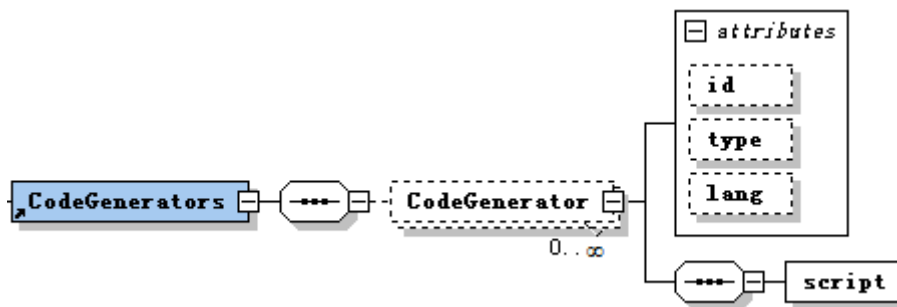


Figure 5.14 CodeGeneratorsType Schema

(i) RefEntities

It used to describe entity references introduced from other models. Its structure is shown in the following Figure 5.15.

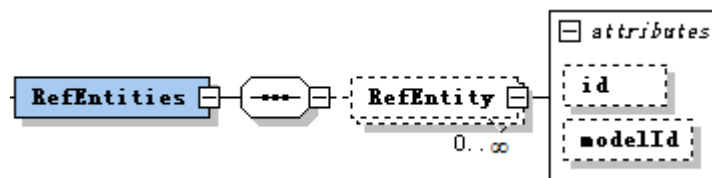


Figure 5.15 RefEntitiesType Schema

(2) Entity

After domain analysis, we can extract and get various domain concepts, which will be instantiated by various concrete entities when domain modelling to express the substantive content of the model. They can be big or small and decided by granularity of modelling. At the time of final code generation, some entities may correspond to a module or correspond to a class. The concrete syntax of description of these entities is defined by an Entity element, and it is a reusable complexType. Its structure is shown in the following Figure 5.16.

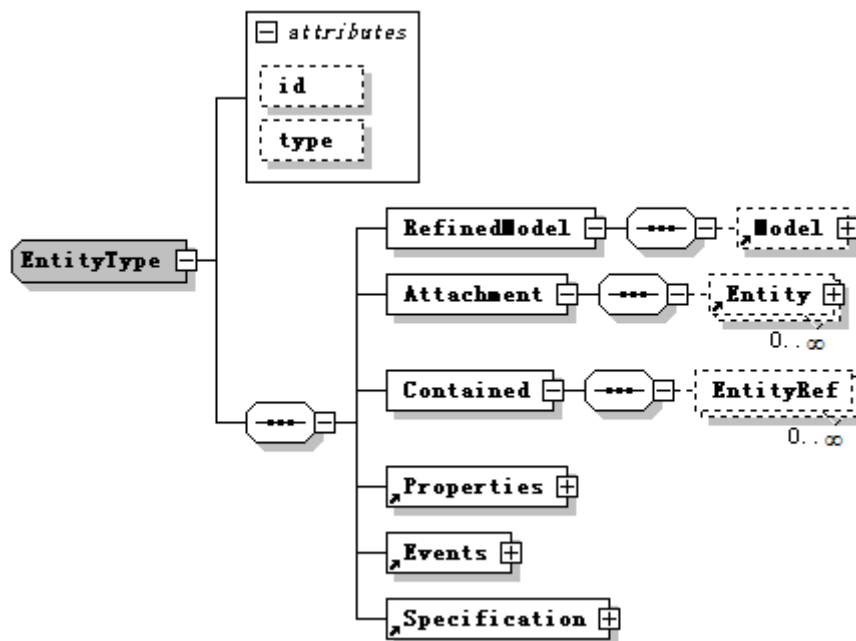





Figure 5. 16 EntityType Schema

	<p>The symbol for a ComplexType.</p>
	<p>It represents the type with two attributes.</p>
	<p>The symbol is used to denote the ComplexType with a series of sub-elements, namely, what kind of sub-element can appear and the appearance order of</p>


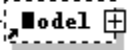
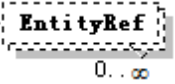
	these sub-elements.
	It only represents a sub-element, which can have its own sub-elements
	It represents a sub-element, what differs from the above one is that it is a reference, here(it has been defined previously as a ComplexType, so there is a small arrow at the lower left of the symbol, and it is similar to a shortcut.). Besides, it is optional (denoted as radiogroup box)
	It represent the number of sub-elements can be 0 or infinite

Table 5. 1 EntityType Schema Element

It is also necessary to further refine modelling to some complex, large granularity domain entity. The same domain object can appear in many domain models and a domain entity can be a function of the business or a non-functional static object. Its syntax is defined by XML Schema. The meaning of each of the various constituent elements can be described as follows.

(a) Attributes

Describe a basic attribute of the modelling entity itself. Among them, id is a unique identifier of the modelling entity itself. Type is used to identify the type of modelling entity and mark it for reuse.

(b) RefinedModel

It is used to express a refinement model included in the modelling entity. Usually, we need to further refine the model by building a sub-model when the meaning or function of a modelling entity is rich.

(c) Attachment

It describes an attachable set of sub entities of a modelling object. For example, in circuit design, a main component of the circuit can be designed as attaching many port components, here, the port components are its sub-entities. The lifecycle between sub-entity and main entity should be consistent. When the main entity is deleted, the sub-entities be removed at the same time.

(d) Contained

Describes some other entity set included in modelling entity objects. The relationship between them and the main entity object is loosed coupled and the deletion of the main entity has no impact on them, so the set just stores their references.

(e) Properties

It expresses the attributes of a set of entities. It is a complexType in Schema and is not affected by which type is used in the definition of the model.

(f) Events

Used for describing some constraints, rules and model object actions in the form of events. The event processing logic is specified by script. It is a complexType in Schema and is the same as the type used in the definition of the model.

(g) Specification:

Expresses a set of some semantic features of entities and is used to describe model specification information such as functions of the model, modelling intentions, required constraints and so on. It is a complexType in Schema and is the same as the type used in the definition of the model.

(3) Relationship

Relationship is used for describing the type of binary relation exiting domain

modelling elements and for building relationships between one entity and another and for the relationship between an entity and an entity relation as well as for relations between different entity relations. The connected modelling elements act as the role of the relationship, and the relationship cannot exist without a role. In XMML, the concrete syntax of Relationship is defined by XML Schema as shown in the following Figure 5.17.

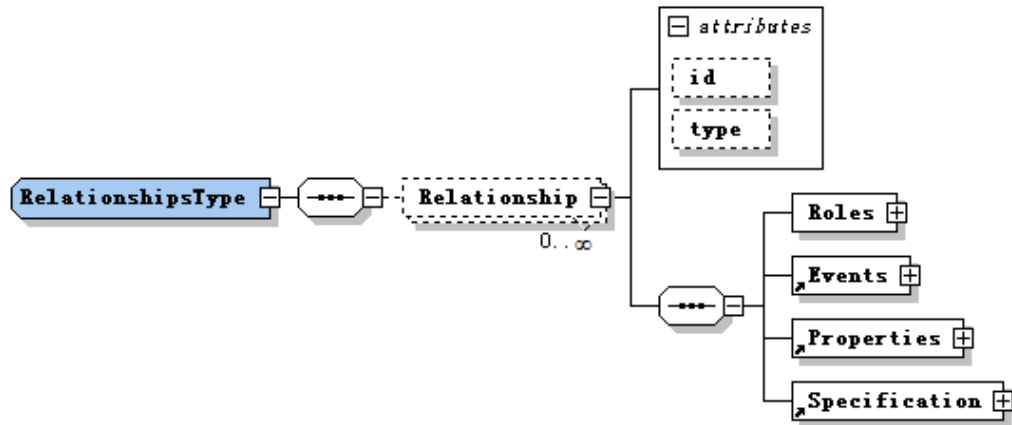


Figure 5. 17 RelationshipsType Schema

The meaning of each of the constituent elements is described as follows:

(a) Attributes

Describe a basic attribute of the modelling entity itself. Among them, id is a unique identifier of the description of the Relationship itself, type describes the type of relationship and marks it for reuse.

(b) Roles

Express role information of two modelling elements linked by the relationship. The role is used to describe information such as the position, effect, identity, etc. of two modelling elements that form binary relationship. It is a complexType element and its structure is shown by the following Figure 5.18.

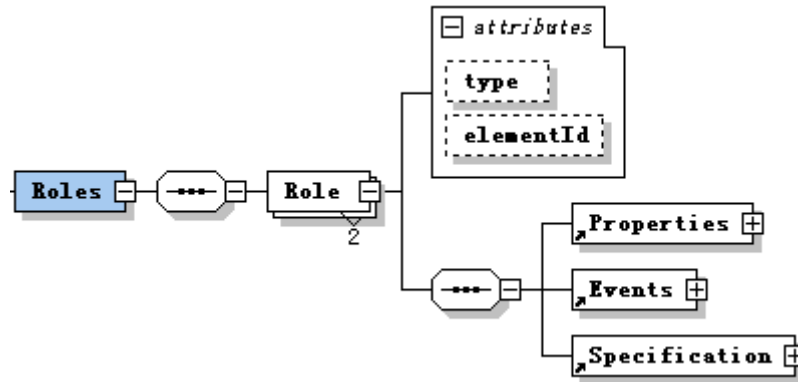


Figure 5.18 RoleElement Schema

Attributes are used to describe basic information of Role and among them, type is used to describe type of role, such as request / response, father / son, etc. with the type being specified by the meta-model developer. elementId is used to record which modelling element is assigned by the role, here this is just a reference to the modelling element. Properties are used for descriptions of some of the Role's user-defined information, such as name, multiplicity, etc. Events are used for the description of some constraints, rules, etc. on the role. The role assigned to some objects will produce a corresponding event. Specification is used to describe some specifications that role should satisfy.

(c) Properties

Express attributes of a set of relationships in Schema. It is a complexType and is not affected by which type is used in the definition of the model.

(d) Events

It describes some constraints, rules and actions of model objects in the form of events. The event processing logic is specified by script. It is a complexType in Schema and is the same as the type used in the definition.

(e) Specification

Expresses a set of some of the semantic features of a related object and is used to describe model specification information such as the function of the

model, modelling intentions, required constraints and so on. It is a complexType in Schema and is the same as the type used in the definition of the model.

(4) Diagram

A diagram is a graphical definition in the model and is used to record visual information relating to a modelling element. It plays the role of a drawing board, and provides an interactive interface with the user. The syntax of Diagram is defined by XML Schema as shown in the following Figure 5.19.

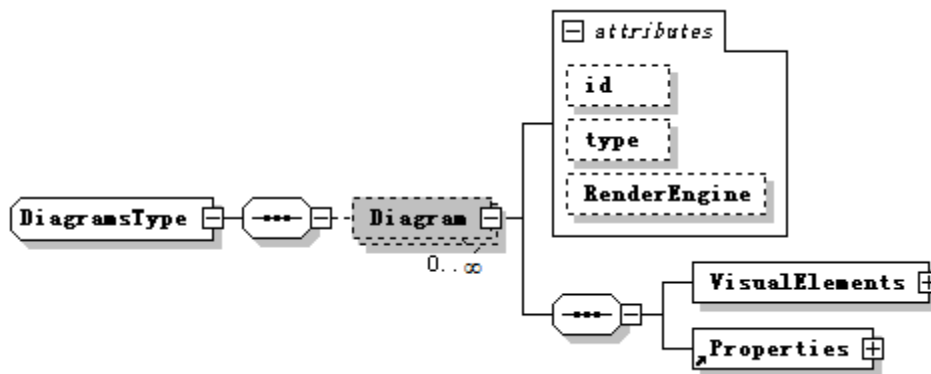


Figure 5.19 Diagram Schema

The meaning of each of the constituent elements is described as follows.

(a) Attributes

Express basic attributes of the diagram itself, among them, id is a unique identifier of the diagram; type is used to describe the type of mark and to mark it for reuse; RenderEngine is used to specify which render engine deals with the diagram. When dealing with graphical displays and interactions, different types of diagram make different requirements on the render engine. For example, there is a lot of difference between a sequence diagram and a class diagram in UML. Therefore, it is necessary to provide the meta-model developer with the approach of freely specifying the render engine.

(b) VisualElements

A visual information set of each modelling element object in the model.

(c) Properties

It used to express the attributes of a set of diagrams. It is a complexType in Schema and is the same as the type used in the definition of the model.

5.5 Formal Specification of Meta-Modeling Language XMML

5.5.1 Formal Semantics of Domain and Meta Domain

The semantics of domain-specific modeling language are divided into two kinds: structural semantics and behavioral semantics, the former relates to models' specification, description and operation, while the later focus on executing semantics of domain-specific, however, the formal definition in the paper is based on domain-specific modeling language and structural semantics of meta-modeling language. With the guide of bottom-up, the concepts of meta-modeling language of all the domain are established on the basis of defining concept of domain.

From the mathematical point of view, the domain composed by all domain models which are described by domain-specific modeling language contains the following three kinds of information:

- (1) The common concepts used for constructing domain model or a mathematical structure set Υ of common primitives;
- (2) All the possible domain model sets R_{Υ} ;
- (3) domain constraint set C acting on R_{Υ} ;

Not all the domain model in the R_{Υ} constructed by Υ is legal (well-formed), only those who meet the domain constraints c are well-formed.

The Υ is abstracted as a set of functional symbol, such as Component (x) means that for $\forall x$, x is a component, here the Υ can be seen as a mapping of functional symbol to nonnegative integer, $\nu \rightarrow \square_+ \nu$ is a functional symbol set used for constructing model, \square_+ is the corresponding set of element number, so the Component (x) can be represented as (Component,1), Component is the name the functional symbol, 1 represents Component is a function of one variable. A set labeled by the model elements is abstract as a character map Σ , a combination of the functional symbol and the character map can be seen a term, which is used to identify an element of a model, such as Component (CA) denotes a element of component type named CA. An algebra inductive method based on Σ and Υ is used to construct a term set $T_Y(\Sigma)$ called as term algebra, and the following definitions are given.

Definition 1 supposes Υ is a symbol set, Σ is a character map, the term of term algebra can be generated as follows:

- (1) $\sigma \in \Sigma$ is a term.
- (2) If $f \in \text{dom}\Upsilon$ and $t_1, t_2, \dots, t_{\Upsilon(f)}$ are terms, so $f(t_1, t_2, \dots, t_{\Upsilon(f)})$ is a term.

So a model can be seen as a domain functional symbol acting on a generated tern set τ_r identified by the model elements, from the mathematical point of view, $\tau_r \subset T_Y(\Sigma)$; the domain model set can be seen as a power set of the term algebra $T_Y(\Sigma)$, namely $R_Y = \rho(T_Y(\Sigma))$.

Based on above analysis, we know that a domain of mathematical concepts composed by the following parts: a character map Σ composed by model identification; a domain symbol set Υ , which directly corresponding to domain construction element; a domain constraint symbol set Υ_c , a extension of Υ , contains all the symbols of derivable model's well-formedness or consistency; a

consistent set C acting on all the model of a domain, C is based on Υ and Υ_c written by a formal language. Next, the formal definition of domain of mathematical concepts will be given.

Definition 2 A domain D is a quadruple $\langle \Upsilon, \Upsilon_c, \Sigma, C \rangle$, Υ and Υ_c are functional symbol set based on ν , Υ_c is an extension of Υ , Σ is a character map, C is a formal expression set $\mathcal{F}(L)$ written by a formal language and expresses constraints rules of a domain.

To map a domain structure of a mathematical concepts to a concrete formal domain structure, such as a domain based on first order logic, to realise consistency checking of a model and attributes analysis, a interpretative mapping from domain of mathematical concepts to a first order logic.

Definition 3 an explanation $\square\square$ can be seen as a mapping δ from model of mathematical domain D to model of first order logic domain D_L , denoted as $\delta: R_Y \rightarrow R_Y^L$, among them $D_L = \langle \Upsilon^L, \Upsilon_c^L, \Sigma, C^L \rangle$.

It required that the predicate symbols of first order logic symbol tables Υ^L and Υ_c^L in D_L directly come from corresponding function symbol of Υ and Υ_c , namely, Υ and Υ^L , Υ_c and Υ_c^L are identical mapping, only the meaning is transformed as first-order predicate from mathematical function; C^L is a group of first-order predicate based on Υ^L, Υ_c^L , which equals the meaning of expression $\mathcal{F}(L)$; they are all based on the same character map Σ . δ is a mapping from mathematical domain model to a first-order logic domain model, δ^{-1} is an inverse mapping of δ .

Theory 1 the mapping from the model of mathematical model to the model of first-order logic is a bijection.

Proof: To any mathematical domain model, namely, $\forall X, \delta^{-1}(\delta(X)) = X$,

thereby it can deduced that $\delta(X) = \delta(Y) \Rightarrow \delta^{-1}(\delta(X)) = \delta^{-1}(\delta(Y)) \Rightarrow X = Y$, so δ is a mapping of one to one (injection). For any model of first-order logic domain, namely, $\forall X_L \in D_L, \exists X, \delta(X) = X_L$, so δ is a surjection. Thereby, δ is a bijection from mathematical domain to first order domain.

It is thus clear that this kind of mapping is a mapping of structure preservation, which can keep its abstract syntax and structural semantic identical. The same reason can prove that $\delta^{-1} : R_Y^L \rightarrow R_Y$ is a bijection preservation of inverse structure between the models of first-order logic domain D_L to the models of mathematical domain D . So the purpose of the checking and analysis of the model itself can be realised by checking and analysis of first-order logic domain D_L . The formal definition of domain-specific modeling language is given by us based on the analysis of the domain of structure semantics (description semantics) of domain-specific modeling language.

Definition 4 a domain-specific modeling language L is a domain described by itself and a two-tuples explained by itself, namely, $L = \langle D, \square \rangle$.

A composing domain of all the legal domain models constructed by the domain-specific modeling language, while all the different domains constructs another domain ——a domain of domain, we called it meta-domain. As the domain depicted by the modeling language, the meta-domain is depicted by the meta-language of modeling language, meta-modeling language. A modeling process is used to construct new domain application model, while the meta-modeling language is a process to define new domain by constructing domain meta-model, here the so-called domain meta-model is a model used for defining domain constructed by meta-modeling language, that is to say the domain is depicted by the meta-model.

The meta-domain of mathematical concepts composed by the following parts: a character map Σ^{meta} composed by a meta-model identification; the

symbol set Υ^{meta} of public primitives or the common concepts used for constructing all the domains, which corresponding to constructing element of meta-domain, such as Entity(x) represents to $\forall x$, x is an entity, here Υ^{meta} can be seen as a mapping from functional symbol to non-negative integer: $\nu \rightarrow \square_+$, ν is a functional symbol set used for constructing meta-model, \square_+ is the set of function's corresponding element number, so the Entity (x) can be denoted as (Entity,1), the Entity is the name of function symbol, 1 represents entity is a function of one variable, the combination of function symbol and character can be seen as a term, which is used for a element of meta-model, such as Entity (Component) means the element of entity type, named Component; a meta-domain constraint symbol set Υ_C^{meta} , is a extension of Υ^{meta} , which contains all the symbols of well-formedness of inferable meta-domain. A set C^{meta} acts on meta-domain and contains consistency constraints of all the domains, C^{meta} is written by the formal language based on Υ^{meta} and Υ_C^{meta} . Next, the formal definition of meta-domain of mathematical concepts will be given.

Definition 5 a meta-domain D^{meta} is quadruple, $\langle \Upsilon^{meta}, \Upsilon_C^{meta}, \Sigma^{meta}, C^{meta} \rangle$, Υ^{meta} and Υ_C^{meta} are the function symbol sets based on ν , Υ_C^{meta} is the extension of Υ^{meta} , Σ^{meta} is the character map, C^{meta} is a formal expression set written by the a formal language, which express constraints rules of meta-domain. The meta-domain D^{meta} expresses the common structure of all the meta-models contained by it.

To map the meta-domain structure of the mathematical concepts to a concrete formal meta-domain structure, such as a domain based on first-order logic, to realise consistency checking and attributes analysis of meta-model, a meta-explanation mapping from the meta-domain of mathematical concepts to

a meta-domain based on first-order logic is established by us.

Definition 6 a meta-explanation $\square\square_{meta}$ can be seen as a mapping δ_{meta} between a meta-model $R_{\Upsilon^{meta}}$ of meta-domain D^{meta} of mathematical concepts and a domain $R_{\Upsilon^{meta}}^L$ of meta-domain D_L^{meta} based on first-order logic, which is denoted as $\delta_{meta} : R_{\Upsilon^{meta}} \rightarrow R_{\Upsilon^{meta}}^L$, among them

$$D_L^{meta} = \langle \Upsilon^{meta^L}, \Upsilon_C^{meta^L}, \Sigma^{meta}, C^{meta^L} \rangle.$$

Theory 2 the mapping δ_{meta} from mathematical domain to first-order logic is a bijection.

The proof is the same to theory 1. This kind of mapping is structure preservation mapping, and it can keep its abstract syntax and structure semantics identical. If δ_{meta}^{-1} is a inverse mapping of δ_{meta} , the same reason can proof that $\delta_{meta}^{-1} : R_{\Upsilon^{meta}}^L \rightarrow R_{\Upsilon^{meta}}$ also is inverse structure preservation bijection between a domain $R_{\Upsilon^{meta}}^L$ of meta-domain D_L^{meta} based on first-order logic to a meta-model $R_{\Upsilon^{meta}}$ of mathematical meta-domain D^{meta} . So the purpose of checking and analysis of meta-model itself can be realised by checking and analysis of a domain of first-order logic. The formal definition of domain-specific modeling language is given by us based on the analysis of the domain of structure semantics (description semantics) of domain-specific modeling language.

Definition 7 a domain-specific meta-modeling language L_{meta} is a two-tuples of described by it and explained by it, namely, $L_{meta} = \langle D_{meta}, \square\square_{meta} \rangle$.

As the domain is depicted by the meta-model of model, meta-domain is depicted by the meta-model of meta-model, namely meta-meta model. The meta-meta model not only depicts all the abstract syntax and structural semantics of meta-domain, but also the common constraints C^{meta} kept by the

all the meta-model of the meta-domain, so only those meta-models constructed by the domain-specific meta-modeling language L_{meta} meeting constraint C^{meta} of meta-domain are well-formed meta-model.

5.5.2 Formal Specification of XMML Meta-Modeling Language

Based on the formal definition of above domain-specific meta-modeling language L_{meta} , according to the structure preservation mechanism, on the basis of improving the existing problems of definition integrity and consistency of XMML meta-modeling language, a formal representation of XMML based on first-order logic domain which is easy to check is established. It focuses on primary meta-modeling element of XMML meta-modeling language as well as its formal depiction of its constraint relationship. So the XMML meta-modeling language L_{XMML} can be seen as a predicate symbol set S_{XMML} decrypting meta-modeling element, an extended predicate symbol set S_{XMML}^C used for property derivation as well as a triad composed by predicate expression set F_{XMML} of consistency constraints acting on all the meta-model.

Definition 8 the XMML meta-modeling language L_{XMML} can be seen as a triad composed by S_{XMML} , S_{XMML}^C and F_{XMML} .

The meta-modeling used for constructing meta-model in S_{XMML} is categorised into types: one is meta-modeling element of entity type, which are represented as node in the diagram form of meta-model; the other is meta-modeling element of relationship type, which are represented as side of meta-model diagram form, when each meta-modeling element is used for constructing meta-model, it is with attributes, such as ID, Type and Specification, .etc. to make the system identify different meta-type, as well as

different entities of the same meta-modeling element, or make the user write the comments. The attributes information contained by meta-modeling language refers to concrete syntax definition of XMML.

Next, a detail formal description of each meta-modeling element as well as constraints expression related to each of them will be given, and the non-formal diagram form supported by its meta-modeling, based on this, to classify the predicate symbol set of XMML and predicate constraint to form the set of S_{XMML} , S_{XMML}^C and F_{XMML} .

(1) Model

Model can be represented as a unary predicate $Model(x)$, which means the metatype of element x is Model, $Model(x) \in S_{XMML}$. For example, a meta-model constructed by XMML has a element ArchA of model type and element EntityA of entity type, so $Model(ArchA)$ is true, $Model(EntityA)$ is false.

The model can contain 4kinds meta-modeling element of entity type, such as Entity, refEntity, Relationship, Constraint, and the hierarchy relation of models can be established by self-contained, its diagram form of meta-modeling as follows:

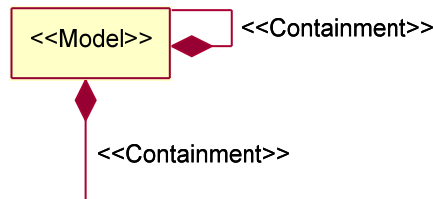


Figure 5. 20 Diagram Form of Model Meta-Type

(2) Entity

The entity can be represented as a unary predicate $Entity(x)$, which means that the meta-type of element x is Entity, $Entity(x) \in S_{XMML}$. The entity can be

contained in model by Containment relation, the Attachment is composed by close containing among entities, the entity Contain relation (EntiCont) is composed by loose containing among entities, the property contain relation (ProvCont) can be used to define the property of meta-typed entity, the role assigned relation (AssginRela) can be used to establish the relation between other meta-modeling element of entity type, but the entity can't contain model, relationship refEntity or Constraints.

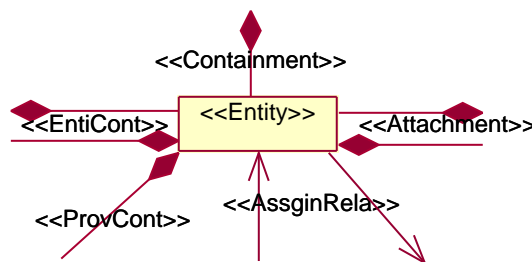


Figure 5. 21 Diagram Form of Meta-Typed Entity

(3) Relationship

The Relationship can be represented as a unary predicate $Relationship(x)$, which means the meta-type of element x is relationship, $Relationship(x) \in S_{XMML}$. The relationship can be contained in Model by Contained Relation, Containment. The property of Relation of meta-type can be defined by property containment relation (ProvCont), to establish explicit relationship of meta-modeling element of entity type by combination of role-assigned relation (AssginRela), but the Relationship can't be self-contained, or contain Model, Entity, RefEntity and Constraint.

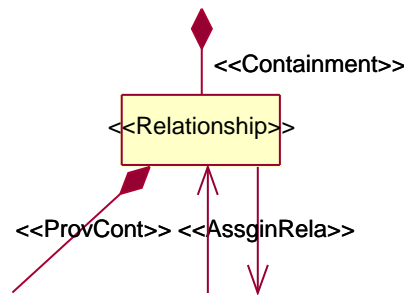


Figure 5. 22 Diagram Form of Relationship of Meta-Type

(4) RefEntity

The reference entity (RefEntity) can be represented as a unary predicate $RefEntity(x)$, which means the meta-type of element x is RefEntity, $RefEntity(x) \in S_{XMML}$. The RefEntity can be contained in Model by Containment, and can point to the referenced other entity by Reference relation to provide modeler with a mechanism that is to refer to other model entity in a model. In addition, the type of \ referenced meta-modeling element referred by reference relation can't be other types excepts entity

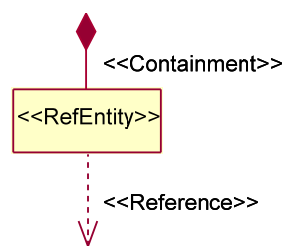


Figure 5. 23 Diagram Form of Referenced Entity of Meta-Type

(5) Property

The data type supported by XMML are Boolean, string and enum, so according to different data types, the Property can be respectively represented as unary predicate $ProvBool(x)$, $ProvString(x)$ or $ProvEnum(x)$, which

respectively means attribute name is x , data type is Bool, String or Enum, $ProvBool(x) \in S_{XMML}$, $ProvEnum(x) \in S_{XMML}$, $ProvString(x) \in S_{XMML}$. The three typed attributes can be contained in model entity and relationship by property contained relationship, ProvCont.

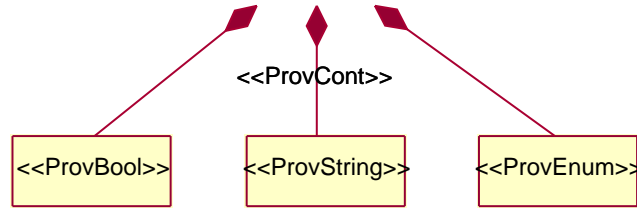


Figure 5. 24 Diagram Form of Attribute Meta-Type

We have three constraints on property's data type.

- (a) Completeness of data type: the data type supported by XMML only is Boolean, string and enum.
- (b) The uniqueness of data type: the data of each meta-property is one of the three types.
- (c) The existence of enum: each meta-property of enum type must define a list of enum value.

If the EnumList(x) is used by us to present meta-property of enum type x in the enum list, the above constraints can be formalised by using first-order predicate expression as follows:

The constraints of data type completeness:

$$\forall x. ProvBool(x) \vee ProvString(x) \vee ProvEnum(x)$$

The constraints of data type uniqueness: it is a conjunction of following three expressions.

$$\forall x. ProvBool(x) \rightarrow \neg ProvString(x)$$

$$\forall x. ProvBool(x) \rightarrow \neg ProvEnum(x)$$

$$\forall x. ProvString(x) \rightarrow \neg ProvEnum(x)$$

The existence of enum values: $\forall x. ProvEnum(x) \rightarrow EnumList(x)$

The above expressions are elements of the set F_{XMML} .

(6) Constraint

The Constraint can be represented as a unary predicate $Constraint(x)$, which means the meta-type of element x is Constraint, $Constraint(x) \in S_{XMML}$. It provides a property of constraint definition, FormulaDomain, which can be used for writing user-defined domain constraint formulas based on meta-model predicate symbol set and extended predicated symbol set when meta-modeling. Due to its definition of constraints of the whole meta-model in the view of a whole, so it is only contained in Model by Containment, while it can't be contained in other meta-modeling elements.

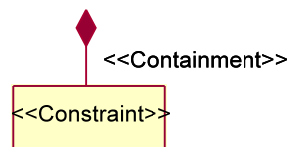


Figure 5. 25 Diagram Form of Constraints Meta-Type

(7) Containment

The Containment can be represented as binary predicate $Containment(x, y)$, which means element x is contained in element y , $Containment(x, y) \in S_{XMML}$.

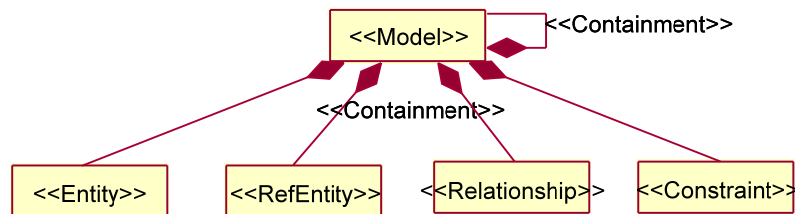


Figure 5. 26 Diagram Form of Containment Meta-Type

We have two constraints on the meta-type connected by Containment.

(a) The side of Containment must terminate at meta-type of model .

- (b) The side of Containment must start form model, entity, refEntiy, relationship or constraints.
- (c) The self-contained relationship only happens to model.

The above constraints can be formalised by first-order predicate formula as follows:

The meta-type constraint of ending of Containment:

$$\forall x, y. Containment(x, y) \rightarrow Model(y)$$

The meta-type constraint of starting of Containment:

$$\forall x, y. Containment(x, y) \rightarrow Model(x) \vee Entity(x) \vee RefEntity(x) \vee Relationship(x) \vee Constraint(x)$$

The constraint of self-contained relationship:

$$\forall x. Containment(x, x) \rightarrow Model(x)$$

The above formals are element s of the set F_{XMML} .

(8) ProvCont

The ProvCont can be represented as binary predicate $ProvCont(x, y)$,

which means element x is contained in element y, $ProvCont(x, y) \in S_{XMML}$.

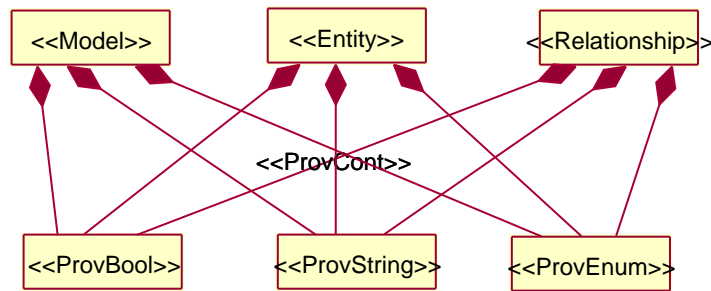


Figure 5. 27 Diagram Form of ProvCont Meta-Type

We have two constraints on the meta-type connected by ProvCont.

- (a) The side of ProvCont must starts from the meta-type of Bool, String or Enum
- (b) The side of ProvCont must terminate at model entity or relationship.

The above constraints can be formalised by first-order predicate formula as

follows:

The constraint of starting of ProvCont:

$$\forall x, y. ProvCont(x, y) \rightarrow ProvBool(x) \vee ProvString(x) \vee ProvEnum(x)$$

The constraint of ending of ProvCont:

$$\forall x, y. ProvCont(x, y) \rightarrow Model(y) \vee Entity(y) \vee Relationship(y)$$

The above formals are elements of the set F_{XMML} .

(9) Attachment

The attachment can be represented as binary predicate $Attachment(x, y)$,

which means element x is attached in element y, $Attachment(x, y) \in S_{XMML}$.

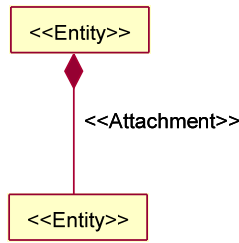


Figure 5. 28 Diagram Form of Attachment Meta-Type



Figure 5. 29 Diagram Form of Self-Attached

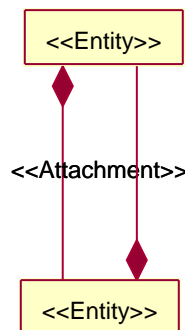


Figure 5. 30 Diagram Form of Forming Attached Loop

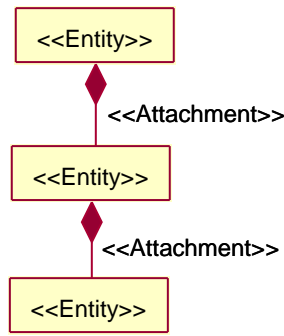


Figure 5. 31 Diagram Form of Attachment Path

We have two constraints on attachment:

- (a) The side of attachment must start from meta-type of entity, and it must terminate at meta-type of entity.
- (b) The attachment is the close contained relation used in different entity type, so the same entity type can't be attached to itself, shown by Figure 5-29.
- (c) If the meta-type of host is deleted, the meta-type of attachment must be deleted together.
- (d) The attachment loop can't be formed between two meta-types of entity, shown by Figure 5-30.
- (e) Due to the hierarchy can be formed by model self-contained relationship, so the level of attachment among entities can't larger than 1, namely it only support the attachment of two entities, the attachment path of 3 different entities can't be formed, shown by Figure 5-31.

Suppose entity element x gets to attachment path of entity element z by entity element y , $AttaPath(x,y,z) \in S_{XMML}^C$, so the above constraints can be formalised by first-order predicate formula as follows:

Meta-type constraints of two ends of attachment:

$$\forall x, y. Attachment(x, y) \rightarrow Entity(x) \wedge Entity(y)$$

Attachment can't be self-contained: $\forall x. \neg Attachment(x, x)$

The constraints between attached elements and hosting elements:

$$\forall x, y. Attachment(x, y) \wedge (\neg Entity(y)) \rightarrow \neg Entity(x)$$

Acyclicity of the attachment among entities:

$$\forall x, y. Attachment(x, y) \wedge (x \neq y) \rightarrow \neg Attachment(y, x)$$

Non-existence of attachment path:

$$\forall x, y, z. Attachment(x, y) \wedge Attachment(y, z) \wedge (x \neq y) \wedge (y \neq z) \rightarrow AttaPath(x, y, z)$$

$$\forall x, y, z. \neg AttaPath(x, y, z)$$

The above formalisms are elements of the set F_{XMML}

(10) EntiCont

The entity-contained (EntiCont) can be represented as binary predicate $EntiCont(x, y)$, which means element x is contained in element y , $EntiCont(x, y) \in S_{XMML}$. There are main two differences between it and attachment, that is to allow self-contained and the deleting of containing entity doesn't affect the existence of contained entity.

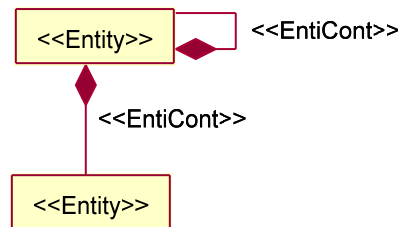


Figure 5. 32 Diagram Form of Meta-Type EntiCont

We have three constraints on EntiCont:

- (a) The side of EntiCont must start from meta-type of entity, and it must terminate at meta-type of entity.
- (b) The EntiCont loop can't be formed between two meta-types of entity.
- (c) Due to the hierarchy can be formed by model self-contained relationship, so the level of EntiCont among entities can't be larger than 1, namely it only support the EntiCont of two entities, the EntiCont path of 3 different entities can't be formed.

Suppose entity element x gets to *EntiCont* path of entity element z by entity element y , $EntiContPath(x, y, z) \in S_{XMML}^C$, so the above constraints can be formalised by first-order predicate formula as follows:

Meta-type constraints of two ends of *EntiCont*:

$$\forall x, y. EntiCont(x, y) \rightarrow Entity(x) \wedge Entity(y)$$

Acyclicity of the attachment among *EntiCont*:

$$\forall x, y. EntiCont(x, y) \wedge (x \neq y) \rightarrow \neg EntiCont(y, x)$$

Non-existence of *EntiCont* path:

$$\forall x, y, z. EntiCont(x, y) \wedge EntiCont(y, z) \wedge (x \neq y) \wedge (y \neq z) \rightarrow EntiContPath(x, y, z)$$

$$\forall x, y, z. \neg EntiContPath(x, y, z)$$

The above formals are elements of the set F_{XMML} .

(11) Reference

The reference can be represented as binary predicate $Reference(x, y)$, which means element x of reference entity type points to referred element y of entity type by reference relation, $Reference(x, y) \in S_{XMML}$.

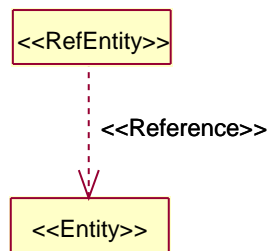


Figure 5. 33 Diagram Form of Reference Meta-Type

We have three constraints on meta-type connected by reference:

- The side of reference must start from meta-type of *RefEntity*
- The side of reference must terminate at meta-type of *RefEntity*.
- The same *RefEntity* only point to a meta-type of entity type.
- The referred entity can't be the attached entity of attachments.

The above constraints can be formalised by first-order predicate formula as follows:

Meta-type constraints of two ends of reference:

$$\forall x, y. \text{Reference}(x, y) \rightarrow \text{RefEntity}(x) \wedge \text{Entity}(y)$$

The RefEntity corresponding to referred entity:

$$\forall x, y, z. \text{RefEntity}(x) \wedge \text{Reference}(x, y) \wedge \text{Reference}(x, z) \rightarrow (y = z)$$

Suppose RefEntity x points to referred entity y attached to entity z and the character is $\text{RefAttaEntity}(x, y, z)$, $\text{RefAttaEntity}(x, y, z) \in S_{XMML}^C$, so the attachments constraints of referred entity as follows:

$$\begin{aligned} &\forall x, y, z. \text{RefEntity}(x) \wedge \text{Reference}(x, y) \wedge \text{Attachment}(y, z) \rightarrow \text{RefAttaEntity}(x, y, z) \\ &\forall x, y, z. \neg \text{RefAttaEntity}(x, y, z) \end{aligned}$$

The above formals are elements of the set F_{XMML} .

(12) Role-assign relation (AssginRela)

AssginRela is used to cooperate with Relationship, and explicitly establish the binary relation among meta-modeling elements of entity type, according to the different directions of the two connected meta-modeling element, we give them different roles, it is a rule that the starting end of the connection is source role, and the target end is target role, corresponding the AssginRela is divided into two relationship types: Source Role Assign Relationship (SRoleAssginRela) and Target Role Assign Relationship (TRoleAssginRela). So if the relationship between two meta-modeling elements of entity type, the source role must be connected with relationship entities by SRoleAssginRela. SRoleAssginRela can be represented as binary predicate, $\text{SRoleAssginRela}(x, y)$, which means the meta-modeling element x of source role is connected with element y of relationship type by SRoleAssginRela, $\text{SRoleAssginRela}(x, y) \in S_{XMML}$. TRoleAssginRela can be represented as binary predicate, $\text{TRoleAssginRela}(x, y)$, which means the relationship element x is connected

with element y of target role by $SRoleAssginRela$,
 $TRoleAssginRela(x, y) \in S_{XMML}$

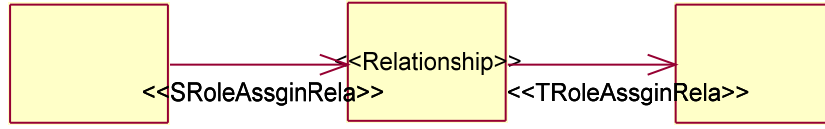


Figure 5. 34 Diagram Form of AssginRela Meta-Type

We have three constraints on the meta-types connected by $SRoleAssginRela$:

- (a) The side of $SRoleAssginRela$ must start from meta-type of entity or meta-type of $RefEntity$
- (b) The side of $SRoleAssginRela$ must terminate at the meta-type of $Relationship$
- (c) Bidirectional relationship between two types of element can't use the same relationship entity.

Suppose a representation of connection that is the source role x is connected to target role z by relationship entity y is $RoleRela(x, y, z)$,
 $RoleRela(x, y, z) \in S_{XMML}^C$, so the above constraints can be formalised by first-order predicate formula as follows:

Constraints of two ends of $SRoleAssginRela$:

$$\forall x, y. SRoleAssginRela(x, y) \rightarrow (RefEntity(x) \vee Entity(x)) \wedge Relationship(y)$$

Constraints of bidirectional relationship:

$$\forall x, y, z. SRoleAssginRela(x, y) \wedge TRoleAssginRela(y, z) \rightarrow RoleRela(x, y, z)$$

$$\forall x, y, z, u. RoleRela(x, y, z) \wedge RoleRela(z, u, x) \rightarrow (y \neq u)$$

We have three constraints on the meta-types connected by $TRoleAssginRela$:

- (a) The side of $TRoleAssginRela$ must start from meta-type of $Relationship$
- (b) The side of $TRoleAssginRela$ must terminate meta-type of entity or meta-type of $RefEntity$

The above constraints by using first-order logic expression can be formalised

as follows:

$$\forall x, y. TRoleAssginRela(x, y) \rightarrow Relationship(x) \wedge (RefEntity(y) \vee Entity(y))$$

5.6 XMML Visual Modelling

5.6.1 Visual Definition Mechanism for XMML

The visual description of XMML provides a graphical means for the modeller to describe the model. As models expressed graphically are much simpler and also easier to understand and communicate, today's popular modelling languages most provide a visual modelling mechanism. For example, in UML, a block diagram is used to express class and arrows are used to express relationships among objects.

Visual modelling is very important with domain-specific modelling languages. In different application domains, it is necessary for various domain concepts to differ in appearance. The visual information for the modelling element should accord with domain concepts in order to directly express semantic information of the modelling language element through its appearance in use. The visual representation corresponding to a modelling element is called a diagram of the modelling element. Considering how people actually make a visual identification of an object, a well-defined diagram can express more information than text to audiences. For example, warning signs on highways can more quickly and effectively express necessary information to people than ordinary text. Therefore, when designing a diagram for a modelling element of a domain-specific modelling language, there is no point in emphasising unity of modelling diagrams like the unified modelling language, UML. On the opposite, its modelling diagram must be able to realise an individual definition of the domain.

In the modelling environments, besides the modelling diagram having appearance performance, another important characteristic is its ability to generate interaction with the modeller. The modeller can change the state of a modelling element or its attributes and make a corresponding response to its visual clues by mouse clicks or key-presses on a keyboard. Examples would include, the state exchange of a switch, or performance of mechanical equipment, etc in the design of circuit. Therefore, a modelling diagram is based on an event-response, and the actual response logic and action is decided by the modelling element itself according to events. It is necessary that the meta-modelling language should be able to model event-responses to the modelling diagram.

In XMML, we show entities and event-response actions by introducing a form of definition mechanism between the modelling element and its diagram and these are independent of each other. In this way we avoid cases of coupling appearing between a modelling element and its diagram and improve reusability of the diagram definition. So, the developer of the meta-model can specify many diagram types for modelling elements, as well as specify the same diagram type for different modelling elements.

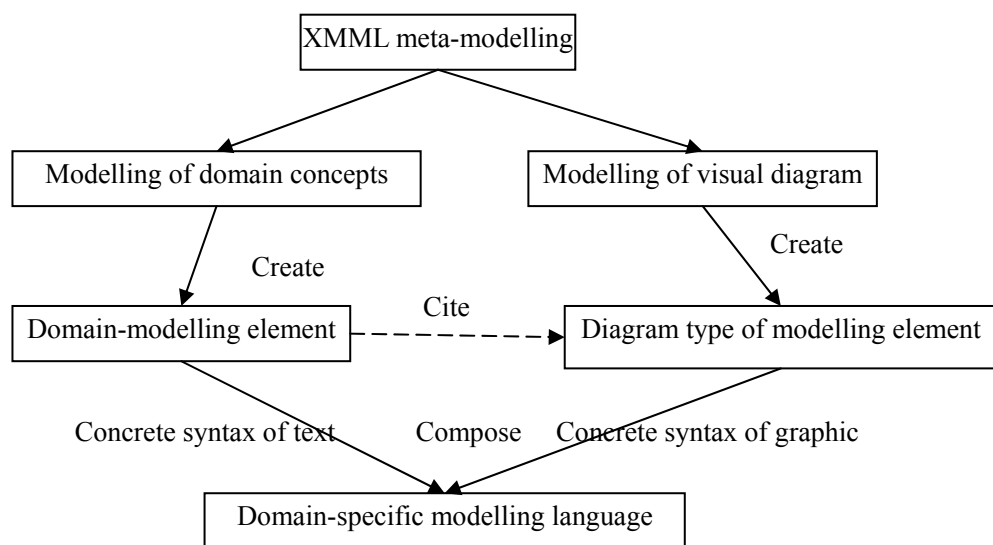


Figure 5. 35 Relationship between Text Concrete Syntax and that of Graphic

Shown in Figure 5.35, the meta-modelling process using XMML can be classified as two different processes: One is the modelling of domain concepts and creating domain modelling elements together with text syntax. The second is modelling to visual diagrams and creating Diagram of modelling elements modelling elements together with graphic syntax. Then relationships are built between domain modelling elements and modelling diagrams by reference to relationships. These combine to enable the visual domain-specific modelling language to be used in domain application modelling.

5.6.2 Primitive Description Scheme of XMML Modelling Elements

The graphic syntax described by the modelling elements of XMML, pay attention to describing the visual appearance and event-responses that the modelling diagram should have. It is not related to the semantics and attributes of modelling elements related to the diagram type. Therefore, it is can be individually used in visual diagram modelling. Diagram syntax is defined by an XML Schema as shown in Figure 5.22.

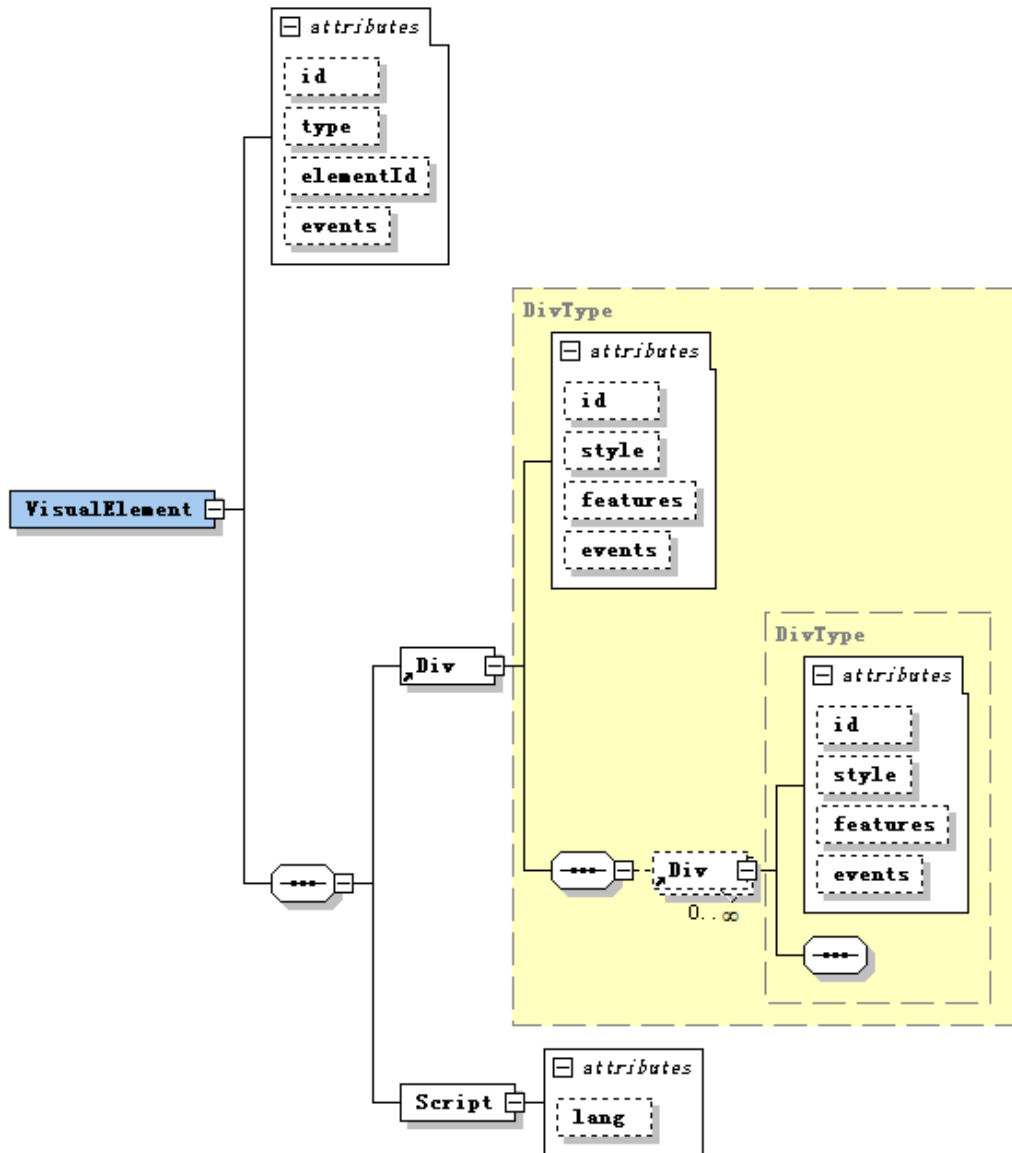


Figure 5. 36 Structure Described by Diagram

VisualElementType is used in describing a complexType of XMML Schema in a modelling diagram. Of the two sub-elements included, Div and Script, Div is used to define the nodes of the diagram, while Script is used to define the diagram as well as the event-handler script for each node. The meaning of each element is described as follows:

(1) Attributes

Describe a basic attribute of the diagram itself. Among them, id is the unique

identifier of the diagram; Type is used to describe type of diagram. elementId is used to illustrate modelling elements represented by diagrams, and to connect visual diagram objects to the element modelled by the attributes. Events are used to describe some event-handler set, which is globally related to the diagram.

(2) Div

Used to describe appearance of each part of the diagram, the Div is similar to a container, which can nest other sub-elements of div to form a DOM tree structure. A Div in the DOM tree is called a primitive node, and each primitive node has three attributes:

Id is a unique identifier of primitive node, to query and operate every Div node by id in script. style is used to describe relationship attributes of visual information. The information shown visually can be defined through style e.g. by the size of a node, or the location, colour, background, or direction of text alignment. The actual usage of style attributes is similar to HTML / CSS and the following grammatical form is adopted: style="Style name: style value; style name: style value; ..." This can make those users who are familiar with HTML / CSS find it easy to reuse CSS mastered by them to design knowledge to describe the visual information of a primitive node. Examples of div and style as follows.

```
<div id="divBox" style="height:80px; width:120px; border:1px solid black; color:red; background: RES (img / component.emf); text-align: center" / >
```

Parsing style attributes controls the graphic render engine. Therefore, the definition ability of primitive appearance is closely related to the graphic render engine, rather than language, and there is no need to change the structure of the primitive description language for upgrading or changing support for style attributes.

Features are used to describe and declare the behaviour characteristics of

primitive nodes, some of which are declared by feature label. Such as: *features* = “*linkable:true; resizable:false; moveable:true; editable:true; hostable: false; selectable:true*”, In the characteristic declaration, *linkable* is used to mark whether the primitive node is connected or not, *resizable* is used to mark whether the primitive node can change size, *moveable* is used to mark whether the primitive node can be individually dragged, *editable* is used to mark whether the primitive element can get input focus as well as whether it’s content be edited or not. *Hostable* is used to mark whether the primitive node can be served as a container to accept other primitives. *Selectable* is used to express whether the primitive can be selected and displayed by highlight background.

The available feature label of *features* attributes do not place limits on these. Like *style*, attribute content can be extended, with the meaning of the extended feature label being parsed by the graphic render engine to achieve extend characteristic behaviour of primitive nodes under conditions of unchanged language structure.

The attributes of events are used to describe events responded by primitive nodes, as well as Script function called when the event is triggered. Such as:

```
events="onclick:fnOnClick(); onload:fnOnload();  
        onkeypress:fnOnkeypress(); ...”
```

The event is mainly initiated by user interaction in the model designer, and it is can be classified in three categories: mouse event, keyboard event and user-defined events. It will call the script function specified in the declaration of events when an event is triggered. The primitive description structure is a DOM tree structure; the transmission mechanism of events in DOM tree is ebullient. Namely, one event generated in sub-node can be automatically spread to all its father nodes, unless a certain level of the parent node has been explicitly suspended at the propagation process of the event. If corresponding events are

specified in a father primitive node, then the script will be called. After finishing script execution, the event is passed to the father node of the layer above, up to the outermost layer Diagram. At the time of event generation, a global event object will be automatically generated at the same time as event generation to record context information of the event. Examples include, sourceObject generates event, coordinate mouse data at time of event, mouse event or key-press information of keyboard event, type of event, etc. The global object can be directly accessed in the event-handler script to get the required information.

(3) Script

Used to define all event-handler scripts declared in primitives. It has lang attributes to explain what language is used to write the script. In script, mainly composed by some script functions, in the modelling environment, all the primitive event-handler scripts are unified, managed and called by the event script manager. When a script definition of a primitive is loaded it will trigger a onload event defined on VisualElement. Here, the event processing script will execute some primitive initialised action according to primitives.

5.7 Summary

In this chapter, a domain-specific visual meta-modelling language XMML is given. It is designed according to the concepts and methods of DSM and used to provide description language support for domain meta-modelling language and domain application modelling in a DSM implementation framework. It can support description and construction of the domain meta-model and the domain application model at the same time.

DSMLs are model description languages that differ from both general programming languages and general modelling languages. When designing XMML, design goals were mainly determined in terms of descriptive ability,

usability and verifiability.

In the abstract syntax model of XMML the, model, entity, relationship, model diagram, visual element and event are all basic elements of its abstract syntax. As abstract syntax elements themselves, they do not have any domain-specific related semantics. They are concerned with the description and definition of model basic concepts and relationships among concepts. Therefore, these abstract syntax basic elements will not only make XMML able to be used for description of the domain meta-model but also enable it to describe the domain application system model. In this way, the two different abstract models can use the same abstract syntax elements to describe their model forming element concepts and relationships.

Corresponding to abstract syntax, the concrete syntax of XMML can be defined by XML schema. As XMML is a visual modelling language, so the concrete syntax of XMML is composed of both text syntax and graphic syntax. The model formed by text syntax and graphic syntax description is expressed on the basis of XML, which is a very popular structural description language that is widely used in various aspects of software. The concrete syntax design is based on XML which ensures that XMML has good machine readability, interoperability and extensibility. At the same time, the concrete syntax defined by XML schema is more easily understood by other application programs, which can parse its architecture and better provide tool support for syntax parsing and verifying of XMML.

The software system model expressed by the text concrete syntax is an abstract representation of the true system, but it is more relative to machine language. Meanwhile, the software system model expressed by graphic syntax is a human-readable abstract representation. Visual modelling capacity is very important for domain-specific modelling languages, in different application domains. This is not least because, the representation of various domain concepts need to be different in appearance, and visual information for the

modelling element should accord with domain concepts and visually convey semantic information of the modelling language elements to users. In XMML, by introducing primitive modelling element to realise encapsulation of appearance of entity modelling element and event response behavior to form the type definition mechanism. Namely, the modelling element is independent of its primitive and in this way not only avoids coupling between the modelling element and its primitive but also increases the reusability of the primitive definition.

Some relationship types among elements implied in XMML modelling elements are obtained through formal definition and analysis of XMML modelling elements. These include: Possessed Type, Refined Type, Referred Type, Role-Assigned type, Attached Type, Contained Type. They provide a basis for formal analysis by XMML language reflection mechanisms and for descriptions of meta-modelling infrastructure.

Chapter 6

Meta-Modelling Infrastructure Based on XMML

One main target of designing the domain-specific visual meta-modelling language, XMML, is to provide basic language support for domain-specific modelling language (DSMLs) Designing, namely, supporting meta-modelling of domain-specific modelling. The architecture of XMML is given in the previous chapter. Meanwhile, this chapter will discuss the implementation of meta-modelling based on XMML on the basis of an analysis of the construction of the XMML language. Corresponding with the characteristics of XMML and the basic requirements of domain-specific meta-modelling, a meta-modelling infrastructure based on XMML is put forward. This provides the necessary modelling support for XMML applications as well as the construction of supporting tools and the development of domain-specific languages.

6.1 Overview of Meta-Modelling

6.1.1 Meta-model and Meta-Modelling

With the research and application of Model Driven development methods, the application model has become the core product of development process of application system software. Meanwhile, modelling languages applied in building model applications together with modelling tools have gradually become the basis for ensuring successful Model Driven development. Especially with domain-specific modelling methods, application software usually involves multiple domains, and modelling different domain application systems often requires different modelling languages together with their

matching modelling tools. Many practical applications have shown that development efficiency using domain-specific modelling languages is about ten times higher than that using UML [66]. However, it is a very difficult task to individually design new modelling languages and modelling tools to meet the requirements of each new domain to be modelled. Therefore, a technology is needed to reduce the costs incurred in developing such modelling tools, and meta-modelling is one approach that can solve this problem. Its core concept is that meta-model developers build a meta-model of the language according to the characteristics of the target modelling language. They then parse or generate a meta-model obtained from corresponding tools to produce the target modelling language together with the modelling tools that support the modelling language.

“Meta” is a prefix from Greek, when the prefix is added to a concept word, and it means it is a transcendence or abstract of the concepts. The simplest explanation of meta-model is that it is a model that describes a model, while meta-modelling describes the activities of creating a meta-model and related artefacts. However, things specified by a meta-model in different domains have different specific meanings. Here, we discuss meta-models and meta-modelling within the field of research of domain-specific modelling languages, so we take a meta-model as a model used to describe some modelling language. At present, there is no standard definition of meta-model and meta-modelling. It is generally agreed that a meta-model can accurately describe what is required for building a semantic model and its rules. It emphasises that the meta-model describes a modelling language at an abstract level higher than that of the model language itself. The meta-modelling exists for a particular purpose just like the modelling, and is a description of something in the real world.

6.1.2 Framework of Meta-Modelling

So far, there are two different implementation method frameworks for meta-modelling activities and their supporting frameworks. As shown in Figures

6.1 and Figure 6.2 [59] these are meta-modelling based on generic modelling tools and meta-modelling based on a modelling tools generator.

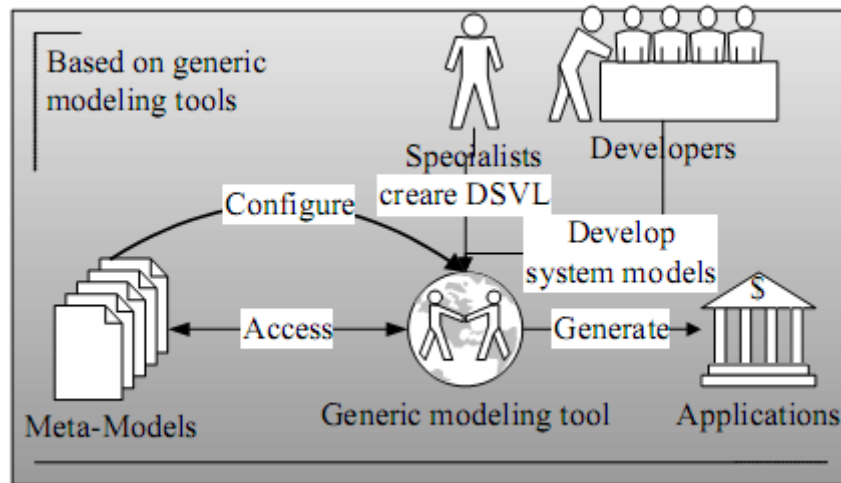


Figure 6. 1 Meta-modelling Based on General Modelling Tool

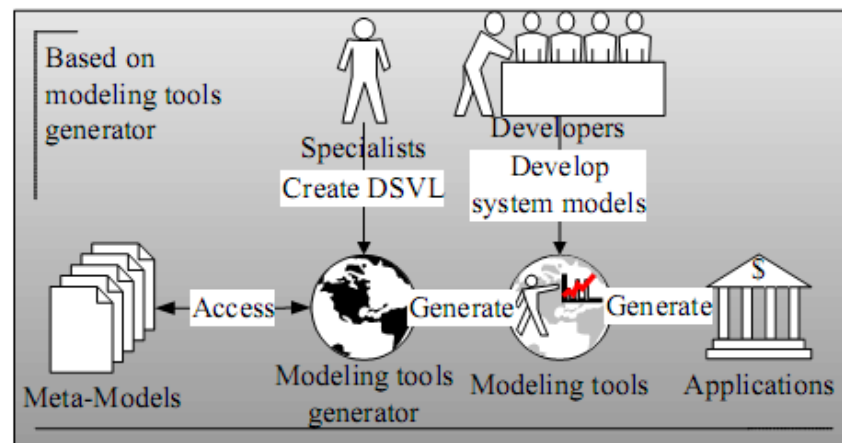


Figure 6. 2 Meta-modelling Based on Modelling Tool Generator

Meta-modelling based on generic modelling tools has generic modelling tools as the core of the meta-modelling implementation framework. First, domain experts can build a meta-model with generic modelling tools to characterise a modelling language. Once built, the meta-model is used to a configure general modelling tools to make it support the modelling language characterised by the meta-model. That is, the general modelling tool can become the specific modelling tool of the modelling language characterised by the meta-model, by configuring the meta-model. The generic modelling tool is

also called the generic modelling environment, it is used to build the meta-model (using meta-meta model to configure general modelling tool) as well as building the model (using the meta-model to configure general modelling tools) [57].

Meta-modelling based on the modelling tool generator does not include the general modelling tool. The first step of meta-modelling is to build the meta-model using the modelling tool generator to characterise the modelling language. However, the configuration documents of the general modelling tool are not generated. The modelling tools which support the modelling language are generated directly.

The two meta-modelling frameworks each have their own characteristics. The first implementation method framework has generic modelling tools at the core. Once the configuration documents have been loaded, a general modelling tool can become a modelling tool which supports the corresponding meta-model (describes the modelling language). There are many meta-modelling tools that adopt the framework, including well known names such as MetaEdit+, GME, DOME, etc. It is helpful to integrate multi-modelling methods and this brings advantages. For example, many meta-models are synchronously introduced into the general modelling tool in MetaEdit+. Here, the general modelling tool can provide good support to multi-meta-models (describing modelling language) to more expediently realise integration of multi-methods.

The second implementation method framework has meta-modelling based on a modelling tool generator. Here, the modelling tool generator does not have configurable ability, but it can generate corresponding modelling tools according to the meta-model. EMF and also GMF based on EMF can provide an exact fit for it. The main advantage of modelling tool generation is that it can provide the user with independent tools. This is helpful when customising, modifying and improving modelling tools. For example, if the modelling tool that is generated does not have some functions that are required (such as code

generation), the user can modify the generated modelling tool to add the required functions. On the opposite, customisation and improvement are very difficult to realise in general modelling tools.

The meta-modelling framework in this thesis is similar to the first example. Namely, a meta-modelling framework based on a general modelling environment is adopted. Meta-modelling and modelling activities are carried out in the general modelling environments. The modelling environment of the target modelling language is constructed by the general modelling environment according to the meta-model, rather than by individual generation of modelling tools specified by the target language. It can be seen from analysis that although support from the general modelling environment at secondary development is less powerful than direct formation of modelling tools, it can make use of extensibility of the meta-modelling language and code generator. Meanwhile in the second example, once the meta-model on which generation of modelling tools is made changes, we have to generate a new modelling tool. So here the modification to the previous generated modelling tool is difficult to retain and maintain. It restricts maintainability of its extensibility.

6.2 Architecture of Meta-Modelling Infrastructure Based on XMML

In a meta-modelling framework based on a general modelling environment, this environment is the core of realising meta-modelling. Its basic task is to provide unified tool support for domain-specific modelling activities. It is not necessary for meta-model developers to specially develop modelling tools for various domain-specific modelling languages. Instead they can focus on the domain meta-model enabling them to concentrate on design and construction of domain-specific modelling languages and so decrease the cost of meta-modelling.

A general modelling environment, Archware based on the general modelling environment of XMML is put forward in this thesis. Its two core functions are as follows: The first is to provide the necessary tools for the meta-model developer to realise domain meta-modelling. The second is to parse the meta-model and generate the modelling support environment required by domain modelling. The basis for the realisation of meta-modelling is a model used for constructing the meta-model, namely the meta-meta-model. Meanwhile, the functions of model parsing and operation are realised by the model reflection interface as shown in the following figure. The meta-meta-model together with the model reflection interface makes up the meta-modelling infrastructure of the general modelling environment. Other parts of the general modelling environment realise access and operation of the model mainly by services provided by meta-modelling infrastructure.

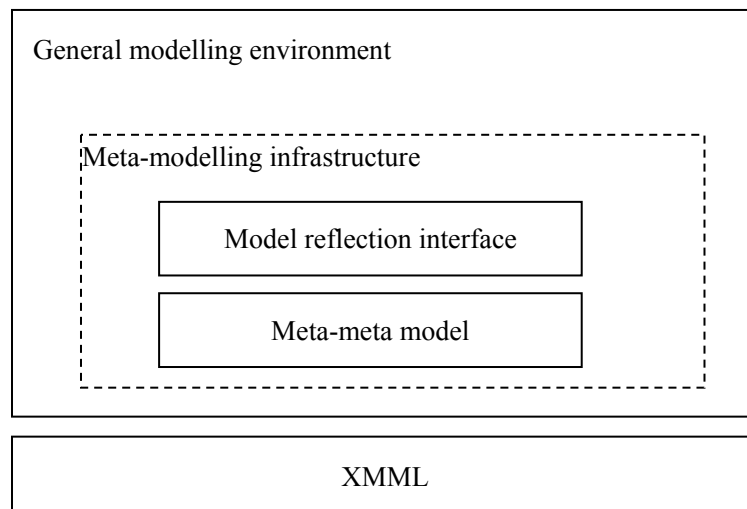


Figure 6. 3 Meta-modelling Infrastructure Based on XMML

In the meta-modelling framework in this thesis, XMML is the foundation on which the general modelling environment is realised. The models (meta-meta-model, domain meta-model, domain application model) used and generated during the process of modelling activities adopt XMML as their basic descriptive language. Although their abstract syntax and semantics are different, they share the concrete syntax defined in XMML. The purpose of using the

same basic language to describe various different models is that it is helpful to simplify the parsing mechanism of the model, and to enable the unified reflection interface to use the AST of the various domain modelling languages. Therefore, the meta-modelling infrastructure becomes a basic modelling data definition constructed on XMML and the service interface layer of basic language.

6.2.1 Meta-Meta-Model

The model of the DSML can itself be described by other meta-models. In our meta-modelling framework, all the domain meta-models can be described by an independent meta-model. For the sake of differentiating, the independent meta-model is called the meta-meta-model and it is the descriptive language model of the domain-specific modelling language. It is the key to meta-modelling, which can enable all the modelling languages to be described in a unified way.

The meta-meta model mainly concentrates on how to express the various constructs of the domain-specific modelling language. These include the modelling concepts and relationships as well as the rules between concepts when designing and developing domain-specific languages. Finally it realises a description of the abstract syntax of the domain-specific language, its concrete syntax and semantics, etc. In the meta-model, the representation of abstract syntax is expressed by Instanced the modelling elements of meta-meta-models which also characterise the domain modelling concepts of the domain-specific modelling language as well as the relationships between and among them. In the actual meta-modelling activities, it is necessary to define the abstract syntax of the target DSML together with its concrete syntax and semantics.

Concrete Syntax modelling is carried out mainly according to graphic syntax. Because text syntax is realised through the text syntax of XMML, only graphic

syntax is needed to design and construct according to the target domain. Semantics modelling is described by event-rules as well as coupled code generation logic, etc. A general advanced programming language is usually used to express them and enable model semantics to become an organic part of the model processing logic by timely calls of the general modelling environment.

In essence, a meta-meta-model is also a domain-specific modelling language, and it is the modelling language which is specifically applied in the field of the design of domain-specific modelling languages. But the main differences between a meta-meta-model and a general meta-model lie in the fact that the meta-meta-model is a basic model embedded in general modelling environments and an important part of meta-modelling infrastructure. Its modelling elements and modelling structure are fixed to the meta-model and the meta-model developer constructs and develops the domain meta-model according to the domain using only these.

The extraction and organisation of the modelling elements of meta-meta-model result from analysis and research result previously given. In the meta-meta model, the modelling elements can be classified into two categories: modelling entities and the relationships between them. Meta-meta model modelling entities are used to describe domain meta-model entity type and relationship types. Meanwhile, meta-meta-model relationships are used to describe relationships of these entity type and relationship type these entity types and relationships among the relationship types.

The five types of modelling entity in the meta-meta model can be described as follows:

(1) Model Type of Entity Element

It used in modelling the model type of the target domain modelling language. Its instance objects correspond with various models of domain meta-model.

Usually a domain-specific modelling language is used for more than one

model and this includes the case model, structure model, logic model, etc and their meta-types are Model Type of Entity Element of the meta-meta-model. The meta-model developer can describe the available model types of the target domain modelling language in this way.

(2) Entity Type of Entity Element

It used in modelling the entity type of the target domain modelling language. Its instance objects corresponded with the various domain meta-model entity types. The entity modelling elements of the meta-model are obtained from analysis of the target domain and concepts extracted from domain entities. The actual source of each depends on the target Of Meta-Modelling and the methods of meta-modelling adopted. Literature [107] concludes that there are four main sources of meta-model elements according to 23 practical cases. These are: concepts put forward by domain experts or developers, goal achieving, the appearance of system to be built and changes in the product line. Modelling elements extracted on the basis of the above four sources, can if they are entity type modelling elements, be described by meta-meta model entity type, entity elements.

(3) Relationship Type of Entity Element

It used in modelling relationship types of the target domain modelling language. Its instance objects corresponded with various domain meta-model relationships types. Some modelling elements obtained by domain analysis are relevance modelling elements. They are related to the Entity type to be associated with to characterise existing modelling entity relationships, as well as to the role they play in the relationship. The relationships cannot exist alone and will have real meaning only together with the actual entity type e.g. relationships of request / response / own / affiliate, etc. They are examples of relationship type entity elements of the meta-model.

(4) Diagram Type of Entity Element

It used in modelling the visual diagram type of the target domain modelling language. Its instance objects corresponded with various diagram types of the domain meta-model. In an actual domain application model, the same model can be expressed by multi-diagrams. These diagrams can either be of the same type or of different types. Many diagrams of the same type can be used to express each part of the whole model or to express different aspects of the model with different diagrams. In the general modelling environment, diagrams of various types can correspond to their own graphics rendering engines. These are registered with the general modelling environment in the form of plug-ins. The meta-model developer can specify the available diagrams and relationships to the graphic rendering engine, and the related information is described by diagram type entity elements of the meta-meta-model.

(5) GraphicObject Type of Entity Element

It used in modelling the visual representation type of various visual modelling elements of the target domain modelling language. Its instance objects corresponded with various graphic object types of modelling entities. The best feature of a visual domain-specific modelling language is that it can specify the corresponding visual appearance for various modelling objects of the domain application model to directly express domain model content. Instance objects of GraphicObject type are related to diagram(graphic) type of model. The same domain modelling element can appear in different graphics and express different appearances. Description information of GraphicObjects can be defined by the meta-model developer with each GraphicObject corresponding to its description. The apparent structure of the GraphicObject, response event between GraphicObject and user interaction as well as appearance logic transformation are defined GraphicObject descriptions. Therefore, definitions of the GraphicObject are composed of appearance description and the processing logic of interactive events.

Modelling entity elements of the above five kinds of meta-meta model are

the basic components used in building the domain meta-model. They must be organically associated if they are to integrally express the information for constructing the domain meta-model. Therefore, it is also necessary to make modelling relationships exist among the various modelling elements by introducing relationships in the meta-meta model.

In the meta-meta model, six types of relationship are used to describe relationships among various meta-model modelling elements.

(a) Possessed Type of Associated Element

It used to describe membership relationships among modelling elements. There are hierarchical relationships among the various modelling elements of the domain meta-model. For example father and son, whole and part, one to many between model and entity, model and graphic, and graphic and graphic relationships. The possessed type of meta-meta model is used to describe modelling elements in this category.

(b) Refined Type of Associated Element

It used to describe the corresponding relationships between domain meta-model entities and refined model type, when the refined modelling of the domain entity object is carried out in the domain-meta model. Not all model types can become refined model objects and not all entities can carry out refined modelling. Therefore, meta-model developers must, according to the characteristics of the target domain applications, make related rules for model refinement operations. That is, during meta-modelling, the refined type of meta-meta model needed to specify a refinement model type which is allowed by the refinable entity modelling element of the domain meta-model.

(c) Referred Type of Associated Element

It used to describe relationships between various model type and modelling entity type of the domain meta-model. To ensure uniqueness of definition of the

modelling entity, a modelling entity object can be defined in a model. The “possessed” relationship is just used to define rule association. Once the entity object is defined, it can be referred to by other models. However, in some situations, not all typological models can refer to an entity type of entity object. For example, in the UML, the use case objects defined in use case diagrams can not be referred to by sequence diagrams. Therefore, it is necessary for the meta-meta model to describe this kind of reference rule by means of a referred type of associate element in the domain meta-model. Besides, in the domain meta-model, there are reference relationships between various modelling elements and visual graphic primitives. Such relationships can be one-to-one, one-to-many or many-to-many. The referred type of associated element can be used to specify that a modelling entity element can have a particular appearances and which entity types can refer to a particular graphic primitive. A structure of separation and loose coupling among them can be realised by reference association.

(d) RoleAssigned Type of Associated Element

It used to describe the role-assigned type in various domain meta-model Relationship Type and modelling element types. Meta-model Relationship Type is a special entity element, its instance objects in the domain model are Relationship elements, expressing binary relationships among modelling elements. There are two information roles (source role and target role) in associated elements. When the associated element is used to connect two modelling elements, both roles are needed to respectively assign to corresponding connected elements. In some situations, they can be assigned to specified modelling elements. Therefore, the corresponding role-assigned relationship can be built between an relationship entity element and a reflexive relation or between two modelling elements by a role assigned type of associated element during meta-modelling.

(e) Attached Type of Associated Element

It used to describe attached relationships existing among various entity modelling elements of domain meta-models. In domain application models, some domain entity objects are used in combination. Meanwhile, some entity modelling elements cannot exist alone but must be attached to other entities to be meaningful. One example would be port objects in the software architecture model. These must be attached to components or connectors in use. This attached relationship among entity elements is described by the attached type of associated element of the meta-meta model, which describes a corresponding relationship between parasite and host.

(f) Contained Type of Associated Element

It used to describe contained relationships existing among various entity modelling elements of domain meta-models. This is an associated type among entities that are looser than attached types. The attached type of relationship forms a parasite / host relationship in associated entities. This means that when a host entity object is deleted, the entity object attached to it is also deleted. But the contained relationship describes a logic contained relationship among entity modelling elements, and there is no identity existing in their life cycles. For example, a boundary object of the UML is just a boundary container and plays the role of logic grouping. When the boundary objects are deleted, the modelling objects included in the boundary cannot be deleted at the same time.

Besides the above core modelling elements in the meta-meta model, there are other assistant modelling elements used for annotation, grouping elements, etc.

In the development of some complex domain-specific modelling languages, this language will not be completely described just by one domain meta-model. The large number of modelling elements involved as well as their associated relationships would make the domain meta-model too large and complex and so difficult to understand and maintain. Therefore, there are many types of model definitions of domain meta-modelling in the meta-meta model, each type of

model just pays attention to some parts of the constructive information of the meta-modelling. The various models combine into an complete meta- model. In order to make the organisational structure of a meta-model clear and its function definite, a parallel model organisational structure is defined in the meta-meta model. It is composed by many types of model:

(1) Model defined by meta-model element.

It used to define all models that are likely to be needed in the meta-model as well as modelling elements that are likely to be used in the model. There are six modelling elements of this kind: Model Type entity, Entity Type entity, Relationship Type entity, Diagram Type entity and Possessed Type of Associated Element. When meta-modelling, Model defined by meta-model element is the model that is built first. We can use this to define and declare all the modelling elements that will be used in the meta-model. Then we can use these modelling elements in other types of model by means of referencing.

(2) Model defined by meta-model diagram.

It used to define modelling element diagrams used in the meta-model, and to connect graphic primitives to their diagrams, and then to declare relationships between diagrams and modelling elements expressed by them by reference association. The modelling elements of such models are as follows: GraphicObject Type entity, Referred Type of Associated Element, Possessed Type of Associated Element, etc. While others use: diagram type entity, entity type entity and relationship type entity to refer to the above definition model of meta-model elements.

(3) Model defined by reference relationship among meta-model entities.

It used to define the reference relationship that is likely to exist between model type entity and entity type entity. There is only one kind of such modelling element, namely a Referred Type of Associated Element. The reference relationship is used to describe what entity elements of other models

can be referred to in a model.

(4) Model defined by role definition of meta-link element.

It used to define role assignment rules for all associated elements in the meta-model. There is only one kind of such modelling element, namely a RoleAssigned Type of Associated Element. Meanwhile, entities of entity type and relationship type are from the *definition model of the meta-model element*.

(5) Model defined by relationship among meta-entity elements.

It used to define implicit relationships existing among the entity elements of the meta-model, such as attached relationships and contained relationships. There are two kinds of such modelling elements: Attached Type of Associated Element, Contained Type of Associated Element.

(6) Model defined by refinement relationship among meta entities.

It used to define corresponding relationships between certain meta-model entity elements and refinement models. Here, there is only one kind of modelling element, namely a refined type of associated element. The relationship between the entity and refinement models is one-to-many, and an entity can have many types of refinement model to which to correspond with.

In the actual meta-modelling, a part of a definition model can be possibly used. However, a model of the same type may have more complex meta-modelling. The general modelling environment can combine these definition models which compose the meta-model, and provide information meta-model access and operation through the model reflection interface.

6.2.2 Model Reflection Interface

In the domain-specific modelling framework given in this thesis, the advanced programming language, is used to describe model event processing logic, specification of model and model members, etc. in domain meta-modelling and

domain application modelling. These programming codes written by the advanced programming language will be timely called at design design times of modelling or run times of the code generator processing model. These programming codes usually need to dynamically access the meta-model or some information of the application model of the meta-model.

The model reflection interface is a programming interface which is designed to meet actual requirements. It can provide dynamic design and runtime context using the general modelling environment, code generator and the programs of the model's high-level programming language. The model reflection interface is a basic service provided by the general modelling environment, and it is an important constituent part of the meta-modelling infrastructure. It mainly provides two kinds of services: the first is a service of accessing the meta-model, and the second is to provide a service regarding the operating information of model instances. The general modelling environment itself will access meta-model construction information by using the model reflection interface to build the instance modelling environment of the meta-model. Meanwhile the meta-model event-processing program needs services provided by the model reflection interface to access and operate objects of model instances as well as the content of related modelling processing attributes.

The realisation of the model reflection interfaces is based on XMML. This meta-modelling language provides unified text syntax across the domain meta-model and the domain application model. The same model resolution mechanism can be used to construct an abstract syntax tree and then the model reflection interface is used to provide unified model access and operation interface service with external objects.

The model reflection provides services mainly by the following interfaces.

(1) Access interface of meta-model.

GetMetaModels():ModelTypes;

Return set of Model Type of Entity Element defined in the meta-model.

GetMetaEntities(Model: ModelType): Entities;

Return set of meta-modelling Entity Type of Entity Element defined in given model type.

GetMetaRelationships(Model: ModelType): Relationships;

Return set of meta-modelling Relationship Type of Entity Element defined in given model type.

GetMetaDiagrams(Model: ModelType): Diagrams;

Return set of meta-modelling diagram Type of Entity Element defined in given model type.

GetMetaGraphicObjects(Model: ModelType, Diagram: DiagramType): GraphicObjects;

Return set of GraphicObject Type of Entity Element defined in given model type.

GetMetaRefineModels(Entity: EntityType): ModelTypes;

Return set of Refined Type of Associated Element that can be built in a given entity.

All the above interfaces are mainly used to query elements and their attributes defined in the meta-model or to filter a corresponding element set by an associated condition.

(2) Operation interface of modelling element objects

CreateElement(MetaTypeName: string): Element;

This is a factory method used to create an element of specified type name.

DeleteElement(Element: ElementType): Boolean;

Used to delete the given modelling element object.

GetElementProperties(Element: ElementType): PropertiesType;

Used to capture the given attribute object of modelling elements, an attribute object is an attribute set of a series of element objects. After the attribute object has been obtained, the method of attribute object is called to query or change the specified attribute value.

TriggerElementEvent(Element: ElementType, EventName: string): Boolean;

Used to trigger an event processing program defined in a given element.

ApplyElementSpecs(Element: ElementType, EventName: string): Boolean;

Used to call a specification code segment defined in a given elements.

GetElements(Model: ModelType, ElementType: string): ElementTypes;

Used to capture specified type of a set of modelling element objects of a model.

All the above interfaces are mainly used to query and operate the domain application model during the processes of modelling or code generation.

The above gives some of the methods of model reflection interface. In the actual realisation, the model reflection interface is provided with services by a COM (Component Object Model) object. Due to the COM having independent with the programming language, it can flexibly call services provided by the model reflection interface in script language and external plug-ins and so create a general modelling environment with greater extensibility.

6.3 Instances of Meta-Modelling

In this section, the previous meta-modelling element and meta-modelling model structure are used to construct a visual architecture description language (ADL)

to demonstrate how a meta-modelling infrastructure based on XMML can give rise to a meta-model description.

6.3.1 Overview of Software Architecture

Software architecture description language can be seen as a domain-specific modelling language. Its target domain is software architecture, so it is necessary to analyse basic concepts of software architecture to extract its domain concepts and relationships to carry out meta-modelling of the ADL.

Software architecture is the high level logic framework of the target system. Its design includes overall design of system structure, function assignment of each computational element, high-level interaction between units, etc. At present, there is no uniform definition of software architecture. The definition given by D. Garlan and M. Shaw is closest to reality and has been widely accepted in academic circles. It is believed that software architecture is a level of software design process, above the algorithm design of computation process and data structure. It deals with various problems of the overall design of system framework and description. It includes overall organisation and global control structure, communication protocol, synchronisation, data access, function assignment of design elements, physical distribution, composition of design elements, choice of design project, evaluation and realisation, etc.

According to the above definition, software architecture can be abstractly generalised as follows.

Architecture = components + connectors +constrains, namely,

SA = {Components, Connectors, Constrain}

Components are certain business processing, intensive computational elements. They are independent of function or structure, and some large-grain components themselves can be seen as architectures. Also, some small-grained

components can be reused in other components and become constituent part of them.

Connectors bond together not only components, but also some computational elements. Here, the main difference compared with components is that it typically deals with communication among components and interface conversation and adapting and with connecting different components. These become part of the software architecture, and usually perform as frame-type objects or converted objects (calling remote components resources).

Constraints are usually rules or specification information existing in components or connectors. They are used to restrict the architecture style of the software architecture together with those of the functional and non-functional specification of its constituent elements.

A visual software architecture description language should have certain specific language features. It should have graphical syntax, which users will find easy to understand and use. At the same time, it should have the necessary formal syntax and strong tool support based on this.

6.3.2 Example of an ADL Meta-Model

Suppose that examples of ADL to be built here have the following specifications.

- (1) The software architecture model is mainly described by components and connectors, and their interaction specifications with external objects are described by the interface.
- (2) Many interfaces can be defined on each component or connector to interact with external objects.
- (3) Associations between components and other components and also between components and connectors must be carried out by interfaces.

- (4) Components can build a refinement model which is still an architecture model.

According to the ADL specification and its domain concepts, four basic kinds model elements, used in modelling, can be extracted. These are the component, connector, interface and binding interface. They are used for composing the ADL meta-model. A Model defined by meta-model element of the meta-meta model is used to define them, as shown in the following figure

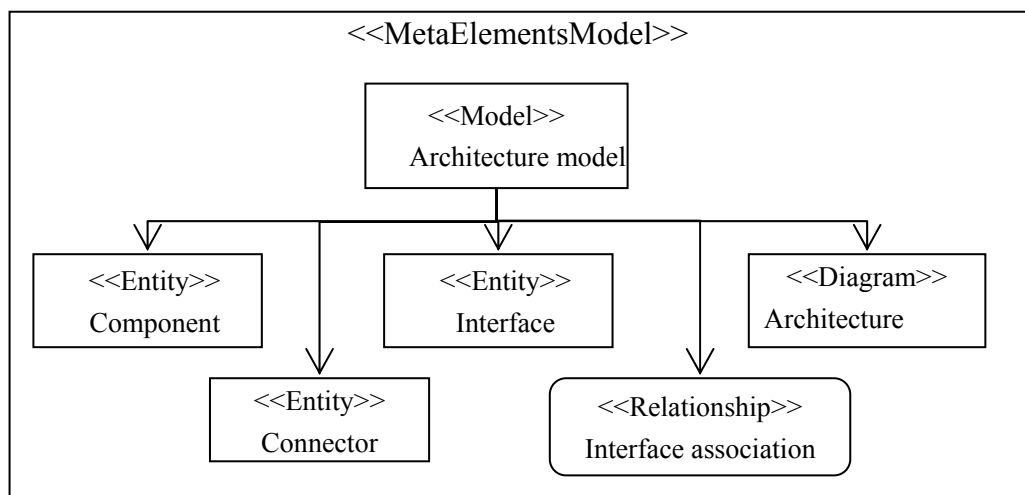


Figure 6. 4 Definition Model of ADL Meta-Model Element

The figure defines: meta-model type (architecture model), meta-entity type (component, connector, and interface), meta-relationship type (interface association) and meta-diagram type (architecture diagram). There are relationships between meta-model type and other types of meta-modelling element. The elements defined in the This model will be referred by other types of model built in the following stages. At the same time, the general modelling environment will generate the modelling tools necessary for the ADL modelling environment. Examples would include those that are used for modelling toolbar buttons and a model selection list using meta-modelling information described

by the model.

After the modelling elements of the domain meta-model have been defined, the relationships among these modelling elements must then be defined. The attached relationship between interfaces, components and connectors among the meta-entity elements can be described by a “relationship” definition model according to the specification of item (2) as shown in the following figure.

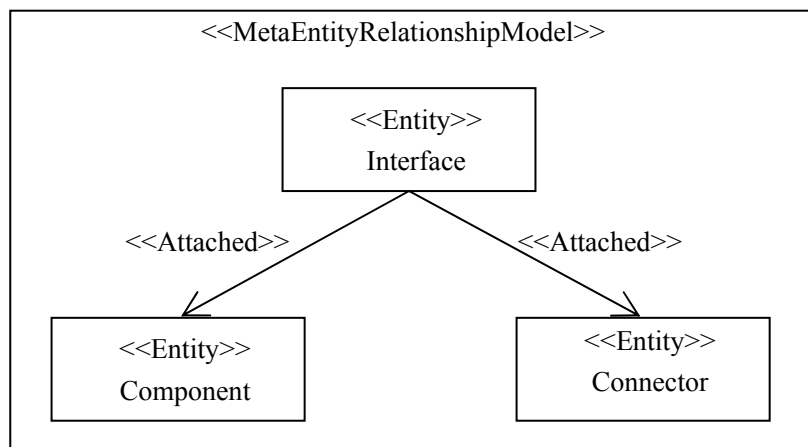


Figure 6. 5 Definition Model of Attached Relationships among ADL Meta-Entity Elements

Components and connectors defined in the above figure will act as host elements of the interface, so the interface can only parasitize components or connectors and can not exist in models on its own.

The Role-Assigned relationships of interface association elements can be described by a “role definition model of meta association element” according to the specification of item (3) as shown in the following figure.

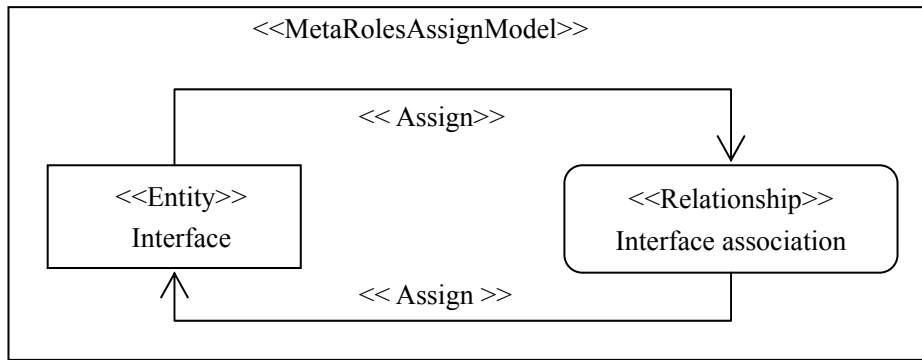


Figure 6. 6 Definition Model of Refinement Relationship for ADL Meta-Entity Element

“Role assign” association describes the interface association source role and target role, which are the only interface elements in the above figure. The “role-assign” association pointing to <<Relationship>> type denotes assignment of source role, while that pointing to <<Entity>> denotes assignment of target role.

The refined model type of component can be described by a “refined relationship definition model of meta-entity element” according to the specification of item (4) as shown in the following figure.

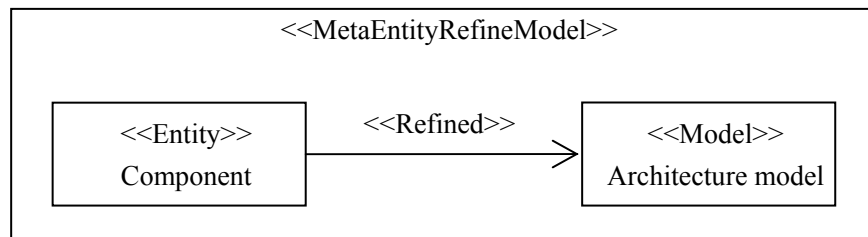


Figure 6. 7 Definition Model of Refinement Relationship for ADL Meta-Entity Element

Finally, reference relationships of the visual diagrams for the ADL modelling elements must also define what can be achieved by the “meta-model diagram definition model” as shown in the following figure.

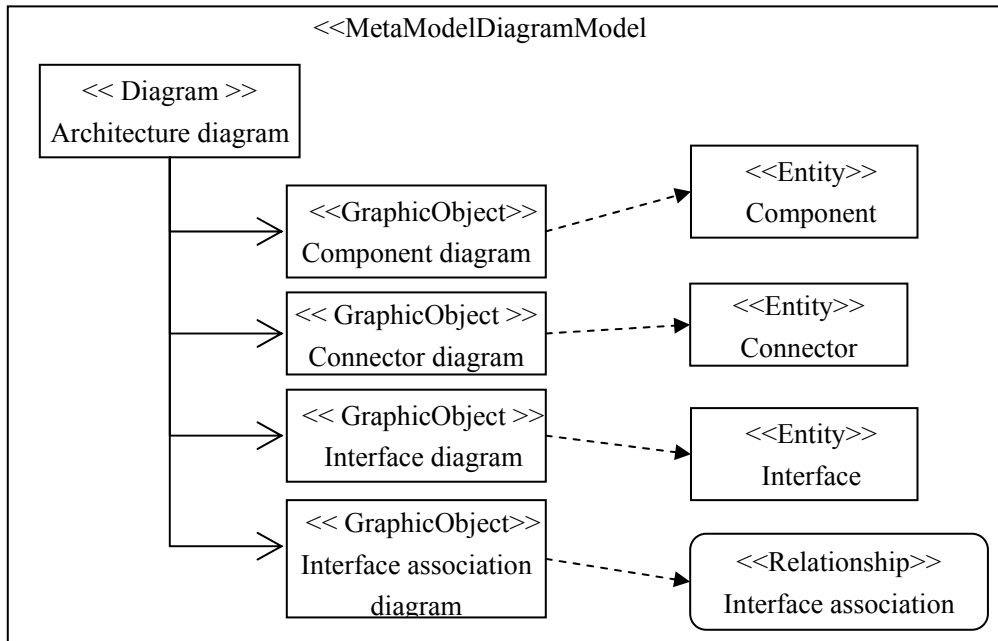


Figure 6.8 Definition Model of Meta-Model Diagram for ADL Meta-Entity Element

<<MetaModelDiagramModel>> <<Diagram>> , etc.	It represents the modeling element type defined by us in the meta-meta model, and it is similar to Stereotype of UML
Pane	It is used to represent EntityElement of meta-metamodel
round Corner Pane	It is used to represent Associated-Element of meta-meta model.
Solid Line	If its type isn't explicitly marked, it is Possessed-Relation, such as in 6.5, it is Attached Relation by explicitly marking.
dotted line	If you don't explicitly mark its type, it is used to represent Referred Relationship.

Table 6. 2 Refinement Relationship of ADL Meta-Entity Element

In the above figure, the corresponding visual diagrams are defined for various meta-modelling elements of the ADL. They are associated with their corresponding meta-modelling elements by reference. In the visual modelling provided by the general modelling environment, these visual diagrams become agents of modelling element objects.

The meta-model examples for the ADL comprise such a group of meta-model definitions. The general modelling environment will get the necessary construction information for the domain modelling environment from the meta-model and automatically construct examples for the ADL modelling environment.

6.4 Summary

In this chapter, methods of implementation of meta-modelling based on XMML are discussed. A meta-modelling infrastructure based on XMML is put forward. Corresponding with the characteristics of XMML and the basic requirements of domain-specific meta-modelling, this provides the necessary modelling support for XMML applications as well as the construction of supporting tools and the development of domain-specific languages.

At present, the implementation of meta-modelling activities and supporting frameworks is either based on meta-modelling using general modelling tools or on modelling tool generators. These two approaches each have their advantages and disadvantages. However, taking account of the extensibility and maintainability of the meta-modelling environment, both use a meta-modelling framework based on a general modelling environment. A meta-modelling framework is particularly suitable for a Model Driven software development method based on DSM.

In a meta-modelling framework based on a general modelling environment,

the general modelling environment is at the core of the implementation of the meta-modelling. It is not necessary for meta-model developers to concentrate solely on developing special modelling tools for various domain-specific modelling languages. They can focus on a domain meta-model that represents the design and construction of a domain-specific modelling language and so reduce the implementation costs of meta-modelling.

A general modelling environment based on XMML, Archware is put forward in this chapter. Its core functions are firstly to provide the domain meta-model developer with the necessary tools to carry out domain meta-modelling. Secondly, it can parse the meta-model and generate the modelling supporting environment necessary for domain modelling. In the architecture of Archware, the design of the meta-meta model and the model reflection interface together constitute the meta-modelling infrastructure of the general modelling environment. The remaining part of the general modelling environment is mainly devoted to services provided by the meta-modelling infrastructure to realise model access and operation.

The meta-meta model is the basis on which visual meta-modelling is realised. Its architecture affects methods of design for the domain meta-modelling language and the meta-modelling process. The meta-meta model has three main constituent parts:

- (1) Entity type elements.
- (2) Relationship type elements.
- (3) The definition model.

There are five kinds of meta-model entity type modelling elements as follows: model type, entity type, relationship type, diagram type and graphicobject type

There are six kinds of meta-model relationship type elements as follows:

possessed type, refined type, referred type, role assigned type, attached type and contained type.

There are six kinds of meta-model definition model as follows: definition model of meta-model element, definition model of meta-model diagram, definition model of meta-model entity referred type, role definition model of meta-relationship element, relationship definition model among meta-entity elements and refined definition model of meta-entity element.

Through the analysis and design of the constituent parts of the meta-meta model, Archware can provide a unified modelling environment in which the supporting ability of XMML language can provide a unified and extendible basic model to be used for domain meta-modelling as well as for domain application modelling.

The model reflection interface is a basic service provided by the general modelling environment. It can provide dynamic design time and runtime context information for the general modelling environment, code generators and advanced programming language programs. The realisation of the model reflection interface is based on XMML meta-modelling language. The XMML meta-modelling language provides unified text concrete syntax for the domain meta-model and the domain application model. The same model parsing mechanism can be used to construct an abstract syntax tree, then the model reflection interface is used to provide unified model access and operation interface services for external objects. There are two kinds of interface type based on the XMML model reflection model. These are the meta-model access type interface and the modelling element object operation type interface. In the implementation of Archware, the model reflection interface uses a COM object to provide services. With COM, the programming language can flexibly call services provided by the model reflection interface in script language as external plug-ins.

Chapter 7

Architecture of General Modelling Environment

In this chapter, domain-specific modelling tools supporting meta-modelling are discussed and a general integrated modelling environment based on XMML, the architecture of Archware, and its design and realisation are given.

7.1 Overview of Modelling Tools

7.1.1 Development of Modelling Tools

The modelling tools are a key element for the realisation of Model Driven development. After designing a domain-specific language, the next important task is to determine how to provide the supporting tools for the modelling language. There are many ways to construct a supporting tool for a modelling language. The options can be listed in ascending order according to development efficiency and degree of automation in the following hierarchy [48]:

- (1) Totally design the modelling tool from scratch.
- (2) Design the modelling tool based on some bottom frames.
- (3) Generate a basic framework for the modelling tool based on some bottom frames using a meta-model then manually add implementation code.
- (4) Fully generate the modelling tool based on some bottom frames using a meta-model.

- (5) Generate the necessary configuration data for the general modelling tool from a meta-model.
- (6) Use a general modelling environment integrating both meta-modelling and modelling.

In the 1970s and 1980s, a large number of the construction methods used to create modelling tools are in level 1 (of the above hierarchy). Many CASE tools were produced during the period. They were all developed according to a particular modelling language. Users could only use the modelling language bound to the CASE tool to develop their systems. They could not adjust or modify the tools or the modelling language. Besides, the heavy workload that accompanied this level of approach to developing modelling languages and their supporting tools seriously hindered the pace of development of these modelling languages and tools. By the 1990s, with the rapid progress of software development technology and development tools, the development approach level for modelling languages and tools improved from level 2 to level 5 (of the above hierarchy). From the early CASE Shells [95] and metaCASE [24] to deuterio meta-modelling tools represented by DOME [25], metaEdit [96, 97] and TBK / ToolBuilder [2], this remarkable progress saw the meta-model being separated from the modelling language and modelling tools and the development of modelling tools becoming based on a meta-model to achieve automatically or semi-automatically generated modelling tools. However, the first drawback of a development approach which separates meta-modelling from modelling is that it militates against rapid design, validation and debugging of the modelling language. From the manual design of the modelling language model through to generation of the modelling language and its tools, we face a long process of design, input, configuration, compiling and testing. This greatly influences the design and development efficiency of the modelling language. In particular, in DSM methodology, the development quality and efficiency of the domain-specific modelling language are related to the success or failure of a

domain-specific Model Driven project. In the context of what is really required, it is inevitable that research and realisation of the modelling environment should turn to an integration of meta-modelling with modelling activities.

7.1.2 Analysis of Several General Modelling Environments

The functional requirements of the general modelling environment can be analysed and checked from the perspectives of both domain meta-modelling specialists and domain application modelling specialists. The domain meta-modelling specialists have the following minimum requirements for the general modelling environment.

- (1) Able to declare and specify the relationship between domain concept entities of meta-model and entity
- (2) Able to declare the above two meta-modelling elements and specify their attributes
- (3) Able to specify basic rules for association between entity objects and how entity objects are to be carried out
- (4) Able to specify graphical or textual symbols for each kind of modelling element
- (5) Able to provide code generators with model accessing services
- (6) Able to generate basic modelling tools from the meta-model

Meanwhile the domain application modelling specialists have the following minimum requirements for the general modelling environment.

- (1) Able to access the model
- (2) Able to create polytype models
- (3) Able to connect entity objects by association objects
- (4) Able to provide graphical interface and support drag and drop of primitive elements of modelling elements
- (5) Able to edit attributes of the model and its member objects

Besides, the general modelling tool should be capable of some of the basic functions of general authoring software, such as copy, cut, paste, undo, redo as well as multiform output of pictures of model documents, etc.

At present, we now have some tools supporting domain meta-modelling and domain application models, such as MetaEdit+ [47] from the MetaCase Company, GME [109] from Vanderbilt college, Microsoft DSL Tools [91] as well as EMF [38] from Eclipse, etc. These tools have their own characteristics for realising supporting domain meta-modelling and domain application models. They also have their own meta-modelling languages, meta-meta models and basic frameworks.

GME is a general modelling environment developed from the modelling research field of electronic engineering. The meta-meta models used by it emphasise concepts of ports and weaken concepts of association (Associations represent conducting wires connecting electronic components in circuit design). Compared to other modelling tools, it is more suitable for modelling systems of electronic engineering domains. The meta-model built by GME is described in a way similar to UML, and differentiates the various element types used by a stereotype method. Meanwhile, the specification description is with a mutated language based on OCL. GME belongs with the modelling tools development methods of level 5 (in the above hierarchy).

DSL Tools is an integrating modelling tool which first appeared in Visual Studio 2005/.NET Framework SDK 3.0. As a commercial modelling tool it depends heavily on Microsoft platforms. DSL Tools can only be used as an extending tool run in the development environment of Visual Studio. All systems modelling generated by DSL Tools can only run on Microsoft platforms and this is even specified in its license agreement. From the perspective of a meta-meta model, DSL Tools cannot provide multi-view modelling. It cannot provide flexible development and customisation in meta-modelling. After generating corresponding modelling tools, users need to make further

development and adjustments. Therefore, it is a modelling tool only at level 3 (in the above hierarchy).

EMF is a java open-source framework and code generation tool, its core meta-meta model is ECore which as a matter of fact is a MOF, therefore, in the strict sense, it is not a modelling tool based on DSM. A concept model of the modelling language, attribute tree list, tool palette, etc. can be defined in EMF by a set of tools. Corresponding Java source code is generated by building a model structure of UML relationships after exporting, which is similar to other java binding frameworks. For example, JAXB or XMLBeans generate java source code after an object-oriented model is given. These are modelling tools at level 4 (in the above hierarchy).

MetaEdit+ is a general modelling environment, integrating domain meta-modelling and the domain application model and is in level 6 (in the above hierarchy). At present, it is the most mature commercial general modelling environment [65] for modelling tools supporting DSM methodology. It is used in visual modelling GOPPRR (graph object property port role relationship) as a basic language to describe the meta-model. The basic modelling elements of the meta-meta model are Graph, Object, Property, Port, Relationship and Role, as shown in Figure 7.1 and Figure 7.2.

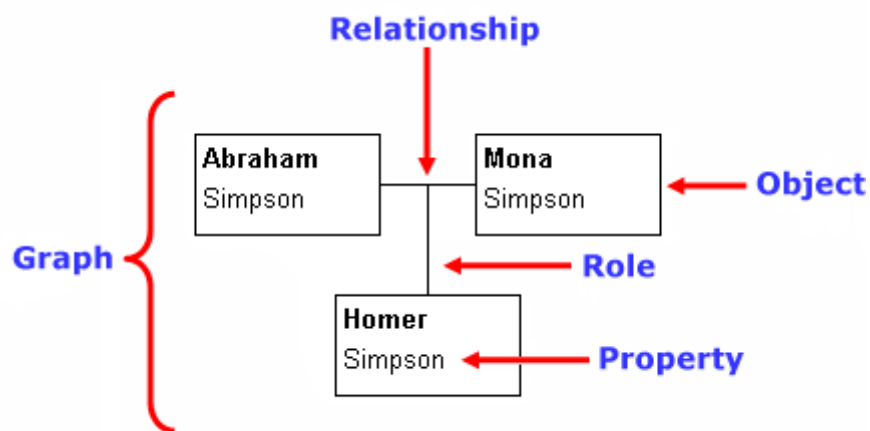


Figure 7. 1 Meta-Model Element of MetaEdit+

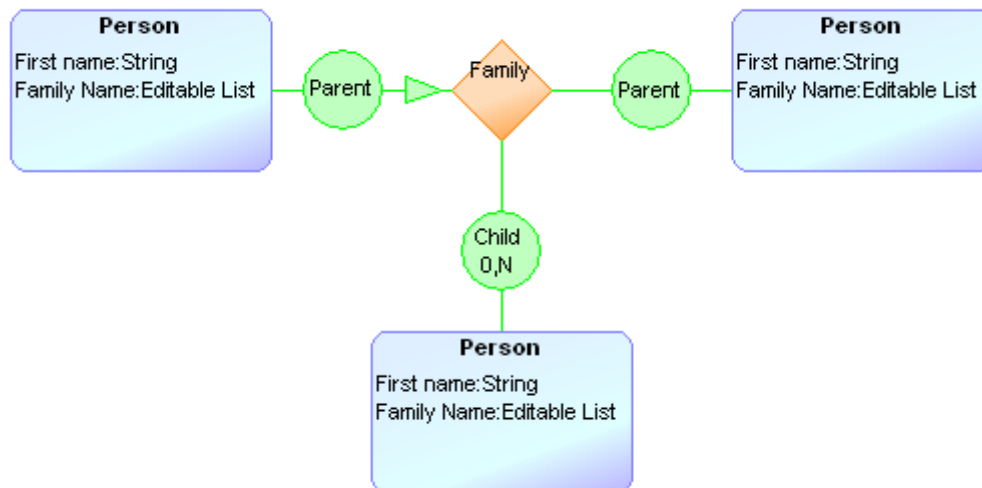


Figure 7. 2 Meta-Model Legend of MetaEdit+

Among them:

- (1) The diagram represents an independent model, expressed as a visual diagram.
- (2) Objects are the main elements of a diagram.
- (3) Relationships are used to connect objects and represent a kind of relationship.
- (4) Role connects an object to a relationship.
- (5) Port is used to define possible added semantics when roles and objects are associated.
- (6) Property denotes features of the above elements.

They are first-order meta-modelling elements of the MetaEdit+ meta-modelling language. The corresponding meta-model development tool is provided in MetaEdit+. This is used to realise description of the domain-specific modelling language model through use of the above meta-modelling elements, then to parse the meta-model obtained and automatically construct corresponding domain-specific modelling tools. But due to the use of its meta-modelling language GOPRR it lacks some key meta-modelling elements relating to the extensibility of meta-modelling, which leads to MetaEdit+ being

limited in usability and extensibility. These deficiencies are mainly embodied in the following aspects.

- (1) Lack of a first-order modelling element to describe relationships among meta-modelling elements.

The meta-meta model of MetaEdit+ only gives entity elements that are used to describe entity elements of the meta-model. It overlooks various associated meta-modelling elements existing in these entity elements. A part of the description mechanism for relationships among entity elements is put on the meta-modelling tools so the user cannot make necessary adjustments and customisations. For example, it cannot specify host objects or container objects for objects, nor can it model reference relationships among modelling elements in MetaEdit+.

- (2) Does not build first-order modelling elements for view and visual graphic primitives

Although the design tools for visual graphic primitives are provided in MetaEdit+, the meta-modelling elements are closer to those used in the meta-modelling of visual modelling languages. For example, diagrams and graphic primitives are not independent modelling elements of the meta-model. Instead they are dealt with as models and attributes related to objects. Therefore, there is no flexible description method for visual meta-modelling which eventually means that multi-types of views cannot be established in the same model. What's more, there is only one kind of fixed diagram primitive representation for objects and there is compact coupling between diagram primitives and objects.

- (3) Lacks a description mechanism for interactive behaviours at design times

The usability of a visual modelling language largely embodies the user's interactive experience provided by the modelling environment. It is a

requirement that the modelling tool can flexibly adjust its own activities according to the features of the modelling language. Meanwhile neither description nor an extended mechanism of inter-behaviours at the model designing stage is provided to meta-model developers. This leads to the designed domain-specific modelling language being only able to interact with users under the inter-behaviour framework of the same modelling tool.

(4) Lacks flexible rules for description mechanism.

There are four kinds of fixed declared descriptive methods for the description of model rules provided in MetaEdit+: connectivity, occurrence, port and uniqueness. So it can only make declarative descriptions of static rules for modelling language elements in these four kinds of fixed rules types. This means that the meta-model developer cannot describe the application logic of some complex rules. At the same time, other rule types cannot be extended and this leads an inability to flexibly answer some of the requirements of the modelling rules.

(5) Lacks functional extensibility mechanism for modelling environment.

The fixed modelling tool construction function provided in MetaEdit+, means that meta-model developers cannot extend or modify the modelling tools after they have been built. While as a general modelling environment, if it cannot provide the necessary second custom development or extensibility mechanism for function plug-ins, there will be some modelling requirements that it will find very difficult to satisfy.

As can be seen from the above analysis, the functional realisation of the general modelling environment is largely limited by the basic modelling language and the meta-meta model that it uses. From the viewpoint of the logic level, the general modelling environment and basic modelling language belong to two different levels. In addition, the general modelling environment is a functional externalisation and embodiment of the basic modelling language and

the meta-meta model.

7.2 Architecture of Archware

Archware is a general modelling environment based on XMML. It is used to design a domain-specific modelling language which carries out domain application modelling and is an integrated meta-modelling and modelling environment. The architecture style of Archware uses architecture of Model / View / Controller, Shown in Figure 7.3.

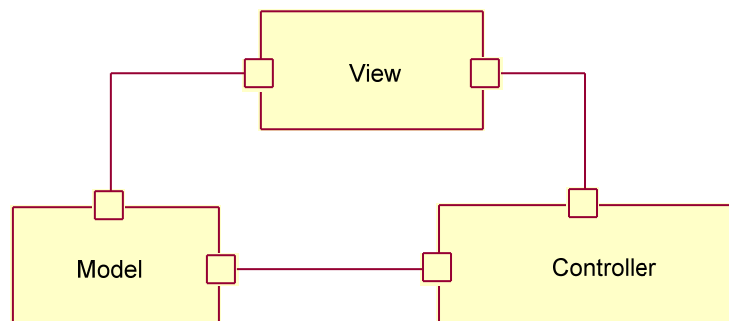


Figure 7. 3 Architecture Style of Archware

The general modelling environment is divided into model, view and controller. Three top-level components are formed: model components, view components and controller components. These components interact with each other through events and orders. When the controller changes model data or properties all dependent views will be automatically updated. Similarly, when the controller changes view, this can get data from the latent model to renovate itself. The following describe the functions of the three main components in the refined architecture model of Archware architecture.

7.2.1 Viewing Component Model

The view component represents the modelling environment interface that users directly operate. For Archware, the functions of the view components mainly lie

in construction modelling operating environments. Examples include constructing corresponding toolbars, attribute editors and in model visual rendering engineering, etc. according to the meta-model as well as the perception of user interactive events and feedback of event processing results. However, event processing responsibility is not included. The view component is complex and it can be further refined as a description of the other architecture as shown in Figure 7.4.

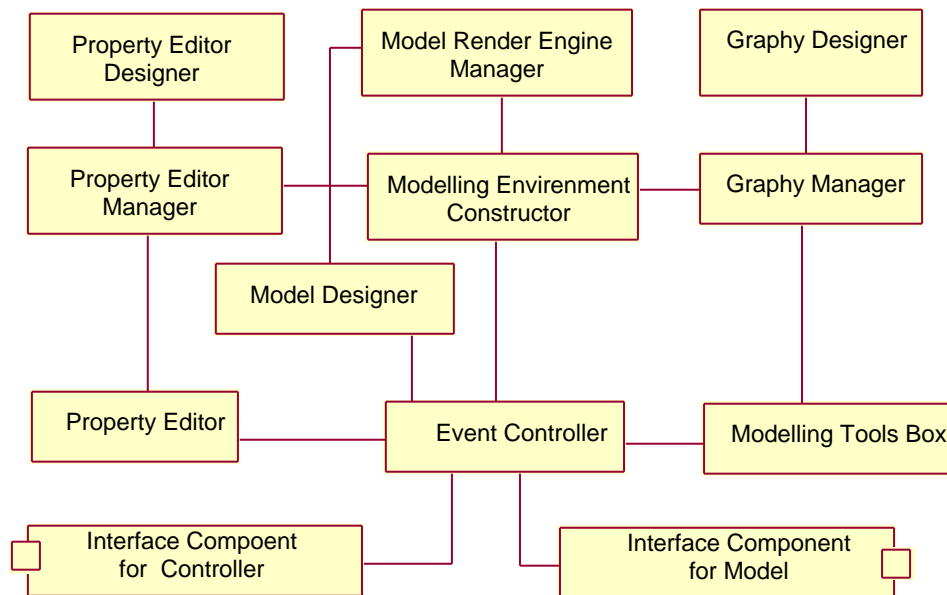


Figure 7. 4 Sub-Architecture of Archware View Components

Event Controller is a core control component of view component architecture. It is responsible for dealing with the scheduling of other components and message passing. It interacts with external model component and controller components by two respective interface components (*interface component for model* and *interface components for controller*).

Modelling Environment Constructor is responsible for constructing a corresponding modelling environment according to meta-model information. The modelling environment is made up of three main components: *Model Designer*, *Modelling Tool Box*, and *Property Editor*. The Modelling

Environment Constructor does not directly construct them, but they are constructed by the corresponding managers (*Model Render Engine Manager*, *Property Editor Manager*, *Graphy Manager*).

The Model Render Engine Manager manages the plug-ins of the graphic rendering engine of the general modelling environments. The plug-in approach is used to achieve extensibility mechanisms for the visual modelling diagrams under the general modelling environment. Users can develop their own rendering engine plug-ins according to special model diagrams, and register them to the model rendering engine manager. The model rendering engine manager chooses a corresponding graphic rendering engine for the Model Designer to drive visual model design according to instructions provided by Modelling Environment Constructors.

Property Editor Designer is used to design special attributes information for each meta-model element. It is an essential activity in modelling the attributes of modelling elements in the meta-modelling activities. Most modelling tools used in the modelling of attributes of meta-modelling elements use an approach based on attribute name and attribute type to describe and provide users with an editing interface in the form of Grid. The advantage of this method is simpler realisation, and it is known as “light weight property modelling”. However, this approach lacks the necessary flexibility when it describes some properties of complex elements and a single editing interface is provided to users. Therefore, another approach known as “heavy weight property modelling” is used in Archware. That is a specialist designer is provided in the meta-modelling environment so meta-model developers are free to develop program logic dealing with each item of attribute information for each meta-modelling element and editing window interface. The *Property Editor* produced during development will be managed by the *Property Editor Manager*.

Graphy Designer is used to design a corresponding visual primitive for each modelling element of the meta-model together with its processing logic for user

interactive events. The designed primitives will be managed together in the Graphy Manager. The Graphy Designer provides meta-model developers with the visual descriptive ability of powerful modelling elements. At present, in some modelling tools, although graphic primitives for custom drawing tools are provided, these tools cannot describe logical interaction of the primitives. Therefore, in these modelling tools, the primitives only appear to the model designers in the form of static graphics and the unified processing logic for interactive events is provided by the *Model Designer*. This leads to meta-model developers being unable to describe and develop dynamic interactive behaviours for the primitives.

The Model Designer is a main component provided to users for visual modelling. It is usually manifested as a “canvas”, and it is an instantiation presentation of diagrams in models. In the modelling process, the Model Designer does not deal with modelling logic but just plays the role of a drawing area. It is mainly used to perceive interactive user events then pass event information to the Event Controller for further event response treatment.

7.2.2 Modelling Component Model

The model component is part of the body of Archware. It is responsible for the realisation of meta-modelling and modelling processing logic as well as some entity data for the basic model. The meta-modelling infrastructure given in the proceeding sections resides in the components. The processing logic packaged by model components and model data is a black-box operation for external components. The model accepts requested data and view actions and the controller returns the final processing result. The model components can be further refined as another architecture model, shown in Figure 7.5.

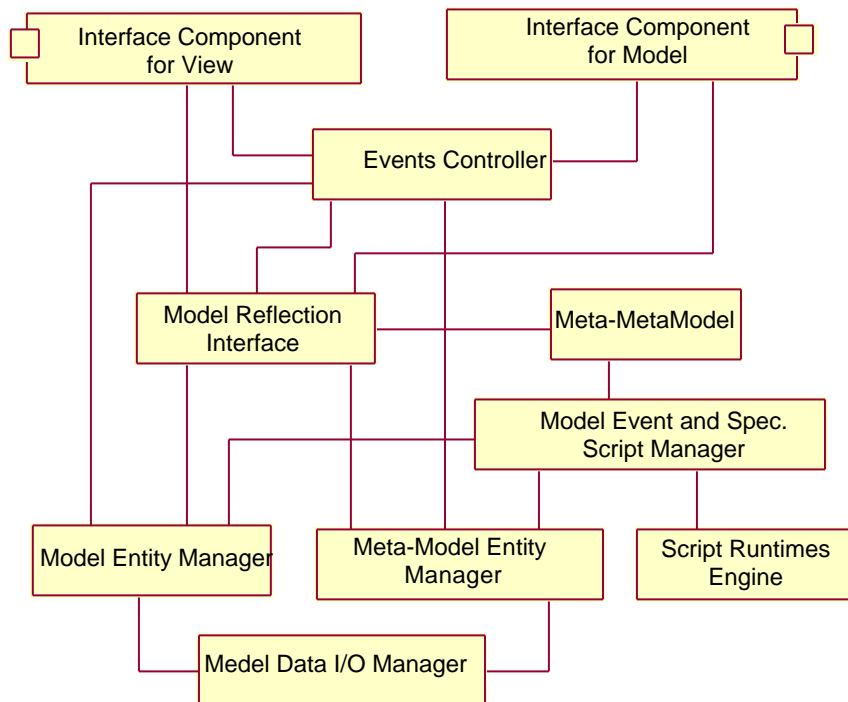


Figure 7.5 Sub-architecture of Archware Model Component

Model Data I/O manager is responsible for reading-writing operations to model data in memory. Its main functions include the serialisation and deserialisation of model entity objects. It deserialises the XML storage representation model and passes the results respectively to the model entity manager and the meta-model entity manager to manage.

The *Model Entity Manager* and *Meta-Model Entity Manager* are respectively used to manage the model and modelling data of the *Meta-Model* to realise operations on modelling element objects, such as adding, deleting and modifying, etc. At the same time, processing to services is realised by the model reflection interface.

Model Events and Specifications are executable logic parts of models, and they are usually script codes written by general advanced languages. They are detached from model modelling-element objects and managed together by *Model Events and Spec. Script Manager*. When they are called, the

corresponding *Scripting Runtime Engine* will be executed and this will return execution results.

The *Events Controller* is used to deal with event requests from internal and external event processing results. External event messages are mainly passed by the *View Interface Component* and the *Controller Interface Component*. At the same time, the two interface components can directly choose *Model Reflection Interface* components according to the type of external request and send the requested result data back.

7.2.3 Controlling Component Model

The controller is used to receive events caused by user operations in view, and these events will be assigned to model components or fed back for view components to deal with. There is no data processing in controller components and the main functions are to recognise, analyse, record, and assign events or requests from view and model components.

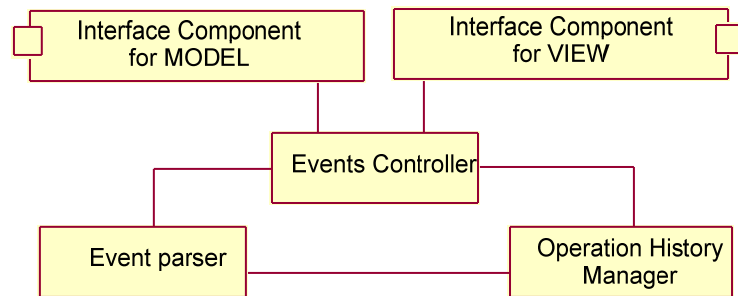


Figure 7. 6 Sub-architecture of Archware Controller

The *Events Controller* is used to receive events or feed back event processing results by the *Model Interface Component* and *View Interface Components*. It simply recognises the newly received events and delivers those events needing further analytical treatment to the *Event Parser* which will carry out a more detailed analytical processing of the event information. Finally the parsed event

will be assigned by the Event Controller.

The Event Parser is mainly used to carry out the necessary decomposition and conversion processes for events from view components and model components. For example, a primitive from a view component creates an event; during event-parser processing it will be converted into a request of a newly built model component element. At the same time, the event parser must still notify the Operation History Manager of events from user operations to track user operation history.

The Operation History Manager is mainly used to generate user tracking records. These operational records will be used to realise undo, redo and logbook functions of the general modelling environment. In some modelling tools, tracking of user operations is realised by saving a snap of the model data. The biggest drawback of this approach is the need to generate a model data snap for every user operation. Where model data is substantial this places heavy demands on system resources, so the time span over which operational history can be recorded is limited. Consequently this approach is not used in Archware, and the operational instructions (the Event Parser having transformed user operation events into operational instructions) sent by the Event Parser are recorded in a stack by the Operation History Manager. When the user withdraw event occurs, the operational instruction is popped from the stack and transformed into the corresponding reverse operation and handed to the Event Handler to assign model components to deal with it. This approach can effectively resolve the problem of excessive occupation of system resources caused by a requirement for snapshots.

The above description of Archware architecture gives the core components of the general modelling environment and describes the functions of components. The design focuses on the language framework of its basic modelling language, XMML. The overall goal is to provide a supporting environment with XMML, enabling both meta-modelling and modelling activities to be effectively realised

in an integrated visual modelling environment.

7.3 Modelling Environment Design for Archware

Archware is a general integrated modelling environment, its core functions are first to provide domain meta-model developers with the tools necessary to carry out domain meta-modelling and second to parse the meta-model and generate the modelling supporting environment needed for domain modelling. In practical project-development applications, there are four main phases to go through from target system spec. analysis to realisation of Model Driven design processes of the target system. These are: domain specification analysis of the target system, domain concepts analysis, meta-model design and target system design. The modelling activities of the two phases of “meta-model design” and “target system design” are completed using visual modelling and in the general integrated modelling environment Archware. As shown in Figure 7.7 and Figure 7.8 meta-modelling of domain-specific and domain application modelling can be carried out in Archware, which in a large part benefits from XMML design. The reason is that the meta-meta model, domain meta-model and domain application model are described by XMML language. So, Archware can adopt the same modelling language syntax parse mechanism to deal with the domain meta-model and domain application modelling, and dynamically construct the corresponding modelling environment according to the meta-meta model and the meta-model.

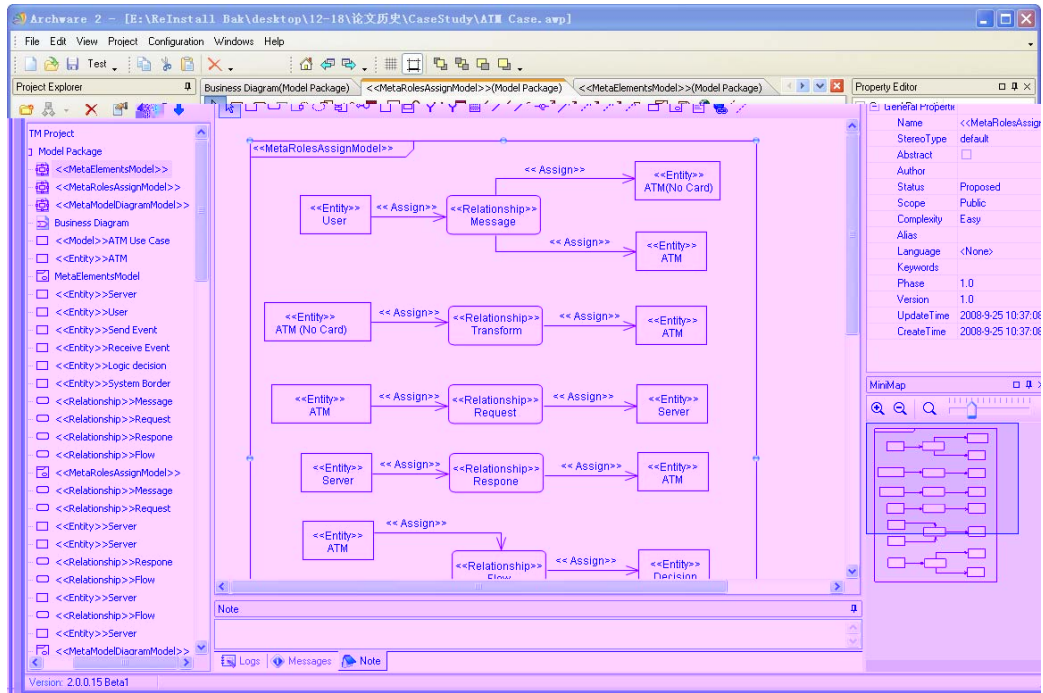


Figure 7.7 Realisation of Domain Meta-modelling in Archware

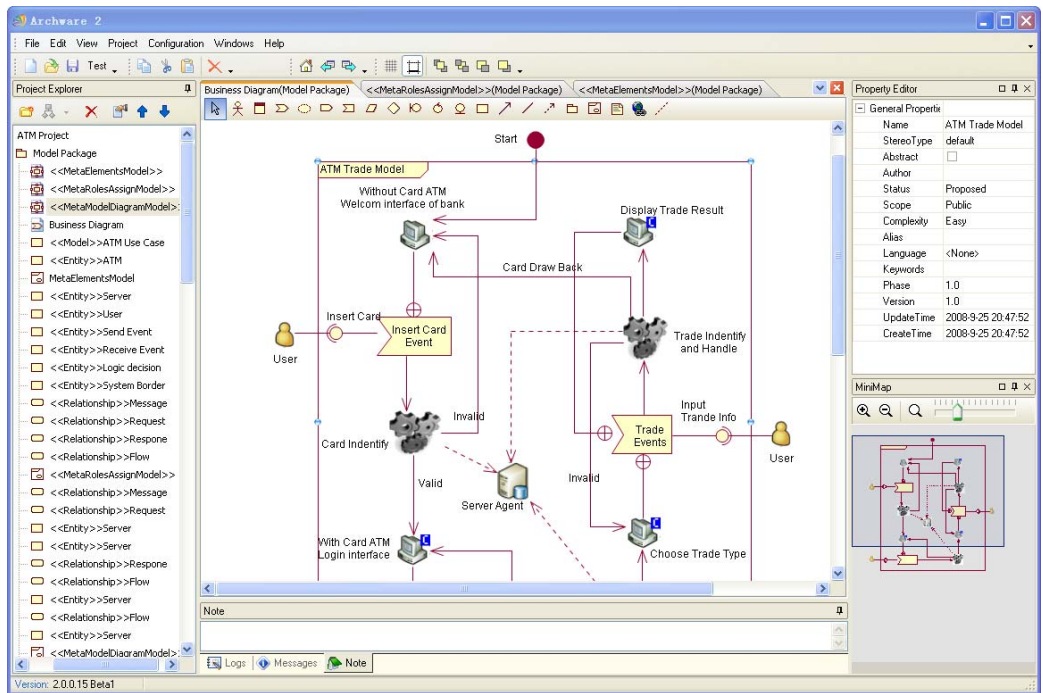


Figure 7.8 Realisation of Domain Modelling in Archware

The main task of the meta-modelling design phase is to describe and define the domain concepts and the relationships among them as produced in the preceding stage. The model designer does this using visual means. Finally the

domain application meta-model is obtained in the form of the domain-specific language model. In the meta-modelling infrastructure of Archware, a domain application is made up of six definition models. These are the: “MetaModel Element Define Model”, “MetaModel Diagram Define Model”, “MetaModel Entity Reference Relationship Define Model”, “MetaModel Relationship Roles Define Model”, “MetaEntities Relationship Definition model” and the “MetaEntities Refine Relationship Define Model” as shown in Figure 7.9.

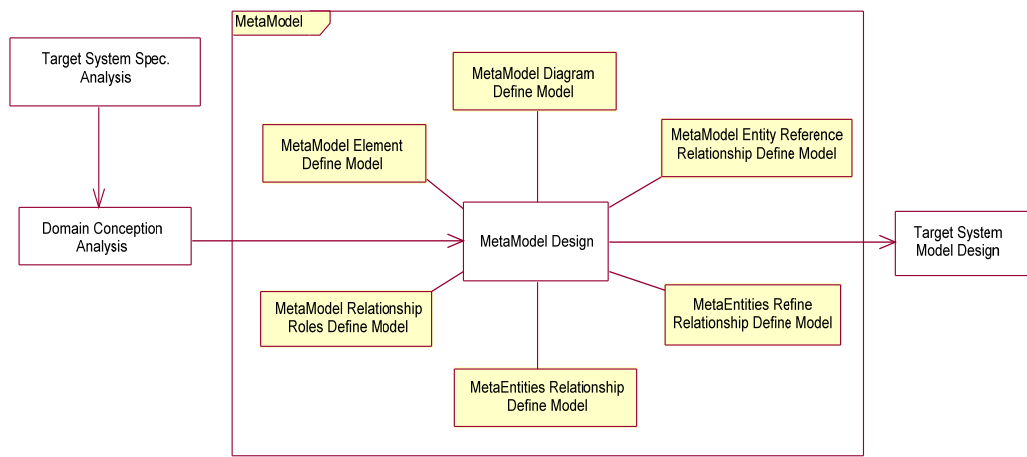


Figure 7. 9 Six Definition Models Involved in Meta-Model Design of Archware

Therefore, the meta-modelling process carried out in Archware is mainly a process of building the six definition models above and using these to respectively depict and describe the components of the domain-specific modelling language and their relationships.

In the target system design phase, domain application modelling is carried out in Archware. Its modelling environment will be automatically constructed by parsing the domain meta-model obtained by the proceeding meta-modelling activities. In the domain modelling process, if it is found that there are still some flawed meta-models, the meta-model design environment can be switched to modify or edit the meta-model, and then switched to the domain modelling environment. But before that, Archware will check the compatibility of the new

meta-model domain model previously designed by the model verification model. An example of an incompatibility that might be found would be where the modelling element used in the previous domain model does not exist in the new meta-model. Here, Archware will use a corresponding dialogue box to ask the user what measures are to be adopted to resolve the incompatibility between the domain application model and the meta-model.

A general visual model designer is provided to the modeller to create a visual modelling environment in Archware. Besides, a set of related supporting tools is also provided to construct an integrated modelling environment. These supporting tools include the editor designer for the modelling element attributes and the primitive designer for the modelling elements.

The property form designer for the modelling elements is mainly used at the meta-modelling phase and by meta-model developers. It can be used to design the corresponding modelling-elements property-editor with some complex attributes. The designer of the modelling-elements property-editor is similar to rapid development tools used in advanced language programs (such as Delphi, VB, etc.) as shown in Figure 7.10.

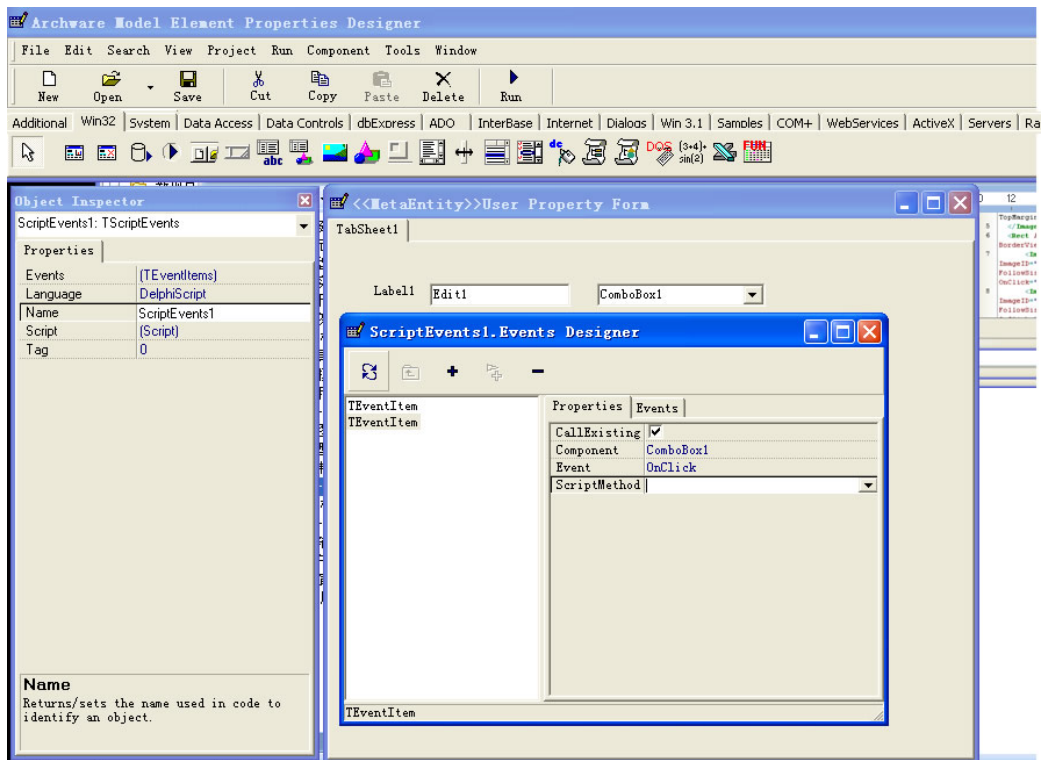


Figure 7. 10 Attribute Form Designer for Archware Modelling Element

The window-interface design area uses various frequently-used form controls, control attribute editors, etc. that are provided in the designer. Developers can design from the top, write programming code and debug code in the designer which provides very flexible tool support with meta-model developers when they design various modelling-element attributes editor-windows. The main difference compared with forms developed by common advanced languages is that the designed forms are executed in interpreted way. They do not need to be compiled in advance and can be executed in Archware. This is because the corresponding window-program virtual-machine is provided in Archware. This can dynamically interpretive execute-attribute editor-forms referred by modelling elements as shown in Figure 7.11.

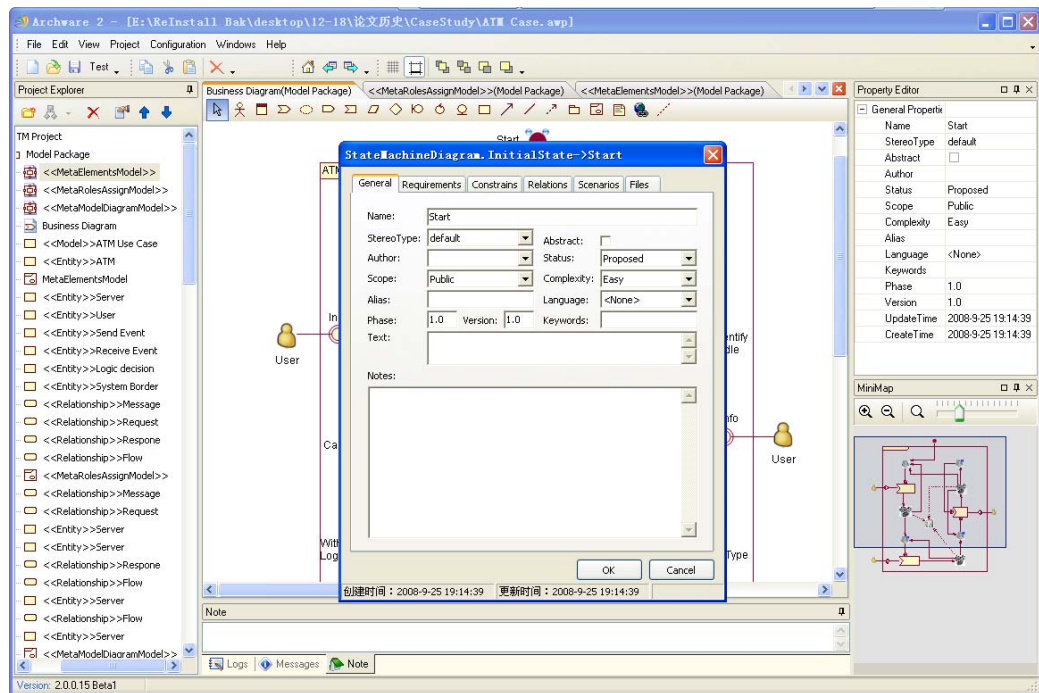


Figure 7. 11 Modelling Element Attributes Editor Form Interpretive Executed by Archware

The attribute editor window development will access the set of attributes that modelling elements have produced through the model reflection interface provided by the modelling environment infrastructure during running. In this way, the modelling-elements attributes-editor does not need to deal with persistent problems.

The modelling-elements graphy-designer is used to design corresponding visual primitives for each modelling element of the meta-model and to write event processing logic when primitives interact with users. The designer is made up of three parts: code editor of graphy physical appearance (shown in Figure 7.12), preview window of graphy physical appearance (shown in Figure 7.13) and script editor of primitive event (shown in Figure 7.14).

The code editor of primitive physical appearance is a tool similar to the text editor of HTML. It provides functions such as syntax highlighting, code completion and tag-matching, etc. The primitive preview window is used to see design results of description code of primitive physical appearance and provides

some of the appearance style adjustment functions, such as adjusting the colour of primitives' borders, text colours of primitives, primitives' background pictures and primitive style attributes. The primitive-events script-editor is used to write corresponding event processing script code for declared events in the nodes of primitives. It also provides functions such as syntax highlighting, code completion and syntax checking of script.

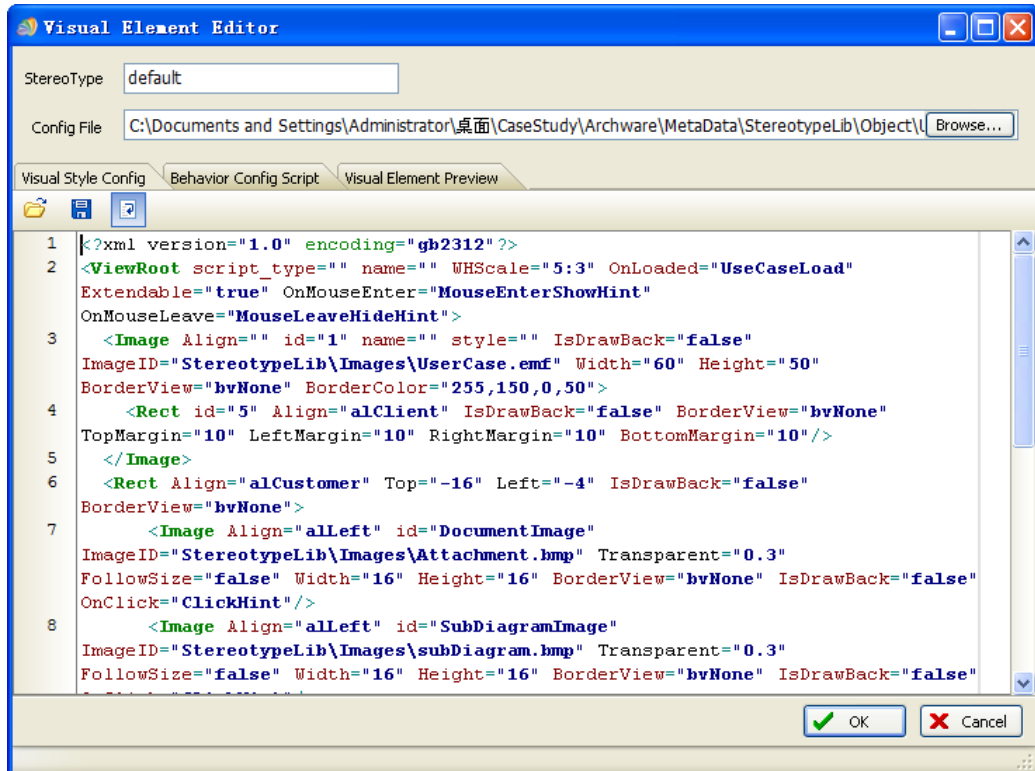


Figure 7. 12 Code Editor for Primitive Physical Appearance

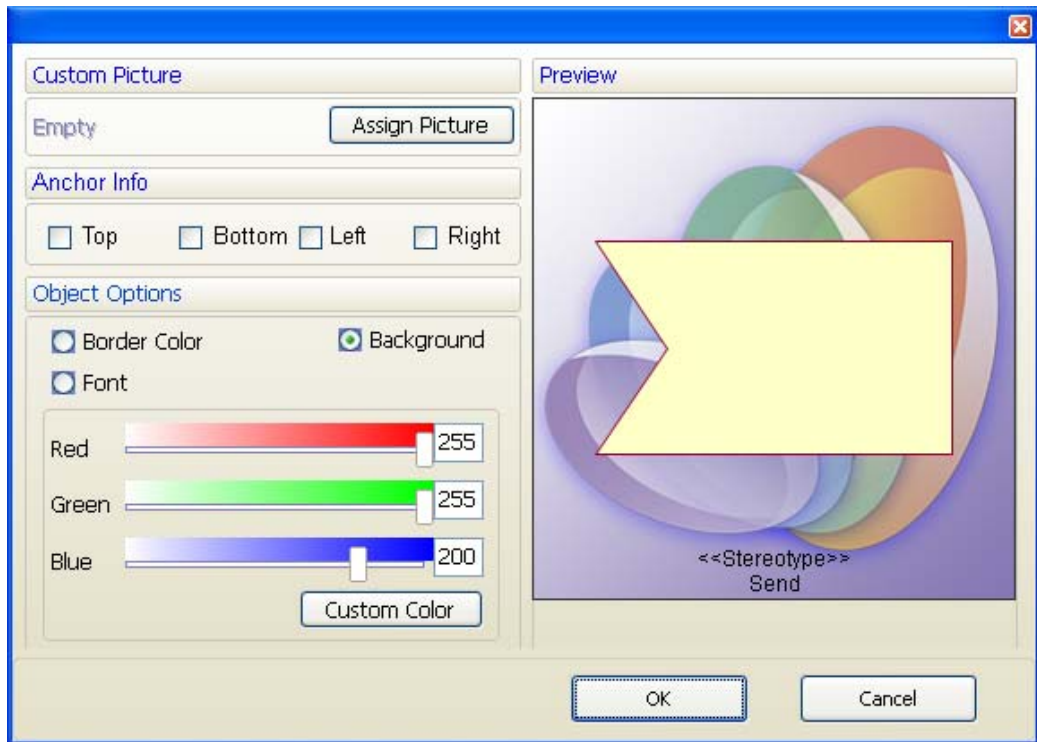


Figure 7. 13 Preview Window of Primitive Physical Appearances

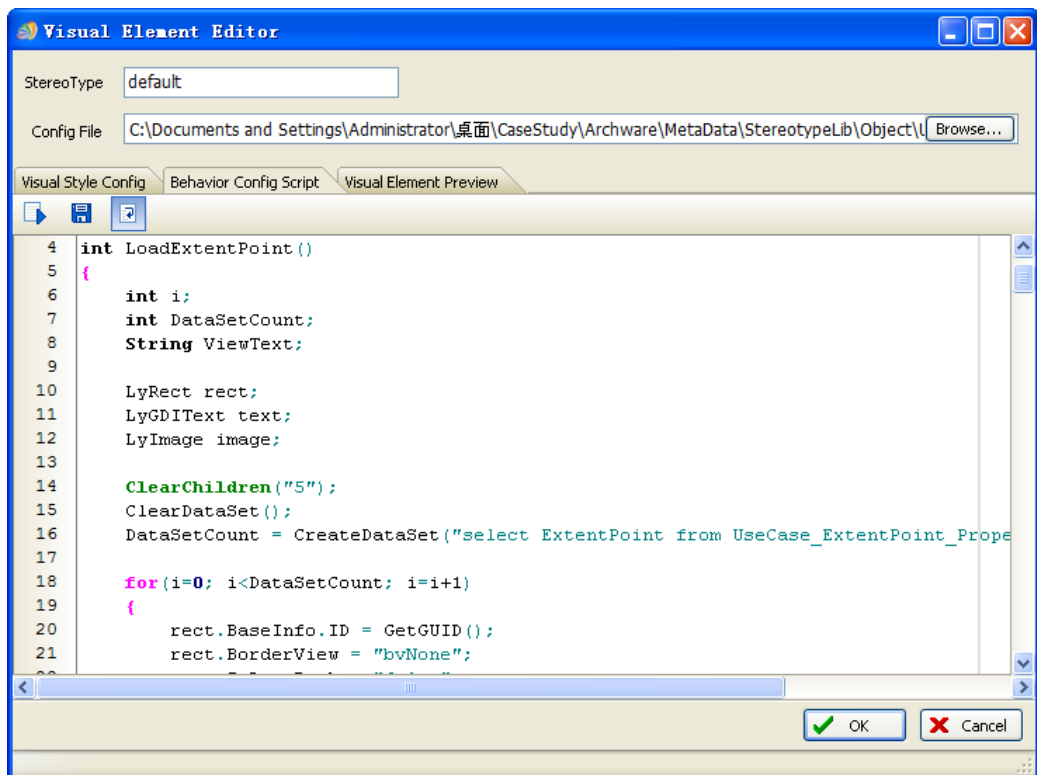


Figure 7. 14 Script Designer of Primitive Events

7.4 Summary

In this chapter, the development of modelling tools supporting Model Driven development is analysed and illustrated and current tools (GME, DSL Tools, EMF, MetaEdit+) supporting domain meta-modelling and domain application modelling are discussed. These are shown leading up to a general integrated modelling environment based on XMML. Meanwhile, the architecture of Archware and its design and realisation are given.

The architectural style of Archware uses MVC architecture to provide a supporting environment. Within this environment, XMML can be applied to realise domain-specific modelling. Meanwhile, meta-modelling and modelling activities can be effectively realised in an integrated visual modelling environment. In the chapter, the core components of the general modelling environment (view component, model component, controller component) are given together with a description of their functions.

The main function of the view component is to construct a modelling operational environment. For example it constructs corresponding tool bars, attribute editors, rendering engines for model visualisation, etc according to the meta-model. It also recognises user interaction events and feedback resulting from event-handling. However, it is not responsible for event handling.

The model component is an integral part of Archware. It is responsible for the implementation of meta-modelling and modelling processing logic together with some basic model entity data. The processing logic and model data encapsulated by the model component relate to external components. The model accepts data and action requests from both the view and the controller and then returns the final processed result.

The controller component is used to receive events initiated by user operations in the view and to assign these events to model components for

disposal or feedback to the view component to handle. The controller component does not carry out data operations. Its main role is in identifying, analysing, recording and assigning events or requests from the view and model components.

Archware's design focus is on the four main processes involved in the implementation of DSM methods. These are domain specification analysis of the target system, analysis of domain concepts, meta-model design and the design of the target system. Meanwhile, a group of related tools serve as the modelling tools and environmental support for these key processes. These are the meta-model designer, the domain model designer, the modelling-element attribute-editor designer and the modelling-element graph designer.

Chapter 8

Case Studies

The case studies in this chapter "ATM transaction processing systems" and "RPG Game Design" are examples of domain-specific modelling applications. Here, the main focus is on domain-specific modelling requirements together with the application of the visual meta-modelling language, XMML and its supporting environment to illustrate the visual modelling process and demonstrate these different types of domain applications.

8.1 An ATM Transaction Processing System

In this case, an ATM (Automatic Teller Machine) transaction processing system which is widely used in banking systems is used to demonstrate the use of DSM and XMML to realise a description of the meta-model of the domain application system and the domain application model.

8.1.1 Modelling Goal

For most ATM transaction processing systems, the interactive processes between the system and the user are very similar. These are card reading, password authentication, service choice, business processing, processing results feedback, exit etc. However, when we look at what is specific to the individual banks, there are some differences. These include differences in the cards, in the information content and format as displayed on the ATM, in the service items in the ATM and in the exchange interface between the ATM and bank's internal data systems. Therefore, if we develop ATM transaction processing systems for many banks, even if the business processing is the same, we still need to customise according to the requirements of the different users. With traditional

methods of development, developers need to locate the source code that must be changed and modify the program by hand until it meets users demands. This is an inefficient method and errors can be easily introduced due to negligence associated with manual processing.

To meet the requirements of such domain applications, domain-specific modelling (DSM) can be used to greatly improve efficiency and quality in domain application development. The solution is first to analyse these target domain systems to identify and extract details of the differences among these systems for use in modelling. Next, these differences are dealt with through a process of customising and adjusting the properties of modelling elements with the support of the code generators. The domain application model based on this can now be built as shown in Figure 8.1.

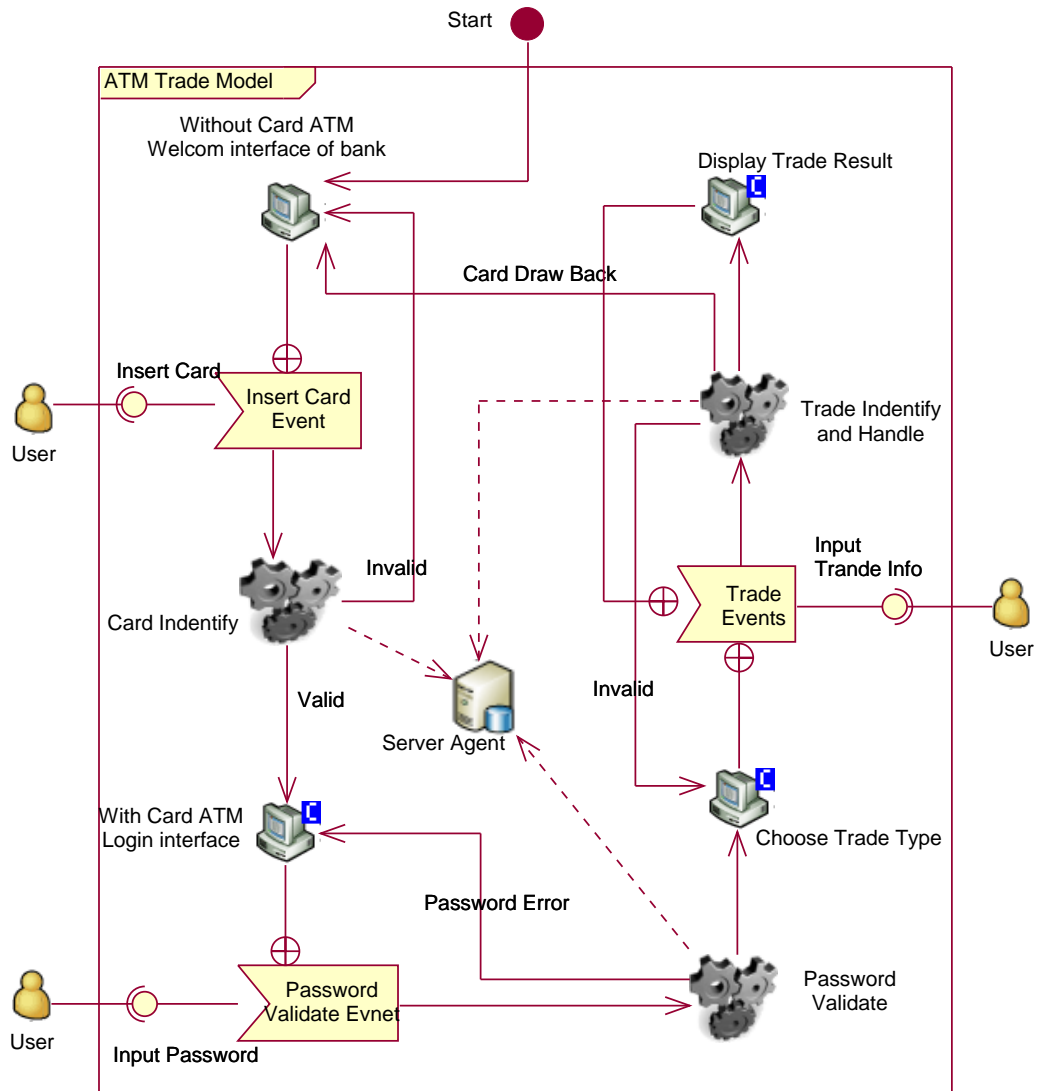



Figure 8. 1 Model of ATM Transaction Processing System

The model depicts the interactive processing business model dealing with transactions between the majority of the bank ATM transaction-processing systems and their users. The modelling element types in the model correspond with domain concepts in the target system as shown in Table 8.1.

A.		Depicts an I/O ATM device such as a monitor, keyboard, card reader etc. used to get users' operation and send back information to users.
----	---	--









B		Depicts an incident initiated by the user, such as insert card, enter the password etc.
C		Depicts an internal processing unit of the ATM, used to make an appropriate disposal of a user event.
D		Depicts proxy unit of ATM accessing host server, which completes ATM request for background data or services.
E		Depicts user operates ATM.
F		The relationship of Event-Capturing, which is the relationship between modeling type A and B.
G		The relationship of Event-Citing, which is the relationship between Modeling type E and B
H		The relationship of Message-passing, which is the relationship between modeling type B and C, C and A.
I		The relationship of service request, which is the relationship between modeling type C and D.

Table 8. 3 Meta-modelling Entity Element of ATM Transaction Processing System

Such a domain application model can easily make adjustments according to the different system requirements of different banks e.g. different welcome interfaces displayed on the screen can be achieved by adjusting or changing the "Welcome Interface of Bank" modelling object; different card types can be identified by adjusting or changing the "Card Identify" modelling object; different back-ground services can be achieved by adjusting or changing "Server Agent" etc. With the cooperation of code generators, these adjusted codes will be automatically generated.

8.1.2 Meta-Model Design

How can we build and describe the domain application model in Archware? The first step is to build a domain application meta-model. That is, to design the various types of modelling elements used in the domain application model by meta-model design as shown in Figure 8.1 which shows some of the entity elements. After this, we must deal with the types of association used in the domain application model.





F		Event-capturing relationship: the relationship between the modelling types A and B
G		Event-inciting relationship: the relationship between modelling types E and B
H		Information-passing relationship: the relationships between modelling types B and C and types C and A
I		Service request relationship: the relationship between modelling types C and D

Table 8. 4 Meta-modelling Associated Element of ATM Transaction Processing System

Besides, there is a modelling element of modelling type: “ATM transaction processing business model” and a graphic modelling element: “graphic of ATM transaction processing business”.

The definitions of these meta-modelling elements are realised through a “Definition Model of Meta-model Elements” as shown in Figure 8.2.

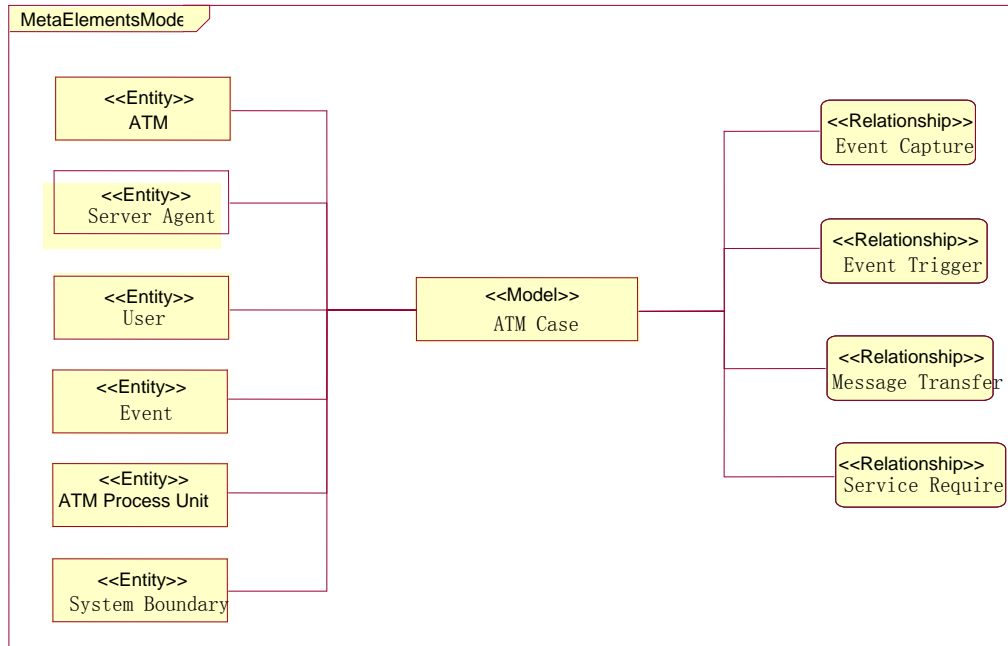


Figure 8. 2 Definition Model of Meta-model Elements

An XMML language fragment used in meta-modelling element definition is shown as follows:

```

<ModelsType>
  <Mode id = '...' type = '...'>
    <Entities>
      <EntityType id = '...' type = '...'>
        <RefinedModel></RefinedModel>
        <Attachment></Attachment>
        <Contained>
          <EntityType id = '...' type = 'TypeA'>
            <RefinedModel></RefinedModel>
            <Attachment></Attachment>
            <Contained></Contained>
            <Properties>
              <Name>ATM</Name>
              .....
            </Properties>
            <Events></Events>
            <Specification></Specification>
          </EntityType>
          .....
        </Contained>
      </Entities>
    </Mode>
  </ModelsType>

```

```

        <Properties>
            <Name> ATM Transaction Processing System </Name>
            .....
        </Properties>
        <Events></Events>
        <Specification></Specification>
    </EntityType>
    .....
</Entities>
<Relationships>
    <Relationship id = '...' type = '...'>
        <Roles>
            <Role type = '...' elementId = '...'>
                <Properties>.....</Properties>
                <Events></Events>
                <Specification></Specification>
            </Role>
            <Role type = '...' elementId = '...'>
                <Properties>.....</Properties>
                <Events></Events>
                <Specification></Specification>
            </Role>
        </Roles>
        <Events></Events>
        <Propertis>
        </Propertis>
        <Specification></Specification>
    </Relationship>
    .....
<Relationships>
<Diagrams>.....</Diagrams>
<Events>.....</Events>
<Properties>
    <Name>A meta-modelling entity element of ATM transaction
processing system </Name>
    .....
</Properties>
<Specifications></Specifications>
<CodeGenerators></CodeGenerators>
<RefEntities></RefEntities>
</Mode>
</ModelsType>

```

8.1.3 Relationships Among Meta Entity Elements

According to the specification and element definition model, we can see that there are no auxiliary relationships or inclusion relationships among elements and each element can appear alone in the model.

8.1.4 Role Definition Model of Meta-relationship Element

Role assigned relationships of elements in interface association are shown in Figure 8.3. “Role Definition Model of Meta-association Elements”.

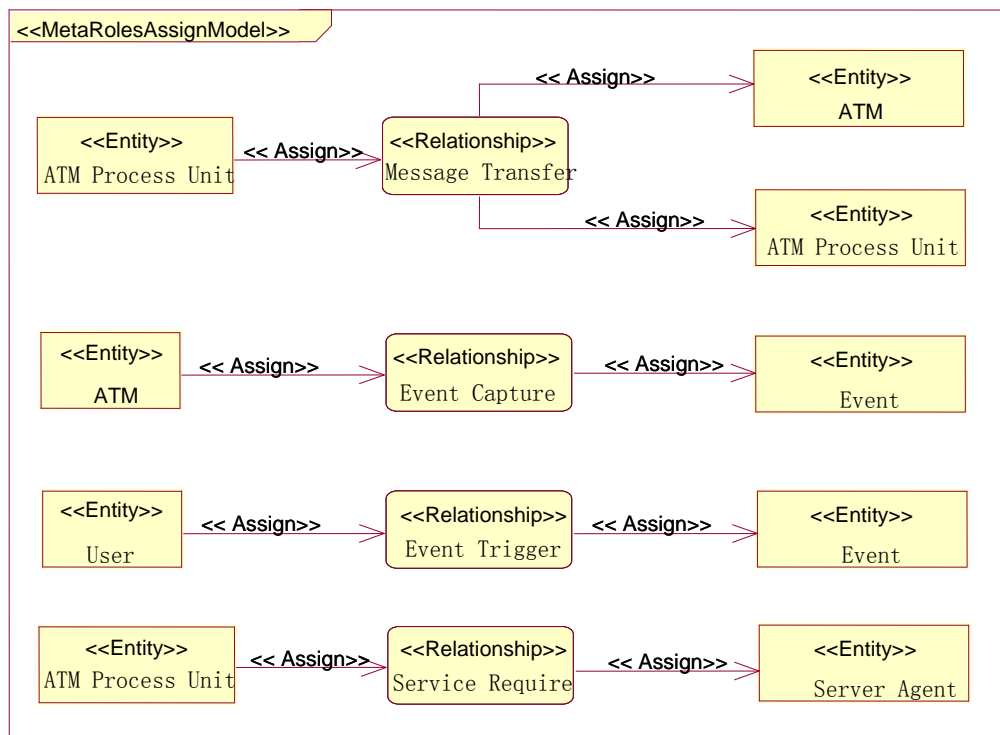


Figure 8. 3 Role Definition Model of Meta-association Element

An XMML language fragment used in the Role definition model of meta-association elements is shown as follows:

```

<ModelsType>
  <Mode id = '...' type = '...'>
    <Entities>
    
```

```

    <EntityType id = '...' type = 'TypeF'>
      <RefinedModel></RefinedModel>
      <Attachment></Attachment>
      <Contained></Contained>
      <Properties>
        <Name>event capture</Name>
        .....
      </Properties>
      <Events></Events>
      <Specification></Specification>
    </EntityType>
    .....
  </Entities>
  <Relationships>
    <Relationship id = '...' type = '...'>
      <Roles>
        <Role type = '...' elementId = '...'>
          <Properties>.....</Properties>
          <Events></Events>
          <Specification></Specification>
        </Role>
        <Role type = '...' elementId = '...'>
          <Properties>.....</Properties>
          <Events></Events>
          <Specification></Specification>
        </Role>
      </Roles>
      <Events></Events>
      <Propertis>
        <Name><<Assign>></Name>
        .....
      </Propertis>
      <Specification></Specification>
    </Relationship>
    .....
  <Relationships>
  <Diagrams>.....</Diagrams>
  <Events>.....</Events>
  <Properties>
    <Name> Role Definition Model of Meta-association Element of ATM
transaction processing system </Name>
    .....
  </Properties>
  <Specifications></Specifications>

```

```
<CodeGenerators></CodeGenerators>  
<RefEntities></RefEntities>  
</Mode>  
</ModelsType>
```

8.1.5 Definition Model of the Meta-model Diagram

Finally, definition of visual graphic reference relationships for modelling elements can be achieved by using a “definition model of the meta-model diagram” as shown in Figure 8.4.

In the above illustration, the corresponding visual diagrams define all kinds of meta-modelling elements. They are linked to their corresponding meta-modelling elements by reference associations. In visual modelling provided by the general modelling environment these visual diagrams are proxies of modelling element objects.

For example for a group of meta-model definition models for meta-model instances of ATM users' balance inquiries, the general modelling environment will get the information necessary for the construction of the domain modelling environment from the meta-model and automatically construct the modelling environment. An example of an ATM transaction processing system is given as follows.

We will take the following "*<<GraphicObject>> user*" and "*<<GraphicObject>> ATM*" as an example of giving definition descriptions for the modelling elements diagram.

```
<<GraphicObject>>User
```

```
<VisualElement id="" type='TypeE' elementId="" events='onloaded:fnOnLoaded ();'>
```

```
  <Div id="" style=' height:80px; width:120px; border:1px solid black; color:red;
```

```
background
```

```
:RES(img/User.bmp); text-align: center">
```

```
  </Div>
```

```
OnLoaded
```

```

        }
    </Script>
</VisualElement>

<<GraphicObject>>ATM
<VisualElement id="" type='TypeD' elementId="" events='onClick:fnOnClick ();'>
    <Div id="" style=' height:80px; width:120px; border:1px solid black; color:red;
background
:RES(img/ATM.bmp); text-align: center">
    </Div>
    <Script>
        int onClick (String id)
        {
            SetAttribute(id, "IsDrawBack", "true" );
            SetAttribute(id, "BackColor", "80, 0, 0, 255");
        }
    </Script>
</VisualElement>

```

An XMML language fragment used in meta-modelling diagram definition is shown as follows.

```

<ModelsType>
    <Mode id = '...' type = '...'>
        <Entities>
            <EntityType id = '...' type = '...'>
                <RefinedModel></RefinedModel>
                <Attachment></Attachment>
                <Contained>
                    <EntityType id = '...' type = 'GOTypeA'>
                        <RefinedModel></RefinedModel>
                        <Attachment></Attachment>
                        <Contained></Contained>
                    <Properties>

```



```

        <Name><<GraphicObject>>ATM</Name>
        .....
        </Properties>
        <Events></Events>
        <Specification></Specification>
    </EntityType>
    .....
</Contained>
<Properties>
    <Name><<Diagram>> ATM transaction processing system
</Name>
    .....
    </Properties>
    <Events></Events>
    <Specification></Specification>
</EntityType>
.....
</Entities>
<Relationships>
    <Relationship id = '...' type = '...'>
        <Roles>
            <Role type = '...' elementId = '...'>
                <Properties>.....</Properties>
                <Events></Events>
                <Specification></Specification>
            </Role>
            <Role type = '...' elementId = '...'>
                <Properties>.....</Properties>
                <Events></Events>
                <Specification></Specification>
            </Role>
        </Roles>
        <Events></Events>
        <Propertis>
            .....
        </Propertis>
        <Specification></Specification>
    </Relationship>
    .....
</Relationships>
<Diagrams>.....</Diagrams>
<Events>.....</Events>
<Properties>
    <Name> Definition Model of Meta-model Diagram of ATM transaction

```

```
processing system </Name>
    .....
    </Properties>
    <Specifications></Specifications>
    <CodeGenerators></CodeGenerators>
    <RefEntities></RefEntities>
  </Mode>
</ModelsType>
```

8.1.6 Summary

In this case study example, we see XMMML used, with the support of Archware modelling tools, to describe the domain related concepts and business processes of bank ATM transaction processing systems and to create the relevant meta-models. A simplified ATM transaction processing system is described by these meta-models.

8.2 RPG Game Design

8.2.1 Introduction to RPG

Role-Playing Game (RPG) is a plot-development oriented type of game. The player takes on one or more specific roles in the virtual world to play games within this special context. Each role has different capabilities according to different episodes of the game and statistical data (such as power, sensitivity, intelligence, magic, etc.) and these attributes will be changed in accordance with the rules of the game. Some game systems can be improved through such changes.

8.2.2 Specification of Target System

- (1) Leading role talks with ‘NPC1’ in the village.
- (2) If the leading role defeats ‘Ogre’, then the ‘NPC1’ will praise him for the good deed and give him ‘Prop A’. If the leading role has not defeated

‘Ogre’, then the ‘NPC1’ will tell him that he will get ‘Prop A’ only after defeating ‘Ogre’ and also tells him the probable location of ‘Ogre’.

- (3) Before commencing the talk with NPC1, the leading role must be face-to-face with NPC1 and so must be in map coordinates adjacent to NPC1.
- (4) The gain of prop will prompt player by message box.

8.2.3 Analysis of Domain Concept

- (1) Leading role: a main character in the game and controlled by the player to drive development of the action of the game.
- (2) NPC: a minor character in the game and controlled by the computer.
- (3) Prop: goods that the leading role of the game can use or equip.
- (4) Game logic engine: game logic calculation engine, is responsible for the calculation of coordinates of each entity in the game, change of game state and so on.
- (5) Message box: displays system information.
- (6) Game state: the state of key tasks of the game.
- (7) Conversation: the conversation between the leading role and a NPC in the game.
- (8) Map scene
- (9) Start, end
- (10) Association
 - (a) NPC query: to query NPC at given map coordinate, if there is NPC, the NPC objects will be returned, or return null.
 - (b) NPC inquiry response: inquiry response of game logic engine to NPC.

- (c) Game states query: query states of key tasks in the game.
- (d) Game states inquiry response: inquiry response of game logic engine to game state.
- (e) Request for getting equip: to request equip for logic engine.
- (f) Responses to the request of getting equip: the system information displays the request; the system message box is shown to map scene.
- (g) Common flow

8.2.4 Design of Meta-Model

(1) Definition model of meta-model element.

Following an analysis of the domain concepts, the domain concepts modelling element can be extracted. Nine basic modelling elements are as follows: 'start', 'end', 'leading role', 'NPC', 'props', 'message box', 'dialogue', 'scene map' and 'association'. There are eight relationships: 'NPC query', 'NPC response to the query', 'game status query', 'inquiry response to game state', 'requests for getting equipment', 'getting response to request for equipment', 'request for system information show' and 'common flow'. The meta-model element-definition model of the meta-meta model is used to define these as shown in Figure 8.5.

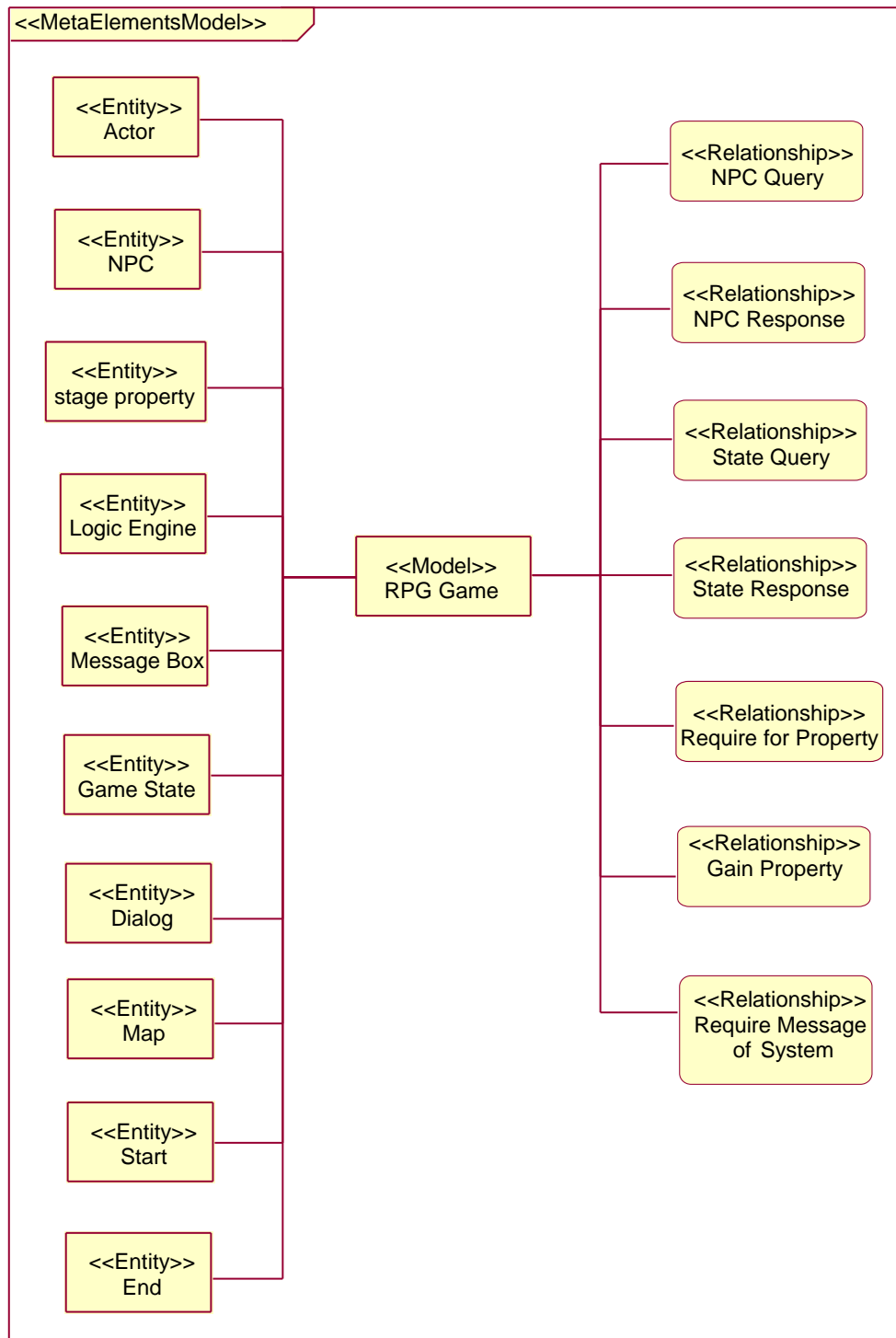


Figure 8. 5 Definition Model of Meta-model Elements

(2) Modelling definitions of attached relationships among meta-entity elements.

According to the specification and element definition model, we can see that

there is no attached relationship among the various elements and each element can appear singly in the model.

(3) Role definition model of meta-association element

The role of element interface-associations in assigning relationships is shown in Figure 8.6.

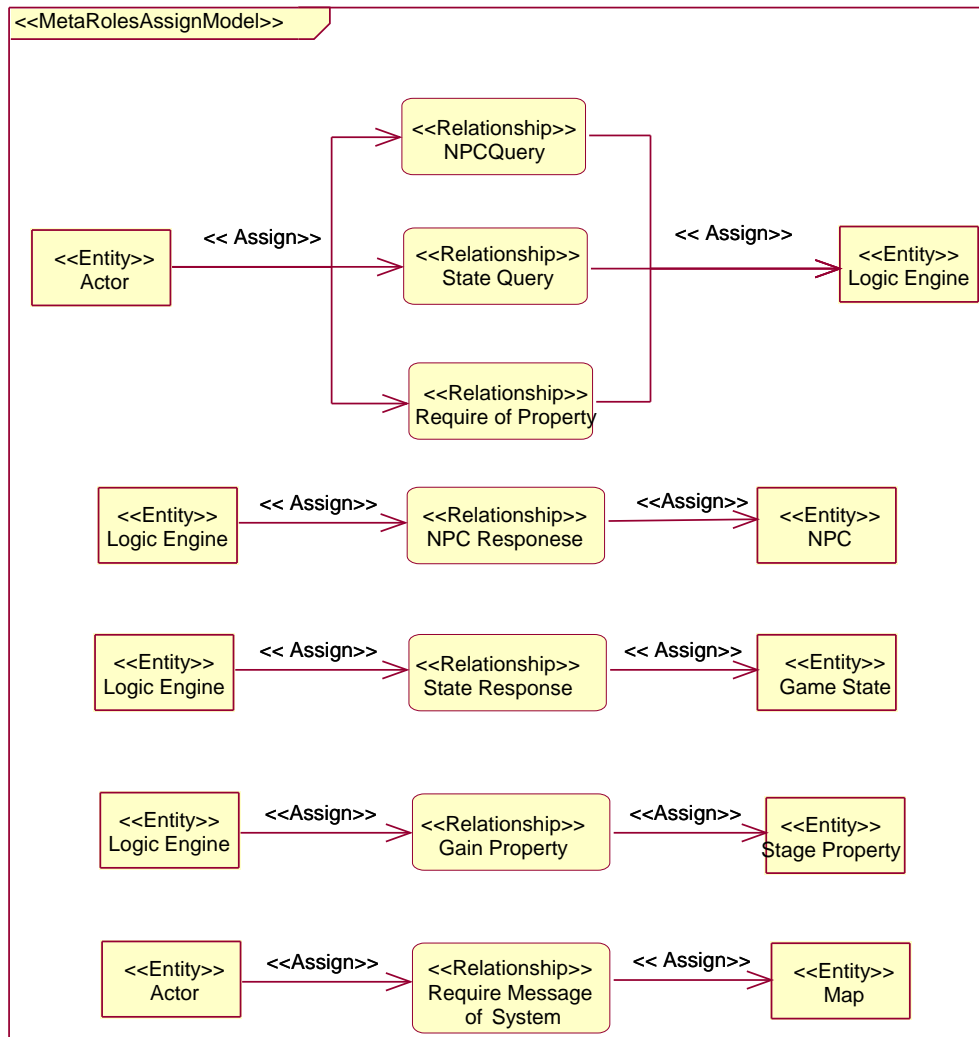


Figure 8. 6 Role Definition Model of Meta-association Elements

(4) Definition model of meta-model diagram

Finally, definitions of the reference relationships of the visual primitives of the modelling elements can be realised by using the “meta-model primitive

definition model” as shown in Figure 8.7.

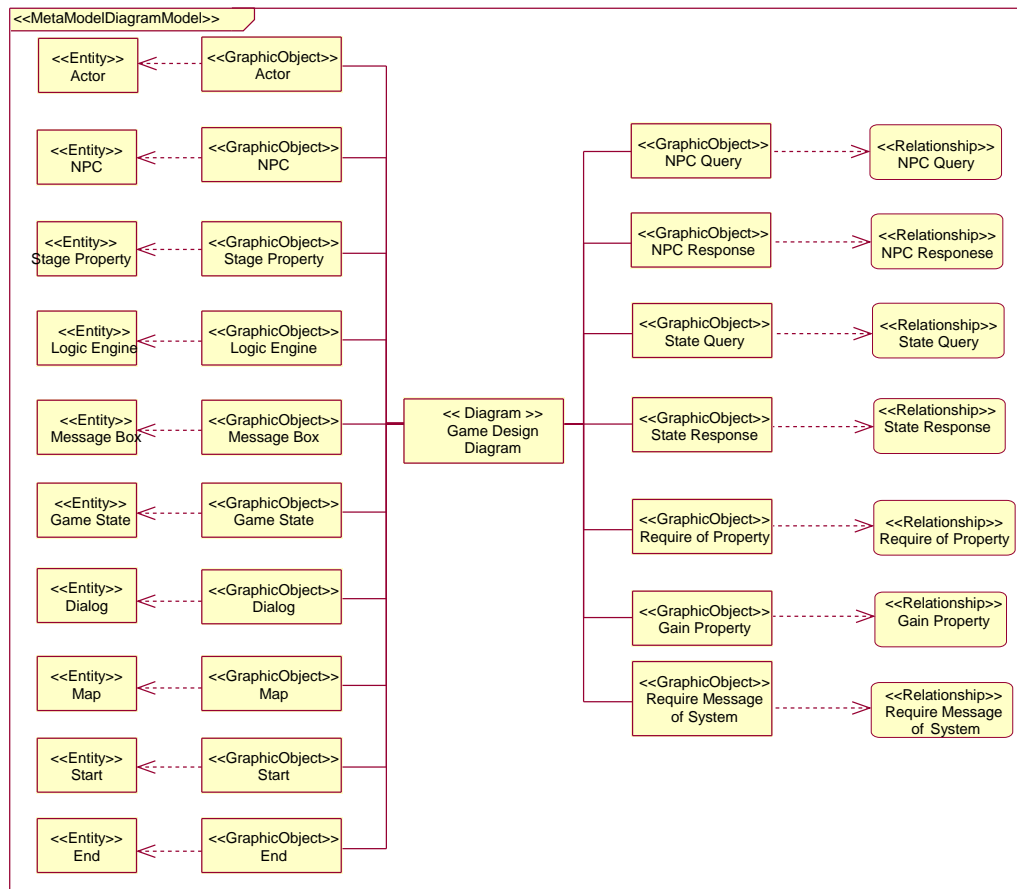


Figure 8. 7 Definition Model of Meta-Model Diagram

8.2.5 Design of Target System

Domain modelling of balance inquiries is carried out based on the various elements of RPG created above. See figure 8.8.

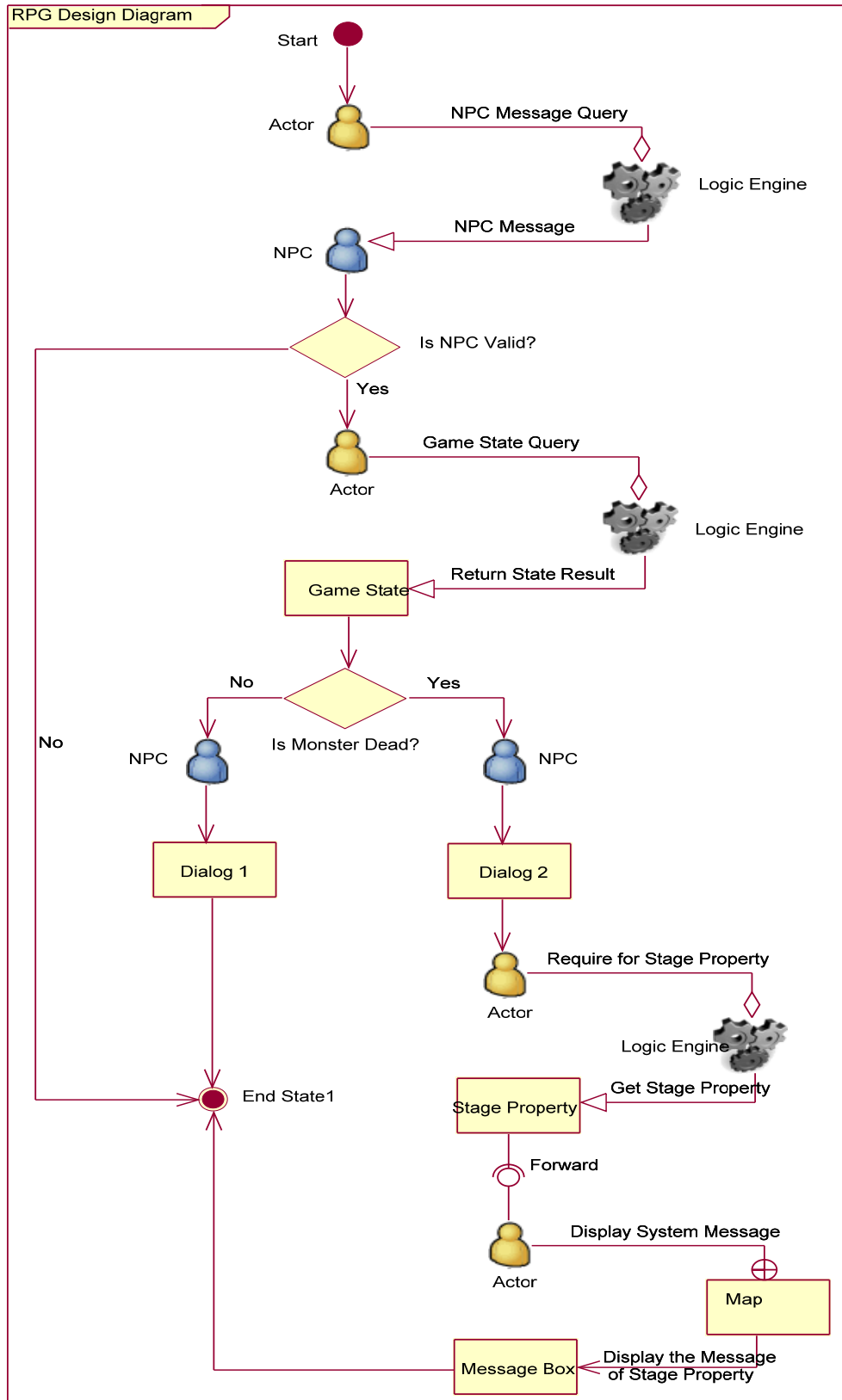


Figure 8. 8 Design of Target System

8.2.6 Summary

In this case study, we extracted a fragment of a game development process. With the support of Archware modelling tools, XMML language describes domain related concepts and the flow of the development process of this role-playing game.

8.3 Summary

In this chapter, several case-study applications are used to illustrate the visual modelling process and demonstrate domain-specific modelling requirements. They are examples of the use of the visual meta-modelling language, XMML together with its supporting environment to carry out visual modelling.

When DSM methods are used in Model Driven development, the first task is to determine the design target. Here, the domain analysis phase involves acquiring knowledge of a unique domain, abstract domain modelling concepts and technical terms together with an understanding of several appropriately sized business entities existing in the target domain system. Next come the design and construction of the domain meta-model. Here, the main task of the meta-model design phase is to describe and define the domain concepts together with the relationships between and among the concepts obtained during the preceding phase. This is achieved through visual means in the model designer. Finally come the domain-specific language model and the domain application meta-model. When using Archware to carry out meta-modelling, the domain application meta-model comprises six main definition models: the meta-model element-definition model, the meta-model diagram-definition model, the meta-model referred-relationship definition model, the meta-relationship element-role definition model, the meta-entity element-relationship definition model and the meta-entity meta-refinement definition model. The meta-modelling process carried out in Archware is represented by the process of

building the above six definition models in the model designer. Together they depict and describe the components of the domain-specific modelling language and the relationships between and among them. The target system design phase follows on after the domain meta-model has been finished. The domain application modelling activities in this phase can be carried out in Archware, its modelling environment will parse and construct the domain meta-model obtained from the preceding meta-modelling activities. If design flaws are discovered in the meta-model during the domain modelling process, modifications or editing can be carried out in the meta-model design environment. This enables timely verification of the results of meta-modelling and effectively improves modelling efficiency.

The above work finished the first step of model-driven development using DSM method, that is the domain model is established, which can be translated into code by code-generator, then the codes can be interpreted or compiled into executable code. By providing automation transform, the code generator realises the productivity and quality advocated by the DSM method. From the viewpoint of the modeller, generated code is complete. It means generated code is full, executable and that quality is ensured. That is to say, there is no need for the manual rewriting of code or for additional manual operations on the codes after code generation.

Compared to the traditional development methods, the DSM separates the business modeling of system and implementation technique, the creator of the domain model uses the meta-modeling language XMML and the visual modeling environment Archware to establish domain-specific meta-model, which is used to build domain model, so the modeling activities of domain-specific is completed, here, the creator of domain-model can be either traditional software developers or those users who master domain knowledge, as long as they mastered the domain knowledge, and correctly use the visual modeling language Archware, the creation of domain model will be finished by

them. In this way, the creators of domain-model can focus on business logic of system, and don't need to think the concrete implementation technique, so the domain trained personnel can be directly take part in the creation activities of domain model, and make the created domain model can correctly express the customers' practical needs, so the situation that the system can't correctly reflect customers' requirement which is caused by a different understanding between software developers and domain experts can be avoided, so the development efficiency can be improved.

The business modeling of the DSM focus on domain-specific modeling knowledge, and it doesn't need to think if the meta-model fits to the needs of other domain modeling, just as the above case studies, almost completely different meta-models are built by the meta-modeling activities of the two systems, these meta-models are mainly used for modeling of model in the domain, and they are not general, as such, the meta-modeling activities is much purer and simpler.

Chapter 9

Conclusion and Future Work

9.1 Summary of Thesis

This thesis puts forward a domain-specific visual meta-modelling language, XMML based on a study of Domain Specific Modelling (DSM) methodology. XMML aims to provide a concise, flexible, scalable and formal visual meta-modelling language as a basis for resolving problems with domain-specific modelling languages and the tools that support them. A new basis for developing meta-modelling language solutions is proposed to promote the rapid design and customisation of visual domain-specific modelling languages together with their corresponding supporting environments. It offers new theory and practice to boost the spread and application of Model Driven development methods. Based on research in domain-specific visual meta-modelling languages, the key technologies and main research results are as follows:

- (1) The focus of this analysis and research has led to the development of a new domain-specific visual meta-modelling language XMML. Its abstract syntax model has resulted from an analysis and study of the basic elements that lie at the core of a visual meta-modelling language. These include models, entities, relationships, model diagrams, visual elements and events.
- (2) A general concrete syntax based on this is used to describe a meta-meta model based on XMML together with a domain meta-model and a domain application model. The unified model reflection interface can be used to operate these different types of model.

- (3) A descriptive method is given for specifying domain modelling language semantics based on events together with methods of producing separate modelling elements, visual primitives and visual interactive descriptions. A solution to modelling language visualisation is provided together with solutions for achieving extensibility.
- (4) A meta-modelling infrastructure based on XMML is designed through a study of meta-modelling and its methods of implementation. Among these, a meta-meta model is taken as the basis for realising visual meta-modelling. Meanwhile its architecture leads to a design method for the domain-specific meta-modelling language and the meta-modelling process. Meta-meta models as described by XMML bring together meta-model entity elements, meta-model association elements and meta-model definition models. Five kinds of meta-model entity elements are given: model type, entity type, relationship type, diagram type and graphicObject type. Six kinds of relationship modelling element are given: possessed type, refined type, referred type, role-assigned type, attached type and contained type. Six kinds of meta-model definition models are given: meta-model element definition model, meta-model diagram definition model, entity reference relation definition model, meta-relation role definition model, definition model for relationships among meta-entities and definition model for refined relationships among meta-entities.
- (5) A general integrated modelling environment of MVC (Model/View/Controller) style is constructed based on an analysis of the development of tools supporting Model Driven development and some current general integrated modelling tools.

9.2 Conclusion

In recent years, Model Driven software development has been considered to be a further software construction technology following object-oriented software development methods and with the potential to bring new breakthroughs in the research of software development. With deepening research, a growing number of Model Driven software development methods have been proposed. The model is now widely used in all aspects of software development. One key element determining progress in Model Driven software development research is how to better express and describe the models required for various software components. OMG has been vigorously promoting UML and related technologies, but with little effect. It has become evident that a general modelling language, such as UML based on an object-oriented paradigm, cannot effectively and accurately express and describe various domain-specific business and software models at higher levels of abstraction. Those who seek to define a general modelling language and use it for modelling all problem domains, then teach experts in these domains how to use it to describe various domains, will not be effective. Modelling languages should have diversity and different applications need domain-specific modelling languages that suit them. Meanwhile, different software development organisations must also be able to define their own modelling languages and tools according to their own particular conditions. Therefore, the design and construction of modelling languages should be agile, economic and efficient. What's more, the development of modelling languages, design technologies and tools can be seen returning to the developers and domain experts. This is because only the creators and users of these models can really know what kind of modelling language they need. At present, Model Driven software development is still not mature relative to some other widely used software development technologies. It is still in the research and exploration phase so it seems too early at present to try to establish a general, standard modelling language. Meanwhile

overemphasis on a unified modelling language and standard Model Driven development can be detrimental to development and progress in this field of study.

Research and discussion in the thesis highlights distinctions between domain-specific modelling methods and the majority of existing Model Driven development methodologies. Domain-specific modelling methods can be successfully applied to actual software development projects, which need a flexible and easy to extend, meta-modelling language to provide support. There is a particular requirement for modelling languages based on domain-specific modelling methods in Meta-modelling as most general modelling languages are not suitable. At the same time, domain-specific modelling methods can improve abstract levels in Model Driven software development, further improve development efficiency in software development projects, reduce development costs for software projects and make full use of a variety of reusable assets accumulated over time by software development organisations. A meta-modelling language must be easy to use, extensible and support the necessary tools. It can be used expediently in Model Driven software development processes by development organisations of all sizes to finally achieve automated production of software systems. Such requirements would be difficult to achieve using a formal data modelling language based on mathematical theory.

The research results of this thesis offer a feasible solution to current problems. The thesis focuses on implementation of domain-specific modelling methods, combines this with engineering thinking and gives a general guiding implementation framework. Here, the emphasis is on object-oriented analysis, personnel organisation and layered design of software architecture for domain-specific modelling. The "domain" is stressed as a keystone of software design and development and this is what most differentiates the approach from general software development process and methods. This seeks to return the

focus of software development to problem domains and objective business processes and away from concerns of the software technology itself, and so effectively detect and address fundamental issues affecting software development.

Concerning the design of meta-modelling languages, the visual meta-modelling language XMML put forward in the thesis has a concise core design and is easy to extend and use. Its purpose is to realise rapid design and construction of various domain-specific modelling languages and to reduce development costs associated with application area modelling language and realisation difficulties in supporting tools. The design of XMML takes account of a wide range of language technologies and can effectively determine and realise visual descriptions, domain objects, descriptions of relationships and rules and can support model analysis and verification and does so with user-friendly technologies.

In the area of supporting tools, a meta-meta model designed on XMML is given. As an important part of the infrastructure of the general integrated modelling environment, the meta-meta model provides a group of general basic component meta-model elements together with the relationships between elements for the construction of the domain meta-model. It can support multi-view, multi-level description of the domain model. Developers or domain experts can complete the design and construction of the domain-specific meta-model and the domain application model in the integrated modelling environment, Archware.

9.3 Criteria for Success and Analysis

In domain-specific modelling development, use of the domain-specific visual meta-modelling language XMML put forward in the thesis and its general integrated modelling environment Archware can:

- (1) Enable the domain meta-model developer to rapidly and flexibly design a modelling language suited to target domain application by effectively improving the level of abstraction in system development and facilitating direct system modelling according to the domain concepts of the domain modelling users.
- (2) Support description of multi-view and multi-level. The model diagram is separated from the model entity in XMML and it becomes a meta-modelling element for the visual description of complex system models from different angles. Meanwhile, refined type association elements are defined according to the meta-meta model of XMML and can gradually decompose and refine a complex domain model and its modelling elements from different abstract levels.
- (3) Provide a formal definition mechanism for the semantics of modelling elements and modelling rules. In XMML, with the support of the general modelling environment, the executable definitions and descriptions of formal semantics are provided to the corresponding modelling element by programming event-elements during modelling and code generation.
- (4) Provide a flexible model reflection interface. The model reflection interface in the general integrated modelling environment can provide model structure, introspection of meta-modelling data and formal modelling development. It can dynamically provide the necessary context information to the domain-specific modelling language.
- (5) Support the definition of interactive behaviours between the flexible appearance of visual modelling elements and modelling. This is in order to provide specifications to the corresponding graphic render engine for visual representations of different domain modelling languages and to customise primitive nodes of main modelling elements and responding

logic for interactive events.

9.4 Future Work

The thesis has laid the foundation necessary for research in descriptive languages through further study in key technologies of visual meta-modelling languages based on Model Driven development. However, areas remain for improvement and further implementation. Further work could be fruitfully carried out as follows:

- (1) Study relating to application theory. The thesis focuses on the design and definition of domain-specific meta-modelling languages. There is room for further discussion and study on areas concerning practical applications of this theoretical basis. These could include the use of XMML to specify and define corresponding modelling to realise support of model evolution, verification and so on.
- (2) The research and realisation of code generators. For example, a study of how best to define and develop the appropriate code generators, based on domain model descriptions, to enable the domain application model to generate executable target language code.
- (3) Project Practice. Meta-modelling language research based on Model Driven development needs to interact well with engineering practice. It will be necessary to test the existing research results in engineering practice and conduct timely discussion on issues that arise in order to further improve the modelling language modelling and its mechanism.

References

- [1] K. Z. Ahmed and C. E. Umrysh, *Developing Enterprise Java Applications with J2EE and UML*, Addison-Wesley, 2001.
- [2] A. Alderson, “Experience of Bi-Lateral Technology Transfer Projects”, *Proceedings of the 2nd IFIP WG8.6 Working Conference on Diffusion, Transfer and Implementation of Information Technology*, Technical Report SOCTR, Staffordshire University, 1997.
- [3] P. Allen and S. Frost, *Component-Based Development for Enterprise Systems: Applying the SELECT Perspective*, Cambridge University Press, 1998.
- [4] J. Bentley, “Programming Pearls: Little Languages”, *Communications of the ACM*, Vol. 29, No. 8, pp. 711-721, 1986.
- [5] P. A. Bernstein, “Middleware: a Model for Distributed System Services”, *Communications of the ACM*, Vol. 39, No. 2, pp. 86-98, 1996.
- [6] S. Beydeda, M. Book and V. Gruhn, *Model Driven Software Development*, Springer, 2005.
- [7] J. Bezivin, O. Gerbe, “Towards a Precise Definition of the OMG / MDA Framework”, *Proceedings 16th Annual International Conference on Automated Software Engineering*, pp. 273- 280, 2001.
- [8] G. Booch, J. Rumbaugh and I. Jacobsen, *The Unified Modeling Language User’s Guide*, Addison-Wesley, 1998.
- [9] G. Booch, A. Brown, S. Iyengar, J. Rumbaugh and B. Selic, “An MDA Manifesto”, *Business Process Trends/MDA Journal*, May 2004.
- [10] F. P. Brooks and H. J. Kugler, “No Silver Bullet: Essence and Accidents of Software Engineering”, *Information Processing 86*, Amsterdam: Elsevier

References

Science, (North Holland), pp. 1069-1076, 1986.

[11] A. Brown, "MDA and Today's System",

<http://www.ibm.com/developerworks/cn/rational/r-mda/>, 2005.

[12] Y. Chen, W. Qiu, X. Du and C. Peng, "Model Driven Development Methodology for Embedded Systems", *Journal of Computer-Aided Design & Computer Graphics*, Vol. 18, No. 2, 2006.

[13] T. Clark, A. Evans, P. Sammut and J. Willans, "Applied Metamodelling: A Foundation for Language Driven Development", www.xactium.com, 2004.

[14] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*, Addison-Wesley, 2001.

[15] "Code Generation Network", www.codegeneration.net.

[16] S. Cook, G. Jones, S. Kent and A. C. Wills, *Domain-Specific Development with Visual Studio DSL Tools*, Addison Wesley, 2007.

[17] K. Czarnecki and U. W. Eisenecker, *Generative Programming: Methods, Tools, and Applications*, ACM Press / Addison-Wesley, 2000.

[18] A. V. Deursen, P. Klint and J. Visser, "Domain-Specific Languages: An Annotated Bibliography", *ACM SIGPLAN Notices*, Vol. 35, No. 6, pp.26-36, 2000.

[19] E. W. Dijkstra, "The Humble Programmer", *Communications of the ACM*, Vol. 15, pp. 859-866, 1972.

[20] S. Dmitriev, "Language Oriented Programming",

<http://www.onboard.jetbrains.com/is1/articles/04/10/lop/6.html#con>.

[21] S. Dmitriev, "Details of LOP",

<http://www.onboard.jetbrains.com/is1/articles/04/10/lop/3.html#dlop>.

[22] Dsmforum, "Enterprise Apps in Smartphones",

<http://www.dsmforum.org/phone.html>.

[23] Dsmforum, “Programming Microcontrollers”,

<http://www.dsmforum.org/voicemenu.html>.

[24] A. Endres and H. Weber, “Software Development Environments and CASE Technology”, *Proceedings of European Symposium on Software Development Environments and CASE Technology*, Knigswinter, Germany, Vol. 509, pp. 81–91, June 1991.

[25] E. Engstrom and J. Krueger, “Building and Rapidly Evolving Domain-Specific Tools with DOME”, *Proceedings of IEEE International Symposium on Computer-Aided Control System Design*, pp. 83-88, 2000.

[26] S. Fan and N. Zhang, “Survey of Generative Programming”, *Computer Science*, Vol. 32, No. 3, 2005.

[27] G. Froehlich, H. J. Hoover, W. Liew and P. G. Sorenson, “Application Framework Issues when Evolving Business Applications for Electronic Commerce”, *Information Systems*, Vol. 24, No. 6, pp. 457-473, 1999.

[28] M. Fowler, "Language Workbenches: The Killer-App for Domain Specific Languages?", <http://martinfowler.com/articles/languageWorkbench.html>.

[29] K. Frank, “Domain Specific Languages vs. UML”,

<http://www.codegear.com/tw/downloads>.

[30] R. Franklin, R. Jacques and S. Ulrich, “Concurrent Transaction Frame Logic Formal Semantics for UML Activity and Class Diagrams”, *Electronic Notes in Theoretical Computer Science*, Vol. 95, pp. 83-109, May 2004.

[31] L. Fuentes, M. Pinto and A. Vallecillo, “How MDA Can Help Designing Component and Aspect-based Applications”, *Proceedings of Seventh IEEE International Enterprise Distributed Object Computing Conference*, pp.124~135, 2003.

References

- [32] E. Gamma, R. Helm, R. Johnson and J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Massachusetts, 1995.
- [33] H. Gao, “An Overview of AOP”, *Computer Science*, 2002.
- [34] T. Gardner, C. Griffin, J. Koehler and R. Hauser, “A review of OMG MOF 2.0 Query / View / Transformation Submissions and Recommendations Towards the Final Standard”, *In 1st International Workshop on Metamodeling for MDA*, York, UK, 2003.
- [35] M. P. Gervais, “Towards an MDA-Oriented Methodology”, *Proceedings of the 26th International Computer Software and Applications Conference on Prolonging Software Life: Development and Redevelopment*, pp.265-270, 2002.
- [36] J. Greenfield and K. Short, “Moving to Software Factories”, Visual Studio Team System, Microsoft Corporation.
- [37] J. Greenfield and K. Short, *Software Factories Assembling Applications with Patterns, Models, Frameworks and Tools*, John Wiley & Sons, 2004.
- [38] Gronback, “Richard Podcast on GMF”,
<http://eclipsezone.com/files/podcasts/1-GMF-Richard.Gronback.html>, 2006.
- [39] Ø. Haugen and P. Mohagheghi, “A Multi-Dimensional Framework for Characterizing Domain Specific Languages”, *Proceeding of the 7th OOPSLA Workshop on Domain-Specific Modeling*, Motreal, Canada, 2007.
- [40] J. Heering and P. Klint, “Semantics of Programming Languages: A Tool-Oriented Approach”, *ACM SIGPLAN Notices*, Vol. 35, No. 3, pp. 39-48, March 2000.
- [41] Y. Hu, S. Zhang, M. Wang and H. Zhao, “Research on Model Driven Customization and Release of CAPP Information”, *Machine Tool & Hydraulics*, Vol. 35, No. 6, 2007.

References

- [42] Q. Huang, C. Chai, “A Meta Model Model Driven Architecture Tool Based on SACRED”, *Journal of Zhejiang University (Engineering Science)*, Vol.41, No. 9, 2007.
- [43] J. Hou, J. Wan and S. Wang, “A Model Driven WEB Application Development Method”, *Journal of Sichuan University (Engineering Science Edition)*, Vol. 39, 2007.
- [44] L. Ji, “Model Driven Web Services Composition”, *Computer Engineering*, Vol. 33, No. 18, 2007.
- [45] JetBrains, “Meta Programming System”, <http://www.jetbrains.com/mps/>.
- [46] H. Jiang, D. Lin and X. Xie, “The Formal Semantics of UML State Machine”, *Journal of Software*, Vol. 13, No. 12, pp. 2244-2250, 2002.
- [47] S. Kelly, L. Kalle and R. Matti, “MetaEditp: A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment”, *Proceedings of the 8th International Conference on Advanced Information Systems Engineering(CAiSE'96)*, Heraklion, Crete, Greece, pp. 1–21, 1996.
- [48] S. Kelly and J. P. Tolvanen, *Domain-Specific Modeling: Enabling Full Code Generation*, John Wiley & Sons Inc., 2008.
- [49] S. Kelly and J. P. Tolvanen, “Visual Domain-Specific Modeling: Benefits and Experiences of Using MetaCASE Tools”, *International Workshop on Model Engineering(ECOOP 2000)*, Cannes, France, June 2000.
- [50] K. A. Kettler, J. P. Lehoczky and J. K. Strosnider, “Modeling Bus Scheduling Policies for Real-Time Systems”, *Proceedings of the 16th IEEE Real-Time Systems Symposium*, Pisa, pp.242-253, 1995.
- [51] A. Kleppe, J. Warmer and W. Bast, *MDA Explained: The Model Driven Architecture: Practice and Promise*, Addison-Wesley Professional, Revised & Revised edition, 2003

- [52] A. Kleppe and J. Warmer, “Unification of Static and Dynamic Semantics of UML, a Study in Redefining the Semantics of the UML Using the pUML OO Meta Modeling Approach”, Technical Report, Klasse Objecten, Available at <http://www.klasse.nl/papers/unification-report.pdf>, July 2001.
- [53] D. E. Knuth, “Backus Normal Form vs. Backus Naur Form”, *Communications of the ACM*, Vol. 7, No. 12, pp. 735-736, 1964.
- [54] D. E. Knuth, “Literate Programming”, *The Computer Journal*, Vol. 27, No. 2, pp.97-111, May 1984.
- [55] T. Kosar, P. M. Lopez, P. A. Barrientos and M. Mernik, “A Preliminary Study on Various Implementation Approaches of Domain-Specific Language”, *Information and Software Technology*, Vol. 50, No. 5, pp. 390-405, 2008.
- [56] Q. Lan, “Research on the Key Technology of Executable Meta-Model”, *Doctoral Thesis of Jilin University*, April 2006.
- [57] A. Ledeczi, A. Bakay, M. Maroti, P. Volgysei, G. Nordstrom, J. Sprinkle and G. Karsai, “Composing Domain-Specific Design Environments”, *IEEE Computer*, Vol. 34, No.11, pp. 44–51, November 2001.
- [58] K. Li, Z. Chen, H. Mei and F. Yang, “An Introduction to Domain Engineering”, *Computer Science*, Vol. 26, No. 5, 1999.
- [59] H. Liu, Z. Ma and W. Shao, “Research on Meta-Modeling Technology”, *Journal of Software*, Vol. 19, pp. 1317-1327, 2008.
- [60] S. Lohr, *the Programmers Who Created the Software Revolution*, Basic Books, October 15, 2002.
- [61] H. Ma, B. Xie, Z. ma, N, Zhang and W. Shao, “PKUMoDEL: A Model Driven Development Environment for Languages Family”, *Journal of Computer Research and Development*, Vol. 44, No.4, 2007.
- [62] S. J. Mellor, A. N. Clark and T. Futagami, “Guest Editors' Introduction:

- Model Driven Development”, *IEEE Software*, Vol. 20, No. 5, pp.14-18, 2003.
- [63] S. Mellor and M. Balcer, *Execable UML: A Foundation for Model Driven Architectures*, Addison-Wesley, 2002.
- [64] M. Mernik, J. Heering and A. M. Sloane, “When and How to Develop Domain-Specific Languages”, *ACM Computing Surveys*, Vol. 37, No. 4, pp. 316-344, December 2005.
- [65] “MetaCase, Success Stories”, <http://www.metacase.com/cases/index.html>.
- [66] MetaEdit Inc., “Domain-Specific Modeling with MetaEdit+ 10 Times Faster than UML”, White Paper, 2005.
- [67] B. Meyer, C. Mingins and H. Schmidt, "Providing Trusted Components to the Industry," *IEEE Computer*, Vol. 31, No. 5, pp. 104-105, 1998.
- [68] J. Miller and J. Mukerji, “MDA Guide Version 1.0.1”, *OMG*, <http://www.omg.org/docs/omg/03-06-01.pdf>, 2003.
- [69] C. Mosher, S. Microsystems, “New Specification for Managing Meta-Data”, *Development of Java Technology and Application*, Chinese Association of Automation, pp.9~16, 2003.
- [70] H. Mouratidis, P. Giorgini and G. A. Manson, "Integrating Security and Systems Engineering: Towards the Modelling of Secure Information Systems," *Proceedings of 15th International Conference on Advanced Information Systems Engineering(CAiSE 2003)*, Klagenfurt, Austria, Vol.2681, pp.63–78, June 2003.
- [71] D. R. Musser, G. J. Derge and A. Saini, *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*, Addison-Wesley, 2001.
- [72] OMG, “Meta Object Facility Specification”, Version 1.3.
- [73] OMG, “Unified Modeling Language Specification”, Version2, <http://www.omg.org/uml>, 2003.

References

- [74] OMG, “XML Metadata Interchange Specification”, Version 1.2, <http://www.omg.org/xmi>, 2003.
- [75] OMG, “the Common Warehouse Metamodel Specifications”, <http://www.cwmforum.org>, 2003.
- [76] OMG, <http://www.omg.org/mda>.
- [77] OMG, “MDA Guide Version1.0”, <http://www.omg.org/cgi-bin/doc?mda-guide>.
- [78] J. Osis, E. Asnina, “Enterprise Modeling for Information System Development within MDA”, *Proceedings of the Annual Hawaii International Conference on System Sciences*, Waikoloa, Big Island, Hawaii, pp. 490–501, 2008.
- [79] J. Osis, E. Asnina, A. Grave, “MDA Oriented Computation Independent Modeling of the Problem Domain”, *Proceedings of the International Working Conference on Evaluation of Novel Approaches to Software Engineering*, Barcelona, Spain, pp. 66–71, 2007.
- [80] Y. Papakonstantinou, H. Garcia-Molina and J. Ullman, "MedMaker: A Mediation System Based System Based on Declarative Specifications", *Proceeding of the 12th International Conference on Data Engineering*, New Orleans, LA, USA, pp. 132-141, 1996.
- [81] D. L. Parnas, “On the Design and Development of Program Families”, *IEEE Transactions on Software Engineering*, Vol. 2, No.1, pp. 1-9, 1976.
- [82] S. Parr and R. Keith-Magee, “How to Apply MDA to Simulation”, *Simulation Technology and Training Conference*, 2004.
- [83] S. Parr and R. Keith-Magee, “The Next Step-Applying the Model Driven Architecture to HLA”, *Proceedings of Spring Simulation Interoperability Workshop*, pp. Id 03S-SIW-123, 2003.

References

- [84] J. J. Petro, M. E. Fotta, D. B. Weisman and P. James, "Model-Base Resue Repository-Concepts and Experience", *Proceedings of the Seventh International Workshop on Computer-Aided Software Engineering*, pp. 60-69, 1995.
- [85] J. D. Poole, "Model Driven Architecture: Vision, Standards And EMerging Technologies", *Position Paper Submitted to ECOP 2001 Workshop on Metamodeling and Adaptive Object Models Hyperion Solutions Corporation*, 2001.
- [86] A. Radeski, S. Parr, R. Keith-Magee and J. Wharington, "Component-Based Development Extensions to HLA", *Proceedings of the 2002 Spring Simulation Interoperability Workshop*, pp. 02S-SIW-046, March 2002.
- [87] X. Ren and Y. Chang, "Discuss of Model Driven Architecture", *Computer Knowledge and Technology*, No 13, 2007.
- [88] L. Réveillère, F. Mérillon, C. Consel, R. Marlet and G. Muller, "A DSL Approach to Improve Productivity and Safety in Device Drivers Development", *Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE,2000)*, Grenoble, France, 2000.
- [89] H. Sang, X. Shen, "Technology of Java-on-Silicon", *Computer Science*, Vol. 28, No. 4, 2001.
- [90] A. J. Sánchez-Ruiz, M. Saeki, B. Langlois and Roberto Paiano, "Domain-specific Software Development Terminology: Do We All Speak the Same Language?", *Proceedings of the seventh OOPSLA Workshop on Domain-Specific Modeling*, ACM-SIGPLAN, October 2007.
- [91] R. Seeley, "ADT at Gartner ITxpo: Gates sees More Modeling, Less Coding", *Application Development Trends* 3/30/2004, <http://www.adtmag.com/article.aspx?id=9166>, 2004.
- [92] B. Selic, "The Pragmatics of Model Driven Development", *IEEE Software*,

Vol. 20, No. 5, pp. 19-25, 2003.

[93] C. Simonyi, M. Christerson and S. Clifford, “Intentional Software”, *Proceedings of ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Portland, Oregon, USA, October 2006.

[94] C. Simonyi, “Anything You Can Do, I Can Do Meta”, <http://www.technologyreview.com/Infotech/17969/?a=f>, January 2007.

[95] K. V. Slooten and S. Brinkkemper, “A Method Engineering Approach to Information Systems Development”, *Proceedings of the IFIP WG8.1 Working Conference on Information System Development Process*, pp. 167-186, 1993.

[96] K. Smolander, “OPRR: a Model for Modelling Systems Development Methods”, *Proceedings of the Second Workshop on Next Generation of CASE Tools*, pp. 224–239, May 1991.

[97] K. Smolander, K. Lyytinen, V. P. Tahvanainen and P. Marttiin, “MetaEdit: a Flexible Graphical Environment for Methodology Modeling”, *Proceedings of the Third International Conference on Advanced Information Systems Engineering*, Trondheim, Norway, pp.168–193, 1991.

[98] J. Sprinkle and G. Karsai, “A domain-specific Visual Language for Domain Model Evolution”, *Journal of Visual Languages and Computing*, Vol.15, pp. 291-307, 2004.

[99] N. Su, Y. Lei, Q. Li and W. Wang, “Application of Model Driven Architecture in M&S”, *Computer Simulation*, Vol. 24, No. 12, 2007.

[100] W. Tang and W. Mo, “Model Driven Intelligent Form System Frame Oriented to Fields”, *Journal of Beijing University of Aeronautics and Astronautics*, Vol. 33, No. 9, 2007.

[101] X. Sun, M. Yu and H. Yu, “Research on Applying of Language Oriented Programming”, *Computer Engineering and Design*, Vol. 28, No. 21, 2007.

- [102] S. Thibault, *Domain-Specific Languages: Conception, Implementation, and Application*, PhD Thesis, University of Rennes 1, France, 1998-10
- [103] S. Thibault and C. Consel, "A Framework of Application Generator Design", *ACM SIGSOFT Software Engineering Notes*, Vol.22, No.3, pp.131-135, 1997.
- [104] S. Thibault, R. Marlet and C. Consel, "Domain-Specific Languages: from Design to Implementation – Application to Video Device Drivers Generation", *IEEE Transactions on Software Engineering (TSE)*, Vol.25, No. 3, pp. 363-377,1999.
- [105] T. Tourwe, J. Brichau¹, A. Kellens and K. Gybels, "Induced Intentional Software Views", *Computer languages systems & structures*, Vol. 30, No. 1-2, pp. 35-47, 2004.
- [106] A. Toval and J. L. Fern, "Formally Modeling UML and its Evolution:A Holistic Approach", *Proceedings IFIP Conference on Formal Methods for Open Object-Based Distributed Systems IV*, Stanford, California, USA, pp. 183-206, September 2000.
- [107] J. P. Tolvanen and S. Kelly, "Defining Domain-Specific Modeling Languages to Automate Product Derivation: Collected Experiences", *Proceedings of The 9th International Conference on Software Product Lines: (SPLC 2005)*, Rennes, France, pp. 198–209, 2005.
- [108] J. Uldrich, *Jump the Curve: 50 Essential Strategies to Help Your Company Stay Ahead of Emerging Technologies*, Platinum Press, January, 2008.
- [109] Vanderbilt University, GME, <http://www.isis.vanderbilt.edu/Projects/gme/>, 2005.
- [110] L. R. Varney and D. S. Parker, "Generative Programming, Interface-Oriented Programming, and Source Transformation Systems" *OOPSLA / GPCE Workshop on Software Transformation Systems*,

Vancouver, October 24, 2004.

[111] M. Völter and J. Bettin, Patterns for Model Driven Software-Development, Version 1.4, May 10, 2004.

[112] W3c, “Document Object Model (DOM) Technical Reports”, Available from <http://www.w3.org/DOM/DOMTR>.

[113] Y. Wan, D. Feng, L. Liu and T. Yang, “Designing for Disaster-Tolerance Based on DSS of Model Driven”, *Computer engineering and Application*, Vol.43, No. 26, 2007.

[114] J. Wang, “Formal Description of CRM Macro Framework”, *China United Telecommunications Corporation*, 2003.

[115] H. Wang, D. Zhang and J. Zhou, “MDA-Based Development of E-Learning System”, *Proceedings of 27th Annual Intl on Computer Software and Applications Conference*, pp. 684~689, 2003.

[116] K. Wang, S. Zhang, J. Zhou and H. Zhao, “Semantic Model Driven Approach for Data-attribute Matching”, *Application Research of Computers*, Vol. 23, No. 10.

[117] B. Wen, Z. Wang, L. Wang and W. Feng , “Research and Implementation of Software Integration Based on Model”, *Computer Engineering and Design*, Vol 28, No. 23, 2007.

[118] M. P. Ward, “Language Oriented Programming: Software Concepts and Tools”, Computer Science Department Science Labs, South Rd Durham, DH1 3LE, January 17, 2003.

[119] A. Wu, J. Wu, J. Chen and C. Liu, “Research and Implementation of Model Driven Distributed Testing of Automation”, *Software engineering and Application*, Vol. 43, No. 10, 2007.

[120] G. Xu, J. Gu and H. Che, *System Science*, Shanghai Science and

Technology Education Press, 2003.

[121] W. L. Yeung, “Checking Consistency between UML Class and State Models Based on CSP and B”, *Journal of Universal Computer Science*, Vol.10, No. 11, pp. 1540-1559, 2004.

[122] F. Yuan, “MDA: Model Driven Architecture”,
<http://blog.csdn.net/yuandafeng/archive/2007/11/28/1905750.aspx>

[123] X. Zhan, H. Miao, and L. Liu, “Formalizing the Semantics of UML Statecharts with Z”, *Proceedings of the 4th International Conference on Computer and Information Technology (CIT '04)*, IEEE Computer Society, pp.1116-1121, 2004.

[124] Q. Zhang, B. Tan and C. Tan, “Implementing Low-Coupling Module with Generative Programming Methods: The Theory and Practice of AOP”, *Journal of Computer Applications*, Vol. 25, No. 3, 2005.

[125] W. Zhuo and M. Gu, “Feature Refinement-Based Generative Programming Method”, *Computer Science*, Vol. 33, No. 6, 2006.

[126] D. Zhou, Y. Ye, G. Hu and H. Cai, “A Study of MDA-Based GIS Applied System Development”, *Journal of Jilin University (Earth Science Edition)*, Vol. 36, No. 4, pp. 653-658, 2006.

Appendix A

Syntax of XMML

XMML is a meta-modelling language based on XML, the meta-model as built together with the specific application system model based on the meta-model use the same concrete syntax structure. The descriptions of the two models are finally persisted as a XML file, therefore, the concrete syntax definition of XMML is described by XML Schema as follows:

A.1 ModelType Schema

```
<xs:complexType name="ModelsType">
  <xs:sequence>
    <xs:element name="Model" minOccurs="0"
maxOccurs="unbounded">
      <xs:complexType>
        <xs:complexContent>
          <xs:extension base="ModelType">
            <xs:sequence>
              <xs:element ref="Entities" minOccurs="0"/>
              <xs:element ref="Relationships" minOccurs="0"/>
              <xs:element ref="Diagrams" minOccurs="0"/>
              <xs:element ref="Events" minOccurs="0"/>
              <xs:element ref="Properties" minOccurs="0"/>
              <xs:element ref="Specification" minOccurs="0"/>
              <xs:element ref="CodeGenerators" minOccurs="0"/>
              <xs:element name="RefEntities">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="RefEntity" minOccurs="0"
maxOccurs="unbounded">
                      <xs:complexType>
                        <xs:attribute name="id"/>
                        <xs:attribute name="modelId"/>
                      </xs:complexType>
                    </xs:sequence>
                  </xs:complexType>
                </xs:element>
              </xs:sequence>
            </xs:extension>
          </xs:complexContent>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:complexType>
```



```
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
<xs:attribute name="id"/>
<xs:attribute name="type"/>
</xs:extension>
</xs:complexContent>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
```

A.2 PropertiesType Schema

```
<xs:complexType name="PropertiesType"/>
  <xs:element name="Properties">
    <xs:complexType>
      <xs:complexContent>
        <xs:extension base="PropertiesType">
          <xs:sequence>
            <xs:element name="Property" maxOccurs="unbounded">
              <xs:complexType>
                <xs:sequence>
                  <xs:element ref="Properties"/>
                </xs:sequence>
                <xs:attribute name="type" use="required"/>
                <xs:attribute name="name" use="required"/>
                <xs:attribute name="value"/>
              </xs:complexType>
            </xs:element>
          </xs:sequence>
          <xs:attribute name="propertyMgr"/>
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
  </xs:element>
</xs:sequence>
</xs:complexType>
```

A.3 EventsType Schema

```
<xs:complexType name="EventsType">
  <xs:sequence>
    <xs:element name="Event"
maxOccurs="unbounded">
      <xs:complexType>
        <xs:attribute name="type"/>
        <xs:attribute name="scriptId"/>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>
```

A.4 SpecificationType Schema

```
<xs:complexType name="SpecificationType">
  <xs:sequence>
    <xs:element name="SpecsItem"
maxOccurs="unbounded">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="content"/>
        </xs:sequence>
        <xs:attribute name="id"/>
        <xs:attribute name="type"/>
        <xs:attribute name="lang"/>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>
```

A.5 EntityType Schema

```
<xs:complexType name="EntityType">
  <xs:sequence>
    <xs:element name="RefinedModel">
      <xs:complexType>
        <xs:sequence>
          <xs:element ref="Model" minOccurs="0"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    <xs:element name="Attachment">
      <xs:complexType>
```

```

    <xs:sequence>
      <xs:element ref="Entity" minOccurs="0"
maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="Contained">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="EntityRef" minOccurs="0"
maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element ref="Properties"/>
<xs:element ref="Events"/>
<xs:element ref="Specification"/>
</xs:sequence>
<xs:attribute name="id"/>
<xs:attribute name="type"/>
</xs:complexType>

```

A.6 RelationshipsType Schema

```

<xs:complexType name="RelationshipsType">
  <xs:sequence>
    <xs:element name="Relationship" maxOccurs="unbounded">
      <xs:complexType>
        <xs:sequence>
          <xs:element ref="Properties"/>
          <xs:element ref="Events" minOccurs="0"/>
          <xs:element name="sourceRole"/>
          <xs:element name="targetRole"/>
          <xs:element name="combinedFeatures" minOccurs="0">
            <xs:complexType>
              <xs:sequence/>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
  <xs:attribute name="id"/>
  <xs:attribute name="name"/>

```

```

<xs:attribute name="type"/>
<xs:attribute name="feature"/>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>

```

A.7 Roles Schema

```

<xs:element name="Roles">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Role" minOccurs="2" maxOccurs="2">
        <xs:complexType>
          <xs:sequence>
            <xs:element ref="Properties"/>
            <xs:element ref="Events"/>
            <xs:element ref="Specification"/>
          </xs:sequence>
          <xs:attribute name="type"/>
          <xs:attribute name="elementId"/>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>

```

A.8 Diagram Schema

```

<xs:complexType name="DiagramsType">
  <xs:sequence>
    <xs:element name="Diagram" minOccurs="0"
maxOccurs="unbounded">
      <xs:complexType>
        <xs:complexContent>
          <xs:extension base="DiagramType">

```

```
<xs:sequence>
  <xs:element name="VisualElements">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="VisualElement" minOccurs="0"
maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element ref="Properties"/>
</xs:sequence>
<xs:attribute name="id"/>
<xs:attribute name="type"/>
<xs:attribute name="RenderEngine"/>
</xs:extension>
</xs:complexContent>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
<xs:complexType name="DiagramType"/>
```

A.9 VisualElement Schema

```
<xs:element name="VisualElement">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="Div"/>
      <xs:element name="Script">
        <xs:complexType>
```

```
<xs:attribute name="lang"/>
</xs:complexType>
</xs:element>
</xs:sequence>
<xs:attribute name="id"/>
<xs:attribute name="type"/>
<xs:attribute name="elementId"/>
<xs:attribute name="events"/>
</xs:complexType>
</xs:element>
<xs:complexType name="DivType">
<xs:sequence>
<xs:element ref="Div" minOccurs="0"
maxOccurs="unbounded"/>
</xs:sequence>
<xs:attribute name="id"/>
<xs:attribute name="style"/>
<xs:attribute name="features"/>
<xs:attribute name="events"/>
</xs:complexType>
```

Appendix B

Publications by Candidate

- [1] Z. Liang, S. Li, H. Liao, Q. Duan, H. Zhou and H. Yang, “A Multiple-Tier Model Manipulation Architecture on Enterprise Decision Making”, In Proceedings of IEEE Computer Software and Applications Conference 2003, Dallas, Texas, October 2003
- [2] Z. Liang and H. Xin, “A Study on Software Process Management based on CMM/CMMI”, *Journal of Yunnan University*, Vol. 25, No. 6, pp.100-104, 2003.
- [3] H. Zhou, Q. Duan, Z. Liang, S. Li and H. Yang, “A Multiple-tier Distributed Data Integration Architecture”, *Seventh World Conference on Integrated Design and Process Technology (IDPT)*, Austin, Texas, December 2003.
- [4] Y. Jin and Z. Liang, “The Application Research of Web Services in Enterprise Data Integration”, *Journal of Yunnan University*, Vol. 26, No. 7, pp.58-63, 2004.
- [5] Q. Duan, Z. Liang, H. Zhou, H. Liao and H. Yang, “Developing Distributed Virtual Machines for the Tri-Integration-Pattern Based Platform (TIPBP)”, *Proceedings of IEEE International Workshop on Service-Oriented System Engineering (SOSE2005)*, Beijing, China, October 2005.
- [6] H. Zhou, X. Sun, Z. Liang and H. Yang, “XMML: A Visual Metamodeling Language for Domain-Specific Modeling and Its Application in Distributed Systems”, *Proceedings of 12th IEEE International Workshop on Future Trend of Distributed Computing Systems*, Kunming, China, pp. 133-139, 2008.