

A DEMONSTRATION-BASED APPROACH FOR  
DOMAIN-SPECIFIC MODELING LANGUAGE CREATION

by

HYUN CHO

DR. JEFF GRAY, COMMITTEE CHAIR  
DR. BARRET BRYANT  
DR. JEFFREY CARVER  
DR. MARJAN MERNIK  
DR. RANDY SMITH  
DR. EUGENE SYRIANI

A DISSERTATION

Submitted in partial fulfillment of the requirements  
for the degree of Doctor of Philosophy  
in the Department of Computer Science  
in the Graduate School of  
The University of Alabama

TUSCALOOSA, ALABAMA

2013



## ABSTRACT

Model-Driven Engineering (MDE) is a promising approach for addressing the issues of complex and large software system development that enables software engineers to develop software systems with high-level abstract models. In MDE, models are first-class entities of software system development and can improve the understanding of problem domains. In addition, models are used to predict the quality and performance of software systems.

Within the context of MDE, Domain-Specific Modeling Languages (DSMLs) are developed to describe notions of a specific domain using either textual or graphical syntax. DSMLs provide a language that has abstractions and notations, as well as precise and concise modeling constructs, for specific domains (e.g., automotive, avionics, finance, and etc). DSMLs assist domain experts in describing their problems closer to the problem domain when compared to General-Purpose Modeling Languages (GPMLs), such as Unified Modeling Language (UML) or programming languages.

DSMLs have been shown in the literature to provide several benefits, such as productivity improvement, quality improvement, and reduction of miscommunication. However, development of new DSMLs can be challenging and requires much time and effort. In addition, the current state of DSML is still in its infancy compared to the tools and resources available for creation of programming language environments.

This dissertation investigates a new approach for DSML creation that allows domain experts to have a more prominent role in describing the languages that they use. The core contributions of the dissertation are focused on three aspects related to domain-specific modeling language creation: 1) enable the creation of DSMLs in a demonstration-based approach by recording and analyzing the operational behavior exhibited by a domain expert as they model notions of their domain, 2) enable domain expert verification of the inferred language by exploring the model space, and 3) enable domain expert verification of the inferred language by exploring the model space.

The objectives and contributions of the research will be explained in detail in this dissertation, combined with case studies from several domain modeling languages to demonstrate how a domain expert can build their own DSMLs in practice.

## DEDICATION

This dissertation is dedicated to everyone who helped me and guided me through the trials and tribulations of creating this manuscript. In particular, my family and close friends who stood by me throughout the time taken to complete this work.

## LIST OF ABBREVIATIONS AND SYMBOLS

<i>AG</i>	Attributed Graph
<i>BFS</i>	Breadth First Search
<i>BNF</i>	Backus-Naur form
<i>BPMN</i>	Business Process Model and Notation
<i>CCS</i>	Calculus of Communicating System
<i>CSP</i>	Communicating Sequential Processes
<i>DaRT</i>	Data Parallelism to Real Time
<i>DESERT</i>	DEsign Space ExploRation Tool
<i>DFA</i>	Deterministic Finite Automata
<i>DHCP</i>	Dynamic Host Configuration Protocol
<i>DNS</i>	Domain Name System
<i>DSE</i>	Design Space Exploration
<i>DSLs</i>	Domain-Specific Languages
<i>DSM</i>	Domain-Specific Modeling
<i>DSMLs</i>	Domain-Specific Modeling Languages
<i>EBNF</i>	Extended Backus-Naur Form
<i>ERD</i>	Entity Relationship Diagram
<i>EMF</i>	Eclipse Modeling Framework
<i>EUP</i>	End User Programming

<i>FODA</i>	Feature-Oriented Domain Analysis
<i>FSM</i>	Finite State Machine
<i>GEF</i>	Graphical Editing Framework
<i>GEMS</i>	Generic Eclipse Modeling System
<i>GME</i>	Generic Modeling Environment
<i>GMF</i>	Graphical Modeling Framework
<i>GPLs</i>	General-Purpose Languages
<i>GPMLs</i>	General-Purpose Modeling Languages
<i>GReAT</i>	Graph Rewriting and Transformation Language
<i>GUI</i>	Graphical User Interface
<i>IEEE</i>	The Institute of Electrical and Electronics Engineers
<i>KM3</i>	Kernel Meta Meta Model
<i>LOTOS</i>	Language Of Temporal Ordering Specification
<i>MARS</i>	MetAmodel Recovery System
<i>MLCBD</i>	Modeling Language Creation By Demonstration
<i>MDA</i>	Model-Driven Architecture
<i>MDD</i>	Model-Driven Development
<i>MDE</i>	Model-Driven Engineering
<i>MDSD</i>	Model-Driven Software Development
<i>MIC</i>	Model Integrated Computing
<i>MOF</i>	Meta-Object Facility
<i>MRL</i>	Model Representation Language
<i>MVC</i>	Model-View-Controller

<i>OCL</i>	Object Constraint Language
<i>OCP</i>	Open/Closed Principle
<i>OMG</i>	Object Management Group
<i>OBDD</i>	Ordered Binary Decision Diagrams
<i>PbE</i>	Programming-By-Example
<i>PCB</i>	Printed Circuit Board
<i>PERT</i>	Program Evaluation and Review Technique
<i>PIM</i>	Platform Independent Model
<i>PMF</i>	Pattern Modeling Framework
<i>PSM</i>	Platform Specific Model
<i>QbE</i>	Query-by-example
<i>RCP</i>	Rich Client Platform
<i>SDL</i>	Specification Description Language
<i>SVG</i>	Scalable Vector Graphics
<i>SysML</i>	Systems Modeling Language
<i>UCMs</i>	Use Case Maps
<i>UI</i>	User Interface
<i>UML</i>	Unified Modeling Language
<i>VDM</i>	Vienna Development Method
<i>XMI</i>	XML Interchange Metadata
<i>XML</i>	Extensible Markup Language
<i>WYSIWYG</i>	What You See Is What You Get

## ACKNOWLEDGEMENTS

My sincerest gratitude goes to my advisor, Dr. Jeff Gray, for his consistent support, encouragement, and care for me over the past years. Dr. Gray was always there to listen and to give advice during the whole period of my graduate study. In each step toward the completion of my Ph.D. degree, Dr. Gray has offered a great deal of effort to help me form ideas, give research directions and advice, revise the publications and presentations, and refine and improve the quality of my research results.

I would also like to extend my gratitude to the other members of my committee. Dr. Barrett Bryant led me to the world of programming languages and compilers. The knowledge I learned from his course has been very useful to my doctoral research. Dr. Marjan Mernik helped me enrich and deepen my understanding of the concepts of “Domain-Specific Languages” through his enlightening course.

To Dr. Jeffrey Carver, Dr. Randy Smith, and Dr. Eugene Syriani, thank you for your efforts to help me develop the necessary knowledge and skills in throughout the stages of my Ph.D. study. I greatly appreciate your precious time and effort in serving as my committee member and sharing your experience of graduate study with me.

I also will never forget the support and help from current and previous SoftComers. To Dr. Yu Sun, Dr. Qichao Liu, Dr. Robert Tairas, and Dr. Zekai Demirezen, thank you for everything you gave to me, and I cherish every moment we had together in the past years. To Dr.

Ferosh Jacob, Amber Krug, Songqing Yue, and Jonathan Corley, I really appreciate our friendship and all the wonderful and fun time together as a collaborative team.

Finally, I am grateful to the financial support from the UA Department of Computer Sciences, and the National Science Foundation CAREER Grant (No. 1052616).

## CONTENTS

ABSTRACT .....	ii
DEDICATION .....	iv
LIST OF ABBREVIATIONS AND SYMBOLS .....	v
ACKNOWLEDGEMENTS .....	viii
LIST OF TABLES .....	xiii
LIST OF FIGURES .....	xiv
1 INTRODUCTION .....	1
1.1 Domain-Specific Modeling Languages (DSMLs) .....	5
1.2 Key Challenges in DSML Development .....	8
1.3 Research Goals and Overview .....	10
1.3.1 Creating DSMLs By Demonstration to Simplify DSML Creation .....	11
1.3.2 Model Space Exploration for Verification of Inferred Metamodel and Semantics .....	12
1.3.3 Metamodel Design Patterns Assist in Inferring Metamodel .....	13
1.3.4 Semi-Formal Approach to Model Requirements of a DSML .....	13
1.4 The Structure of the Thesis .....	14
2 INTRODUCTION TO DSMLs .....	16
2.1 MDE and DSMLs .....	16
2.2 Components of DSMLs .....	22
2.3 DSML Development Process .....	28
3 SYNTAX MAP: MODELING LANGUAGE FOR DSML REQUIREMENTS .....	35
3.1 Requirements Modeling .....	36
3.1.1 Natural Language .....	37
3.1.2 Semi-Formal Methods .....	39
3.1.3 Formal Methods .....	40
3.2 Goals and Requirements of Syntax Map .....	41

3.3	Design of Syntax Map.....	44
3.3.1	Concrete Syntax of Syntax Map .....	44
3.3.2	Metamodel of Syntax Map.....	47
3.4	Guidelines for Describing DSML Requirements using Syntax Map.....	48
3.5	Application of Syntax Map .....	50
3.6	Related Work.....	52
3.7	Conclusion.....	54
4	METAMODEL DESIGN PATTERNS .....	56
4.1	Approach of Metamodel Design Pattern Mining .....	57
4.1.1	Context Setting.....	58
4.1.2	Identification of Metamodel Design Problems .....	60
4.2	Identification of Metamodel Design Patterns.....	62
4.2.1	Metamodel Pattern for Base Metamodel .....	62
4.2.2	Metamodel Pattern for Typed Relationships .....	64
4.2.3	Metamodel Pattern for Containment.....	65
4.3	Application of Metamodel Design Patterns .....	67
4.4	Related Work.....	68
5	INTERMEDIATE DESIGN SPACE.....	70
5.1	Introduction to Intermediate Design Space.....	72
5.2	Concrete Syntax Identification.....	74
5.3	Graph Construction .....	79
5.4	Metamodel Inference.....	86
6	MODEL SPACE EXPLORATION .....	96
6.1	Process of Model Space Exploration .....	97
6.2	Model Instantiation and Clustering.....	98
6.3	Consideration of the Number of Models for Model Space Exploration .....	107
6.4	Related Work.....	112
6.5	Summary .....	113
7	CASE STUDIES APPLYING MLCBD.....	114
7.1	Development of a Finite State Machine Modeling Tool.....	114

7.1.1	FSM Requirements Modeling with Syntax Map .....	115
7.1.2	FSM Development with GMF .....	117
7.1.3	FSM Development with MLCBD.....	124
7.1.4	Comparison FSM Development: MLCBD vs. GMF .....	127
7.2	Development of Network Diagramming Tool .....	130
7.2.1	Network Diagramming Tool Requirements Modeling with Syntax Map ..	130
7.2.2	Development of Network Modeling Tool using GME .....	132
7.2.3	Development of Network Modeling Tool using MLCBD.....	140
7.2.4	Comparison between MLCBD and GME.....	144
7.3	Summary .....	146
8	FUTURE WORK.....	148
8.1	Enhancements to the Intermediate Design Space Capability .....	148
8.1.1	Improve Metamodel Design Patterns.....	149
8.1.2	Consideration about Dynamic Semantics .....	150
8.2	Enhancements to the Model Space Exploration Functionality .....	150
8.2.1	Enhance Model Space Exploration Algorithm .....	151
8.2.2	Improve Layout Management.....	152
8.3	Improvements for the Syntax Map.....	153
8.3.1	Formalization of Syntax Map .....	153
8.3.2	Tool Support .....	154
9	CONCLUSION.....	155
9.1	Syntax Map .....	156
9.2	Intermediate Design Space.....	156
9.3	Model Space Exploration .....	158
9.4	Dissertation Conclusion .....	159
	LIST OF REFERENCES.....	160

## LIST OF TABLES

Table 2.1 Classification of Model Transformation.....	22
Table 3.1 Concrete Syntax of Syntax Map.....	45
Table 4.1 Listing of Example Domains for Representative DSMLs.....	59
Table 5.1 Candidate Concrete Syntax.....	77
Table 5.2 Annotated Concrete Syntax .....	78
Table 6.1 Samples of Model Instances .....	105
Table 6.2 The Number of Diagramming Elements in Major Diagrams .....	109
Table 7.1 Overview of Comparison.....	146

## LIST OF FIGURES

Figure 1.1 Flexibility vs. Abstraction of Languages.....	4
Figure 1.2 Software Development Paradigms: Conventional vs. Model-Driven .....	7
Figure 2.1 Model-Driven Architecture: PIM and PSM .....	18
Figure 2.2 DSML Components and their Relationship .....	23
Figure 2.3 Abstract Syntax of Simplified FSM .....	24
Figure 2.4 Concrete Syntax for FSM.....	24
Figure 2.5 Concrete Syntax for Actor.....	25
Figure 2.6 Denotational Semantics of Simple FSM Described by the Z Notation.....	27
Figure 2.7 DSML Development Process .....	30
Figure 3.1 Syntax Map Metamodel .....	47
Figure 3.2 DFA for Checking the Multiples of 3.....	50
Figure 3.3 The Skeleton of the Syntax Map .....	51
Figure 3.4 The Complete Syntax Map for the DFA .....	52
Figure 4.1 Feature Model of DSML Concrete Syntax.....	60
Figure 4.2 Base Metamodel Design Pattern .....	63
Figure 4.3 Metamodel Design Pattern for Typed Relationships.....	64
Figure 4.4 Metamodel Design Pattern for Containment.....	66
Figure 5.1 Overall Process of MLCBD .....	70
Figure 5.2 Process of Intermediate Design Space .....	73

Figure 5.3 Domain Models of Process Control .....	76
Figure 5.4 Results of Graph Builder .....	81
Figure 5.5 Results of Graph Annotator .....	85
Figure 5.6 Base Metamodel Design Pattern and its Graph Representation .....	87
Figure 5.7 Combined Graph Representation: Adjacency Matrix and Cardinality Matrix .....	88
Figure 5.8 Metamodel Inference Algorithm: Main Algorithm .....	90
Figure 5.9 Metamodel Inference Algorithm: Instantiate Tree .....	91
Figure 5.10 Metamodel Inference Algorithm: Merge Tree .....	91
Figure 5.11 An Example of Row-Column Representation and Decision Tree .....	92
Figure 5.12 Merged Decision Tree .....	94
Figure 5.13 Inferred Metamodel .....	95
Figure 6.1 Process of Model Space Exploration .....	98
Figure 6.2 The Y-Chart for Model Space Exploration .....	100
Figure 6.3 Main Algorithm of Model Space Exploration .....	102
Figure 6.4 Algorithm for Model Space Exploration .....	102
Figure 6.5 Graph Search: Breadth First Search with Backtracking .....	103
Figure 6.6 Numbered Graph Representation .....	104
Figure 6.7 The Number of Model Instances vs. Modeling Elements .....	111
Figure 7.1 An Example of FSM .....	115
Figure 7.2 Syntax Map for FSM .....	116
Figure 7.3 GMF Tooling Workflow .....	118
Figure 7.4 Ecore model for FSM [Kermeta] .....	119
Figure 7.5 Specifying a Graphical Definition in GMF .....	120

Figure 7.6 Refine Graphical Definition .....	121
Figure 7.7 Tooling Definition Model.....	122
Figure 7.8 Mapping Model and Generation Model .....	123
Figure 7.9 FSM Tool Created using GMF .....	123
Figure 7.10 Creating FSM By-Demonstration.....	124
Figure 7.11 Annotating Concrete Syntax in MLCBD .....	125
Figure 7.12 Creating FSM while Inferring Static Constraints.....	126
Figure 7.13 Syntax Map for Network Digramming.....	131
Figure 7.14 Sample Network Digramming Model .....	132
Figure 7.15 User Interface of GME .....	133
Figure 7.16 Metamodel for Network Diagramming .....	134
Figure 7.17 Aspects of Network Diagramming Metamodel.....	136
Figure 7.18 Concrete Syntax Specification .....	137
Figure 7.19 Creating a New Network Model.....	138
Figure 7.20 Network Models in GME .....	139
Figure 7.21 Image Files of Network Hardware Components .....	140
Figure 7.22 Demonstration of the Network Model in MLCBD .....	141
Figure 7.23 Examples of Model Space Exploration .....	142
Figure 7.24 Summary of Model Space Exploration .....	143

## CHAPTER 1

### INTRODUCTION

Along with the advances in hardware and software technology, end users' demands for computer systems have also increased. Consequently, software size is continually growing and software is becoming more complex than ever before. As a result, many software development projects are often unmanageable, run over planned budget and schedule. In addition, software can become challenging to maintain and may even fail to satisfy the desired level of quality and miss the stated customer requirements. The term “software crisis” [Naur and Randell, 1969] was coined several decades ago, but is still appropriate for describing the current state of software development.

In connection with the software crisis, Fred Brooks mentioned in his book, *The Mythical Man-Month: Essays on Software Engineering*, that software development is an inherently challenging process due to both essential and accidental complexity [Brooks, 1987]. The essential complexity represents the core of the difficult problems that software developers have to solve. The difficulties of understanding the problem domain and identification and development of the conceptual constructs (e.g., core data structures and their interrelationships, as well as algorithms and the behavioral consequences of their combination) that compose the abstract software entity are examples of essential complexity. In addition, the inherent characteristics of software that Brooks identifies (e.g., invisibility, changeability and conformity)

are the other parts that contribute to the essential complexity. The accidental complexities are the challenges regarding concrete software development and testing processes (e.g., specific languages and platforms that must be used to represent the software). In the past several decades, much effort has been made to help software engineers address these complexities in order to develop quality software systems while achieving quality attributes (e.g., productivity, simplicity, reliability, and maintainability).

Developing higher levels of abstraction in programming languages is one of the strategies that has been put forth to address the issues of accidental complexity. The history of programming languages can be traced back to the mid-1940s. In 1945, von Neumann proposed a computer architecture that can perform different tasks by storing programs in memory [Backus, 1978]. Since then, many programming languages have been proposed, evolved, and disappeared. For example, machine languages are typically hardware dependent and required programming at a low-level of abstraction. Thus, machine languages are rarely used in modern software development environments due to an average software engineer's difficulty in understanding the language, which can contribute to increased software complexity. As a result, machine language code is generated by high-level language compilers. The idea of low-level representations being generated from higher level specifications has been a constant trend over the past decades. The notion of generative programming allows expression of core problem concepts in languages that are closer to the abstractions of the problem domain, rather than the solution space [Czarnecki and Eisenecker, 2000].

Among all the efforts in the history of programming languages, Brooks states that raising the level of programming language abstraction is the “most powerful stroke for software productivity, reliability and simplicity” [Brooks, 1987]. This is because the raised level of

abstraction assists in capturing only the details relevant to the target computing environment, and as a consequence, hides the underlying implementation information [Lenz and Wienands, 2006].

As shown in Figure 1.1, the level of abstraction is raised from machine language to assembly language, to high-level and object-oriented programming languages. As the level of abstraction is increased, software engineers generally lose fine-grained control of the underlying computing environment (e.g., direct memory access and device control), but they can be isolated from irrelevant low-level implementation details. Therefore, software engineers can become more immersed in the problems they need to solve.

On the other hand, the flexibility of the programming languages, in terms of applicable domains, is slightly narrowed as the level of abstraction is raised. High-level languages tend to offer a very narrow set of language constructs to solve a specific domain problem, and may not support certain features (e.g., controlling underlying details of the execution platform). For example, Java could be said to be less flexible than C because Java does not provide core language constructs to access the execution environment (e.g., memory and devices), which makes it more challenging to use Java to develop software that can utilize and manipulate hardware resources.

Domain-Specific Modeling Languages (DSMLs), which are placed at the top of the hierarchy in Figure 1.1, are languages designed and implemented to satisfy specific domain needs [Gray et al., 2007; Lédeczi et al., 2001]. DSMLs are widely used as a concrete and mainstream Model-Driven Engineering (MDE) methodology [Schmidt, 2006], which considers models as first-class entities instead of program codes. MDE has been applied to develop complex and large-scale software systems by decoupling the description of the essential

characteristics of a problem from the details of a specific solution space (e.g., middleware and programming languages).

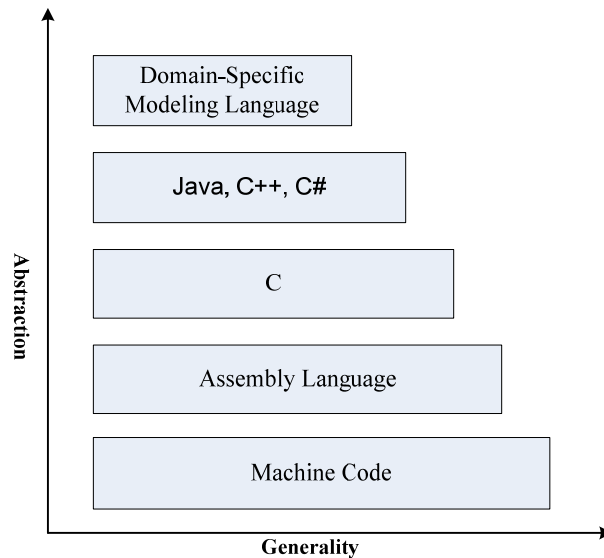


Figure 1.1 Flexibility vs. Abstraction of Languages

DSMLs allow software engineers, or even end-users (e.g., domain experts), to describe requirements, design, and test cases of a software system while focusing on specific domain concepts, rather than solutions that are intertwined with the underlying computing environment [Schmidt, 2006]. In addition, integration with model transformation and code generators allow DSMLs to generate automatically the desired software artifacts (e.g., programming code, simulation scripts, and XML deployment description).

Models help software engineers and domain experts to understand a problem domain. Software engineers are not able to see every aspect of a system at once. It can be difficult to understand both the behavior and structure of a large software system solely through a

programming language. However, if the software system is described using models, a software engineer can understand a system more as a whole because models can abstract and visualize notions of the domain at higher levels. With increased understanding of a software system, software engineers and other stakeholders are able to predict quality, performance, and patterns of the software system evolution at early development phases, without the need for code. In addition, software engineers and customers can make more accurate and valuable decisions in software design, implementation, and maintenance [Kelly and Tolvanen, 2008].

A model-driven approach can also contribute toward the concept of end-user software development [Burnett et al., 2004] and reduce knowledge and expertise gaps between software developers and domain experts. As such, models can improve communication between stakeholders. Design intents can be understood by stakeholders when problems and solutions are described using models. This is possible because modeling languages tend to be designed to represent notions of the problem domain with high-level abstractions from the underlying domain.

### 1.1 Domain-Specific Modeling Languages (DSMLs)

DSMLs introduce a new software development paradigm. The main notion of DSML-based software development is “everything is a model” [Kurtev et al., 2006], and the goal of DSMLs is to help software engineers or domain experts develop quality software using models, which can describe a specific problem domain with models that raise the level of abstraction. Specifically, DSMLs can describe the properties of a system (or a domain) with a high-level of abstraction and a set of platform-independent notations. Thus, DSMLs can offer several benefits. For instance, DSMLs can reduce the chances of software failures by minimizing

miscommunications between software engineers and domain experts and encourage involvement of domain experts in software development [Burnett et al., 2004]. In general, software engineers are skilled at programming, but may not thoroughly and correctly understand problems of every domain that they encounter. Yet, in contrast, domain experts have deep knowledge and experiences about their domain, but may have no expertise about building software systems using programming languages. The knowledge and expertise gap between software engineers and domain experts make communication difficult and present a challenge to develop software systems that meet the demands of the end-user. However, DSMLs are able to ease the difficulty by offering precise and concise notations, which are commonly understood by domain experts without additional descriptions or explanations. In addition, claims of productivity gains in very specific domains using DSMLs have been reported by a factor of 5 to 10 [Kelly and Tolvanen, 2000].

As shown in Figure 1.2, conventional software development practice builds software systems (or packages) using platform-specific models (e.g., programming languages such as Java, C/C++, C#) based on the results of the problem domain analysis. Thus, conventional software development has “the wide conceptual gap between the problem and the implementation domains of discourse” [France and Rumpe, 2007]. The models built by users must conform to the definition of the metamodel [Atkinson and Kuhne, 2003; Gray et al., 2007], which specifies the entities, associations and constraints for the DSML. A metamodel defines abstract syntax in a similar way as a grammar specifies the abstract syntax for a programming language. Thus, a model conforms to its defining metamodel [Kurtev et al., 2006] in the same way that a program conforms to the programming language in which the program is written. To describe notions of a specific domain visually, concrete syntax is also associated with the metamodel. Elements of

concrete syntax are often described using either a textual notation, graphical symbols or both. In addition, semantics can be associated with a metamodel to specify properties and behaviors of a modeling language. However, the current manner in which semantics is specified varies across metamodeling tools and is still at an early stage of investigation.

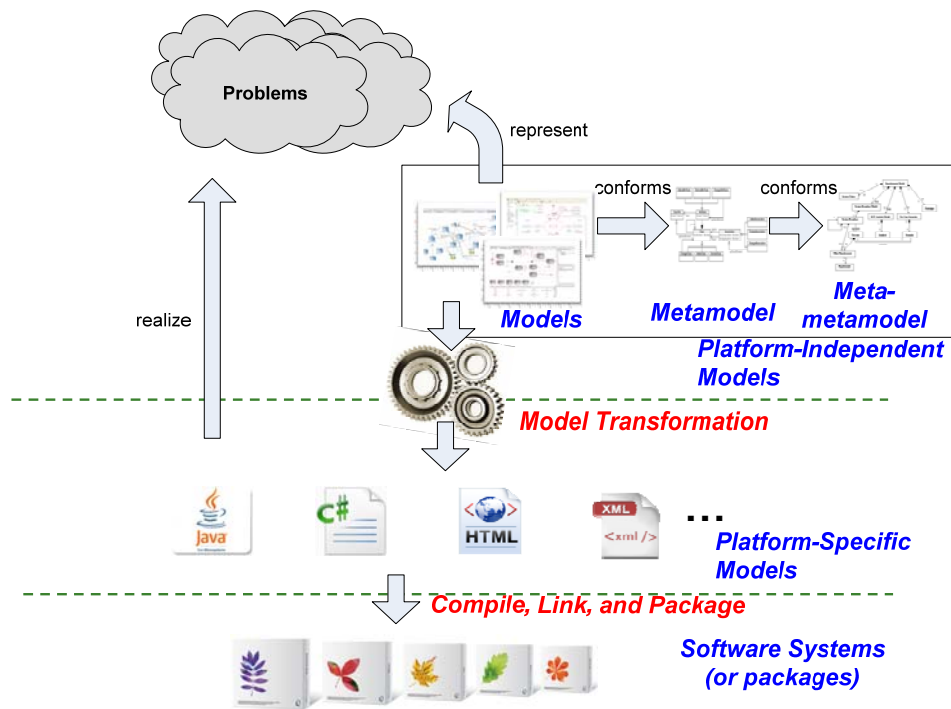


Figure 1.2 Software Development Paradigms: Conventional vs. Model-Driven

Similar to models, a metamodel must also conform to the definition of a meta-metamodel, which is a core modeling language that conforms to itself and used to define other modeling languages for different domains. The common meta-metamodeling languages at this layer are Meta-Object Facility (MOF) [MOF, 2011], and Kernel Meta Meta Model (KM3) [Jouault and Bézivin, 2006]. As a comparison to programming language specification, a meta-metamodel is

similar to EBNF (Extended Backus Naur Form). A meta-metamodel is used to define metamodels representing new modeling languages in the same way that EBNF is used to describe grammars for new programming languages. The four-layer modeling architecture is depicted in Figure 1.2

As shown in this figure: 1) models conform to a metamodel; 2) a model transformation translates a model representation into some lower-level representation, such as a specific programming language, which is then compiled into an executable system that has a correspondence to the associated model.

## 1.2 Key Challenges in DSML Development

As discussed in the previous sections, MDE is a promising approach to address issues in conventional software development by introducing models as first-class entities. However, there are several challenges and limitations that have emerged. Because DSML development requires both domain knowledge and modeling language development expertise (e.g., metamodeling experience), end-users and domain experts who are not computer scientists often find the traditional approach for creating DSMLs daunting due to the following challenges.

*DSML Challenge 1:* In practice, domain models are often specified with unconstrained environments such as word processors or drawing/presentation tools, rather than formal metamodeling tools. For example, Microsoft Visio is a popular tool used in many domains to design drawings and representations of domain-based designs. Due to the increasing number of sketch-based input devices (e.g., tablets) and a general preference for hand-drawn sketches in capturing high-level requirements and software design, modeling tools will need to adjust to new

forms of input [Ossher et al., 2010]. This is particularly challenging for demonstration-based language design because the initial sketch is rather unconstrained and needs to process a wide range of open notations for different domains. Recognizing new shapes that are more informally drawn in new environments and on new input devices is a challenge that will increase in need.

*DSML Challenge 2:* Modeling language creation requires familiarity with metamodeling environments. Domain experts who are not familiar with metamodeling often do not understand the deep implications of domain analysis in DSML implementation. The common practice of DSML development is to define a metamodel for the specific domain based on the captured notations. As the metamodel specifies the abstract syntax and static semantics for the domain, domain experts need to understand the implications of metamodeling before they develop their own DSML. The lack of language creation expertise, especially metamodeling expertise for modeling languages, may undermine the quality of a DSML implementation if developed by a domain expert who lacks a background in computer science.

*DSML Challenge 3:* The captured visual notations and notions of a domain tend to be informal and incomplete, often requiring multiple iterations to reach a final version of the DSML. However, the iterative process of creating a DSML is tedious, error-prone and time-consuming if done manually. Therefore, simplifying (or automating) the DSML development process such that domain experts can create their own languages may offer an advantage.

*DSML Challenge 4:* Specifying the semantics of a modeling language using formal techniques is often challenging even for computer scientists. The major focus of many by-demonstration approaches is on structural and syntactic issues, with little contribution toward mechanisms for describing semantics. Enabling end-users to describe the semantics of their free-form language will require new innovations to separate the underlying formality needed from the

level of abstraction expected by an end-user. This is perhaps one of the most challenging issues facing flexible modeling, and without progress in the area, the tools and associated languages will not be as effective in providing a complete automated solution for performing many tasks that are typical in other modeling contexts.

*DSML Challenge 5:* Having a set of complete and precise requirements is a critical factor for the success of software development. However, eliciting such requirements is not easy because requirements are captured through informal activities (e.g., brainstorming, workshops, direct observation, questionnaires, and interview) [Nuseibeh and Easterbrook, 2000]. In addition, requirements are often expressed using natural languages, which are inherently ambiguous and fuzzy [Rolland et al., 2003]. Although many researchers and practitioners have introduced several approaches, methods and tools for eliciting, specifying and managing requirements of generic software systems, little attention has been paid to the area of DSML requirements management even though specifying requirements of a DSML requires different principles and expertise than those of general software application requirements specification. For example, requirements of a DSML need to describe abstraction mechanism (e.g., abstract notions of a domain and map to concrete syntax) [Liskov and Zilles, 1975], consistent management of similar notions and concrete syntax consistently, and extensibility of a language.

### 1.3 Research Goals and Overview

To address the difficulty of creating DSMLs, the research in this dissertation provides a user-centered DSML creation approach. The goal of the approach is to enable domain experts, who have broad domain knowledge and experiences but do not have programming language development expertise, in creating their own DSMLs. To achieve the research goal, the approach

implements DSML creation tasks combining techniques such as By-Demonstration, graph transformation, a demonstration-based inference engine, metamodel design patterns, and model space exploration. The research is focused on an investigation into techniques that allow a modeling language to be inferred from a set of domain model examples that are provided by the end-user who is a domain expert.

### *1.3.1 Creating DSMLs By Demonstration to Simplify DSML Creation*

To address the challenges of DSML creation as mentioned in the last section, a DSML development framework has been designed and implemented. The demonstration-based approach is called Modeling Language Creation By Demonstration (MLCBD). The MLCBD framework enables domain experts to create a DSML by directly performing edit operations on concrete examples (i.e., a set of domain model examples), combined with user refinement and automated inference processes. To demonstrate the notions of a domain, the MLCBD framework provides predefined shapes. If there is no appropriate predefined symbol in the framework, a domain expert performing the demonstration may use sketch-level shapes or images that represent the notions of their domain. After domain experts demonstrate the modeling language that they desire, the various components needed to represent a DSML (e.g., abstract syntax, concrete syntax, and semantics) are generated using machine learning techniques, as provided by an inference engine that supports MLCBD. The inference engine transforms a set of domain model examples into a set of graph representations in order to avoid a representation mismatch. The DSML language components are inferred from the set of domain model examples. As will be presented in Chapter 5, the MLCBD framework enables domain experts to be isolated from

the challenges of DSML creation and the associated complex language development tasks (i.e., specifying abstract and concrete syntax, and defining semantics).

### *1.3.2 Model Space Exploration for Verification of Inferred Metamodel and Semantics*

The second contribution of this research includes a Model Space Exploration feature associated with MLCBD, which enables domain experts to verify the correctness of the inferred modeling language components. Although the MLCBD framework offers an integrated DSML development environment such that domain experts can build their own DSMLs by demonstrating a set of domain model examples, the syntax and semantics inferred from the set of domain model examples may not describe the domain correctly and fully. Demonstrating every aspect of the domain requires demonstration of both positive aspects (i.e., concepts that represent the domain correctly) and negative aspects (i.e., models not allowed in the domain and illegal relationships between model elements) [Kirsopp and Shepperd, 2002]. The level of demonstrations needed to realize the complete language that is desired requires the domain expert to invest much time and effort. To address the demands required for demonstrating a sufficient number of model examples, the idea of Model Space Exploration is borrowed from ideas that were initiated in computational learning theory [Valiant, 1979] and design space exploration [Oliveira et al., 2010]. During Model Space Exploration, a set of new candidate domain model instances are generated from a previously inferred metamodel. The domain expert plays the role of oracle and is asked to indicate whether each generated instance accurately describes the concepts within the domain. The results from the candidate model instances are fed back to the inference engine to complement the existing understanding of the syntax and semantics that were inferred previously using a set of domain model examples. We believe that

Model Space Exploration can reduce the effort and time that domain experts spend in demonstrating a large set of domain model examples as well as verify inferred syntax and semantics.

### *1.3.3 Metamodel Design Patterns Assist in Inferring Metamodel*

To support the inference of a DSML, the idea of metamodel design patterns has been investigated within the MLCBD framework [Cho and Gray, 2011]. Initially, the MLCBD framework creates a set of metamodel design pattern instances to complement the demonstrated model examples. The idea of metamodel design patterns is an extension of the concept of design patterns [Gamma et al., 1995] as applied to metamodel design. Metamodel design patterns define consistent solutions for recurring metamodel design issues and represent the collection of generic metamodel primitives that occur frequently in the specification of metamodels.

To generate the abstract syntax and corresponding semantics for a DSML, the MLCBD framework performs an isomorphic test in order to find a set of metamodel design patterns that matches to a set of domain model examples. The MLCBD framework then composes the set of metamodel design patterns into a metamodel while adding any required generalizations (e.g., inheritance). With metamodel design patterns, the MLCBD framework can reduce the complexity in inferring a metamodel and the corresponding static semantics from a set of domain models.

### *1.3.4 Semi-Formal Approach to Model Requirements of a DSML*

The idea of a Syntax Map is also introduced in the approach to assist development and verification of a DSML. Syntax Map is another DSML that is designed to model the

requirements of a DSML using graphical notations. Syntax Map captures requirements of a DSML according to syntax usage scenarios. Specifically, requirements of a DSML are modeled based on causal relationships between responsibilities of one or more classifiers and relationships. In addition, Syntax Map can convey both concrete syntax and abstract syntax, as well as constraints associated to the syntax in a compact form. Syntax Map enables reasoning about missing or redundant requirements of a DSML by checking conflicted syntax usage scenarios.

#### 1.4 The Structure of the Thesis

This introductory chapter summarized the focus of research on DSML creation from a set of end-user demonstrated examples. The set of challenges associated with this research area was presented along with the research goals that address those challenges. The remainder of the dissertation is organized as described in the rest of this section.

Chapter 2 describes background information related to the research of this dissertation. Chapter 3 introduces a semi-formal approach, named Syntax Map, which can be used to describe the requirements of a DSML. A Syntax Map focuses on the usage of each modeling element in a DSML and offers a set of graphical symbols to model the requirements of DSMLs. Chapter 4 discusses the idea of metamodel design patterns, which are used in the metamodel inference process of MLCBD. Chapter 5 covers the specific details of MLCBD, including the description about the main steps and implementation details of the approach. Related work is discussed to highlight the unique features and contributions of MLCBD. Chapter 6 motivates the need for Model Design Space, which generates sample domain models from the inferred metamodel and constraints. The domain expert is asked to confirm the correctness of the generated model instances to help refine the inferred metamodel. Case studies are presented in Chapter 7 to show

how MLCBD supports end users building their own DSMLs. Chapter 8 outlines the future work of the research, and Chapter 9 concludes the work of this dissertation by summarizing the contributions.

## CHAPTER 2

### INTRODUCTION TO DSMLs

This chapter provides background information relevant to the research of this dissertation. First, MDE will be introduced in Section 2.1 with further discussion on Domain-Specific Modeling (DSM) and DSMLs. This chapter will also outline the key components of DSML creation in Section 2.2. General software development processes and activities for DSML development will be given in Section 2.3. Finally, because the main contribution of this research focuses on creating visual (or graphical) DSMLs, relevant information about DSML design principles and considerations will be discussed briefly in Section 2.4.

#### 2.1 MDE and DSMLs

Throughout the history of the intersection of software engineering and programming languages, most new development paradigms that have emerged made claims about offering higher levels of abstraction as a benefit of some new idea. The general claim is that some newly introduced language or methodology is able to alleviate impediments of earlier efforts in order to accelerate software development and improve quality. The typical claim is that the raised level of abstraction makes it possible for software engineers to develop far more complex software systems without increasing the associated effort. For example, procedural programming languages (e.g., C, COBOL and FORTRAN) represented the dominant paradigm in the 1960s/1970s. However, procedural programming languages lacked expressiveness and

modularity concepts compared to object-oriented programming languages, such as Java, C++, and C#. In addition, procedural programming languages have supported fewer reusable libraries and application frameworks (or platforms), which can minimize the need to reinvent common services.

Since the 1990s, object-oriented programming languages (e.g., Java and C++) were introduced with more powerful expressive language constructs and have assisted software engineers in maintaining and reusing various software systems [Booch, 1997]. Object-oriented programming languages have many advantages over conventional programming languages. For example, the languages assist in promoting modularity, modifiability, and maintainability. However, the principal advantage claimed for object-oriented programming languages is that they promote reuse of valuable intellectual assets that are captured in code, which are described in the form of classes or objects. However, object-oriented programming languages have reached a complexity ceiling due to the fast growth of dependent platforms and middleware complexity, and the inability of expressing domain concepts effectively [Schmidt, 2006].

In the last two decades, MDE has attracted considerable attention from both academia and industry as a promising approach to address platform complexity and the need to express domain concepts, which are difficult in programming languages. MDE moves the focus of software development from programming language codes to models as first-class entities of software development. The concepts of MDE are established by combining many other approaches, such as Domain-Specific Languages (DSLs), software factories, Model-Integrated Computing (MIC) [Karsai et al., 2004], Model-Driven Software Development (MDSD), model management, and language-oriented programming [Bézivin 2005]. The major contribution for a new era of modeling was made by the Object Management Group (OMG), who offered a

conceptual framework to support Model-Driven Development (MDD). Specifically, OMG launched the concept of Model-Driven Architecture (MDA) [MDA, 2011], which consists of three ideas: direct representation, automation, and open standards. As shown in Figure 2.1, MDA describes the problem domain using a Platform-Independent Model (PIM), which can describe the problem domain independently of the underlying computing environments. From the PIM, a Platform-Specific Model (PSM) can be generated by applying model transformation techniques.

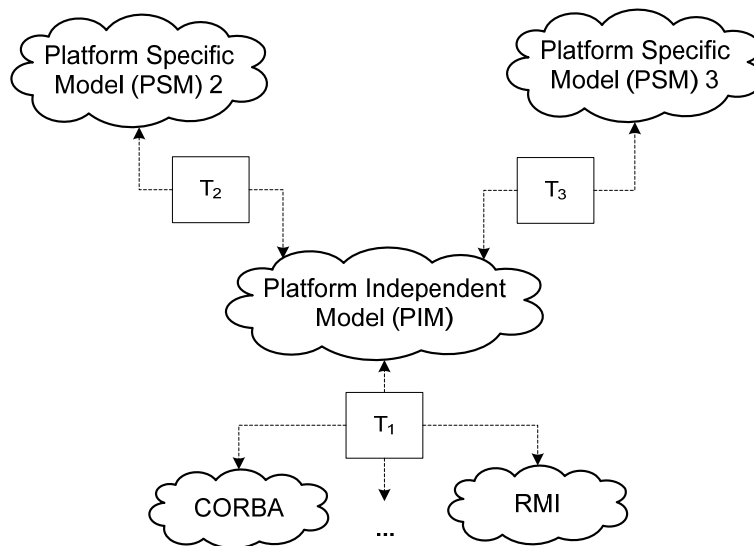


Figure 2.1 Model-Driven Architecture: PIM and PSM

A PSM contains the details of the implementation or solution space of a software system. Thus, MDA can shift the focus of software development from the technology domain (e.g., programming languages, and platforms) towards the concepts of the problem domain (e.g., important abstractions in the domain of interest). In addition, model transformation can automate software development while minimizing manual and tedious mappings between the abstraction layers.

The MDA concepts of direct representation and model transformation are germane to two key technologies of MDE: DSMLs and model transformation. DSLs [Visser, 2007] can be defined by the following definition: “*A domain-specific language is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain*” [Deursen et al., 2000].

Conventionally, most computer languages came into existence as dedicated languages for solving problems in a specific domain. For example, COBOL was introduced to develop business processing applications, and FORTRAN was designed for numeric computation and scientific computing. As the majority of programmers in various domains adopted those languages (e.g., COBOL and FORTRAN) to solve their own domain problems, languages often evolved to solve general kinds of problems, regardless of the area or domain. In MDE, the Unified Modeling Language (UML) is a well-known general-purpose modeling language (GPML). UML was not originally designed for MDA, but it has become a standard GPML because it contains numerous diagrams, constructs and views that can be used to represent various perspectives of a system.

However, GPMLs also have some drawbacks. First, due to a large number of diagrams, constructs, and views, deep modeling expertise is required to develop precise and concise models. For example, the current UML 2.4.1<sup>1</sup> core specification is defined in 978 pages, not including supplementary specifications. Thus, not everyone is able to create a quality model using a GPML

---

<sup>1</sup> UML 2.4.1 Specification, <http://www.omg.org/spec/UML/2.4.1/>

like UML. Second, GPMLs may not be well-matched to model a specific problem domain because the GPML syntax and semantics are not obvious due to their generality. This has created an impedance mismatch between the underlying concepts of domain and the concepts of GPMLs. In addition, this hampers the precise and concise description of the domain concepts.

DSMLs are proposed as a promising approach to address the deeper learning curve and the associated challenges of model comprehension. As textual variants, DSLs are languages whose type systems are tailored for a particular domain such as finance [Brand et al., 1996; Deursen and Klint, 1998], software architecture [Medvidovic and Rosenblum, 1997], operating system specialization [Pu et al., 1997], and multimedia [Stevson and Fleck, 1997].

The key characteristics of DSMLs are summarized below:

- DSMLs offer only a limited suite of notations and abstractions compared to GPMLs. A DSML focuses on a specific-domain and is designed to help domain experts improve their understanding of representation problems in their domain.
- DSMLs are often declarative and can be viewed as specification languages.
- DSMLs are built on the domain users' vocabulary.
- The syntax of a DSML is designed to raise the level of abstraction so that it can hide the inherent accidental complexities of programming languages.
- DSMLs typically require less time and effort to learn the syntax and to develop models, compared to GPMLs.

In addition, both DSLs and DSMLs offer several other benefits that include [Gray et al, 2007; Kelly and Tolvanen, 2008]:

- An ability to describe a problem domain with the idioms and abstraction that are commonly used in the domain.
- Improvements in productivity, reliability, maintainability, and portability. [Herndon and Berzins, 1988; Kieburtz et al., 1996; Deursen and Klint, 1998]
- Support for reuse of domain knowledge as embodied in language constructs.

As introduced in Section 1.1, the top-most layer of the MDE model architecture is called the meta-metamodel layer, which is a core modeling language that conforms to itself and can be used to define other metamodeling languages for different domains. Meta-Object Facility (MOF), Ecore, and Kernel Meta Meta Model (KM3) [Jouault and Bézivin, 2006] are examples of meta-metamodeling languages. The layer below the top-most layer is the metamodel layer. The models at this layer conform to the meta-metamodel and are used to define the core of a DSML. A common metamodeling language at this layer is the UML metamodel. Models at the third level conform to the metamodel to which they are associated. The models in this layer represent the instance models that users create and manipulate to model the underlying real system. Finally, the real system is placed at the bottom layer, which is mapped and generated from the instance models [Kurtev et al., 2006].

Model transformation is another core technology of MDE [Sendall and Kozaczynski, 2003]. Examples of model transformation include code generation from models, model synchronization and mapping, model evolution, and reverse engineering. Code generation, reverse engineering, and refactoring are also examples of model transformations. Due to its importance in MDE, many researchers and practitioners have proposed various model transformation techniques, as summarized in Table 2.1.

Table 2.1 Classification of Model Transformation (Adapted from [Mens and Gorp, 2006])

	Horizontal	Vertical
Endogenous	Refactoring	Formal Refinement
Exogenous	Language Migration	Code Generation

Horizontal transformation is a transformation where the source and target models reside at the same abstraction level. On the contrary, vertical transformation is a transformation where the source and target models reside at different abstraction levels. Endogenous transformations are transformations between models expressed in the same language. Refactoring is an example of an endogenous transformation because it changes the internal structure of a software system without changing observable behaviors. Refactoring also falls into horizontal model transformation because the source and target languages do not change. Exogenous transformations occur between models expressed using different languages. The typical example of exogenous transformation is code generation, where a set of models is translated into source code. As source and target models have different level of abstractions, code generation is also an example of a vertical model transformation.

## 2.2 Components of DSMLs

In this section, we present elements of DSMLs in order to understand how and what domain experts need to provide when trying to develop their own DSMLs. Similar to programming languages, DSMLs consist of five tuples: Abstract syntax (A), Concrete Syntax

(C), Semantics (S), Mapping from concrete syntax to abstract syntax ( $M_{CA}$ ), and Mapping from abstract syntax to semantics ( $M_{AS}$ ) [Chen et al., 2009]. DSML components and their relationship are illustrated in Figure 2.2.

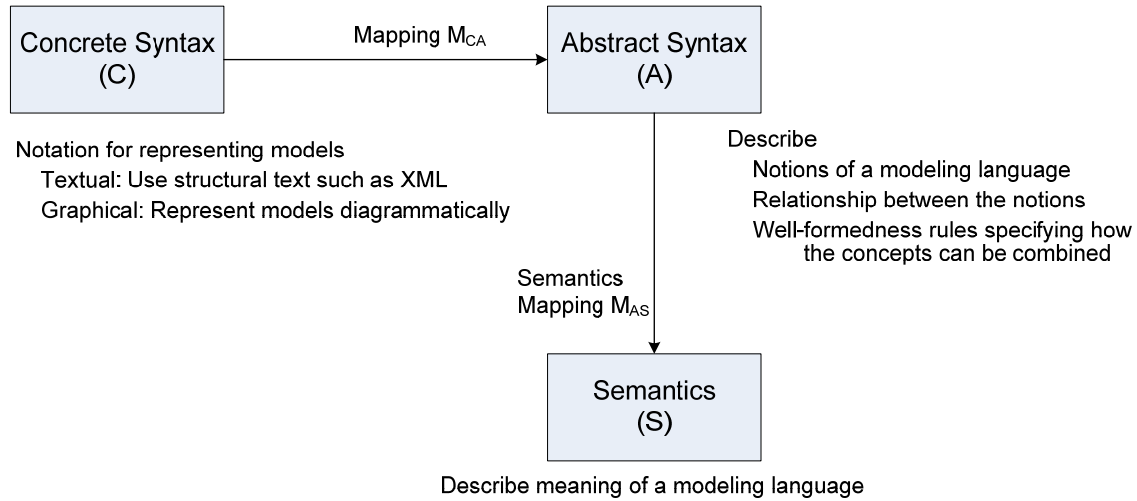


Figure 2.2 DSML Components and their Relationship

The Abstract syntax (A) describes the core concepts and notions of the language, the structural relationships between the language concepts, and the constraints that define how core language elements can be combined to describe domains. Figure 2.3 shows a metamodel for a simple Finite State Machine (FSM) that consists of two modeling elements, *State* and *Transition*. *State* is linked with other *State* through *Transition*. *FiniteStateMachine* is defined to represent the simple FSM and contains all metamodel elements.

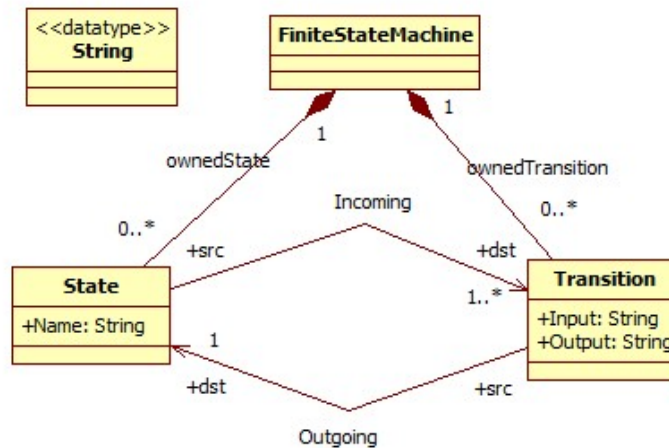
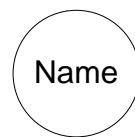
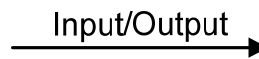


Figure 2.3 Abstract Syntax of Simplified FSM

Concrete syntax (C), also called surface syntax, is primarily concerned with the concrete representation of a language [Milanović et al., 2009]. Concrete syntax defines every detail about the notations used to express models, and either a textual, graphical, or mixed form can be used for specifying concrete syntax [Erwig, 1998]. Figure 2.4 shows the possible concrete syntax for a simple FSM, where a circle is associated to *State*, and a directed line is mapped to *Transition*.



(a) State

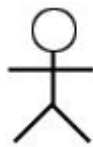


(b) Transition

Figure 2.4 Concrete Syntax for FSM

In general, each concrete syntax element must be mapped to the underlying structure of the abstract syntax [Grunske et al., 2008]. Typically, one abstract syntax element can be mapped

to more than one concrete syntax element to satisfy various usage purposes and requirements of technical spaces. UML is an example that uses multiple concrete syntax elements to satisfy various usage purposes. The abstract syntax of UML, which is defined in the form of the UML metamodel, can be associated with two concrete syntaxes: one for human interaction and the other for computer-to-computer communication. For human interaction, abstract syntax of UML is visualized using graphical notations, and humans are able to design a software system using the graphical notations within a tool. Each UML modeling tool uses its own data representation to manage abstract syntax and corresponding concrete syntax of UML. So, if designers use different UML modeling tools, a UML model maintained under a tool's data structure cannot be loaded into another UML modeling tool. The eXtensible Markup Language (XML) Metadata Interchange (XMI) [OMG XMI, 2011] is standardized by OMG to exchange meta information between UML modeling tools. Figure 2.5 shows two different concrete syntax specifications. Figure 2.5(a) is a graphical representation of an actor in a Use Case model, and Figure 2.5(b) is part of the XMI specification of an actor.



(a) Graphical Representation

```
<packagedElement xmi:type="uml:Class" xmi:id="UseCases-Actor"
  name="Actor">
  <ownedComment xmi:type="uml:Comment"
    xmi:id="UseCases-Actor-_ownedComment.0"
    annotatedElement="UseCases-Actor">
    <body>An actor specifies a role played by a user or
      any other system that interacts with the
      subject.</body>
  </ownedComment>
```

(b) Part of Actor XMI specification

Figure 2.5 Concrete Syntax for Actor

While the syntax of a DSML is focused on the form of its expression, the semantics of a DSML is concerned with the meaning of grammatically correct models. A DSML can have two types of semantics: static semantics and dynamic semantics. Static semantics focus on “*what something is*” and describe invariants on elements of abstract syntax and/or on relations between those elements. Static semantics include all of the possible sets of components and their relationships, which are consistent with well-formedness rules. Dynamic semantics describe the behaviors of the language. Dynamic semantics often specify the evolution of the state of the modeled artifact along some time model. Generally, defining dynamic semantics is much more difficult than static semantics.

Semantics can be defined several ways. Operational, Denotational, and Axiomatic are well-known approaches for semantics specification [Zhang and Xu, 2004]. Operational semantics is interested on the effect of how a computation is produced. Thus, operational semantics describe how to execute programs of a language or how to change the state. Operational semantics is useful as a basis for implementation. Denotational semantics is given by mathematical function, which maps a mathematical meaning to each syntax element. Denotational semantics is used for proving properties of a language. Axiomatic semantics is also called Floyd-Hoare logic [Hoare, 1969] and provides a logical system, which is built from axioms and inference rules for proving partial correctness properties of a program.

Figure 2.6 shows the denotational semantics of a simple FSM. The Z notation is used to describe the semantics of a simple FSM.

$[STATES, INPUTS, OUTPUTS]$

(a) Set Definition

$q0 : STATES; NULL : OUTPUTS; FINALS : \mathbb{F} STATES$
$STATES \neq \emptyset \wedge (\exists n : \mathbb{N} \bullet \#STATES \leq n) \wedge INPUTS \neq \emptyset \wedge q0 \notin FINALS$

(b) An Axiomatic Definition

<div style="display: flex; justify-content: space-between; align-items: center;"> <span>FSM</span> <span>—</span> </div> <div style="padding: 5px;"> <math>transition : STATES \times INPUTS \rightarrow STATES; output : STATES \times INPUTS \rightarrow OUTPUTS</math>  <math>current : STATES</math> </div> <div style="padding: 5px;"> <math>dom\ output = dom\ transition \wedge q0 \in dom\ (dom\ transition) \wedge FINALS \cap dom\ (dom\ transition) = \emptyset \wedge</math>  <math>FINALS \subseteq ran\ transition \wedge STATES \setminus \{q0\} = ran\ transition \wedge</math>  <math>STATES \setminus FINALS = dom\ (dom\ transition)</math> </div>
---

(c) Schemata

Figure 2.6 Denotational Semantics of Simple FSM Described by the Z Notation [Dimitrov, 2010]

Figure 2.6(a) defines the basic sets of a FSM. *STATES* is a non-empty final sets of states, *INPUTS* is the set of input events, and *OUTPUTS* is the set of all output events. Figure 2.6(b) is an axiomatic definition where *q0* is the initial state, *NULL* is a special symbol, and *FINALS* is possibly an empty subset of *STATES*. Figure 2.6(c) is the schema of a FSM that specifies the behaviors of a program. An FSM consists of a transition, output, and current. Transition and output are in the same domain of the Cartesian product of *STATES* and *INPUTS*. The initial state *q0* is part of the transition function domain and is the only state that has no input. *FINALS* has only input, and all states, except the final state, have to have both input and output.

### 2.3 DSML Development Process

To develop a system effectively and manage the progress of the development systematically, a well-formed software development lifecycle is needed with precisely defined activities. Many researchers and practitioners have proposed and applied new software development processes. The Waterfall, Iterative and Incremental, and Spiral models are well-known software development lifecycle models proposed to address issues that occur during software development.

The Waterfall model was proposed by Winston Royce [Royce, 1987] to enable structured software development by executing a series of development activities (e.g., requirement analysis, design, implementation, and testing) sequentially. Although the Waterfall model is still used to develop software, it has been attacked for being too rigid and unrealistic when a system under development needs to accommodate quickly changing customer needs. In addition, the Waterfall model takes a long time to yield a working version of a system.

The basic idea of an iterative and incremental model is to develop a software system while taking advantage of what was learned during the development of previous iterations [Larman and Victor, 2003]. In an iterative and incremental approach, a system is divided into small parts, and built by implementing and integrating the small parts incrementally and iteratively through a mini-Waterfall model.

A prototyping model is based on the assumption that customers may not know all of their requirements at the beginning of development [Naumann and Jenkins, 1982]. Generally, at the beginning of software development, customers define objects that they wish to address with a

system, but do not have the requirements describing the system features and capabilities in detail. Moreover, requirements are changed or enhanced as customers have a better understanding while the development is in progress. To address these issues, the prototyping approach creates an artifact that focuses on user interfaces without having substantive implementation in a short time. Assessment of the prototype is performed against customer requirements. Thus, a prototyping model helps customers to understand and identify what they require from the system. The main application areas of a prototyping model are to assist with feasibility and usability tests.

The Spiral model is another improvement that combines ideas from several software development processes. The Spiral model was proposed by Boehm to guide multi-stakeholder concurrent engineering of software intensive systems [Boehm, 1988]. The two main features of the spiral model are its cyclic approach and its adherence to a set of milestones. The cyclic approach enables the development of a software intensive system incrementally while decreasing the degree of software development risks.

A DSML can be developed using any of the development models just described, but we recommend the iterative and incremental approach because domain users may not have a full set of domain modeling language requirements when elaborating on the needs of their language [Mernik et al., 2005]. The potential high-level of potential maintenance issues is one of the factors that also suggest an iterative and incremental model [Nakatani et al., 1999].

Unlike general application development, DSML development activities are slightly different. As mentioned in the previous section, a DSML consists of three components (i.e., abstract syntax, concrete syntax, and semantics) and these components are interrelated by mapping rules. Thus, DSML development needs to consider how to design and implement each component independently and assemble them seamlessly.

As shown in Figure 2.7, DSML development requires a collaboration of two experts: domain experts and language development experts.

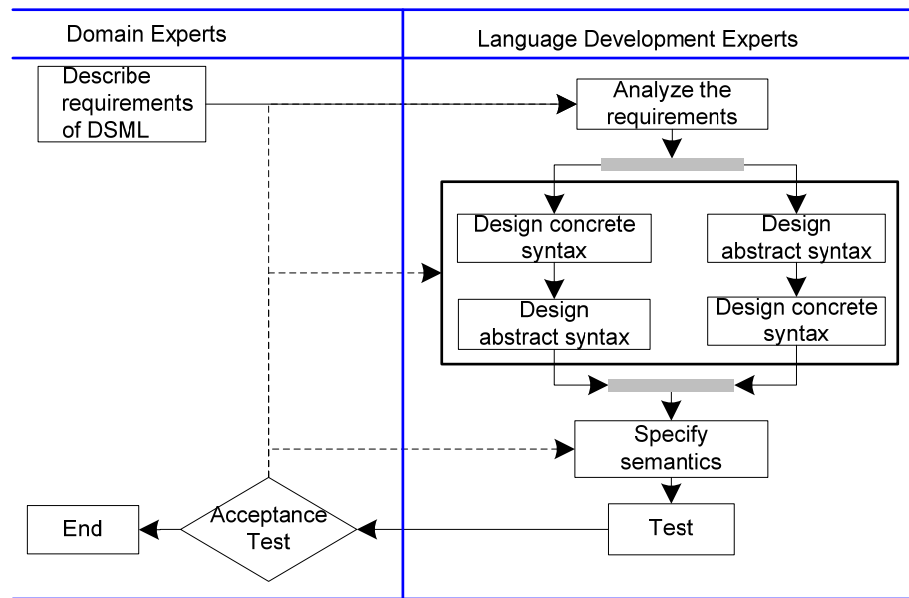


Figure 2.7 DSML Development Process

Domain experts have profound knowledge and experiences within the domain, but do not have language development expertise. The main roles of domain experts are eliciting requirements of the DSML and validating whether the developed DSML meets the requirements for a new language describing the domain. On the other hand, language development experts build a DSML based on requirements specified by domain experts. The major activities of DSML development are analyzing the requirements of the DSML, designing abstract syntax and concrete syntax, specifying semantics, and performing tests.

Requirements analysis is the first step of DSML design. The main activity of requirements analysis is to identify the notions of the domain by acquiring and consolidating

information in the domain in order to design the syntax and specify the semantics. The goals of requirements analysis for DSMLs are as follows:

- **Identify stakeholders of the domain.** Several different types of stakeholders may use a DSML with different purposes. For example, domain experts may use a DSML for describing their domain, and software engineers will use a DSML to understand a domain before they develop a software system for the domain. Because stakeholders have different views and goals, identification of domain stakeholders should be carried out before the other tasks to define the language scope correctly.
- **Define the domain scope.** Domain scoping identifies appropriate boundaries for the domain and considers the existing and expected instances (or notions) within a domain [Pohl et al., 2005].
- **Identify domain notions and notations used in the domain.** Although no DSML may exist for a specific domain, a domain expert may define a set of notions and corresponding notations (or symbols) to model in their domain. In some cases, domain notions are maintained in the form of a data dictionary, and domain notations can be found in documents such as design documents, meeting minutes and presentation slides. Thus, these notions and notations should be identified and documented for the syntax design and semantics specification.

After completing requirements analysis, language development experts need to determine which syntax (i.e., either abstract or concrete) is developed first. Generally, the concrete syntax

is explored before developing a DSML. This is often the case because domain experts may use symbols (or notations) that represent the idioms of discourse and jargon of specific problem domains to document and/or share the notions of the domain with the domain stakeholders. Thus, many DSMLs may begin by designing concrete syntax and then extracting the essence of the language during the abstract syntax design. Abstract syntax-driven DSML development may be preferable if the domain expert is not clear on some new aspects of the domain. This situation often happens when the domain is relatively new and subject to change due to external forces (e.g., new requirements placed on the domain). Thus, both domain and language development experts often work together to concretize the key concepts of the domain and characterize the abstractions required in the domain, with late elaboration of the abstract syntax [Wile, 1997].

As mentioned above, domain notions are captured during requirements analysis. The captured notions are specified using a modeling language during the abstract syntax design phase. UML and Eclipse Modeling Framework (EMF) are examples of modeling languages that are often used to specify abstract syntax specification. To develop quality abstract syntax, Karsai et al. [Karsai et al., 2009] proposed abstract syntax design guidelines. First, abstract syntax and concrete syntax should be aligned to each other. To align both syntax elements, a language designer should keep a one-to-one mapping between abstract syntax and concrete syntax. If similar or duplicated notations exist, they should be merged into single abstract syntax elements. Second, abstract syntax should be designed in a modularized manner. In accordance with the increased complexity of modern software systems, abstract syntax tends to describe complex and large-scale software systems. Thus, keeping abstract syntax modularized can help language maintenance and evolution.

Designing the concrete syntax of a DSML is similar to defining a set of symbols for the domain and mapping those symbols to abstract syntax. The main activity of concrete syntax design is in deciding whether to use a textual or graphical representation after weighing the advantages and disadvantages of the representations against the domain users' preferences [Grönniger et al., 2007]. The advantages of textual representation are that textual representation takes less space than graphical representation when presenting the same amount of information. In addition, textual representation requires neither a specific platform nor a specific environment for reading maintaining, or developing the representation. On the other hand, graphical representation may help users to create a model quickly by supporting drag-and-drop model definition features. In addition, the learning curve may be shortened in both cases, reducing the amount of anxiety that users may have while learning new concepts and/or notations.

In general, we can consider two types of concrete syntax users (e.g., computing devices and humans); each type of user may suggest a different approach to designing the concrete syntax. Computing devices often are connected on a network and able to communicate with each other to share information or collaborate to produce outputs. Concrete syntax plays a key role in making data sharing easy and efficient among heterogeneous computing devices because the syntax defines the structure and representation of information. For this type of concrete syntax user, interoperability among the heterogeneous computing devices will be the most important concern when designing concrete syntax because computing devices may have different computing capabilities and data formats (e.g., big or little endian). In addition, integration is another important factor for designing concrete syntax. Applications (or tools) may define their own concrete syntax, and the difference makes it difficult to integrate applications even when they are executing in the same computing device. XMI is a good example that addresses concrete

syntax design issues for machine-to-machine information exchange. XMI is the standard for representing object-oriented information using XML and enables interoperability among heterogeneous computing devices [OMG XMI, 2011]. In addition, because XMI is capable of representing many forms of object-oriented information, it is used to support lightweight integration among Java applications, the Web, XML, and different kinds of models [Grose et al., 2002].

Another user of concrete syntax is a human who mainly uses the concrete syntax to communicate and share information. Designing the concrete syntax for a human is more difficult than designing for computing devices because a human can interpret or understand the same syntactical forms differently based on his/her domain knowledge and cultural background. Thus, designing concrete syntax for a human may require multi-disciplinary skills such as computer language design, cognitive science, psychology, and graphic design [Selic, 2009].

After the abstract and concrete syntax is designed for a DSML, the semantics should be specified and associated to the abstract syntax. Specifying the semantics of a language involves three activities: 1) understanding the designed syntax of a language, 2) choosing a semantic domain, which is the underlying formalism used to define the language, and 3) mapping from the syntax to the semantic domain [Harel and Rumpe, 2004].

## CHAPTER 3

### SYNTAX MAP: MODELING LANGUAGE FOR DSML REQUIREMENTS

The success of software system development and evolution depends on how well the needs of its users and its environment are met [Nuseibeh and Easterbrook, 2000; Parnas, 1999]. A good set of requirements are critical for the success of software development projects because requirements are the primary tool for communication among stakeholders and the measure of software quality. In addition, requirements are used to describe what users want from the system, to partition the work out for contract, and acts as a basis for verification and validation.

All deliverables of software system development should be documented and designed to conform to the requirements [Crosby, 1979]. However, descriptions of requirements often evolve as development of a system progresses. For example, at the systems requirements elicitation phase, the requirements define the functional features that the system must provide, as well as the non-functional needs (e.g., performance, reliability, and usability). At the architectural design and detailed design phases, the requirements need to describe the cost and benefits, as well as the potential changes needed to address architecture and component design decisions. In addition, the qualification requirements of the system are also described for system testing and acceptance testing [IBM Rational Doors]. Thus, requirements management is often called the umbrella process because requirements span all phases of the software development lifecycle rather than being specific to any particular life cycle stage.

However, describing a good set of requirements is challenging because it generally can be obtained only after analyzing the problem space. The problem space is the space where

“simple” solutions would not adequately solve the problem. In addition, the problem space is less constrained than the solution space and comprises ill-defined and conflicted descriptions [Cheng and Atlee, 2007]. To address the difficulties of obtaining a good set of requirements, many researchers and practitioners have introduced various techniques such as Goal-oriented [France and Rumpe, 2007;], Use Case [Fantechi et al., 2002; Maiden and Robertson, 2005], and Scenario-based [Maiden and Robertson, 2005; Uchitel et al., 2002] requirements engineering.

Although many approaches and techniques have been proposed to develop and manage the requirements of application domains, there is little research and empirical evidence on how to apply those approaches and techniques to manage the requirements of computer languages, specifically modeling languages. In addition, domain experts may suffer from the lack of an appropriate method, guidance, or tool support for capturing the requirements of a DSML.

In this chapter, we introduce an approach, named Syntax Map, for capturing and managing the requirements of DSMLs. A Syntax Map aims to help communication between domain experts and programming language development experts by capturing the requirements of a DSML, specifically DSMLs represented with graphical notations. Because miscommunication is one of the factors leading to failure of software system development, Syntax Map addresses those issues by offering intuitive graphical notations to model the requirements of graphical DSMLs precisely and concisely.

### 3.1 Requirements Modeling

Eliciting and analyzing requirements are essentially a cooperative work because only domain experts really know the deepest details of the problem, but an analyst is often needed to

help the domain experts in fully and correctly describing it. Thus, choosing the right requirements modeling approach plays an important role in accelerating cooperative work.

### *3.1.1 Natural Language*

Natural language is commonly and heavily used to describe the requirements of software systems because the initial requirements are often gathered from informal activities (e.g., user interviews or brainstorming) or from documents of previous projects. In addition, natural language is expressive and flexible for processing such initial inputs [Hsia 1993; Kaindl et al., 2002; Berry 2003; Luisa et al., 2004]

However, due to the informal nature of natural language, requirements described by natural language are highly prone to ambiguity and may not give the same interpretation when read by different persons. Thus, to minimize and avoid ambiguity and misinterpretation, requirements should be specified with the following characteristics [Wieggers, 1999; IBM Rational DOORS]:

- **Requirements should be described completely and correctly.** All required functionality and necessary information should be included in the requirements specification. However, identifying missing requirements or information is not an easy task. To write requirements completely and correctly, requirements should be written by focusing on user tasks rather than on system functions. In addition, graphical and/or formal methods can help to reveal incompleteness within the requirements specification.

- **Requirements should be described consistently.** Some requirements are often conflicted with other requirements or with requirements focused on a different level (e.g., user requirements vs. system requirements). Thus, to keep the requirements consistent, they should be reviewed by all stakeholders whenever change requests are made.
- **Requirements should be verifiable.** To be verifiable, the requirements must use statements that can be verified by examination, analysis, test, or demonstration. Thus, requirements statements should avoid using subjective or subjective words such as “easy,” “efficiently,” and “adequately.” In addition, verifiable requirements help to eliminate ambiguity in requirements.
- **Requirements should be traceable.** Because requirements affect all development activities (e.g., design, implementation and testing), requirements traceability plays an important role in several ways. First, requirements traceability ensures that the final deliverables should satisfy initial customer demands. Second, if requirements traceability is done properly, it can be used to estimate and evaluate project progress, especially cost, time, and resources. Finally, requirements traceability can help in analyzing the impacts of a system from change requests.

To satisfy the characteristics of good requirements as described above, each requirements statement should have: 1) a user who benefits from the requirement or drives the need for the requirement, 2) a state that the user wants to reach after requirements are executed, and 3) a metric or some mechanism to verify the correctness of the requirements.

### 3.1.2 *Semi-Formal Methods*

Semi-formal methods use formal syntax (e.g., graphical notation such as diagramming techniques and/or tabular techniques) to model a domain, but their semantics are not formally defined using mathematical notations (e.g., logics and set theory). By using graphical notations, Semi-formal methods can analyze and design software systems with a high-level of abstraction [Schmidt, 2006; Watson, 2008] and is able to validate requirements and design at an early stage of development. In addition, semi-formal methods can minimize miscommunication.

A well-known semi-formal requirement model is a Use Case model, which is currently considered the state-of-the-art for describing and modeling requirements because it offers several benefits [Fowler 2003; Cockburn, 2000]. First of all, requirements can be elicited and specified from the user's point of view. In general, the context of a system is defined as a part of requirements elicitation and analysis, and users' of the system are defined in terms of input. Users are often ignored or described ambiguously when specifying requirements with natural language. A Use Case model can tackle this issue by asking to describe requirements from a user's point of view. Second, a Use Case model can minimize miscommunication between stakeholders by providing different levels of abstraction. If software engineers talk to non-technical stakeholders (e.g., marketing team and/or financial team), they explain and discuss the system with graphical notations of the Use Case model. If software engineers need to talk to software architects, they can use both graphical notations of a Use Case model and its specification, which describes detailed functional and non-functional requirements of the Use Case.

### 3.1.3 *Formal Methods*

Formal methods can address issues of natural language and semi-formal methods for modeling requirements by describing requirements with a very precise mathematical notation. Model-driven, Process algebra, and axiomatic methods can be used to model requirements formally.

Model-driven formal methods describe requirements using a set of models, which are specified as objects and their operations. Model-driven formal methods need to explicitly define the types of objects of concern and utilize primitive, predefined operations in defining higher-level operations [Pedersen and Klein, 1988; Aichenig, 1999]. B [Abrial, 2005; Julliand and Kouchnarenko, 2007], Z [Potter et al., 1996; Woodcock and Davies, 1996], and Vienna Development Method (VDM) [Björner and Jones, 1978; Pedersen and Klein, 1988] are examples of formal methods. Process algebra originated from the mathematical theory of automata. The term process algebras were coined in 1982 by Jan Bergstra and Jan Willem Klop [Bergstra and Klop, 1982] and the process algebras focus on the specification and manipulation of process terms, which are induced by a collection of operator symbols and interaction and communication between processes to achieve a common goal [Fokkink, 2009]. CCS (Calculus of Communicating Systems) [Milner, 1989; Bruns, 1997] and CSP (Communicating Sequential Processes) [Hoare, 1978; Roscoe, 1997] are examples of process algebras.

Axiomatic formal methods [Hoare, 1969] use axioms to define properties of systems but the axioms are restricted to equations. Axiomatic formal methods are well-suited for algebraic specification of abstract data types, and ACT ONE [Claßen, 1989; Claßen et al., 1993] and its

extension ACT TWO [Fey, 1988] are languages for axiomatic formal methods. Both languages use algebraic specifications with conditional equational axioms to specify a system's behaviors formally.

### 3.2 Goals and Requirements of Syntax Map

A language typically consists of three components: abstract syntax, concrete syntax, and semantics. Semantics define the meaning of the language and can be specified using one or more formal methods (e.g., Denotational Semantics, Axiomatic Semantics, Operational Semantics, or Attribute Grammars). Concrete syntax is used to represent the surface-level notion of a language. In programming languages, the concrete syntax is often textual and described using EBNF. In modeling languages, the concrete syntax could be textual, graphical, or mixed. Abstract syntax describes the notion of a language, the relationships between the notions, and the well-formedness rules, which specify how the concepts can be combined. Generally, the abstract syntax is defined using an EBNF for programming languages and a metamodel for modeling languages.

To develop a new graphical DSML, the requirements of the DSML can be described using three language components: the abstract syntax, a mapping relationship between the abstract syntax and concrete syntax, and semantics. However, it is not easy to describe the requirements of a DSML completely and precisely, and domain experts may suffer from the lack of an appropriate method, guidance, or tool support for capturing the requirements of a DSML. We have investigated an approach to address these issues and have developed the idea of a Syntax Map, which assists in capturing the requirements of a new graphical DSML. The Syntax Map's focus is to assist domain experts, who may have little experience or knowledge about

development of graphical modeling languages, in describing the requirements of their own graphical DSML. The Syntax Map captures elements of syntax and the relationship (or structures) between the elements.

The detailed requirements of a Syntax Map are specified as follows:

- A Syntax Map should be able to describe the requirements of DSMLs, specifically graphical DSMLs, using graphical notations.
- A Syntax Map should be able to define classifiers, which are modeling elements that describe behavioral or structural features in a system. Actors, Use Cases, classes, and interfaces are examples of classifiers.
- A Syntax Map should be able to define at least three relationships: association, aggregation, and inheritance. The relationship is a connection between modeling elements and can assist in defining the semantics of a model. Association, aggregation, and inheritance are the three key relationships in modeling a domain, especially in Object-Oriented Modeling [UML Infrastructure; UML Superstructure]. Association represents a structural relationship between two model elements and depicts the possible connections from one instance of a classifier to another instance of a classifier. In addition, association describes the direction of navigation between linked modeling elements. An aggregation relationship represents the whole-part relationship between classifiers; a classifier is a part of (or subordinate to) another classifier [Barbier et al., 2003]. The aggregation relationship is often used to represent the containment relationship by describing how classifiers are assembled or configured together to create a more

complex classifier. The inheritance relationship expresses the *is-a* relationship between two classifiers. It encourages the reuse of existing data and code. When classifier *A* inherits from classifier *B*, we say *A* is the subclass of *B*, and *A* can access all the attributes and methods of *B*.

- Each classifier and relationship in a Syntax Map shall be associated to the relevant concrete syntax. Concrete syntax visualizes the notion of abstract syntax using textual, graphical or mixed representation and helps domain experts to understand the language. The association should be a one-to-one correspondence between the abstract syntax and concrete syntax. By limiting the correspondence between abstract syntax and concrete syntax as a one-to-one mapping, notations used in the Syntax Map can address concerns of precision and concise expressiveness [Moody 2009; Goodman, 1968].
- Each concrete syntax element of a Syntax Map should be distinguishable and unique in the language.
- A Syntax Map should be able to describe the cardinality (or multiplicity) between model elements linked by relationship. The cardinality expresses the upper or lower limits when two classifiers are linked by a relationship.
- Each Syntax Map should have one start symbol, and each Syntax Map should be able to have multiple end symbols. A Syntax Map should have at least one start symbol and one end symbol to be a valid model.

### 3.3 Design of Syntax Map

This section describes the design of the concrete and abstract syntax of the Syntax Map based on the requirements from Section 3.2.

#### 3.3.1 *Concrete Syntax of Syntax Map*







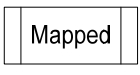

In this section, we describe the concrete syntax of the Syntax Map. As described in Chapter 2, DSML development may begin by designing either the abstract syntax design or the concrete syntax design. Considering the abstract syntax first is useful when a domain is mature and programming language development experts have deep domain knowledge and experience. However, DSML development that is driven by concrete syntax is generally used more frequently because identifying the concrete syntax is relatively easier than identifying the abstract syntax. Also, domain experts may have already defined or used a set of notations to model and communicate with their stakeholders.

Syntax Map development begins by defining the concrete syntax first. To describe the requirements of a graphical DSML, the Syntax Map provides eight symbols, as shown in Table 3.1. To define the concrete syntax of a Syntax Map, we analyzed several diagrams (e.g., Activity diagram, Flowchart, and Entity-Relationship Diagram) and then selected notations that have a high probability of uniqueness among domain experts.

A valid Syntax Map must contain at least two symbols, *Start* and *End*, which represent the start and end of the DSML requirements description. These two symbols are represented by a circle (*Start*) and bar (*End*). Similar to declarative programming (or modeling) languages,

Symbol *Start* and *End* are defined to structure requirements of a DSML according to each syntax usage scenario.

Table 3.1 Concrete Syntax of Syntax Map

Symbol	Name	Description
	Start	Indicate start of the Syntax Map.
	End	Indicate end of the Syntax Map.
	Classifier	Represent classifiers (or entity) in abstract syntax.
	Relationship	Represent relationship between classifiers.
	Attribute	Associate attribute to classifier. If necessary, relationship can have an attribute.
	Data Type	Represent attribute type. Basic data type (e.g., String and Numbers) is provided.
	Mapped	Mapping between abstract syntax and concrete syntax.
	Link	Link between the Syntax Map elements.

A classifier is denoted as a rectangle, and at least one classifier should be linked to the *Start* and *End* symbols. If necessary, attributes can be associated to a classifier. When defining attributes, the type of the attribute should be provided. Primitive data types (e.g., String and Number) are provided as built-in Data Types. In addition, domain experts can define composite data types by combining primitive data types. To represent relationships between classifiers, a rounded rectangle is placed between classifiers.

To link between the Syntax Map symbols, two types of link symbols are provided. One is a line and the other is an arrow-headed line. If an arrow-headed line is used to link between classifiers and relationships, it implies that there is directional dependency or information flow between the two. If a line is used, information can flow in both directions between classifiers and relationships. After domain experts connect classifiers to relationships, they need to specify the attributes of the relationship. The relationship has four attributes to characterize the relationship between classifiers: *Type*, *Directional*, and *Cardinality of Source and Target*. Attribute *Type* defines four types of relationships by default: association, aggregation, composition, and inheritance. If a domain expert assigns one of these types as the name of the relationship, the Syntax Map automatically sets the *Type* attribute corresponding to the assigned name. If a domain expert assigns a name that is different from the name in the *Type* attribute, then the type of the relationship needs to be defined. The attribute *Direction* specifies the direction of information flow or dependency. The semantics of the *Direction* attribute is determined by combining the types of links that connect between classifiers and relationships. The *Cardinality of Source and Target* describes the number of possible classifiers linked by the relationship. Finally, the symbol *Mapped* is used to map the abstract syntax to the concrete syntax. *Mapped* has two attributes: *Constraint* and *Rendering*. The former is used optionally when an abstract syntax needs to be mapped to multiple different types of concrete syntax. The mapping constraint is specified informally using natural language because the Syntax Map is used to capture requirements of a graphical DSML in the early development stage. The *Rendering* attribute specifies a path where a graphic symbol is stored. When a domain expert specifies the path attribute in *Rendering*, the Syntax Map reads the file and displays the concrete syntax in the symbol *Mapped*.

### 3.3.2 Metamodel of Syntax Map

Based on the requirements and specifications described in the last section, a Syntax Map metamodel is defined in Figure 3.1.

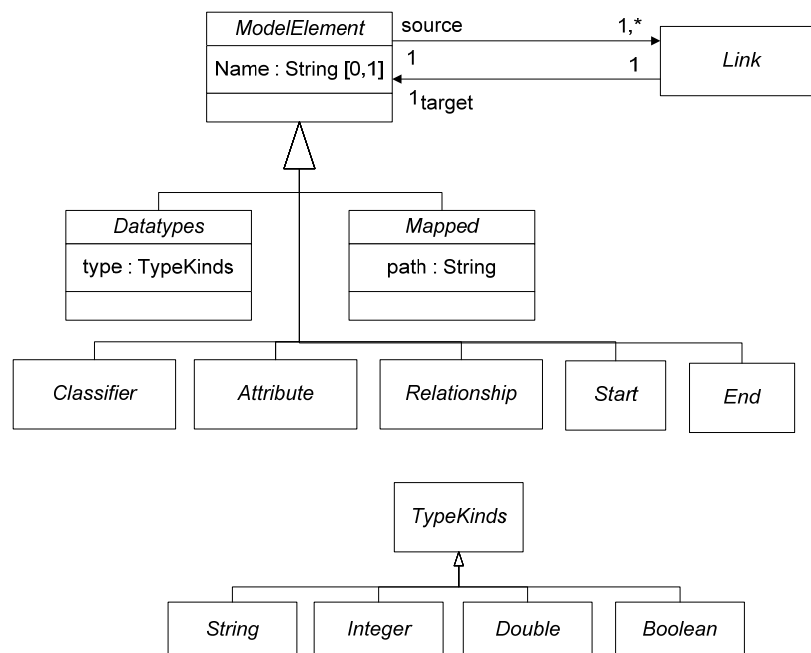


Figure 3.1 Syntax Map Metamodel

*ModelElement* is the top-most base class for all modeling elements (except *Link*). *ModelElement* has string type attribute *Name*. The multiplicity of attribute *Name* is defined 0 or 1, because some modeling elements have attribute *Name* and some do not. For example, modeling elements classifier, attribute, and relationship have attribute *Name*, but modeling elements *Start* and *End* do not.

Because each modeling elements need to be connected to at least one or more other modeling elements, a *ModelElement* is associated with *Link* in a bi-directional manner. If *ModelElement* is the source, it can have more than one *Link*. On the contrary, if *ModelElement* is target, a *Link* can be connected with only one model element. Thus, *Link* and *ModelElement* are linked by a one-to-one relationship.

The *Mapped* class maintains the information about concrete syntax and an additional attribute called *Path* is defined to specify the location of the concrete syntax notation. The *Datatypes* class defines the data types that are used for a Syntax Map. To define the data types, the *Datatypes* class refers to the *TypeKinds* class, which is the base construct for representing datatypes; it has four subclasses (e.g., String, Integer, Double, and Boolean).

### 3.4 Guidelines for Describing DSML Requirements using Syntax Map

A good set of requirements usually contains at least three characteristics. First, each requirement should be a complete sentence because single words, phrases, and collections of acronyms may be interpreted several ways according to each reader's background and experiences. Thus, a requirement statement should have at least a subject and a predicate. Second, a requirement may describe a defined, desirable quality (e.g., performance, reliability, or usability). Finally, a requirement should contain metrics or a mechanism for testing against the requirement. In order to describe the requirements of a graphical DSML using the Syntax Map, the following steps are recommended to domain experts:

- **Start with a classifier.** Classifiers represent notions that are commonly understood in a domain. A Syntax Map can describe the requirements of a DSML based on the use cases of classifiers.
- **Model flow from left to right.** In general, a flow is implicitly modeled to run from left to right or from top to bottom.
- **Define classifiers.** A Classifier is an abstract metaclass that is used to describe a set of instances that have common features. The classifier is used for defining a namespace, type, and redefinable elements.
- **Add attributes to each classifier if necessary.** A classifier may have structural and/or behavioral characteristics in a specific domain. An attribute is one way to represent the characteristics of a classifier.
- **Define relationships and relate relationships with relevant classifiers.** Relationships play important roles in modeling languages because they describe structural relationships between classifiers.
- **Specify additional attributes to relationship** (e.g., relationship type, cardinality, and directional information). This can be added to provide semantic information about the relationship after it is assigned to classifiers.
- **Associate concrete syntax.** If the concrete syntax of a graphical DSML has been identified already, the concrete syntax can be associated with the corresponding abstract syntax element.

### 3.5 Application of Syntax Map

This section shows how to use the Syntax Map for describing the requirements of a DSML. To describe the requirements of the abstract and concrete syntax of a DSML, we will use a Deterministic Finite Automation (DFA) as an example language. Consider the specific DFA, shown in Figure 3.2, which computes whether the input binary numbers are multiples of 3. If a binary number 11 is entered, the DFA jumps to  $S_1$  for the first 1 and then returns to  $S_0$ , which is the accepting state that represents the input binary number is multiples of 3. As shown in Figure 3.2, the DFA uses three unique symbols, *Accepting State*, *State* and *Transition*, to model the DFA for checking multiples of three.

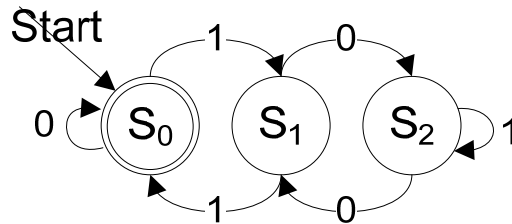


Figure 3.2 DFA for Checking the Multiples of 3

To describe the requirements of the DFA with a Syntax Map, a domain expert first creates the skeleton scenarios by placing classifiers and relationships between the Syntax Map symbol *Start* and *End* according to their usage scenarios. For instance, if the current state is the *Accepting State* and a transition is triggered, the next state is determined by the transition

condition and can be either *Accepting State* or *State*. In addition, *State* can transition to either *Accepting State* or *State*.

The skeleton Syntax Map for a DFA is shown in Figure 3.3.

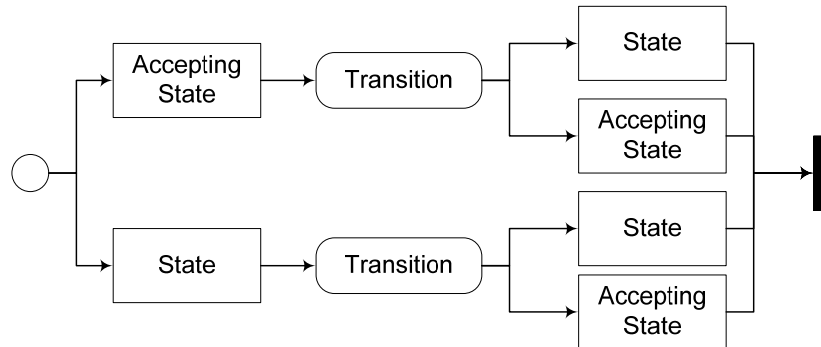


Figure 3.3 The Skeleton of the Syntax Map

The upper part describes the state transition scenario from the *Accepting State*, and the lower part is for the *State*. After the skeleton of a Syntax Map is created, a domain expert can add additional information to each Syntax Map modeling element if necessary. If the classifiers (or relationships) have the same attributes across the Syntax Map, a domain expert specifies the attributes to only one classifier. Then, the rest of the classifiers will share the attributes. For example, although another classifier *State*, which is connected with symbol *End*, is modeled without any attributes, attribute *Name* will be associated automatically with classifier *State*, which is connected to symbol *Start*. The complete Syntax Map is shown in Figure 3.4.

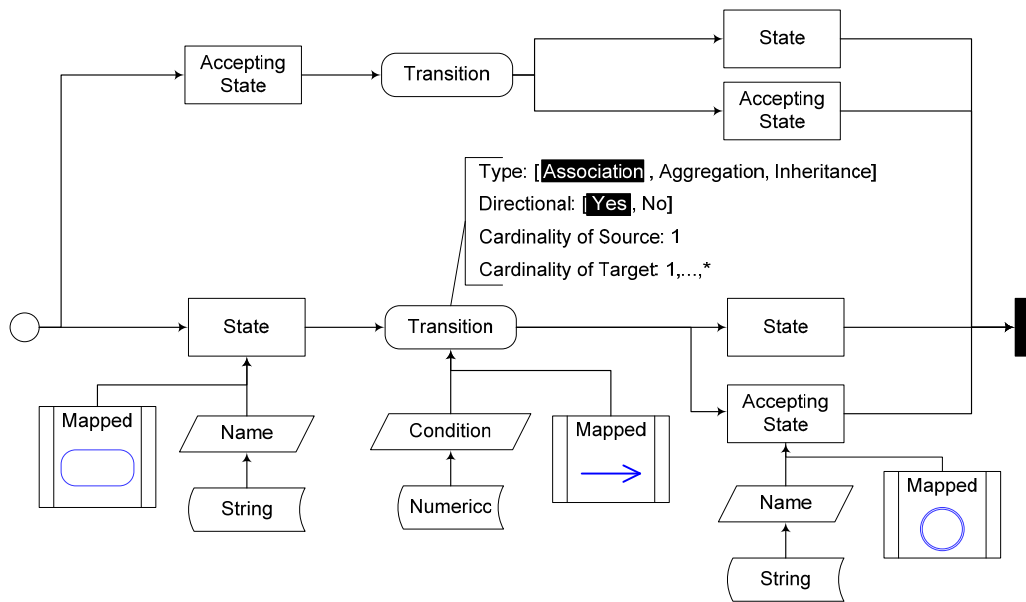


Figure 3.4 The Complete Syntax Map for the DFA

### 3.6 Related Work

Use Case Maps (UCMs) were proposed by Amyot et al. to provide “*a notation to aid humans in expressing and reasoning about large-grained behavior patterns in systems*” [Buhr and Casselman, 1995]. The basic idea of UCMs is to capture the requirements of a system by introducing a scenario-based software engineering technique, which can “*describe causal relationships between responsibilities of one or more use cases*” [Buhr and Casselman, 1995; UseCase Maps]. Due to the nature of a scenario, UCMs are useful in capturing informal (or functional) requirements, and validate logical errors in requirements. In addition, UCMs can be transformed into Language Of Temporal Ordering Specification (LOTOS) [Amyot and Logrippo, 2000] or Specification Description Language (SDL) [He et al., 2003]. Due to simplicity, UCMs have been applied successfully for capturing requirements of software systems, documenting

standards, and evaluating the alternates of architecture. The main ideas of UCMs (e.g., scenario-based requirements capture and graphical representation) influenced the design of the Syntax Map. To develop a graphical DSML using the Syntax Map, domain experts should first define the concrete syntax and then the abstract syntax. In order to design the abstract syntax, domain experts may examine possible scenarios of each element of the concrete syntax, and focus on the possible relationship combinations among the elements of the concrete syntax. Thus, similar to UCMs, the Syntax Map helps domain experts to design the abstract syntax by considering the usages (or relationships) of each concrete syntax element.

A Syntax Graph [Taylor, 1961] is represented in a directed graph and also called a syntax diagram, or syntax chart. The syntax graph was first used to document the syntax of ALGOL 60 in a condensed form for reference during compiler development. The syntax graph is similar to a flow chart, which can represent flow. It is designed to be able to define language constructs as metalinguistic formulas, which consist of metalinguistic variables and basic symbols. The syntax graph helps in checking the syntax of a program and can be used to train programmers. Because the syntax graph is useful to illustrate syntax structure and/or data structure visually, it is used to illustrate the syntax for several different contexts, such as SQL [IBM Database Fundamental; Oracle Syntax Diagram] and web services [SharePoint 2010 REST]. Similar to the syntax graph, the Syntax Map can also be used to document abstract syntax or illustrate syntax structure. The Syntax Map can be used as an input in order to develop a graphical DSML through transformation.

### 3.7 Conclusion

Requirements are key artifacts for developing and evaluating a software system and, thus, many researchers and practitioners have invested much time and effort to develop and manage a good set of requirements. For instance, the Institute of Electrical and Electronics Engineers (IEEE) define standard guidelines and processes for developing and managing requirements, such as the IEEE Recommended Practice for Software Requirements Specifications (IEEE Std 830-1998). However, specifying and maintaining quality requirements are not easy tasks because requirements are gathered through informal methods (e.g., interviews and brain storming) and are often described using natural languages, which are inherently ambiguous. To address these issues, researchers and practitioners have proposed many approaches, methods, and tools.

In this chapter, we presented a semi-formal modeling approach (named Syntax Map) for specifying the requirements of DSMLs. The requirements for a DSML should describe all three components of the language (i.e., abstract syntax, concrete syntax, and semantics). Domain experts need to specify consistent and verifiable requirements of a DSML of three components by referring to appropriate guidelines and methods. To address this need, Syntax Map offers a set of graphical notations for specifying the requirements of a DSML. In addition, Syntax Map offers several advantages, as follows:

- Syntax Map encourages domain expert involvement. One of the issues for quality software development is that domain users are inadequately involved in requirements elicitation. In addition, domain experts may not have expertise for

describing their requirements using traditional notations suitable for computer scientists. To encourage domain expert involvement in describing quality requirements, a Syntax Map provides a small set of graphical notations that can assist domain experts in describing a DSML based on the usage scenarios of each language.

- A Syntax Map helps domain experts and programming language development experts to verify the requirements of a DSML. With a Syntax Map, the requirements of a DSML are described by classifiers and their usage scenarios. This allows domain experts to describe the requirements directly related to each DSML.
- Each path in a Syntax Map represents a logically meaningful modeling unit. By breaking the requirements of a graphical DSML into several logically meaningful modeling units, domain experts and programming language development experts can share their understanding about the DSML they are developing. In addition, each modeling unit helps domain experts and programming language development experts to find missing and/or redundant requirements of a DSML.

## CHAPTER 4

### METAMODEL DESIGN PATTERNS

Software reuse helps to develop quality software while shortening development time and minimizing investment [Krueger, 1992; Mockus, 2007; Mohagheghi and Conradi, 2008]. Recently, design patterns have been adopted widely as a type of software reuse, especially for design reuse, because they reflect the experience and knowledge of designers who have successfully solved recurring problems in different contexts. Similarly, even though DSMLs are developed to be used for a specific domain, there exist recurring problems when designing modeling languages regardless of the domains of interest. Thus, extending the notion of design patterns to metamodel design can contribute to the design quality of a metamodel. In addition, metamodel design patterns are able to guide metamodel inference.

In the MLCBD approach, the abstract syntax (or metamodel) is inferred by the Metamodel Inference Engine on the graph representation with the concrete syntax (see Chapter 5). Generally, metamodel inference can be considered a special case of inductive learning, which induces output by learning from examples [Higuera 2005; Michalski, 1983]. To infer a quality metamodel, the inference engine requires a large set of training data, which contains positive examples (i.e., a set of data that belongs to the target) and negative examples (i.e., a set of data that does not belong to the target), but preparing such training data is challenging in practice [Kirsopp and Shepperd, 2002; Cho et al., 2011]. To address the issue of preparing training data

for inductive learning, we introduce the notion of metamodel design patterns and how those patterns assist in generating a basic training data set.

#### 4.1 Approach of Metamodel Design Pattern Mining

Metamodel design patterns extend the notion of design patterns onto metamodel design in order to provide solutions for recurring problems when designing a metamodel. To mine metamodel design patterns, we took the following steps [Cho and Gray, 2011]:

- **Step 1 - Context setting:** To identify issues of metamodel design, we reviewed the concrete syntax of several DSMLs and modeled their commonality and variability. Because complete DSMLs are challenging to obtain from industrial settings, we include GPMLs such as UML diagrams, assuming that each diagram can be tailored for a specific domain. A feature model [Kang et al., 1990] was used to summarize our understanding of commonality and variability in the DSML examples that we analyzed.
- **Step 2 - Identification of metamodel design problems:** Based on the feature model that was created from the analysis of DSML concrete syntax, we observed several metamodel design challenges. To derive the recurring metamodel design idioms, we focused on the commonalities in the DSML feature model. These commonalities represent a few of the common features at the core of metamodeling.

- **Step 3 - Metamodel design pattern creation:** Based on the identified problems from Step 2, we searched and analyzed relevant metamodels and created a metamodel design pattern for each problem identified.

#### 4.1.1 Context Setting

To identify the commonly recurring metamodel design problems, we examined the concrete syntax of several DSMLs (see Table 4.1 for example domains), with a specific focus on classifiers and relationships. To generalize the concrete syntax of DSMLs, we assume that most modeling languages commonly use a *Box-and-Line* style, even though there is some disagreement in the community on how to interpret and understand the syntax and semantics of graphical languages [Balakrishnan and Reps, 2010; Kopp et al., 2009; Petre, 1995]. Typically, *Boxes* represent the instances of the domain concepts such as key functionalities or behaviors, and *Lines* that connect *Boxes* describe how the connected *Boxes* communicate or are related to each other syntactically and semantically. The key benefit of using the *Box-and-Line* style is its simplicity, and thus, many modeling languages inherently contain the notion of *Box-and-Line* even though they are realized with different concrete syntax. For example, Petri Nets define four basic symbols (i.e., Places, Transitions, Directed arcs, and Marks) to model and analyze reachability, liveness, and boundedness of concurrent discrete event systems [Murata, 1989; Ouardani et al., 2006]. Places and Transitions, denoted by circles and rectangles (or bars), correspond to *Boxes*; Directed arcs, represented by arrows, correspond to *Lines*.

Table 4.1 Listing of Example Domains for Representative DSMLs

Domain	Diagrams	Brief Description	Key Modeling Elements	Containment/ Nesting		Relationship	
						Style/ Boundedness	
Concurrent Discrete Event System Modeling	Petri Net	Modeling systems with concurrency and resource sharing	Place, Transition (C), Directed Arc (R)	A	N	Directed	Closed
Data Modeling	ERD	Model the logical structure of database	Entity(C), Relation(R)	N	N	Directed	Closed
Project Management	Gantt Chart	Model project activities with relevant information (i.e., duration, cost, ...)	Task(C), Predecessor (R)	N	N	Directed	Open
	PERT Chart	Identify the critical path of the project by modeling the sequence of tasks	Task(C), Directed arcs (R)	N	N	Directed	Closed
Electronic Circuit Design	Schematic Diagram	Represent how electronic components are connected with others	Component (C), Line(R)	N	A	Undirected	Closed
	PCB Layout	Show the placement of electronic components on printed circuit board	Hole (C), Line (R)	N	N	Undirected	Closed
SW Design	Flowchart	Model process or algorithm	Symbols (C), Connector(R)	N	N	Directed	Closed
	Component Diagram	Represent static structure of components and their relations	Component, Interface, Port (C), Connector (R)	A	A	(Un)Directed	Both
	UseCase Diagram	Describe system functionalities or behaviors with UseCase and Actor	UseCase, Actor (C), Relation (R)	N	A	(Un)Directed Typed	Closed
	Class Diagram	Describe the static structure of the system in terms of classes	Class (C), Relation (R)	N	N	(Un)Directed Typed	Closed

Based on this observation, prior to identifying metamodel design patterns, the concrete syntax of DSMLs should be identified and generalized from model instances. In particular, we paid close attention to what and how many modeling entities are used in DSMLs, and what and how the relationships link modeling elements both syntactically and semantically.

#### 4.1.2 Identification of Metamodel Design Problems

Based on the analysis of existing DSMLs, we identified the commonality and variability among several DSMLs and derived a feature model as shown in Figure 4.1.

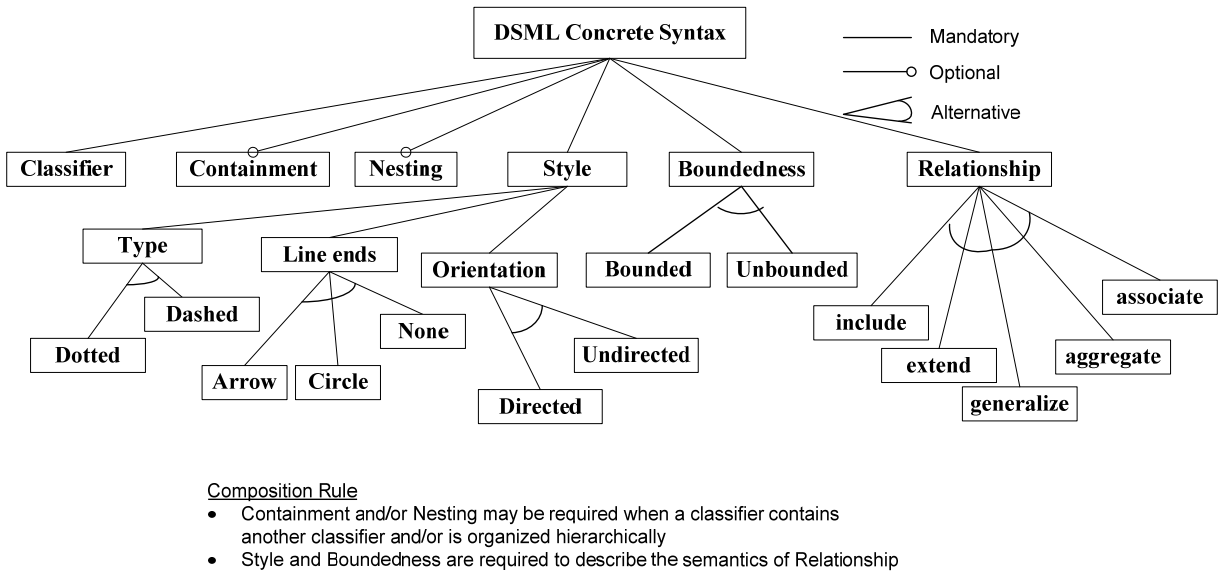


Figure 4.1 Feature Model of DSML Concrete Syntax

Four major features (i.e., Classifier, Relationship, Style, and Boundedness) are defined as mandatory features. There are two other features (i.e., Containment and Nesting), which describe characteristics of a classifier, that are defined as optional features. In addition, Sub features of

Type, Orientation, Line Ends, and Boundedness are defined as an alternative feature because a relationship can have only one kind of Type, Orientation, Line ends, and Boundedness.

Based on the feature model in Figure 4.1, we derived the following questions that relate to metamodel design challenges:

- *How to design a metamodel if the concrete syntax of the DSML consists of simple boxes and lines?* This question will examine how to design a metamodel for a very primitive concrete syntax, which consists of classifiers and association relationships. Thus, the solution for this problem will be the base metamodel, and metamodels for complex DSMLs will be designed by extending this base metamodel.
- *How to design or evolve a base metamodel if the concrete syntax is more complex (e.g., classifiers are linked with several different types of relationships)?* This is generally required for both GPMLs and DSMLs. For example, in a Use Case diagram, a Use Case can be linked with other Use Cases that include or extend the relation. This question may also be important in the design of DSMLs, which heavily depend on relationships between classifiers to describe domain knowledge.
- *How to represent boundedness of a relationship?* Generally, most DSMLs implicitly enforce that both ends of a relationship are bounded to classifiers to represent which classifier drives a behavior and which classifier reacts to the action. In some cases, one end of the relation can be open. A DSML for representing chemical structure [Bentley et al., 1987] can be a good example for

this case because some chemical structures have lone pairs of electrons, which are not involved in chemical bond formation, as well as bonding pairs.

- *How to design a metamodel to represent containment and nesting?* Some DSMLs may contain one or more types. Petri Nets and Activity Diagrams are examples of languages that have containment. As mentioned above, Petri Nets are defined with four modeling elements (i.e., Places, Transitions, Directed Arcs, and Marks). Places represent the pre- and post-state of a system by transition, and a transition shows the place where events occur. Directed arcs show the direction of a transition. Transitions between places are determined by the contained number of tokens in a place and are fired when one or more start places, linked to the same transition, contain enough tokens to satisfy the firing condition. Nesting can be a special case of containment and used to control the level of abstraction by organizing classifiers hierarchically.

## 4.2 Identification of Metamodel Design Patterns

Based on the questions described in Section 4.1.2, we introduce a set of solutions in this section. The solutions are presented by investigating several metamodels, including UML. We use object-oriented notations, such as those used in class diagrams, to represent metamodel designs.

### 4.2.1 Metamodel Pattern for Base Metamodel

Extension to a base metamodel is proposed as a candidate solution for the first question related to metamodel design when the concrete syntax consists of boxes and lines. Consideration

of metamodel design for the simple *Box-and-Line* style DSMLs is important because this style may be used when requirements of a DSML are captured at an initial sketch level, which may occur at the early stage of DSML development. This issue emerges when a domain needs to be modeled with a very high level of abstraction.

In the *Box-and-Line* style, *Boxes* are generalized as a set of classifiers and *Lines* are mapped to relationships. As a relationship normally links two classifiers, one for the source classifier and the other for the target classifier, the classifier and relationship are linked with two association relationships, source and target.

Multiplicity is assigned to the association in order to specify the number of participating instances. In addition, it can also be used to describe the boundedness of a relationship. For example, Figure 4.2(a) shows the relationship links for two classifiers with source and target, which denotes the situation where at least one source and target exist due to both the multiplicity of source and target being specified as one-to-many. On the contrary, in Figure 4.2(b), the multiplicity of source and target is set to one-to-many and zero-to-many, respectively. This means that there exists at least one source, but the target may or may not exist in the relationship.

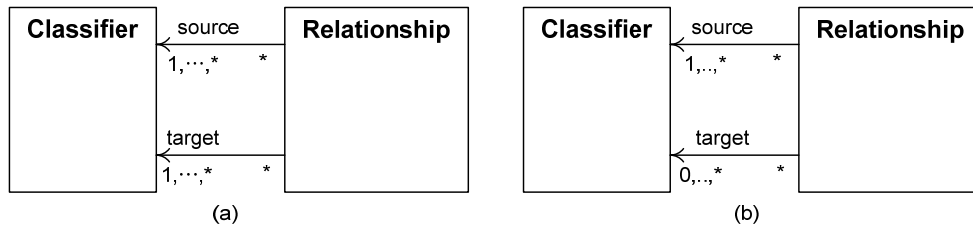


Figure 4.2 Base Metamodel Design Pattern

#### 4.2.2 Metamodel Pattern for Typed Relationships

Associations represent a common relationship type in DSMLs. However, several types of relationships may exist to enrich the semantics between linked model elements. For example, a Use Case diagram has two typed relationships, such as include and extend. A Class Diagram has three typed relationships (i.e., inheritance, aggregation, and composition) in addition to association.

Several metamodel designs for typed relationships and classifiers have been presented in the literature. Figure 4.3(a) is from the UML Superstructure Specification v2.4 [UML Superstructure], and Figure 4.3(b) is simplified from Ouardani et al. [Ouardani et al., 2006].

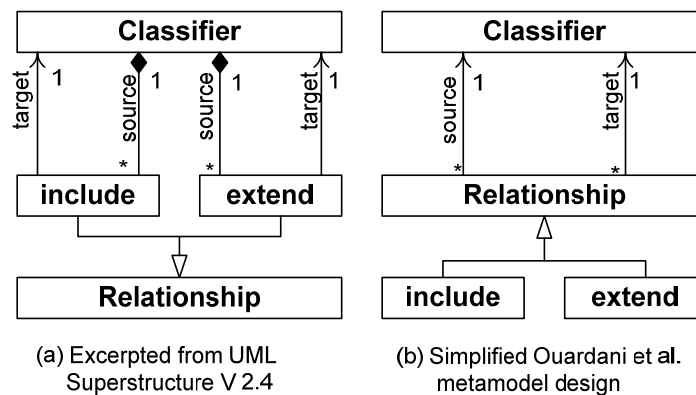


Figure 4.3 Metamodel Design Pattern for Typed Relationships

(adapted from [Ouardani et al., 2006] and [UML Superstructure])

Although the number of participating elements is equal, the two metamodel designs, which are shown in Figure 4.3., are different in two key ways: linked elements and a typed

relationship to the linked metamodel elements. First, when looking at the linked elements, both metamodels are designed to inherit typed relationships (i.e., include and extend) from the common parent relationship. However, in Figure 4.3(a), each typed relationship is linked with a classifier, but in Figure 4.3(b) the classifier and relationship are linked to each other instead of linking the typed relationships.

In addition, the two metamodels use different typed relationships between the classifier and relationship. In Figure 4.3(a), two different relationships, composition and association, are used to link between classifiers and typed relationships (i.e., include and extend). In this metamodel, a composition relationship may be introduced to describe that the source classifier is strongly dependent on the target classifier. But in Figure 4.3(b), the association relationship is used for both source and target links. Typically, association is used to link classifiers weakly, and composition is used to describe a part-whole relationship. However, because the two relationships are relevant to each other and the semantics of the two are defined slightly differently among OO modeling approaches [Albert et al, 2003], it is difficult to say which one is more appropriate. In general, we believe that association is to be preferred to composition if there is no clear part-whole relationship.

#### *4.2.3 Metamodel Pattern for Containment*

Containment represents a part-whole hierarchy and is used to raise the level of abstraction by grouping large and complex model elements with a simple element. The Composite design pattern [Gamma et al., 1995] is commonly used for designing containment needs, but containment also can be designed without using the Composite design pattern. Three different containment metamodel designs are shown in Figure 4.4.

Figure 4.4(a) uses the Composite design pattern to design a metamodel that represents containment. The design leverages the benefits of design patterns, such as facilitating the addition of new kinds of classifiers and recursive composition. Figure 4.4(b) represents containment with a unary composition relationship. Although Figure 4.4(b) represents a viable design option for containment, the design is only applied for containing the same type of classifiers and may violate the Open/Closed Principle (OCP) [Martin, 1996] when a container needs to include new kinds of classifiers.

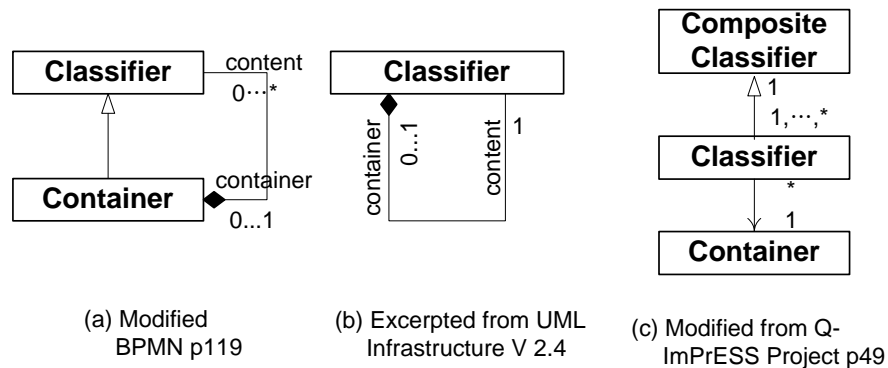


Figure 4.4 Metamodel Design Pattern for Containment

(adapted from [UML Infrastructure], [OMG BPMN], and [QImPRESS])

In Figure 4.4(c), the classifier is inherited from *Composite Classifier*, which represents an abstract classifier that can have sub-classifiers. In addition, a classifier is linked with the *Container* through an association relationship. The intent of the design is to treat the container differently from the contents by introducing *Container*, which may have different characteristics than other classifiers. For example, a deployment diagram (or allocation diagram) may be used to illustrate how physical resources (i.e., storages, processors, and network interfaces) are allocated

onto execution environment vertices. Physical resources are often composed of the same or other physical resource (i.e., a network interface may have a processor and storage to manage network packets) to provide their own functionality, but the instance of the composed physical resources are treated as composite physical resources. However, vertices are composed of physical resources to offer services, but they can be designated as a container rather than composite entities because they are grouped logically. The advantage of the design is that containers can specify classifiers to be contents of the container through the links between classifiers and container.

As described above, each metamodel design has its own intent, but the metamodel design for containment can be unified into Figure 4.4(a), which is an overly general design for containment that has the flexibility of extension for adding new classifiers or other properties. For example, if a container should have different characteristics that are abstract (or logical) from classifier, an attribute may be added to the classifier class to represent that need. Moreover, Figure 4.4(a) can represent nested containers without additional descriptions.

### 4.3 Application of Metamodel Design Patterns

Describing the applicability of a metamodel design pattern is an important factor in characterizing its usefulness and promoting its understanding. The metamodel design patterns introduced in the last section can form the basis for designing the metamodel for a DSML. The first metamodel design pattern can be applied to design a simple *Box-and-Line* style DSMLs. The second and third patterns can be used to describe metamodels that support typed relationships and containment, respectively. In addition to these applications, metamodel design patterns can be used for composing and inferring metamodels.

Metamodel composition [Karagiannis and Höfferer, 2008; Emerson and Sztipanovits, 2006; Karsai et al., 2004; Mapelsden et al., 2002; Lédeczi et al., 2001a] is a technique that creates a new metamodel by reusing all or part of existing metamodels. To make metamodels reusable and/or composable, the metamodels are refined to abstract metamodels that are not designed for specific DSMLs, but capture general structures and behaviors of DSMLs. The proposed metamodel design patterns are elicited from commonality analysis and can represent general characteristics of DSMLs, much like abstract design patterns.

Metamodel inference is the other application area of metamodel design patterns. Metamodel inference has recently been considered as an application of grammar inference [Berwick and Pilator, 1987; Fu and Booth, 1986; Gold, 1967] and used to recover metamodels from existing model instances [Favre, 2004; Javed et al., 2008; Liu et al., 2010]. To infer a metamodel accurately, a metamodel inference engine may require a large set of training data [Kirsopp and Shepperd, 2002]. However, having a large set of existing training data may not be practical in many cases. To complement the lack of training data, metamodel design patterns can be used as a supplementary aid to generate representative instances for metamodel inference through the commonality provided by DSMLs for recurring metamodel design problems.

#### 4.4 Related Work

Mernik et al. [Mernik et al., 2005] defined DSL development phases with five process areas such as decision, analysis, design, implementation, and deployment. They also identified patterns for each of those process areas. For example, either language exploitation pattern or language invention pattern can be used to design a DSL. The language exploitation pattern is a pattern to guide how to tailor existing General-Purpose Languages (GPLs) or DSLs. On the other

hand, the language invention pattern guides DSL development from scratch. Although patterns listed and structured in [Mernik et al., 2005] are well-defined, the patterns are described at a much higher level of abstraction and focus on more of a textual DSL development process.

Elaasar et al. [Elaasar et al., 2006] proposed the Pattern Modeling Framework (PMF) to specify and detect patterns in MOF-compliant modeling frameworks and languages. PMF was designed to conform to OMG's metamodeling architecture. *EPattern* is the top-most layer and represents a meta-metamodel. Metamodel patterns are specified by conforming to an *EPattern* and then metamodel instances can be instantiated from metamodel patterns. *EPattern* is also applied to detect patterns. To detect metamodel patterns, the detection algorithm checks whether each *EPattern* element conforms to a defined type and verifies associated constraints.

Schäfer et al. [Schäfer et al., 2011] proposed a pattern-based approach to develop a metamodel. To create a metamodel, the approach applied patterns in a stepwise manner from requirements elicitation to metamodel definition. For example, language elements are identified at the requirements phase and modeled in a Use Case diagram. The language elements are associated with appropriate language patterns to form an intermediate metamodel. Additional language patterns are applied to the intermediate metamodel until the approach can build a complete metamodel.

## CHAPTER 5

### INTERMEDIATE DESIGN SPACE

This chapter describes the main contribution of this dissertation for applying by-demonstration concepts across the modeling language development lifecycle. The overall process of the MLCBD framework is depicted in Figure 5.1.

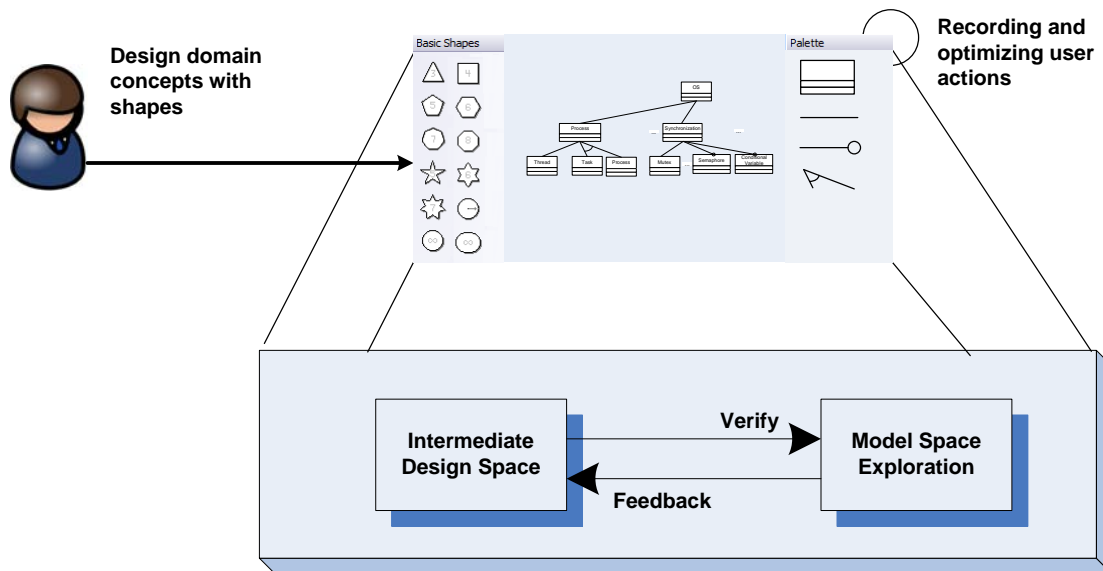


Figure 5.1 Overall Process of MLCBD

The framework consists of two parts: the front-end and back-end modules. The front-end module is designed to address the first and second DSML challenges (i.e., preference of office tools for domain modeling and lack of familiarity with metamodeling environments). The front-

end module provides an environment where domain experts can model concepts in the domain using free-forms or sketch-level forms. The back-end module implements algorithms to resolve the remaining DSML development challenges, such as capturing concrete syntax, inferring abstract syntax, and formalizing static semantics based on the output of the front-end module.

The front-end module consists of two components: the modeling canvas and the recording engine. The main issue of the first DSML development challenge is to recognize new shapes that are drawn informally using free-forms and/or sketch-level shapes, for which domain experts can understand without additional information. To support free-form shapes, Chen et al. [Chen et al., 2008] developed a software design tool named SUMLOW to capture and formalize sketch-level UML constructs. SUMLOW used a gesture recognition technique to recognize sketch-level modeling elements progressively and formalize them while minimizing user interaction. Ossher et al. [Ossher et al., 2010] introduced the concept of flexible modeling tools, which use predefined free-form shapes for modeling pre-requirements. They built a prototype to combine the advantages of office automation tools and traditional modeling tools.

In the MLCBD framework, a modeling canvas supports a combination of these two approaches. The framework provides pre-defined modeling elements as a default, as well as a shape authoring tool. To model a domain with free-form shapes, end-users need to draw shapes that represent their domain and register them as pre-defined shapes using an authoring tool. In addition, the modeling canvas is integrated with the recording engine to capture user actions. The recording engine captures user actions; in particular, it records sequences of user actions that demonstrate domain concepts on the canvas. The recording engine also optimizes the captured actions by pruning unnecessary actions.

The back-end module, as shown at the bottom of Figure 5.1, consists of two major components: the Intermediate Design Space and Model Space Exploration. The core functions of Intermediate Design Space are (1) initially inferring an intermediate metamodel and semantics (i.e., static constraints) using a set of domain model examples, and then (2) completing the metamodel and static semantics based on feedback from Model Space Exploration. After the intermediate metamodel and semantics are generated by the inference engine, the Model Space Exploration component generates a set of model instances and then presents the generated model instances to a domain expert who plays the role of an oracle to offer feedback about which model instance correctly reflects the notions of the domain and which do not. The information provided by the domain expert and a set of model instances are fed back to the Intermediate Design Space to adjust the metamodel and static semantics that are inferred at the previous step. Finally, the metamodel and static semantics are specified by iterating between the Intermediate Design Space and Model Space Exploration components until the output of the approach meets the user confirmed requirements of the target DSML.

This chapter specifically focuses on the Intermediate Design Space. The details of the processes and algorithms required for inferring a metamodel and its static constraints are described in the following sections.

## 5.1 Introduction to Intermediate Design Space

The main purpose of the Intermediate Design Space is to infer the metamodel and semantics based on a set of domain model examples. A set of domain model examples is a small number of models that the domain expert provides, and each domain model example describes the particular notions and intents of a domain using the front-end module. Although the approach

offers a modeling environment, creating a large set of domain model examples is a tedious and error prone process. As a result, domain experts may tend to demonstrate only a few of the positive domain models. Thus, the goal of the Intermediate Design Space is to generate a metamodel and its associated static constraints from a small set of domain model examples, which satisfy the intent of a domain expert.

As shown in Figure 5.2, the Intermediate Design Space consists of two modules: Graph Construction and Inference Engine.

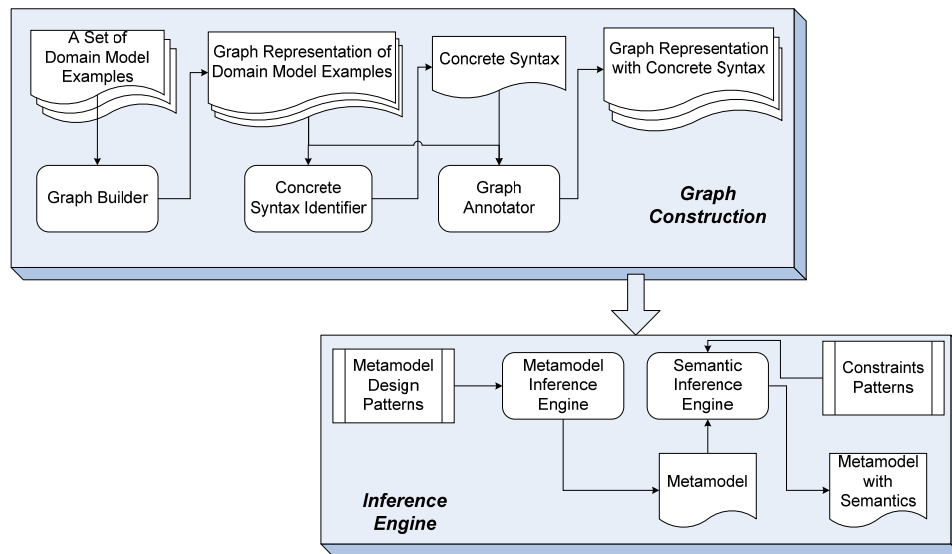


Figure 5.2 Process of Intermediate Design Space

The Graph Construction component transforms a set of domain model examples that are created by domain experts into a set of attributed graphs. This is done prior to inferring the metamodel. During the process of graph construction, concrete syntax elements are identified (or verified) and appropriate attributes are appended to the identified concrete syntax by domain experts. After a set of attributed graphs are created from the set of domain model examples, the

graphs are passed into the inference engine to infer the metamodel and its associated static constraints. When inferring a metamodel, the inference engine refers to metamodel design patterns (please see Chapter 4) to complement the lack of a large number of domain model examples. The following sections describe the details of each step for inferring a metamodel and its static constraints.

## 5.2 Concrete Syntax Identification

Concrete syntax describes the concepts of a domain in terms of a human-readable format (e.g., textual, graphical, or both). In a programming language, concrete syntax serves as the basis of a parser that translates a program into an abstract syntax tree unambiguously using the tokens and keywords of the language. However, in modeling languages, concrete syntax is often described using graphical symbols.

Because the concrete syntax describes how modeling concepts (or abstract syntax) are rendered with graphical and/or textual elements, the Concrete Syntax Identifier finds unique modeling elements, which are modeled as vertices in a graph, by traversing each graph representation. The identified unique modeling elements become the candidate concrete syntax after checking against unique symbols that are captured during domain expert demonstration. By doing this, Concrete Syntax Identifier can verify that all unique symbols are captured without duplication or missing model elements. Then, the candidate concrete syntax is reviewed and annotated by the domain expert. Initially, the candidate concrete syntax is selected with respect to the uniqueness of the modeling elements (e.g., shapes and styles), such that different names and labels can be associated to each modeling element. Thus, the MLCBD framework requires interaction with the domain expert to review the candidate concrete syntax and annotate each

unique modeling element with a generalized name that can represent the notion of each modeling concept precisely and clearly. If each unique modeling element has a label, the type of the label (e.g., number or string) also needs to be specified by the domain expert.

During the review and annotation of candidate concrete syntax, domain experts may be asked to assign additional information for links between example model elements, such as directional information. A link is used to connect two or more classifiers and provides a static semantic relationship between connected classifiers. The direction of a link adds constraints between connected classifiers, such as direction of data or control flow. For example, a dependency relationship in UML is used to represent how a change in a model element may affect the semantics of dependent modeling elements. The arrow of a dependency specifies the direction of a relationship between connected modeling elements.

For example, domain experts who design circuits for signal processing may demonstrate their domain similar to Figure 5.3. As shown in Figure 5.3, the domain is described with several components: *Step*, *Transfer Fcn*, *Scope*, *Adder*, *Gain* and directed arrow. The *Step* is a signal and shows zero for negative and one for positive. The *Transfer Fcn* modifies its input signal and produces a new signal as output. The *Scope* is a sink block used to display a signal much like an oscilloscope. The *Adder* generates a signal by adding two input signals, and *Gain* increases an input signal by a given factor. Finally, the directed arrow shows flow of signals.

In the diagrams of Figure 5.3, the *Step*, *Adder*, and *Gain* are shown once in each process control model. But, the others (e.g., *Transfer Fcn*, and *Scope*) are shown multiple times. For example, the *Scope* is shown two times with different names (e.g., *Scope* and *Scope2*) in Figure 5.3(d). The *Transfer Fcn* is also shown twice in Figure 5.3(e) named with *PI Controller* and *Plant*. Based on the concrete syntax analysis, six unique modeling elements (e.g., *Step*, *Transfer*

*Fcn*, *Scope*, *Adder*, *Gain* and directed arrow) are selected as candidate concrete syntax after eliminating duplicated symbols.

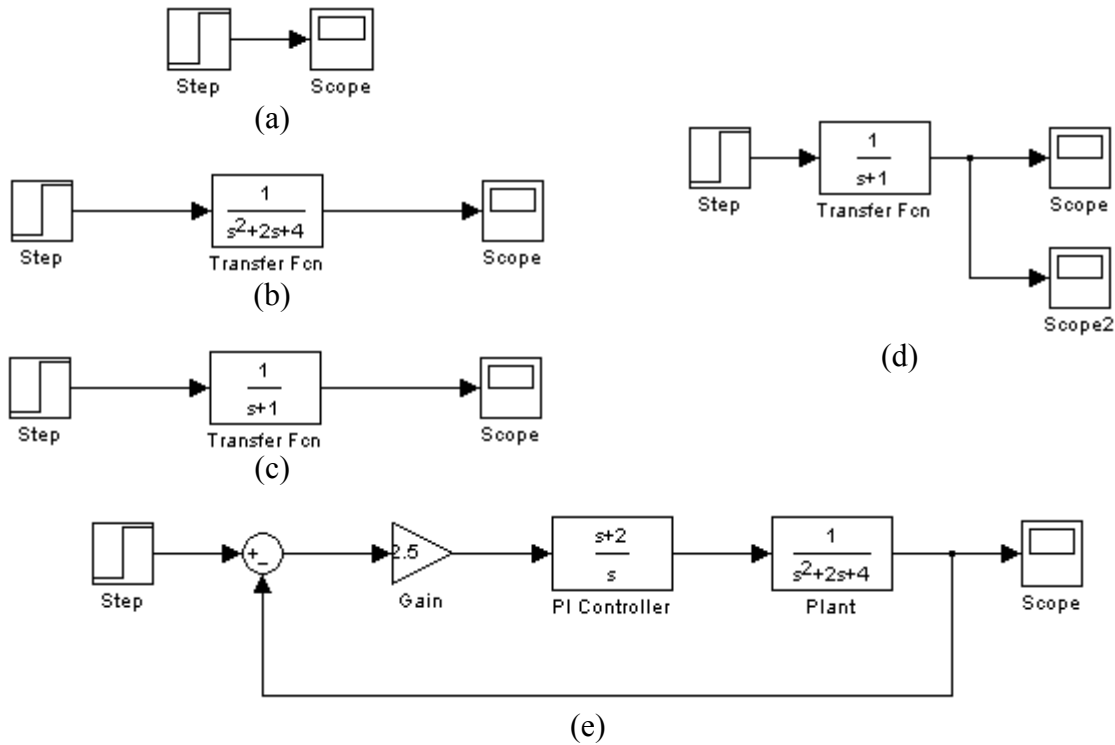






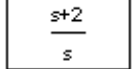
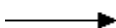
Figure 5.3 Domain Models of Process Control [Simulink]

The list of candidate concrete syntax and its related information is shown in Table 5.1. A candidate concrete syntax could have multiple names and labels because roles and responsibilities of the concrete syntax could be different for each model instance.

Domain experts are asked to review and annotate each symbol with generalized names and labels after the candidate concrete syntax is captured. The possible result of the annotation is shown in Table 5.2. For example, the symbol *Scope* appears in each diagram. In Figure 5.3(d), two *Scope* symbols are used to display the output of *Transfer Fcn* with two different names,

*Scope* and *Scope2*. When capturing a unique modeling element, only one *Scope* symbol is captured but its names (*Scope* and *Scope2*) are associated as attributes.





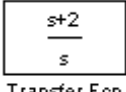
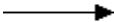
Table 5.1 Candidate Concrete Syntax

	Instance Name	Label
 Step	Step	
 Gain	Gain	2.5
 Scope	Scope, Scope2	
		
 Transfer Fcn	Transfer Fcn, PI Controller, Plant	$\frac{1}{s+1}$ , $\frac{1}{s^2+2s+4}$ , $\frac{1}{s+1}$ , $\frac{s+2}{s}$
		

When domain experts review candidate concrete syntax, they review associated attributes (e.g., name and label) and then annotate each modeling symbol with the name that represents the notion in the domain, as well as the assigned type for the label. In addition, domain experts are asked to specify mandatory attributes such as the type of the symbol and directional information. For example, the *Transfer Fcn* is specified as classifier, and string type attribute *Name* is associated to the *Transfer Fcn*. In addition, two string-type attributes (i.e., Numerator coefficients and Denominator coefficients) are defined to specify coefficients of *Transfer Fcn*'s numerator and denominator, respectively.

In case of an arrow, the link type is defined as an association and directional information is set to *True*, which means the link is directional. Therefore, if a domain expert specifies a symbol as an association, it implies that the symbol will be used as a link between two or more classifiers, and domain experts are asked to assign directional information.

Table 5.2 Annotated Concrete Syntax

	Name	Attribute
	Step	Type = Classifier Name = String StepMax = Integer StepMin = Integer Step = Integer
	Gain	Type = Classifier Name = String Gain = Integer
	Scope	Type = Classifier Name = String
	Adder	Type = Classifier Name = String Operand Left = Double Operand Right = Double
	Transfer Fcn	Type = Classifier Name = String Numerator coefficients = String Denominator coefficients = String
	Link	Type = Association Name = String Directional = True

### 5.3 Graph Construction

Graph transformation is one of the key technologies of MDE because it can assist in the creation, editing, and analysis of a model [Andries et al., 1999; Bézivin, 2005; Gerber et al., 2002]. A graph and its transformation are widely used in the software modeling community and are also a proven approach for representing a high-level programming language formally, which can then be used to generate other types of artifacts by applying production/replacement rules through graph rewriting [Blostein and Schürr, 1999; Saxena and Karsai, 2010]. Transforming platform-independent models into platform-specific models is a common example of graph transformation [Mens and Gorp, 2006]. In our approach, graph constructions are used in two places: the Graph Builder and Graph Annotator phases.

The MLCBD approach uses a graph transformation in order to change the representation of a set of domain model examples into the graph that is used for inferring the metamodel and its associated static semantics, rather than transforming one model to another by applying rules. As shown in Figure 5.2, the process for creating a DSML begins with the transformation of a set of domain model examples into graph representations.

Using the Graph Builder, a set of domain model examples is transformed into a set of graphs. The goal of the Graph Builder is to generate a representation-independent model from a set of domain model examples that are created by a domain expert who demonstrates domain notions using the modeling canvas. Because DSMLs can be developed in various languages, domain models can be described with different representations. For example, to define the syntax of a DSML and maintain model instance data, a DSML may use different file representations

such as XML, text, and binary forms. Although DSMLs may use the same file representation, the schema representing the metamodel for the DSML can be structured differently for each DSML. Thus, the Graph Builder reads a set of domain model examples and transforms them into the corresponding internal graph representation, which are represented as  $G = (V, E, s, t)$ , where  $V$  is a set of vertices,  $E$  is a set of edges, and the functions  $s, t: E \rightarrow V$ , where  $s$  is the source and  $t$  is the target functions.

To generate a graph representation, the Graph Builder maps each modeling element (or classifier) into a vertex and transforms links into edges. In addition, the Graph Builder generates an adjacency matrix that is used to specify the adjacency between modeling elements. The names of the source modeling elements are listed in the left-most column and the names of the destination modeling elements are listed in the first top row. If a cell in the matrix is marked 0, it indicates that there is no relationship between the source and destination modeling elements. If a cell value is larger than 1, it means that the source modeling element can be linked with the destination modeling elements. Figure 5.4 shows the results of the Graph Builder for process control models, which are depicted in Figure 5.3, as well as the corresponding adjacency matrix.

As shown in Figure 5.4, the Graph Builder generates undirected graphs because only limited link information (e.g., name and participated classifiers) is provided when transforming the domain model examples into graphs. Therefore, values of the adjacency matrix that represent dependencies between the source and destination are also marked based on the undirected graph.

For example, when looking at the adjacency matrix for the graphs in Figure 5.4(b) and (c), *Transfer Fcn* in the second row is marked as having a relationship with *Step*, even though *Transfer Fcn* can be the only destination of *Step*.

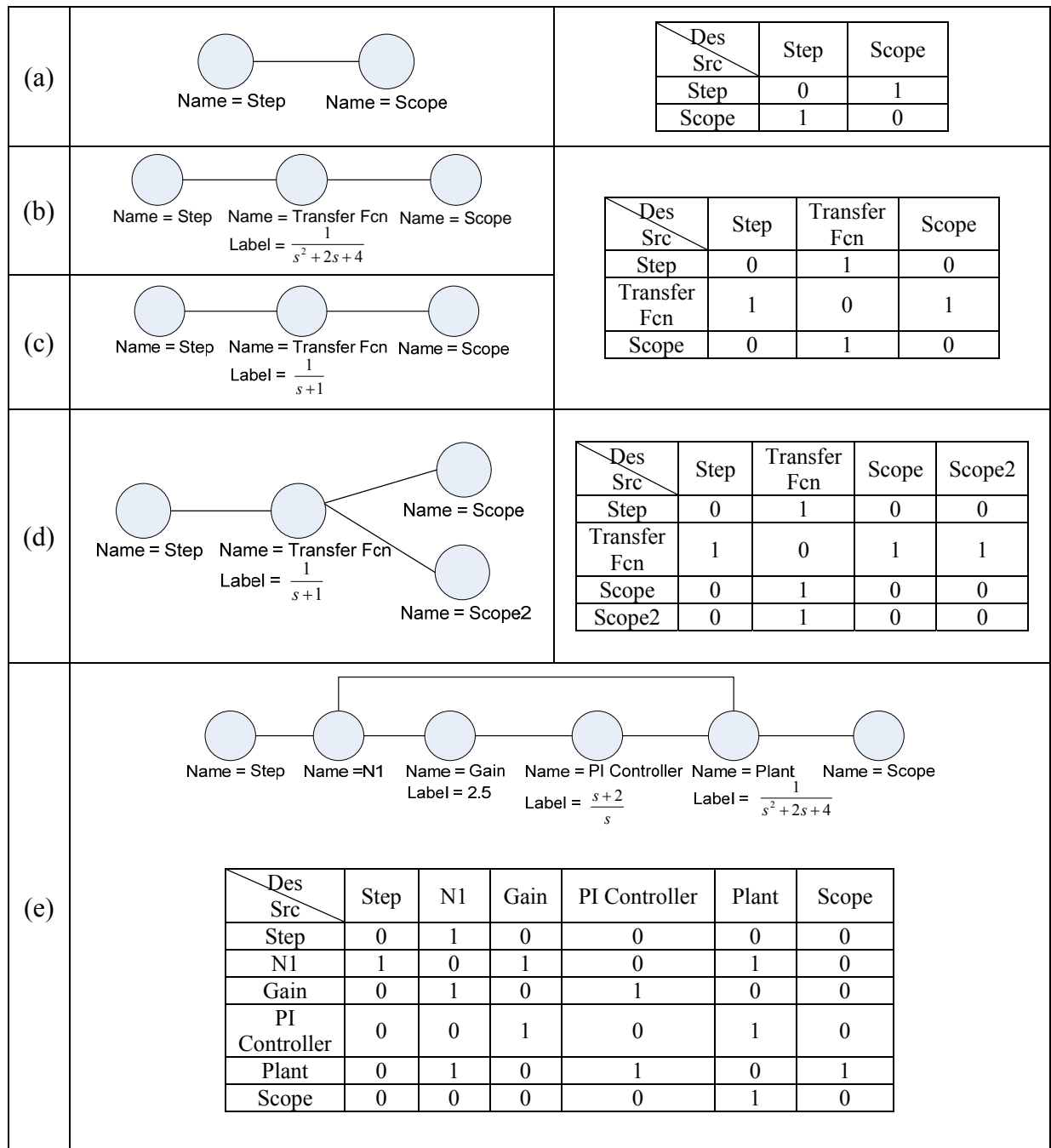


Figure 5.4 Results of Graph Builder

After a set of domain model examples is transformed into the internal graph representation, the Concrete Syntax Identifier defines the concrete syntax of the DSML in a semi-automated manner (i.e., this requires interaction with a domain expert for annotation). After the concrete syntax is given as an example and annotated, the Graph Annotator transforms the outputs of the Graph Builder, a set of undirected graphs, into the attributed graphs by combining the information about the concrete syntax. An attributed graph (AG) [Ehrig et al., 2004] is a graph where some attributes are associated to vertices and edges.

The attributed graph (AG) can be defined as  $AG = (G, D)$  where  $G$  is a normal (un)directed graph,  $G=(V,E,s,t)$ , and  $D$  is the data signature. The data signature defines data and operations. It is defined as  $D = (S_D, O_{PD})$  where  $S_D$  is the type (e.g., Char, String, Nat) and  $O_{PD}$  is constants and/or operations (e.g., in, out, inout, and return).

In our approach, the Graph Annotator converts each vertex,  $V$ , into an attributed vertex that has a pair,  $(name, attrs)$ , where *name* is a generalized name that represents each vertex uniquely in the graph, and *attrs* is a set of attributes attached to the vertex.

In addition, the Graph Annotator converts edges into attributed edges that have a 3-tuple,  $(attrs, src, dst)$ , where *attrs* is a set of attributes attached to the relationship such as type of relationship and directional information, *src* and *dst* are a source and destination vertex linked by the edge, respectively.

To generate a graph representation with concrete syntax, the Graph Annotator takes the following three steps. First, all vertices and edges are renamed with the matched concrete syntax. The name of the vertices and edges are initially assigned with arbitrary instance names, and the other instance names are maintained as an attribute. The instance names are provided by the domain experts when they review and annotate the candidate concrete syntax. After the concrete

syntax is defined, the arbitrary names need to be renamed with their corresponding concrete syntax name. Second, the Graph Annotator checks the link information, which is also added when the domain expert annotates the concrete syntax, and determines whether to change the graph into a directed graph. Initially, the graph representation is constructed using an undirected graph because only limited link information (e.g., name and participated classifiers) is provided when transforming the domain model examples into a set of graph representations. If a domain expert adds additional pieces of information about links (including the direction of a link) at the concrete syntax identification phase, the Graph Annotator transforms the initial graphs into the attributed graphs. Finally, the Graph Annotator completes the graph generation by merging vertices that have the same concrete syntax name.

Figure 5.5 shows the result of the Graph Annotator for the graphs shown in Figure 5.4. As the domain expert annotates each unique modeling element as shown in Table 5.2, the initial graph representations are transformed into attribute graphs that have at least two attributes, Type and Name.

For example, the graph shown in Figure 5.4(a) is changed from an undirected graph to a directed graph because the link is specified as directional, and five attributes (e.g., Type, Name, StepMin, StepMax, and Step) are associated to vertex *Step* to describe characteristics of the step function. Similarly, the two attributes Type and Name are associated to vertex *Scope*. While rewriting the initial graph representation, graphs shown in Figure 5.4(b), (c) and (d) are merged into one graph as presented in (b) because the two graphs shown in Figure 5.4(b) and (c) are identical in terms of the graph representation and adjacency matrix.

Along with rewriting graphs, the Graph Annotator updates the adjacency matrix. As the domain expert annotates the arrow as a directed association, an adjacency that violates the

direction is set to zero. For example, an adjacency from *Transfer Fcn* to *Step* is reverted to zero from one because signals can only flow from *Step* to *Transfer Fcn*.

When rewriting the graph shown in Figure 5.4(d), the resulting graph is also the same as the graph shown in (b), but its adjacency matrix is slightly different to indicate that *Transfer Fcn* can have two *Scope* outputs. To indicate this, the adjacency from *Transfer Fcn* to *Scope* is set to 2 instead of 1. The last graph shown in Figure 5.4(e) is rewritten as (c). *NI*, which is arbitrarily named for *Adder* by the system, is renamed to *Adder* by annotation.

In addition, the *PI Controller* and *Plant* are annotated to *Transfer Fcn*, and the two vertices are merged into one and named *Transfer Fcn* because they are mapped to the same concrete syntax. A recursive link is added representing that *Transfer Fcn* can be linked with some other *Transfer Fcn* as shown in Figure 5.5(c). Along with modifying the graph, the adjacency matrix is reduced because *PI Controller* and *Plant* are merged to *Transfer Fcn*. In addition, the adjacency value from *Transfer Fcn* to other modeling elements is updated considering the link direction.

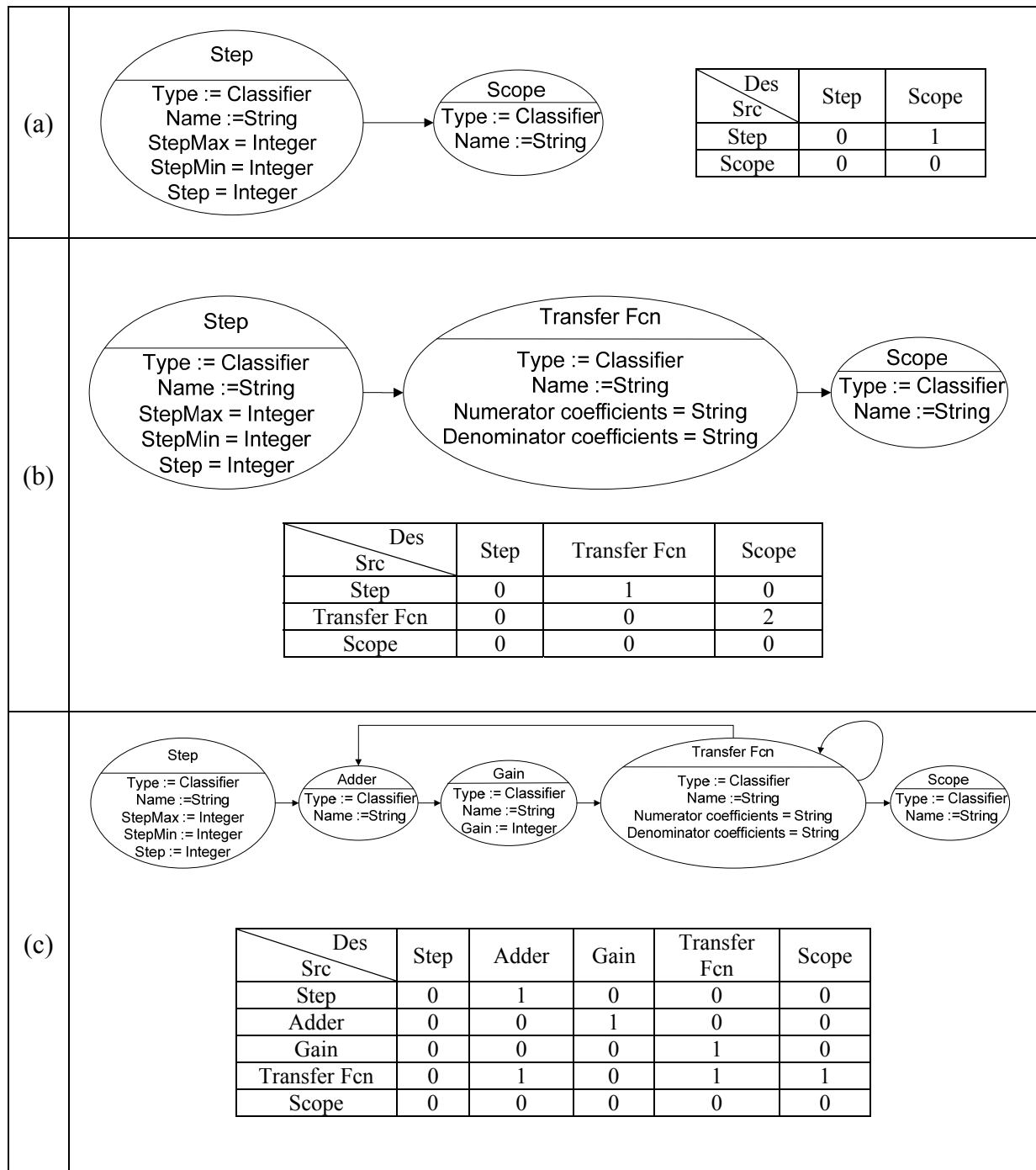


Figure 5.5 Results of Graph Annotator

## 5.4 Metamodel Inference

After a set of domain examples are transformed into a set of graph representations with concrete syntax, the Metamodel Inference Engine infers metamodel and static constraints based on the graph representation. To infer a metamodel, the Metamodel Inference Engine loads each graph representation and then compares the loaded graph representations against a set of metamodel design patterns in order to determine if the graph representation matches a known metamodel pattern. For example, the graph representation of the second process control model instance, shown in Figure 5.3(b), consists of three concrete syntax elements: classifiers (*Step*, *Transfer Fcn* and *Scope*) and a relationship (Link). Classifiers and relationships are mapped onto vertices and edges as shown in Figure 5.5(b). Comparing this graph representation with the set of metamodel design patterns, the Metamodel Inference Engine will find the base metamodel pattern that best matches the graph representation. As shown in Figure 5.6(a), the base metamodel design pattern is designed for DSMLs that consist of simple classifiers and relationships, and can be transformed into three different graph representations as shown in Figure 5.6(b). The top part of Figure 5.6(b) represents two different classifiers that are linked with a relationship, and the middle part describes that two or more of the same classifiers are linked with a circular relationship. The bottom part depicts that the two same classifiers are linked to different classifiers.

While the Metamodel Inference Engine determines the design patterns that are matched to each graph representation of domain model examples, the engine combines each graph

representation of the domain model examples and generates a single graph representation that represents the entire set of domain model examples.

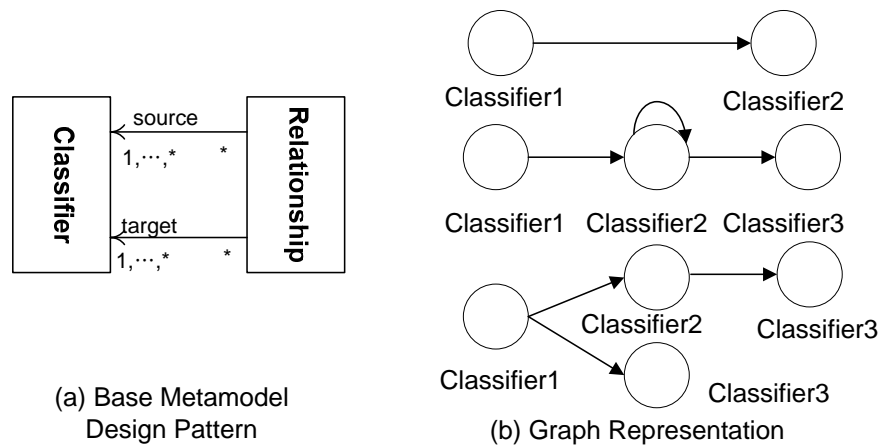
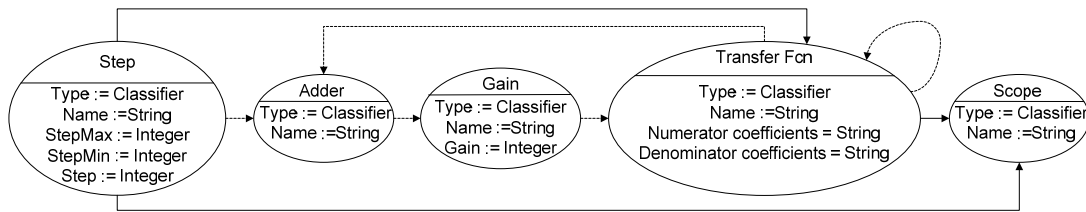


Figure 5.6 Base Metamodel Design Pattern and its Graph Representation

When combining each domain model example, the type of each edge is classified based on the appearance of a vertex. For instance, vertex *Step* and *Scope* are classified as mandatory because they are used in every graph representation. However, vertex *Adder* and *Gain* are optional because they are not present in some domain model examples (e.g., Figure 5.4(a), (b), (c), and (d)).

The result of the graph combination is shown in Figure 5.7. The merged graph is similar to the graph in Figure 5.5(c), but a link between vertex *Step* and vertex *Scope* is added in order to cover the process control model that has only *Step* and *Scope*, such as Figure 5.3(a). In addition, a dotted line is used between other vertices (e.g., *Adder*, *Gain*, and *Transfer Fcn*) to represent that *State* can be used optionally in process control models.

After all graph representations are combined into a single graph, the single graph representation also serves as input to the Metamodel Inference Engine to find the matched metamodel design patterns. Besides combining the graph representations, the Metamodel Inference Engine determines the cardinality between vertices. The identified cardinality is maintained using a cardinality matrix, as shown in the bottom of Figure 5.7(c).



(a) Combined Graph Representation

Des Src	Step	Adder	Gain	Transfer Fcn	Scope
Step	0	1	0	0	1
Adder	0	0	1	0	0
Gain	0	0	0	1	0
Transfer Fcn	0	1	0	1	2
Scope	0	0	0	0	0

(b) Adjacency Matrix

Des Src	Step	Adder	Gain	Transfer Fcn	Scope
Step	0	0,1	0	0,1	1,2
Adder	1,1	0	1,1	0	0
Gain	0	1,1	0	1,1	0
Transfer Fcn	0,1	0,1	0,1	0,1	1,2
Scope	1,1	0	0	1,1	0

(c) Cardinality Matrix

Figure 5.7 Combined Graph Representation: Adjacency Matrix and Cardinality Matrix

Similar to the adjacency matrix, the left-most column represents the source vertices and the top row lists the destination vertices. Values in each cell describe the minimum and maximum appearance of the concrete syntax in the graph. The matrix is initially set to 0 for the minimum and maximum value of each cell. When the first model instance, Figure 5.3(a), is processed, both the minimum and maximum cardinality between *Step* and *Scope* is set to 1. When the second model instance is introduced, the cardinality between *Step/Scope* and *Transfer Fcn* is updated to (0,1). At this moment, the minimum cardinality between *Step/Scope* and *Transfer Fcn* remains 0 because *Transfer Fcn* does not appear in the first model instance. The matrix is finally completed by filling the cardinality between other vertices when the last model instance is processed.

To infer a metamodel, a combined graph representation is tested over a set of graphs, which is instantiated from metamodel design patterns in order to check (sub) graph isomorphism. Thus, the algorithm needs to solve the one-to-many (sub)graph isomorphism problem: one combined graph representation and many graphs instantiated from the metamodel design patterns. To solve the issue, the algorithm extended and modified the algorithm proposed by Messmer and Bunke [Messmer and Bunke, 1999], whose algorithm tests one-to-many (sub)graph isomorphism in quadratic time. In order to test the (sub)graph isomorphism, the algorithm transforms a set of reference graphs into a decision tree and then tests the input graph over the decision tree.

The Metamodel Inference algorithm consists of three parts: *Instantiate\_Tree()*, *CreateDecisionTree()*, and *main* (see Figure 5.8, Figure 5.9, and Figure 5.10, respectively). *Instantiate\_Tree()* generates a set of graphs from metamodel design patterns with size of  $m$ , where  $m$  is the size of the vertex of the combined graph representation.

Let  $G$  be the combined graph representation, and  
 $M$  be the adjacency matrix of  $G$  and represented as  $M = (a_1, a_2, \dots, a_m)$   
where  $m = |V|$ , size of vertex in the graph  $G$ ,  
and  $a_i$  is the row-column element that is attached to vertex  $i$ .

Let  $GT_D = (GT_{D1}, GT_{D2}, \dots, GT_{Dk})$  be a set of graph template  
of metamodel design patterns, and  $k$  is the number of metamodel design patterns

Measure size of vertex,  $m = |V|$ , of the combined graph

Instantiate\_Tree( $GT_D, m$ )

Create the root vertex Root of the decision tree,  $G_M$  if it does not yet exist.

For  $i=1$  to  $k$   
 $G_M +=$  CreateDecisionTree( Vertex Root,  $GT_{Di}$ )  
Next for

Let  $N = \text{Root of } G_M$   
For  $i=1$  to  $m$   
Look up row-column elements that are attached to vertex  $N$ ,  
and find an entry  $a_N$  such that  $a_N = a_k$   
if no matched element is found,  
the graph  $G$  is not isomorphic with  $G_M$ , and exit with failure  
else  $N = N_S$  and mark vertex  $N$  matched  
Next for

Traverse  $G_M$  to find the marked vertices and then transform the vertex with corresponding metamodel design pattern.

Figure 5.8 Metamodel Inference Algorithm: Main Algorithm

```

Instantiate_Tree (GTD, m )
  Instantiate GD = (GD1, GD2, ..., GDk) with size of m
    from graph templates of metamodel design patterns, GTD,
    where GDi = (GDi1, GDi2, ..., GDil) is l different types of graph instances of
      ith metamodel design pattern with vertex size m

```

Figure 5.9 Metamodel Inference Algorithm: Instantiate Tree

```

CreateDecisionTree(Vertex Root, GDi)
  Generate adjacency matrix of GDi,
    MDi = (MDi1, MDi2, ..., MDil), where dimension of MDi is m,
    and MDik = (aik1, aik2, ..., aikm), where aikl is row-column elements
      that is attached to vertex l

  Let N = Root
  For l = 1 to m
    If there is successor vertex NS of N for which aikl = aNS, and N = NS
      Do next
    Else
      Generate a new vertex NS for aikl and make NS a direct successor of N.
      Set N = NS
    End if
  Next for

```

Figure 5.10 Metamodel Inference Algorithm: Merge Tree

As shown in Figure 5.6, each metamodel design pattern is used as a template to instantiate a set of graphs. *CreateDecisionTree()* merges graphs instantiated from *Instantiate\_Tree()* in order to make a decision tree. When graphs instantiated from metamodel design patterns are passed into *CreateDecisionTree()*, a adjacency matrix is constructed and its row-column representation for every graph is passed into *CreateDecisionTree()*.

Figure 5.11 shows how the metamodel inference algorithm creates a decision tree from a set of graphs, which are created from metamodel design patterns.

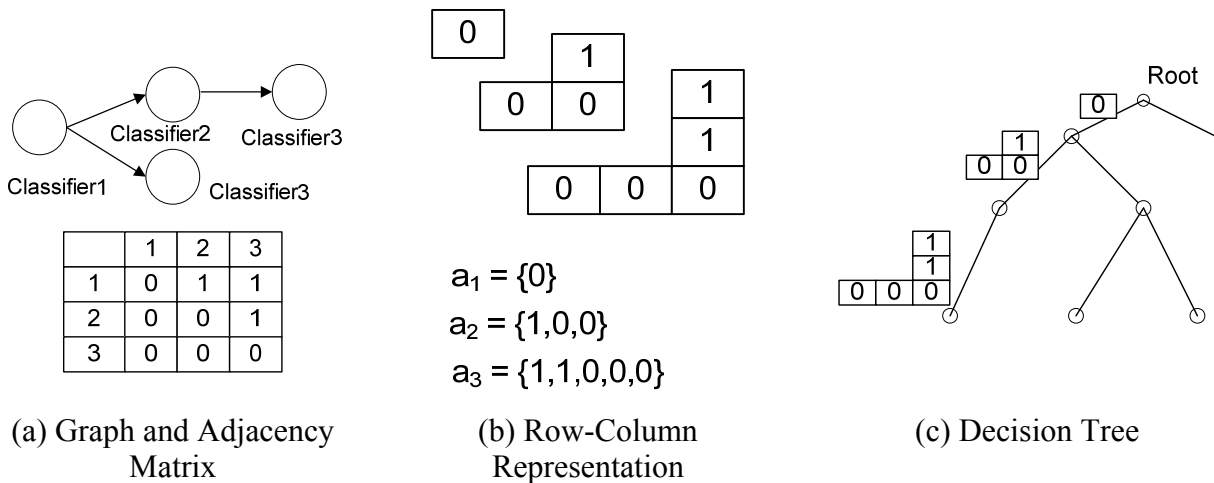


Figure 5.11 An Example of Row-Column Representation and Decision Tree

When a graph and its adjacency matrix are passed into the metamodel inference engine, the metamodel inference engine calculates the row-column representation of the graph from the adjacency matrix. The result of row-column representation is shown in Figure 5.11(b), which illustrates that the row-column representation has different dimensions depending on the vertex.

After the adjacency matrix and row-column elements are identified, *CreateDecisionTree()* builds a new decision tree if it does not yet exist, or merges graphs into the existing decision tree.

When a graph shown in Figure 5.11(a) is passed into *CreateDecisionTree()*, the routine first creates three row-column representations (i.e.,  $a1=\{0\}$ ,  $a2=\{1,0,0\}$ , and  $a3=\{1,1,0,0,0\}$ ) and then creates a new decision tree from the small size of the row-column representation. Therefore, a branch is created for  $a1=\{0\}$  from Root vertex and then another branch is created from  $a1$  for  $a2=\{1,0,0\}$ . Figure 5.11(c) shows the decision tree created using three row-column representations in Figure 5.11(b). While creating the tree, each branch, which connects from one vertex to another, is labeled with a row-column element. The row-column element associated to the branch is used to determine whether the algorithm needs to traverse vertices further down, or whether the input graph is isomorphic.

When a tree is created from the first graph, the Metamodel Inference Engine checks whether the remaining graphs exist, and then expands the tree by processing the graphs if they exist. As shown in Figure 5.6, three graphs are instantiated from the Base Metamodel Design pattern so that the Metamodel Inference Engine has to process two remaining graphs. Figure 5.12(c) shows the decision tree when the Metamodel Inference Engine processes the middle part of the graph in Figure 5.6(b).

Similar to the first graph, the adjacency matrix of the second graph can also be described with a three row-column representation:  $a1=\{0\}$ ,  $a2=\{1,1,0\}$ , and  $a3=\{0,1,0,0,0\}$ . The first row-column representation of Figure 5.12(a),  $a1=\{0\}$ , is the same as the first row-column representation of Figure 5.11(a) - the row-column representation does not make any change to the decision tree. However, the second and third row-column representations create new branches, as shown in Figure 5.12(c).

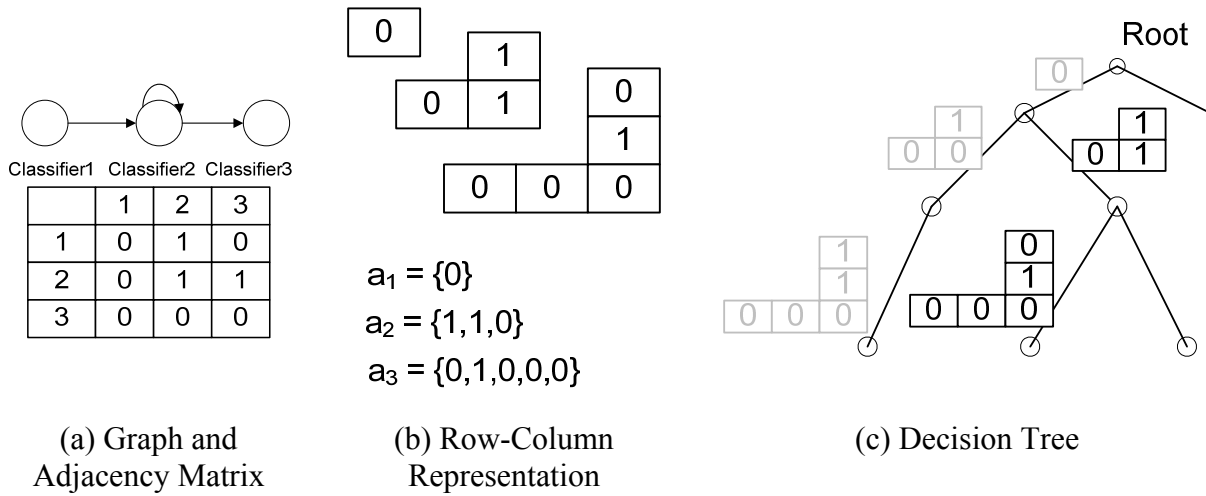


Figure 5.12 Merged Decision Tree

After the first row-column,  $a_1 = \{0\}$ , is processed, *CreateDecisionTree()* creates another branch from the end of  $a_1 = \{0\}$ , for the second row-column representation,  $a_2 = \{1, 1, 0\}$ ; and then, another branch is created from  $a_2 = \{1, 1, 0\}$  for  $a_3 = \{0, 1, 0, 0, 0\}$ . The complete decision tree for the graph representations of the Base Metamodel Design patterns shown in Figure 5.6 is the same as Figure 5.12(c) because the row-column representations of the first instance of the metamodel design patterns (i.e.,  $a_1 = \{0\}$ ,  $a_2 = \{1, 0, 0\}$ ) do not affect the decision tree.

After the decision tree is created, the Metamodel Inference Engine traverses the decision tree with the row-column representations of the combined graph, which is shown in Figure 5.7, in order to find the matched vertex. The output of the Metamodel Inference Engine is shown in Figure 5.13.

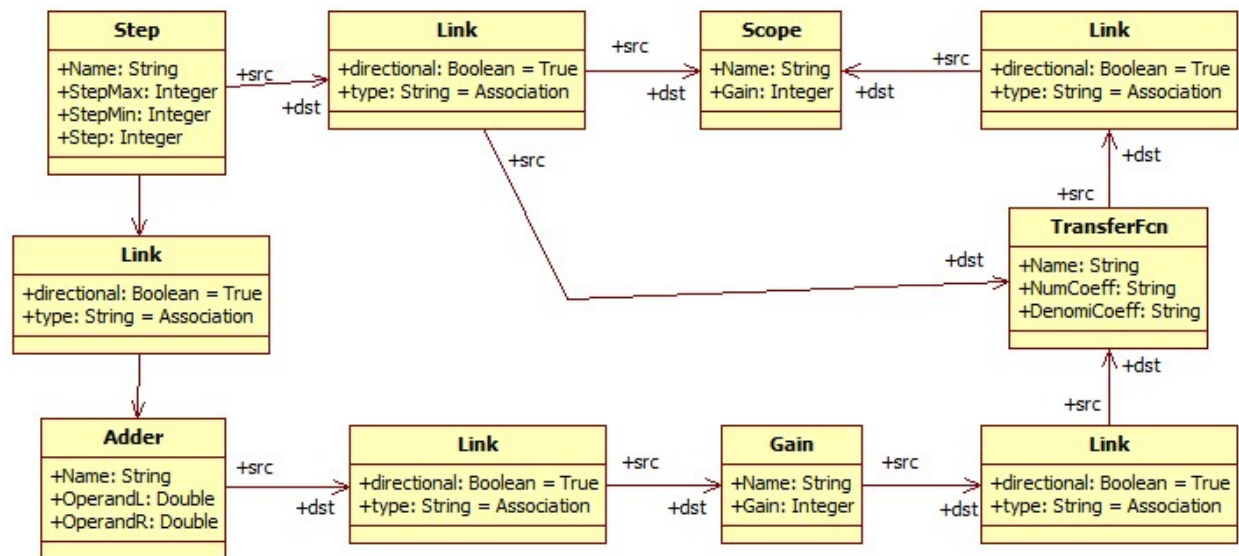


Figure 5.13 Inferred Metamodel

## CHAPTER 6

### MODEL SPACE EXPLORATION

In Chapter 5, we introduced how Intermediate Design Space infers metamodel and static constraints from a set of domain model examples demonstrated by domain experts. However, the inferred metamodel and semantics may not completely reflect the notion of a domain because they are often inferred with a small set of domain model examples. As mentioned earlier, it is challenging to expect that domain experts demonstrate a large set of domain model examples for training the inference engine, even if there is tool support, because modeling every aspect of the domain requires much time and effort. In addition, the example staging process can also be mundane to domain users, especially, when it is performed just for the demonstration purposes.

Thus, Model Space Exploration is introduced as means of verification of the output of the Intermediate Design Space. If the domain experts have modeling language development expertise (which would be uncommon), they can verify the correctness of the inferred metamodel and static constraints by reviewing the output of the Intermediate Design Space. However, as the approach assumes that domain experts have only knowledge of their domain, the MLCBD framework should provide a way to reason about the correctness of the inferred metamodel and static constraints.

The notion of Model Space Exploration is borrowed from the area of Design Space Exploration (DSE), which was originally introduced for hardware/software co-design. The goal of DSE is to automate design (or find design alternatives) from a computational environment in

which hardware and software designers participate in design or modeling guided by a computer system to an extent both computationally possible and desired by the designer [Neema et al., 2003; Saxena and Karsai, 2010; Woodbury et al., 2000].

In our approach, the notion of DSE is extended to verification of the modeling language. Model Space Exploration has been investigated as a way to verify metamodel and semantics inference by exploring model spaces where a set of model instances are generated from the output of the Intermediate Design Space. In addition, Model Space Exploration is applied to identify missing notions, which need to be demonstrated by domain experts for inferring more accurate metamodel and static constraints.

The following sections will describe the process for usage and the algorithms of Model Space Exploration to verify the correctness and completeness of an inferred metamodel and its static constraints.

## 6.1 Process of Model Space Exploration

The overall process of Model Space Exploration in the context of MLCBD is shown in Figure 6.1. Model Space Exploration consists of two parts: Model Space Modeling and Model Space Clustering. The role of Model Space Modeling is generating model instances from the outputs of the Intermediate Design Space; these result in a metamodel, adjacency matrix, and cardinality matrix. The Model Generator creates all computationally possible model instances that represent the notion of the demonstrated domain, both positively and negatively, by combining the output of the Intermediate Design Space with the constraint patterns.

The generated model instances are provided to domain experts in order to obtain feedback about which model instances correctly describe the notions of the domain and which do

not. Feedback is then associated to each model instance in the form of a label, and then model instances are classified and grouped by the Cluster. The Cluster classifies each model instance based on the label and the number of participating modeling elements. After the model instances are clustered, they are transformed into a graph representation and then passed back to the Intermediate Design Space to evolve the metamodel and semantics. A complete DSML is created by iterating over the phases of the Intermediate Design Space and Model Space Exploration.

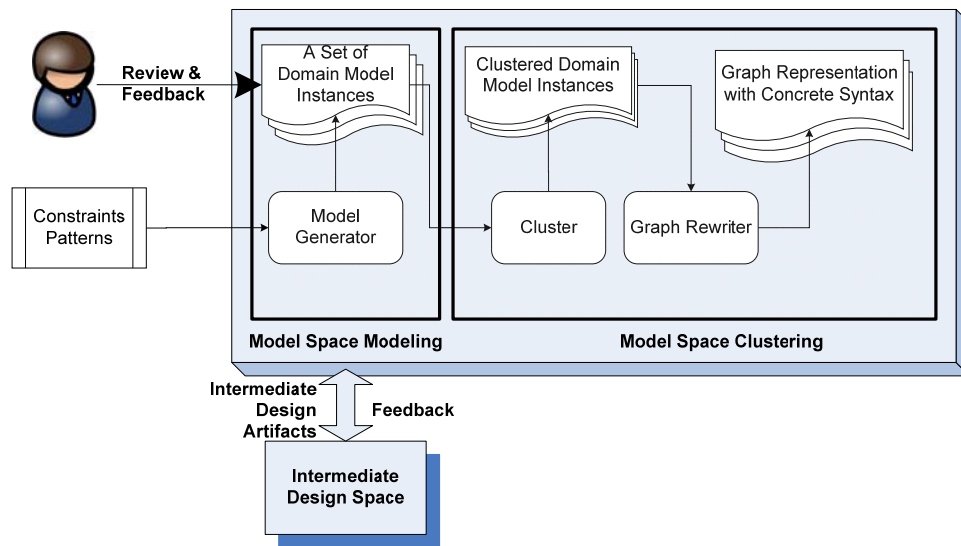


Figure 6.1 Process of Model Space Exploration

In the following two sections, we will describe the details of the algorithm used in Model Space Exploration and investigate issues related to model instantiation.

## 6.2 Model Instantiation and Clustering

Model Space Exploration originated from the notion of design space exploration, with the goal of finding improvements of the inferred metamodel and static constraints by identifying

conflicts, which are recognized by domain experts who may respond differently to the expected classification of a generated model instance. According to Gries [Gries, 2004], “*an exploration algorithm working on the problem space systematically chooses a system configuration, evaluates it, and decides whether this configuration is feasible or not.*” Thus, having a well-designed exploration algorithm is a key factor to determine the success of Model Space Exploration. Several exploration algorithms have been proposed to satisfy certain properties or criteria, which provide the required or alternative characteristics to find a design solution. Simulation-based [Halambi et al., 1999; Eker et al., 2003], equation-based [Blickle et al, 1998; Erbas et al., 2003], and model-based [Bondé et al., 2005; Kangas et al., 2006] are example contests for design space algorithms.

In our approach, we use the graph representation as the basis for developing the exploration algorithm. Graph-based design space exploration is a commonly used approach in design space exploration [Blickle et al, 1998; Oliveira et al., 2010]. In the MLCBD framework, graph representation plays a vital role throughout the Intermediate Design Space. A set of domain model examples is transformed into a set of attributed graphs based on the annotation made by domain experts, and then the Metamodel Inference Engine uses the set of attributed graphs in order to infer the metamodel and static constraints. Thus, we adopted the notion of graph-based Model Space Exploration to reuse graph resources, which are created and maintained in the Intermediate Design Space.

The design of Model Space Exploration follows the Y-chart philosophy [Kienhuis et al., 1997] that is commonly applied in design space exploration (see Figure 6.2). As shown in Figure 6.2, a set of model instances, an inferred metamodel, and constraints are the artifacts of Model Space Exploration. A set of model instances can be considered a set of alternative domain model

choices, which may or may not describe the notions of a domain. A metamodel and its constraints can be used to check conformance. The mapping stage checks whether a set of model instances conform to the inferred metamodel and its constraints. Each model instance is then labeled according to the checked result. Finally, model instances are reviewed by domain experts during the analysis of the exploration.

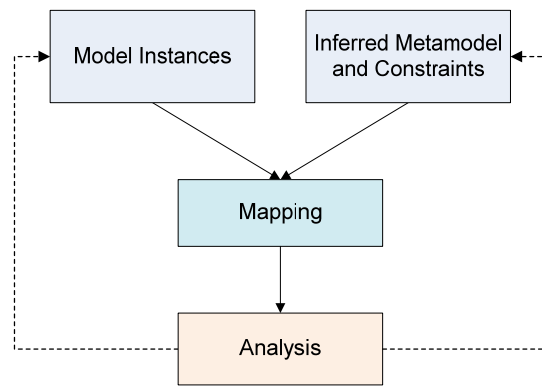


Figure 6.2 The Y-Chart for Model Space Exploration

To create model instances, a graph path finding algorithm is applied to the attributed graphs. The graph path finding algorithm is useful to traverse reachable states exhaustively and can check the syntax and constraints associated to the path. To find all paths between an arbitrary number of two vertices exhaustively, the algorithm was designed by combining breadth-first search and Dijkstra's shortest path algorithm. The Model Space Exploration algorithm is presented in Figure 6.3, Figure 6.4, and Figure 6.5.

The main algorithm shown in Figure 6.3 controls the whole process of Model Space Exploration. Prior to exploring the model space, the algorithm retrieves graph information from the Intermediate Model Space. The algorithm then checks the multiplicity from the cardinality

matrix. The multiplicity information is passed into the path search algorithm, which is shown in Figure 6.5, to determine whether the algorithm can revisit a vertex during path finding. After the algorithm gets information about the graph from the Intermediate Design Space, it explores the positive model and then switches focus to a negative model. For exploration of negative models in Model Space Exploration, the algorithm transposes the adjacency matrix of the initial graph. By transposing the adjacency matrix, dependencies between vertices are reversed.

After all of the necessary information is prepared, a set of model instances is presented and annotated by domain experts. The procedure of annotation is described in Figure 6.4. A queue is created and initialized to store all distinctive paths between arbitrary vertices  $s$  and  $t$  (i.e.,  $s$ - $t$  paths). Then, the algorithm picks up an arbitrary two vertices from the graph and checks whether vertices in the graph are connected with weighted edges. To check whether the graph is weighted or not, function *checkEdgeWeight()* traverses the graph until it finds edges having different weight rather than arbitrarily selected vertices  $s$  and  $t$ . If the graph has non-weighted edges, the algorithm searches all paths  $s$ - $t$  paths using modified breadth-first search (i.e., *searchBFS()*). Otherwise, paths are retrieved using a modified Dijkstra search algorithm (i.e., *searchDijkstra()*). The modified BFS algorithm is shown in Figure 6.4.

When a path between  $s$  and  $t$  is found, the path link is added to *Paths*. The path link is used to instantiate a model by referring back to the inferred metamodel's concrete syntax and static semantics.

```

exploreModelSpace {
  // Initialization
   $G$  = getGraph(); // Get graph information from the metamodel inference
   $bMulti$  = checkMultiplicity(  $G$  );

  // Model Space Exploration
  exploreModelSpace(  $G$ ,  $bMulti$  ); // Explore positive model space
   $G^R$  = reverseGraph(  $G$  ); // Reverse graph in order
  exploreModelSpace(  $G^R$ ,  $bMulti$  ); // Explore negative
}

```

Figure 6.3 Main Algorithm of Model Space Exploration

```

exploreModelSpace( graph  $G$ , Boolean  $bMulti$  ) {
   $Paths$  = {}; // Create a queue for managing paths between arbitrary two vertices
  for each ( $u$  in  $V$  of  $G$ ) {
    for each ( $v$  in  $V$  of  $G$ ) {
      do {
        checkEdgeWeight() ? searchBFS(  $G, src, dst, Paths, bMulti$  ) :
                               searchDijkstra(  $G, src, dst, Paths, bMulti$  );
        instantiateModels(  $Paths$  );
        showModel(  $Paths$  );
      } while(  $Paths$  is not null );
    }
  }
}

```

Figure 6.4 Algorithm for Model Space Exploration

```

searchBFS ( graph G, vertex src, vertex dst, queue Paths, boolean bMulti) {
  for each (u in V) { // initialization
    mark[u] = Unvisited
    d[u] = infinity // Initialize distance vector from src to u
    pred[u] = null // Initialize the predecessor vector, which determines the shortest path
  }
  mark[src] = Visited // Set source vertex src as discovered
  d[src] = 0
  Q = {src} // put src in the queue
  while( Q != Empty ) {
    u = dequeue from Q // get next vertex from queue
    for each (v in Adj[u]) { // get adjacent vertex from adjacent matrix
      if ( v == dst ) { // v is the destination vertex
        if( G[u][v]!=0 ) {
          pred[v] = u // ...and its parent
          mark[v] = Visited //
          if pred not in Paths then add pred to Paths // Add path from src to dst to Paths
        }
        break;
      }
    }
    else if(bMulti==true && (v==src || mark[v]==Visited) ) // Continue if a vertex is
      continue; // source or visited.
    else {
      if ( mark[v] == Unvisited && G[s][v] ) { // Check v is visited vertex
        mark[v] = Visited // ...mark v visited
        d[v] = d[u]+1 // ...set its distance from u
        pred[v] = u // ...and its parent
        append v to the tail of Q // ...put it in the queue
      }
    }
  }
}

```

Figure 6.5 Graph Search: Breadth First Search with Backtracking

For example, the combined graph representation shown in Figure 5.7(a) is simplified in Figure 6.6 for computational convenience. If the search algorithm searches paths between Step and Scope, which is numbered 0 and 4, respectively, *Paths* will contains path information as a sequence of vertices like  $\{ \{0,4\}, \{0,3,4\}, \{0,1,2,3,4\} \}$ .  $\{0,4\}$  represents a direct link from *Step* to *Scope*, and  $\{0,1,2,3,4\}$  represents a series of sequential connection from *Step*, *Adder*, *Gain*, *Transfer Fcn* to *Scope*.

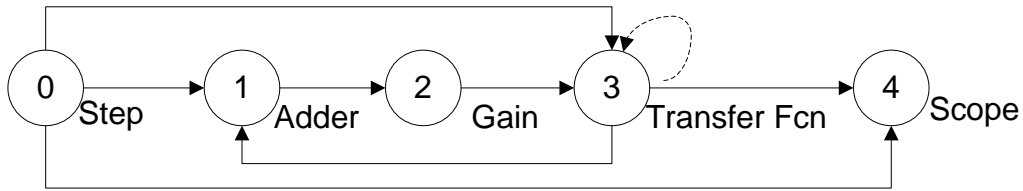
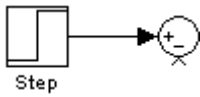
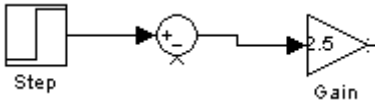
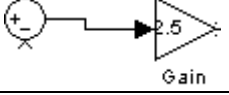
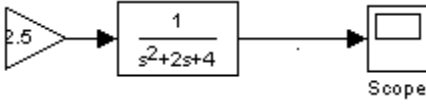
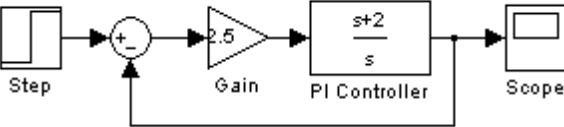
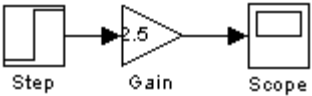


Figure 6.6 Numbered Graph Representation

After all possible paths between an arbitrary two vertices are found and stored in *Paths*, a set of model instances are created based on the path sequences in *Paths*. While a set of model instances is being created by function *instantiateModels()*, each model instance is annotated to indicate whether it conforms to the inferred metamodel and static constraints. As mentioned in Chapter 1, DSMLs should be designed to satisfy the layered modeling architecture, which implies that a model in the top-most layer can conform to itself, and the rest of the models in each layer should be designed to conform to models defined in the layer above. Thus, each model instance created by a path link should be checked for conformance to the inferred metamodel and semantics. The set of model instances are provided and reviewed by domain experts to determine whether the models correctly describe the notion of the domain.

Table 6.1 shows some examples of model instances created by the algorithm described above.

Table 6.1 Samples of Model Instances

	Model Instance	Created From	Expected	Answered
(a)		Positive Graph	True	False
(b)		Positive Graph	True	False
(c)		Positive Graph	True	False
(d)		Positive Graph	True	False
(e)		Positive Graph	True	True
(f)		Negative Graph	False	True

The first five models are created based on a positive graph. The positive graph is created by combining a set of graph representation from domain model examples that were used to infer a metamodel. The combined graph of a process control DSML is shown in Table 6.1(a). The bottom model is generated from the negative graph, which is created by transposing or reversing the adjacency matrix of the combined graph representation.

Thus, a set of model instances generated from the positive graph will conform to the inferred metamodel and semantics. The expected responses from the domain experts regarding the presented model instances are initially set to *True* during the conformance check in function *instantiateModels()* (i.e., all model instances describe the notions of domain correctly). However, models instantiated from the negative graph will be assigned to *False*. In the example scenario shown in Table 6.1, a domain expert answered *False* to all models except the model in Table 6.1(e); the domain expert also indicated *True* to the model instantiated from the negative graph. Theoretically, the first four model instances should be labeled *True* because they are generated from the positive graph and conform to their metamodel and static constraints. But, those models are labeled *False*, which is different from expectation. This may imply that models conform to the inferred metamodel and static constraints, but do not conform to the notions of the process control domain.

The semantics of *Adder* generates a new signal for the next process element after adding two input signals, and *Gain* increases the power or amplitude of a signal from the input to the output. Thus, *Adder* and *Gain* should be connected with appropriate input and output to satisfy their semantics. However, *Adders* shown in Table 6.1(a), (b), and (c) have only one input or no input signals. In addition, *Adders* and *Gains* shown in Table 6.1(a), (b), and (c) are not connected to appropriate output. A process model shown in Table 6.1(d) is *False* because *Scope* is not able to show any signals because no input signals are provided to *Gain* and *Transfer Fcn*. Although these models are correct syntactically and semantically in terms of the inferred modeling language, they are incorrect in terms of notions of the domain. Thus, they are discarded while informing the Model Space Modeling module not to generate them in future Model Space Exploration sessions.

On the contrary, a model shown in Table 6.1(f) is labeled true even though the model is generated from the negative graph. In addition, the model does not conform to the inferred metamodel and static constraints. However, the model satisfies semantics of the process control domain. *Gain* has an input signal to amplify and is connected to its output (*Scope*) which displays the output signal of *Gain*. Thus, the model clearly describes the notions of the process control such that a signal is amplified and then displayed. However, the model was not demonstrated originally by the domain expert because such a wide variety of possible domain models makes it challenging to demonstrate every detail of the domain notions.

After a set of model instances are explored by a domain expert, the models are clustered based on the expectation and domain expert responses in order to improve the previously inferred metamodel and semantics. To cluster the set of model instances, the cluster algorithm finds models that generated different feedback from the expected expert response. For example, model instances in Table 6.1(a), (b), (c), (d), and (f) are initially selected for improving the previously inferred metamodel and static constraints. Based on the selection, each model is examined whether it can be used for improvement or ignored. Model instances in Table 6.1(a), (b), (c), and (d) are classified to be ignored because those models are syntactically correct, but semantically incorrect. However, the model instance shown in Table 6.1(f) is labeled to be updated for improving the previously inferred metamodel and static constraints.

### 6.3 Consideration of the Number of Models for Model Space Exploration

Graph traversal plays a key role in instantiating models for the Model Space Exploration. Each vertex represents a modeling element, and edges between arbitrary two vertices represent the relationship between the two vertices (or the relationships between two modeling elements).

Thus, vertices and edges in paths between arbitrary two vertices (i.e., *s-t paths*) can be subsets of the domain models. In addition, the number of paths and the number of vertices contained in a path affect the quality and effort needed during Model Space Exploration. For instance, if the Model Space Exploration algorithm finds too many short paths, domain experts may need to spend much time and effort to review and annotate semantically incorrect model instances. Thus, estimating the number of paths between *s-t paths* is an important consideration for the Model Space Exploration.

Counting the number of paths between *s-t paths* in a graph is simple if the graph has few vertices. If the graph is a directed graph, the number of paths between *s-t paths* can be counted easily even if done manually. However, counting the number of paths between *s-t paths* becomes complicated and error-prone if a graph has a large number of vertices. Thus, counting the number of paths between *s-t paths* is considered a #P-complete problems such that “*the detection of the existence of a solution is easy, yet no computationally efficient method is known for counting their numbers*” [Valiant, 1979].

Prior to investigating the number of possible model instances for Model Space Exploration, we first investigate the number of diagramming elements in several well-known software analysis and design modeling paradigms (e.g., UML, SysML, Structural Analysis and Design, and Petri Nets) in order to estimate the number of model instances to be explored. As shown in Table 6.2, most diagrams have less than 10 elements to model most notions.

Simple diagrams (e.g., Use Case, Petri Nets, and Context diagram) offer less than 5 elements. For instance, a Use Case diagram can model a requirement with only 3 elements: actors, Use Cases, and relationships. A flowchart is the diagram that has the largest number of diagramming elements and offers more than 20 distinctive elements.

Table 6.2 The Number of Diagramming Elements in Major Diagrams

The number of diagramming elements	Diagram	
	Directed	Undirected
Less than 4	Petri Net, Context Diagram	Use Case Diagram
Between 5 and 10	Statechart Diagram, Activity Diagram, Package Diagram, Entity-Relationship Diagram	Class Diagram, Component Diagram, Deployment Diagram
More than 10	Data Flow Diagram, Flowchart, Specification Description Language (SDL)	Unknown

In addition, as shown in Table 6.2, most diagrams are classified as “directed,” which means that the modeling elements are connected with other modeling elements using a directed relationship. For example, Petri Nets use a directed relationship to represent the direction of transitions. Modeling elements in a Use Case Diagram and Class Diagram can be connected with the directed relationship. But, in most cases, the directed relationship is ignored and used to specifically indicate the direction of controls or messages flow. Thus, we classify Use Case and Class diagrams as undirected.

The relationship between the number of paths and modeling elements are shown in Figure 6.7. To compute the number of paths between  $s$ - $t$  paths, graphs are designed to have a different number of vertices and graph density. The number of paths in Figure 6.7(a) is computed when a graph density is 0.85, and the graph in Figure 6.7(b) is 0.15.

A graph density describes the general level of connectedness, and a graph is complete if all vertices are adjacent to each other. In general, the more vertices that are adjacent, the greater

the graph density. The graph density can be calculated using the following equations, where equation (a) is used for calculating the graph density of the undirected graph, and equation (b) is for the directed graph. Generally, directed graphs are less dense than undirected graphs because directed graphs do not form loops.

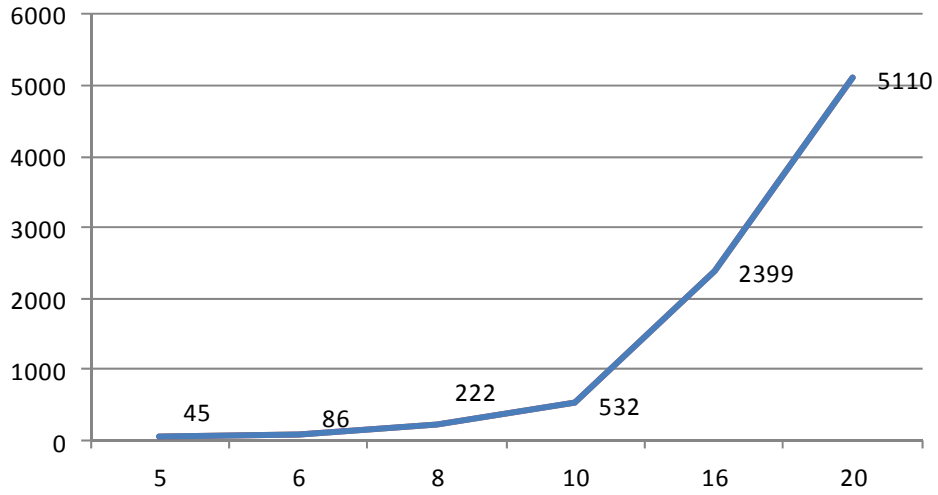
$$\Delta = \frac{2|E|}{(|V| * (|V| - 1))} \quad (a)$$

,where  $V$  and  $E$  are vertices and edges in graph  $G=(V,E)$

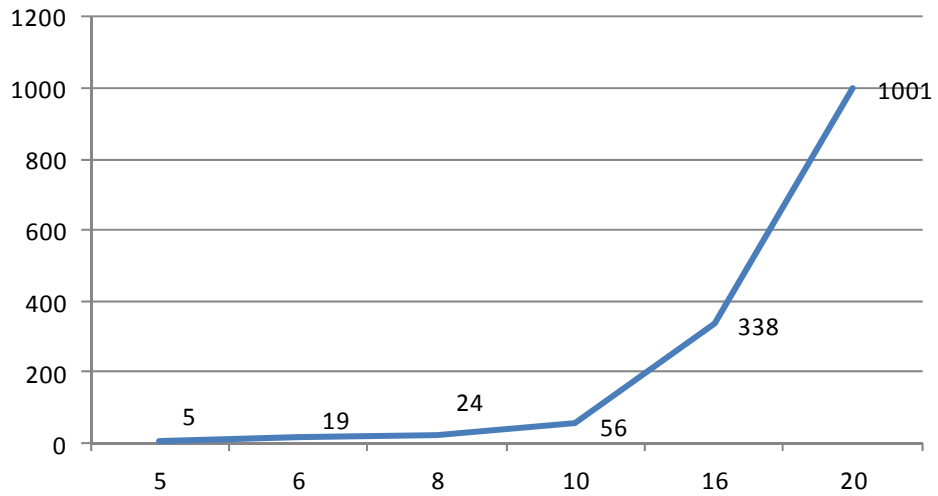
$$\Delta = \frac{2|E|}{(|V| * (|V| - 1))} \quad (b)$$

As shown in Figure 6.7, the number of paths between s-t paths is dramatically different according to the graph density. The higher the graph density, the higher number of paths that can be found between s-t paths. If a graph has more than 10 vertices, the number of paths between s-t paths is increased at least three times compared to a graph that has less than 10 vertices.

Considering the task of Model Space Exploration, from Figure 6.7(b), we can estimate that the positive model instances will be between 20 and 60. The reason is that, as shown in Table 6.2, most diagrams have less than 10 modeling elements, and modeling elements are linked with the directed relationships. This means that models created using those diagramming elements would have from a low to medium level of graph density. For instance, we could have 14 positive model instances and 41 negative modeling instances from network diagramming, which is described in the next chapter. The graph density of the process model is 0.35 and 0.7 for the positive graph and negative graph, respectively.



(a) Graph Density = 0.85



(b) Graph Density = 0.15

Figure 6.7 The Number of Model Instances vs. Modeling Elements

Based on the observation, we suggest that the Model Space Exploration algorithm instantiates a reasonable number of model instances to interact with domain experts if a DSML consists of less than 8 modeling elements. For DSMLs with a larger number of modeling

elements, a possible solution to reduce the burden of responding to the number of generated examples would be to break the exploration process into different stages, rather than providing all of the sample instances at once.

## 6.4 Related Work

Model Space Exploration aims to improve previously inferred metamodel and static constraints by reasoning about the domain through a set of model instances. The notion of Model Space Exploration originated from design space exploration, which is used to choose an optimized solution from a set of alternative design choices.

The DESERT tool suite [Neema et al., 2003] is a domain-independent tool chain for defining design spaces and executing constraint-based design space exploration. To model a design space, DESERT introduced the concept of an AND-OR-LEAF tree and supports a subset of the Object Constraint Language (OCL). AND-OR-LEAF tree captures the relationships between design space properties and represented them hierarchically, and a subset of OCL allows users to specify system requirements (e.g., compatibility constraints and performance constraints) as constraints. In addition, Ordered Binary Decision Diagrams (OBDDs) are used to model and prune the design space based on the constraints. By employing OBDDs, which can search all design spaces exhaustively and prune irrelevant designs, DESERT can create a small and complete design space that satisfies all design constraints. In addition, because DESERT is a domain-independent tool, it can be applied to a variety of exploration problems with a single encoding mechanism at the architecture level.

Schätz et al. [Schätz et al., 2010] proposed an interactive, incremental process using a model transformation technique for deriving possible design alternatives. In their approach,

various solutions to the design problem are offered iteratively to engineers to make design decisions, and the partial solutions generated by each iteration can be combined incrementally to make a complete solution. Model transformation is used to instantiate concrete models (e.g., description of the technical architecture) automatically from the abstract models (e.g., description of components, subsystems, and architecture) by applying relational and declarative rules. Thus, different design options can be created through the model transformation.

The DaRT (Data Parallelism to Real Time) approach [Bondé et al., 2005] applied a Model-Driven Architecture. DaRT needs to define metamodels to specify application, architecture, and software/hardware association. DaRT adopted a model transformation technique to create an optimized association model from metamodels. By adopting MDA, the approach is able to reuse models and unifies the definition of the transformation rules.

## 6.5 Summary

The notion of Model Space Exploration is borrowed from the ideas of design space exploration, which is widely used to automate the search for design alternatives. In our approach, Model Space Exploration is applied to verify the correctness of an inferred metamodel and its static constraints. The verification is based on a confirmation step that involved a domain expert answering questions about a set of generated model instances. If the generated domain models are annotated differently from the expected result, the models are presented to domain experts again to resolve the conflict. The results of the Model Space Exploration are fed back to the Intermediate Design Space to complement the previously inferred metamodel and static constraints.

## CHAPTER 7

### CASE STUDIES APPLYING MLCBD

In previous chapters, we described our approach to create a DSML using a By-demonstration approach. The approach utilizes several techniques such as by-demonstration, grammar inference, and Model Space Exploration. In this chapter, we apply the MLCBD framework for developing a finite state machine (FSM) and a network modeling language. In addition, the two DSMLs are also developed with two other DSML development environments (e.g., Graphical Modeling Framework (GMF) and Generic Modeling Environment (GME)) for comparison purposes.

The rest of this chapter is divided into three sections: Section 6.1 describes the development of a language to represent a FSM, Section 6.2 describes the development of a network modeling language, and Section 6.3 concludes with an evaluation and summary discussion.

#### 7.1 Development of a Finite State Machine Modeling Tool

A Finite State Machine (FSM) is a primitive, but useful computation machine. Some exemplary application areas of FSM are pattern matching, sequential logic circuits modeling, and natural language processing [Hopcroft et al., 2006].

An FSM can be formally defined as a 5-tuple  $(Q, \Sigma, \delta, i, F)$ , where  $Q$  is a finite set of states,  $\Sigma$  is a finite alphabet,  $i$  is the initial state ( $i \in Q$ ),  $F$  is the set of final states ( $F \subseteq Q$ ), and  $\delta$  is the

transition function mapping  $Q \times \Sigma$  to  $Q$ , which implies  $\delta(q,a)$  is a state for each state  $q$  and input  $a$  that is accepted when in state  $q$ .

A simple FSM is shown in Figure 7.1. This FSM is designed to check whether input values are multiples of three.

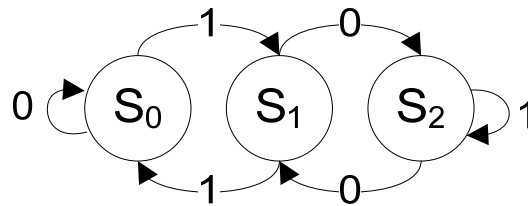


Figure 7.1 An Example of FSM

For instance, if the input value is 3, which is represented 11 in binary, the FSM begins from state  $S_0$  and moves to the next state  $S_1$  by first binary input 1, which corresponds to transition 1 from  $S_0$  to  $S_1$ . Finally, the FSM moves back to state  $S_0$  by the second value, which is also 1.

In this section, we describe how to develop the simple FSM, illustrated in Figure 7.1, with two approaches: MLCBD and the Graphical Modeling Framework (GMF).

### 7.1.1 FSM Requirements Modeling with Syntax Map

In this section, we describe the requirements of the FSM using a Syntax Map. To describe the requirements of the FSM, domain experts first create the skeleton scenarios by placing classifiers and relationships between the Syntax Map symbol *Start* and *End* according to their usage scenarios. For instance, if the current state is  $S_0$  and a transition is triggered, the next

state is determined by the transition condition. Thus, the destination state can be either State  $S_0$  or State  $S_l$ .

A Syntax Map of the simple FSM is shown in Figure 7.2.

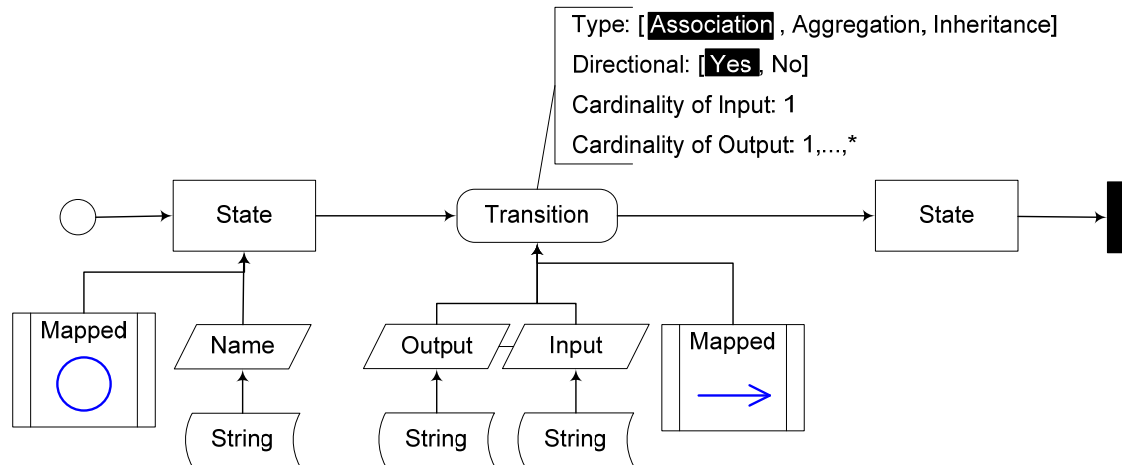


Figure 7.2 Syntax Map for FSM

As shown in Figure 7.2, *State* is assigned to classifier having an attribute, *Name*, with type *String* to denote instance name. If multiple classifiers (or relationships) appear in a Syntax Map, the attributes are automatically associated when attributes are tagged by the domain expert. For example, two *States* appear in Figure 7.2. Attributes are only associated to *State*, located in the left-most side, but not associated to *State* at the right-most end. This implies the same attributes are also associated to *State* at the right-most side if other attributes are not associated to it.

### 7.1.2 FSM Development with GMF

Several metamodeling tools are available to assist in developing graphical editors for a DSML. However, creating a new graphical editor for a DSML can be slow and painful because software engineers need to understand the intricacies of the metamodeling tool. This subsection uses an Eclipse-based metamodeling environment to place into context the details needed to create a DSML using metamodeling tools.

The Eclipse Modeling Framework (EMF) and the Graphical Editing Framework (GEF) are well-known Eclipse frameworks for creating DSMLs. The EMF is “*a modeling framework that exploits the facilities provided by Eclipse and enables software engineers to rapidly build robust tools and other applications based on a structured data model*” [Steinberg et al., 2008]. EMF offers several features such as metamodeling, template-based Java code generation, and XMI serialization and deserialization. The GEF was developed to “*support the creation of rich graphical editors and views for Eclipse-based tools and Rich Client Platform (RCP) applications*” [Rubel et al., 2011]. GEF consists of three components: Draw2D, Zest, and GEF. Draw2d is a standard 2D drawing framework that supports a lightweight drawing. Zest is built on top of Draw2D and provides interfaces to bind Java models. GEF provides APIs for enabling interactive diagrams. Combining EMF and GEF, software engineers can develop any kind of applications that require a graphical editing environment. Modeling applications (e.g., business process modeling and UI design) are well-known domains that use EMF and GEF. However, because EMF and GEF have been developed and evolved for different purposes, several technical challenges (e.g., different command stack) may make it difficult for some developers to

create graphical editors using these two technologies. To address the technical challenges integrating EMF model within the GEF framework, GMF was introduced.

The GMF provides a generative component and runtime infrastructure for developing graphical editors based on the EMF and GEF. The GMF offers editors to model the notation and semantics of a graphical editor, as well as a generator to produce the source code of a graphical editor. In addition, GMF provides a run-time component that provides a consistent look and feel, and allows extending its features by third parties.

As shown in Figure 7.3, to develop an application using GMF, software engineers have to define four different models (e.g., domain model, graphical definition model, tooling definition model, mapping model, mapping model).

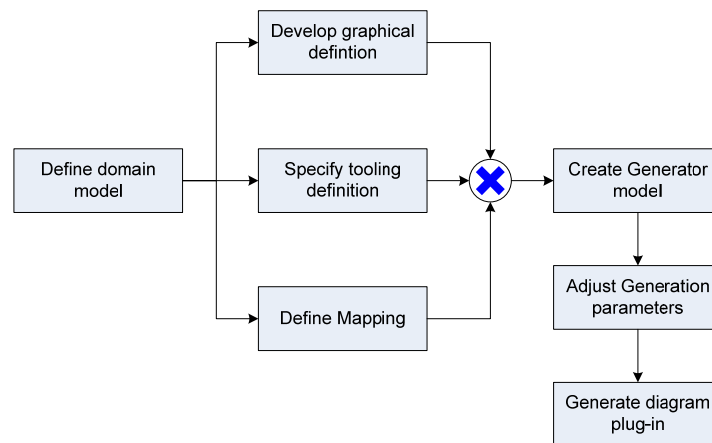


Figure 7.3 GMF Tooling Workflow

The domain model is the basis of all artifacts and defines the abstract syntax of an application. The graphical definition model defines symbols (e.g., figures, nodes) that will be displayed on the diagram. The tooling definition model specifies graphical elements such as

palette, creation tools, and actions. The mapping model specifies how to bind the three models to form a language environment (i.e., it binds the domain model, graphical definition model, and tooling model).

After all models are defined, a generation model is created by combining the four models to generate some external artifact from the model (e.g., source code or some other artifact). Finally, an application is obtained by compiling and linking Java code, which is generated from the generation model.

As a first step of FSM development for this first case study, the FSM domain model is defined as shown in Figure 7.4. The domain model can be defined in several ways: ECore, annotations on a Java interface, UML models, or XML schema. The model presented in Figure 7.4 is defined using Ecore, which is a metalanguage developed based on the OMG's MOF.

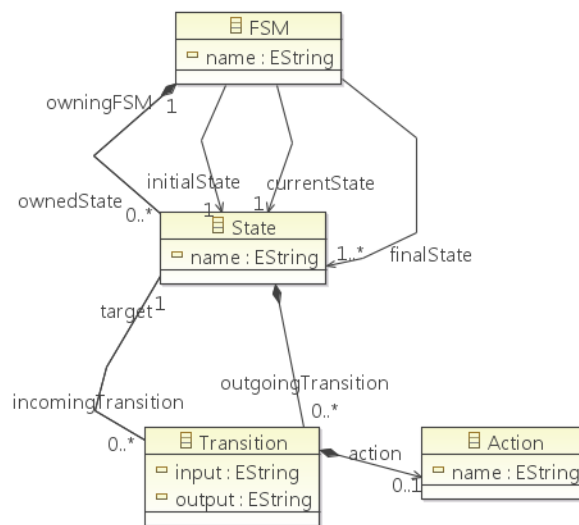


Figure 7.4 Ecore model for FSM [Kermeta]

Class *FSM* is the representative entity that manages all notions in the FSM domain. Classes *State* and *Transition* are defined to describe the notion of *State* and *Transition* in the FSM, respectively. In addition, other properties (e.g., cardinality and roles) are associated to the relationship that links between *State* and *Transition*.

The graphical definition specifies how model elements in the domain model are mapped to graphical elements. As shown in Figure 7.5, all domain model elements and possible graphical representations are listed in a graphical definition dialog, and the mapping between them is specified by selecting a checkbox. At this dialog, domain model elements can be mapped only to fixed graphical notations; e.g., rectangle, line, or text (from left to right in callout). In this example, Class *State* is mapped to rectangle, and its attribute name is linked with text. Similarly, class *Transition* is mapped to line.

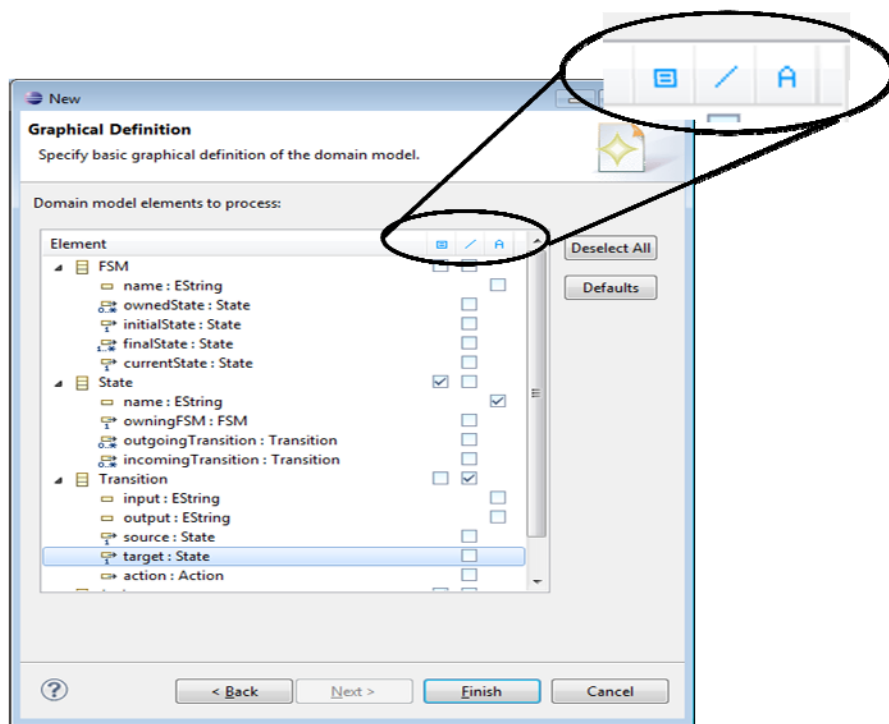


Figure 7.5 Specifying a Graphical Definition in GMF

After the graphical definition model is created, the graphical representation of each domain model element can be further refined with more appropriate graphical representations in order to provide better cognitive effectiveness as link to domain notation. Figure 7.6 shows the graphical definition model for FSM.

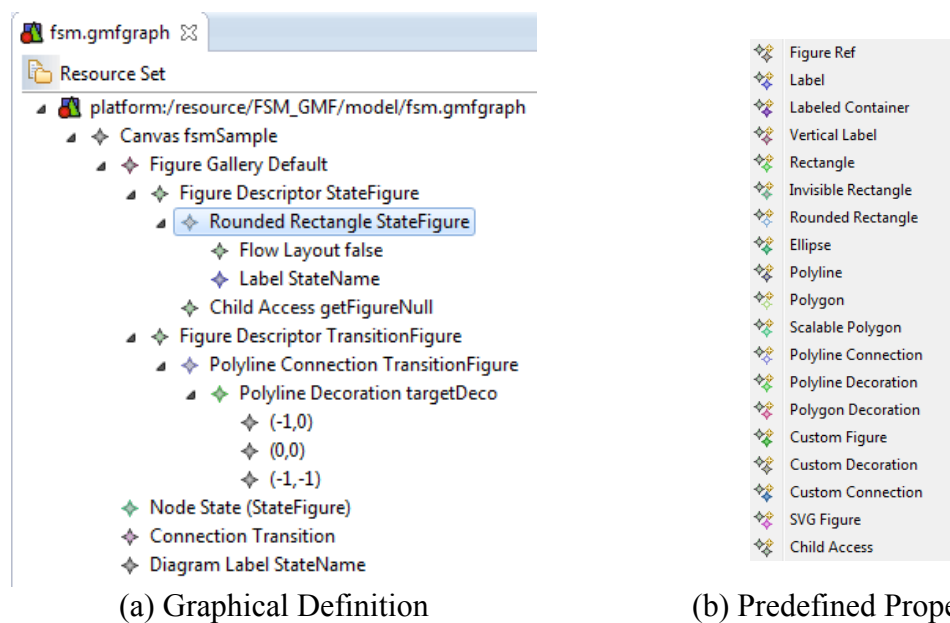


Figure 7.6 Refine Graphical Definition

To refine the graphical representation, the graphical definition model offers predefined shapes (e.g., rounded rectangle, ellipses, and polygon). An SVG (Scalable Vector Graphics) figure also can be used, which is designed with an XML-based scalable vector image that can support interactivity and animation (see Figure 7.6(b)). In order to refine the graphical representation, an existing definition should be removed and then replaced with a new definition. In addition, all the related properties should be specified in accordance with the new graphical definition.

Figure 7.7 illustrates the tooling definition model. This model defines the mapping rule between model elements in the domain model and graphic representations for the palette.

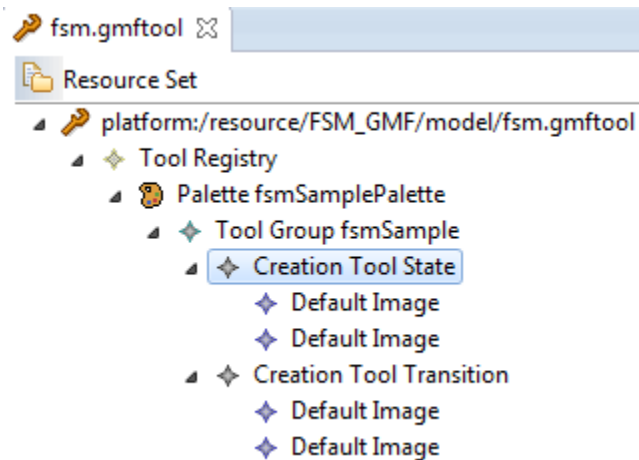
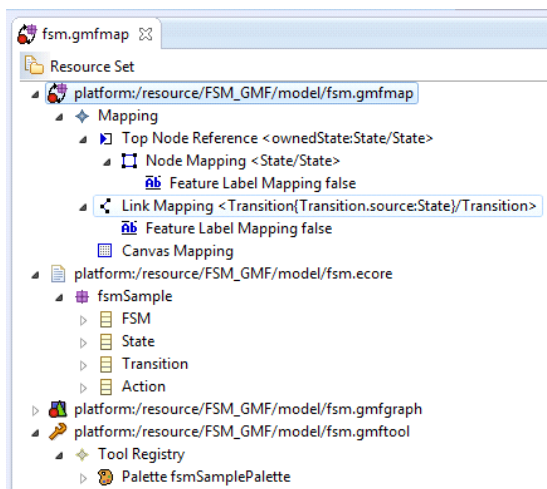


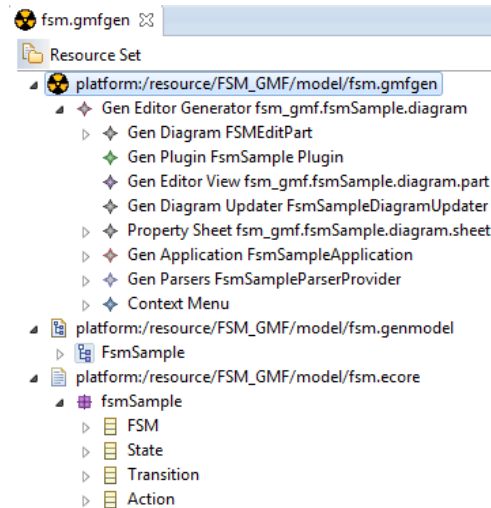
Figure 7.7 Tooling Definition Model

Finally, the mapping model combines all three pieces of model information in order to create a generation model, which contains control parameters for code generation. The mapping and generation model are presented in Figure 7.8(a) and (b), respectively.

Based on the generation model, Java code is generated automatically by a model transformation, which reads the generation model and then produces three plug-in source codes: domain model, diagram editor, and domain editor. The domain model, diagram editor, and domain editor correspond to the Model, View, and Controller (MVC) architecture, respectively. Figure 7.9 shows the FSM modeling tool, which is built using the generated source code from the various language models described in this section.



(a) Mapping Model



(b) Generation Model

Figure 7.8 Mapping Model and Generation Model

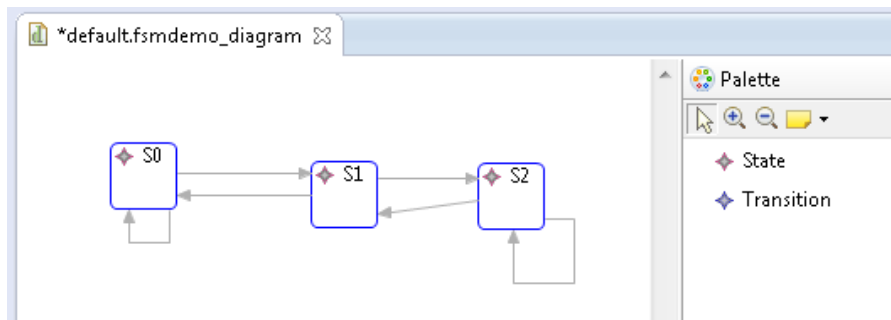


Figure 7.9 FSM Tool Created using GMF

GMF offers more advanced technologies and convenient methods for building graphical editors by combining EMF and GEF. Thus, software engineers can build graphical editors semi-automatically by just defining a few language definition models and mapping them properly. In addition, the GMF user interface offers a consistent look and feel, which may help software engineers maintain their focus of interest when specifying domain instance models.

### 7.1.3 FSM Development with MLCBD

In this section, we describe how to develop the FSM modeling tool with MLCBD. To build a DSML, MLCBD follows three steps: Recording, Creating, and Exploring Model Space. The recording step begins by clicking a button placed at the far left of the toolbar (see Figure 7.10 (a)). When the button is clicked, a new page is created and two stencils, Basic Shapes and Default, are loaded into Visio.

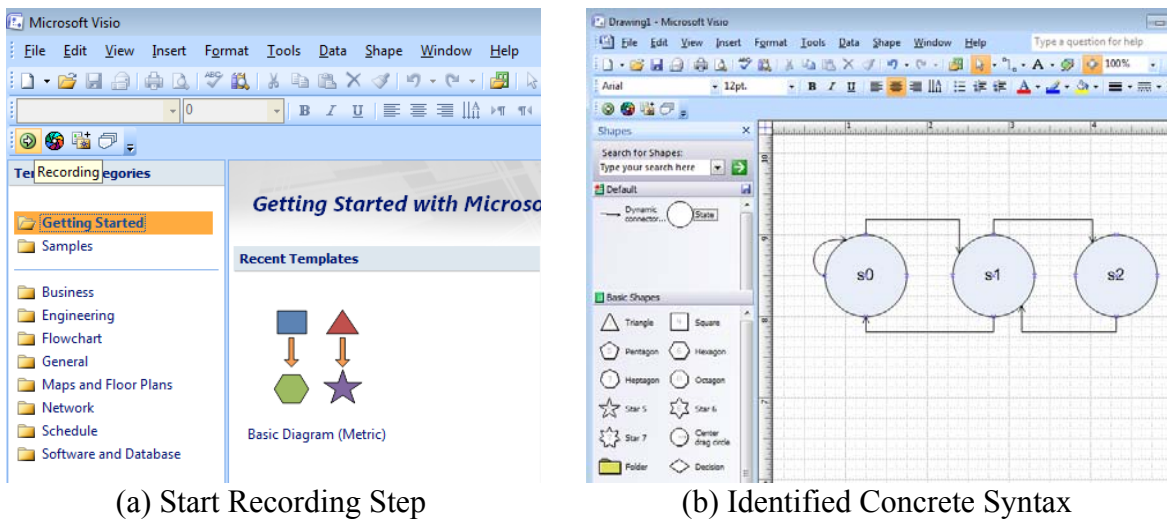
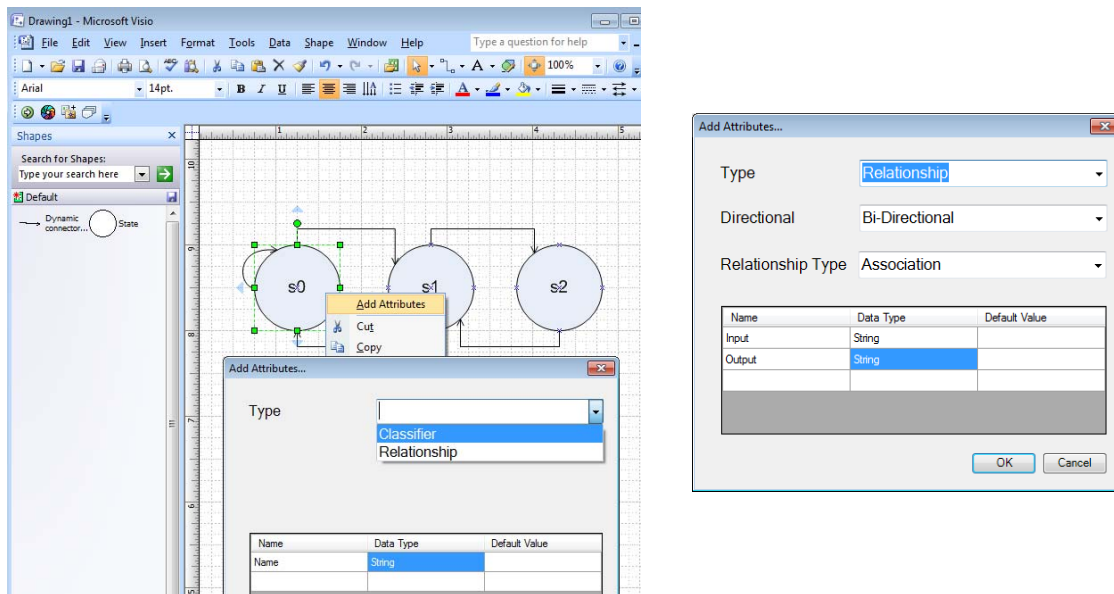


Figure 7.10 Creating FSM By-Demonstration

When a domain expert demonstrates a FSM model in an MS Visio Page (see Figure 7.10(b)) by dragging from the stencil *Basic Shapes* to the editor canvas, each user action is captured by a background process in order to identify the unique symbols as candidate concrete syntax elements. The unique symbols are copied to stencil *Default*, which functions as a concrete

syntax repository. As shown in Figure 7.10(b), only one circle and one line are placed in the stencil *Default*, even though three circles and five lines are used to demonstrate a FSM model.

After completing the FSM model demonstration, a domain expert annotates symbols to add information that guides the metamodel and static constraints inference. Figure 7.11 represents screenshots of the annotation. The annotation is performed on the demonstrated model instead of master symbols in stencil *Default*. As shown in Figure 7.11(a), if a symbol is mapped to a classifier, a domain expert can only add attributes. A domain expert does not need to annotate every symbol. If a symbol is annotated, MLCBD automatically annotates the information to the same type of symbols and corresponding master in stencil *Default*.



(a) Annotating Classifier

(b) Annotating Relationship

Figure 7.11 Annotating Concrete Syntax in MLCBD

For example, if the left-most circle is annotated as depicted in Figure 7.11(a), the rest of the two circles and a circle in stencil *Default* will be attached to the same annotation information

as the left-most circle. If a symbol is a relationship, two additional properties (i.e., Directional and Relationship Type) need to be specified along with attributes.

Finally, the MLCBD framework creates the FSM modeling language while inferring static constraints. As illustrated in Figure 7.12, two types of static constraints (e.g., link boundness and structural constraints) can be inferred while a domain expert demonstrates the FSM concepts. In addition, the “*Not allowed to use shapes in other master*” option is provided to prevent the domain expert from using shapes in another stencil when the FSM modeling tool is created. This is needed because MLCBD is developed over MS Visio, and MS Visio handles a DSML as just a stencil without any underlying knowledge. MLCBD provides the constraints as underlying Visual Basic scripts to enforce the static semantics.

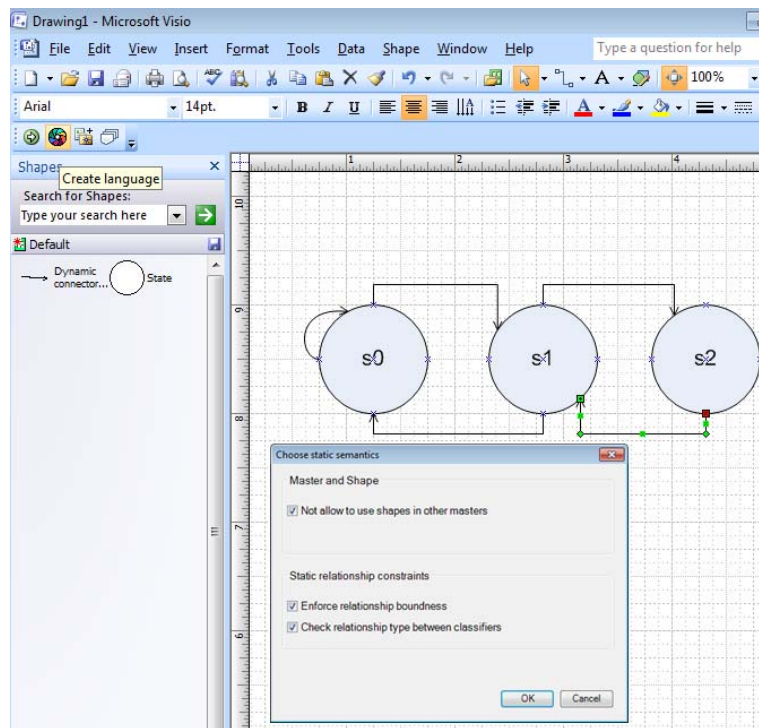


Figure 7.12 Creating FSM while Inferring Static Constraints

#### 7.1.4 Comparison FSM Development: MLCBD vs. GMF

FSM is a useful example for comparison between approaches because it is simple and intuitive. Thus, FSM was chosen in this section to assist with the comparison between MLCBD and more traditional approaches, such as GMF. There are four criteria in which the advantages and disadvantages of the two approaches can be compared:

- *Complexity of development:* The learning curve is an important factor to determine whether to adopt a new technology. Although GMF simplifies graphical editor development by combining EMF and GEF, software engineers need to invest some time and effort to learn how to create and define a new DSML (e.g., domain model, graphical definition model, tooling definition model, and mapping model). For example, software engineers have to possess knowledge about metamodeling for domain model definition and also understand how models are related to each other by specifying the mapping between them. Thus, developing a DSML with GMF requires software engineers to invest some time and effort to learn the GMF tooling mechanism. If domain experts who have domain knowledge (but do not have knowledge about this kind of tool development environment) try to develop their own DSML with GMF, they will likely need to spend more time and effort than software engineers who are experts in software development, but do not have domain knowledge. For many domain experts, the likelihood of actually using a framework like GMF is not possible.

On the other hand, MLCBD does not require the same amount of effort and time as GMF in order to develop a new language. With MLCBD, a DSML can be built by demonstrating the notions of the domain on the modeling canvas. The MLCBD framework does not require language development expertise. Domain experts need only domain knowledge and a basic understanding of Visio in order to develop their own DSML with MLCBD.

- *Completeness of graphical representation:* Both tools support a set of predefined symbols to represent notions of a domain. In addition, both tools also support the use of image files for graphical completeness. However, as shown in Figure 7.5 and Figure 7.6, GMF uses a menu-driven approach in order to replace a graphical definition. For example, to change a graphical definition from a regular rectangle to a rounded rectangle, a language designer must delete the existing graphical definition for a rectangle and then instantiate the context menu to add the new graphical definition. The context menu needs to be instantiated whenever the language designer wants to add additional attributes or styles to the newly added graphic definition.

On the contrary, MLCBD, which supports WYSIWYG (What You See Is What You Get), it is possible to drag and drop new shapes after deleting existing shapes. Thus, a domain expert can change the graphic definition more intuitively than with GMF.

- *Analysis and Debug Capability:* GMF offers a validation feature to check whether a model is correctly specified. If there is an error in any model, the graphical editor cannot be generated. Although GMF can locate where the error occurred,

the changes cannot be propagated to other related models. Thus, a language designer needs to traverse every model to reflect the changes.

- *Language Evolution*: When the domain model is changed or evolved, GMF must change every related model to support backward compatibility. In addition, all source codes should be recompiled to build a new version that contains the changes. GMF does not support backward compatibility. In MLCBD, the domain expert only needs to reload the language definition and re-demonstrate the evolved changes to the DSML.

## 7.2 Development of Network Diagramming Tool

Computer networks have become an important part of daily life by allowing many different types of users to share resources and information. In order to provide more convenient services, a computer network consists of various types of hardware components, which are interconnected by communication channels. Routers, switches, gateways, (cable) modem, servers (e.g., DHCP, DNS, file, mail, and etc), and computers (e.g., desktops, laptops, and tablets) are examples of components that make up a computer network.

A network diagram depicts how hardware components are interconnected with the communication channels. A network diagram is useful to describe the placement of the network's various components and data flows within a network.

This section describes the development of a simple network diagramming tool as a case study of MLCBD. For comparison purposes, a simple network diagramming tool is also developed with the Generic Modeling Environment (GME), a traditional metamodeling tool.

### 7.2.1 *Network Diagramming Tool Requirements Modeling with Syntax Map*

Requirements of the network diagramming tool are illustrated in Figure 7.13 using a Syntax Map. For modeling a network, nine elements (e.g., *Server Farm*, *Media Server*, *Router*, *Network*, *Group*, *Cable Modem*, *Laptop*, *Desktop*, and *Tablet*) are connected to each other with a link. In this case study, a *Server Farm* and a *Media Server* are only connected with a *Router* to provide services. Because a link is an association relationship that is undirected, the reverse connection (e.g. from *Router* to *Server Farm*) is also allowed.

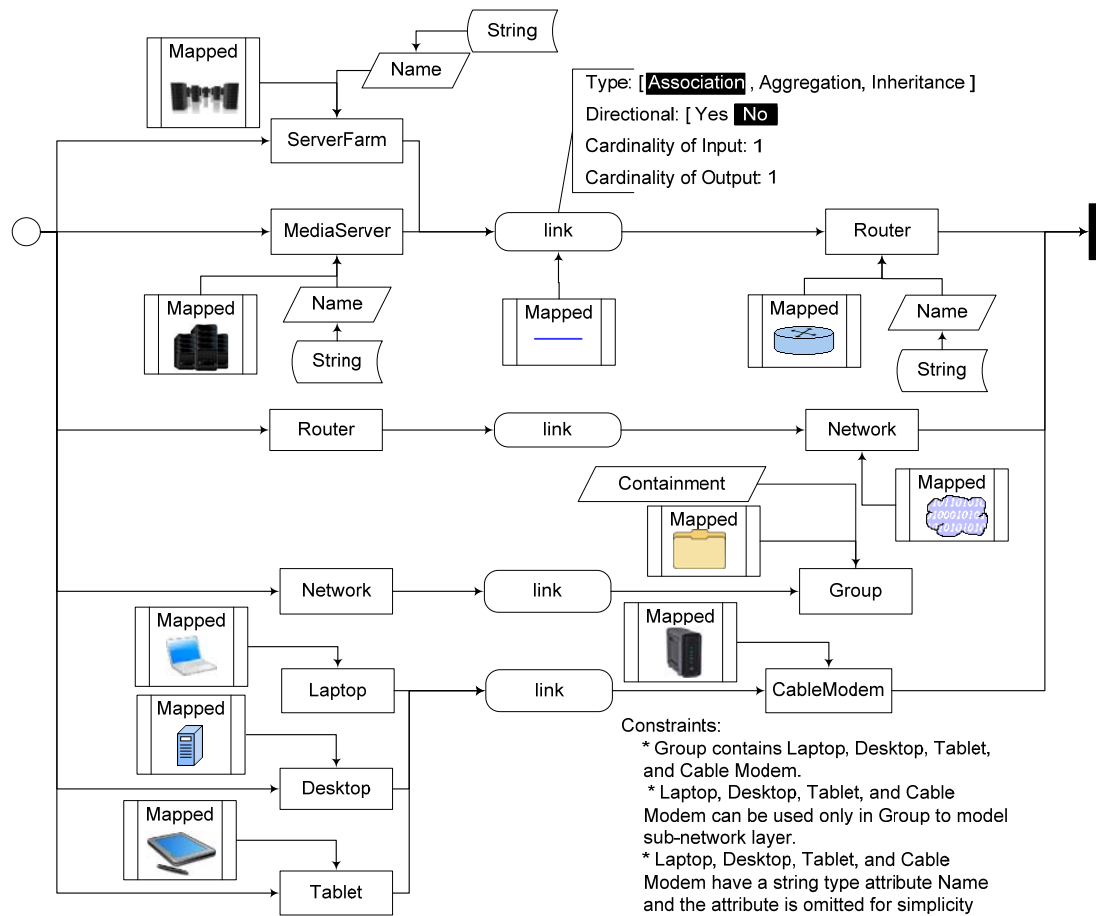


Figure 7.13 Syntax Map for Network Digramming

A *Network* is connected with a *Router* by a link, but it cannot be linked to *Server Farm* and *Media Server* directly. A *Network* is also connected to *Group*, which can define a sub-network domain. *Group* can have different network components. To specify the network elements to be contained in a *Group*, constraints are associated to the *Group*. The constraints are expressed using natural language, but must be formalized to have an operational effect in a modeling tool. In Figure 7.13, constraints specify *Laptop*, *Desktop*, *Tablet*, and *Cable Modem* as contained parts of *Group*. In the sub-network domain, the *Laptop*, *Desktop*, and *Tablet* are linked

to a *Cable Modem* to communicate with each other. As mentioned in the constraints, the string type attribute *Name* is omitted in *Laptop*, *Desktop*, *Tablet*, and *Cable Modem* for simplicity. Figure 7.14 depicts the sample network models that conform to the requirements described using the Syntax Map from Figure 7.13.

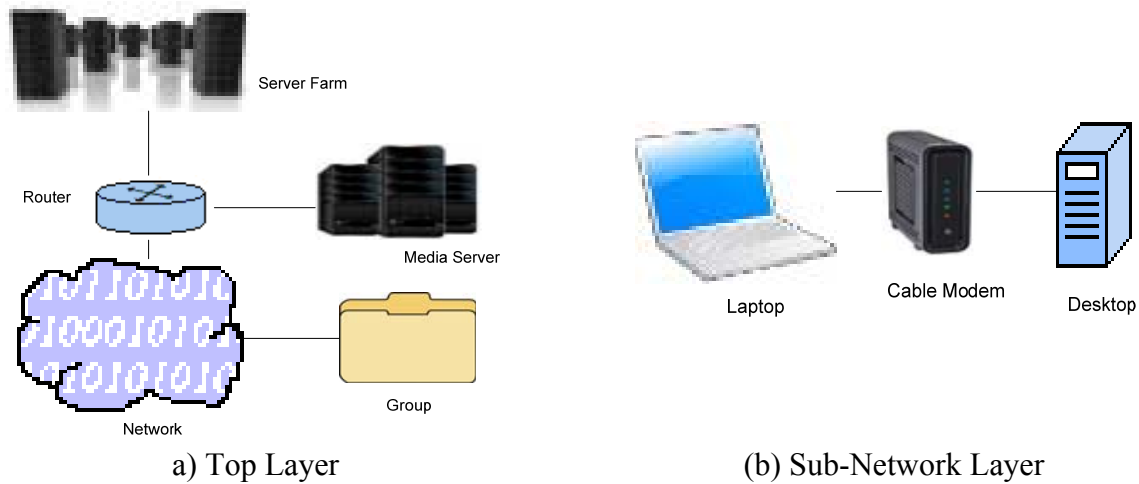


Figure 7.14 Sample Network Digramming Model

### 7.2.2 Development of Network Modeling Tool using GME

The GME is “a domain-specific, model-integrated, configurable program synthesis tool for creating and evolving domain-specific, multi-aspect models of large-scale engineering systems” [GME, 2012; Lédeczi et al., 2001]. GME provides various features to build DSMLs. Metamodeling, hierarchy, multiple aspects, sets, references, and explicit constraints are examples of the capabilities that GME offers for language definition. A metamodel plays a key role in GME and contains all of the syntactic, static semantics, and presentation information about the domain. Hierarchy is introduced to model the containment between modeling elements, and multiple aspects enable multiple views of the models. Sets are used for metamodeling to show a

set of metamodel elements that are related to a particular object. References enable reuse of existing metamodeling elements. Constraints check and/or specify context types and multiplicity using OCL.

To develop a DSML with GME, the language designer must have expertise about the core GME modeling concepts, such as root folder, atom, model, and aspect. The root folder is located at the very top of the project hierarchy, and each modeling project should have at least one root model. Atom is a basic GME entity and has no internal structure except attributes. Model is very similar to atoms, but can contain atoms, other models, and other types of objects. An Aspect is not a modeling element, but is used to provide different views of model structures.

The GME user interface consists of several features, as shown in Figure 7.15.

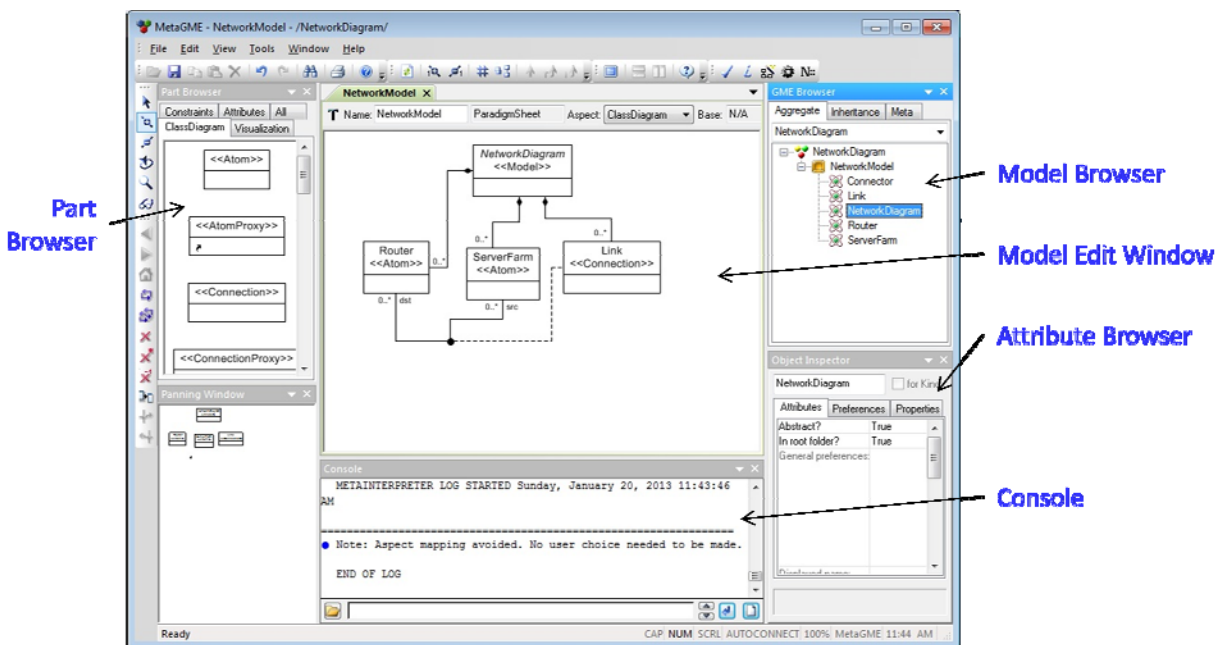


Figure 7.15 User Interface of GME

The Part Browser is a repository that contains all the parts that are required for metamodeling. The Part Browser consists of five different tabs (e.g., Class Diagram, Visualization, Constraints, Attributes, and All) and each tab has a different set of parts to support different modeling notions. For example, the Visualization tab has Aspect and AspectProxy, and SameAspect, which can manage various modeling aspects. The Model Browser consists of three tables: Meta, Aggregate, and Inheritance. The Meta tab lists all metamodeling elements used in the Model Edit Window, and the Aggregate tab shows the project management information hierarchy using tree-based containment. The Inheritance tab informs the type inheritance of a model. The attribute browser lists all attributes and preferences of an object. The Console area displays error information during model interpretation (i.e., transformation of the model to some other form, such as source code).

The metamodel for the network diagramming tool is shown in Figure 7. 16.

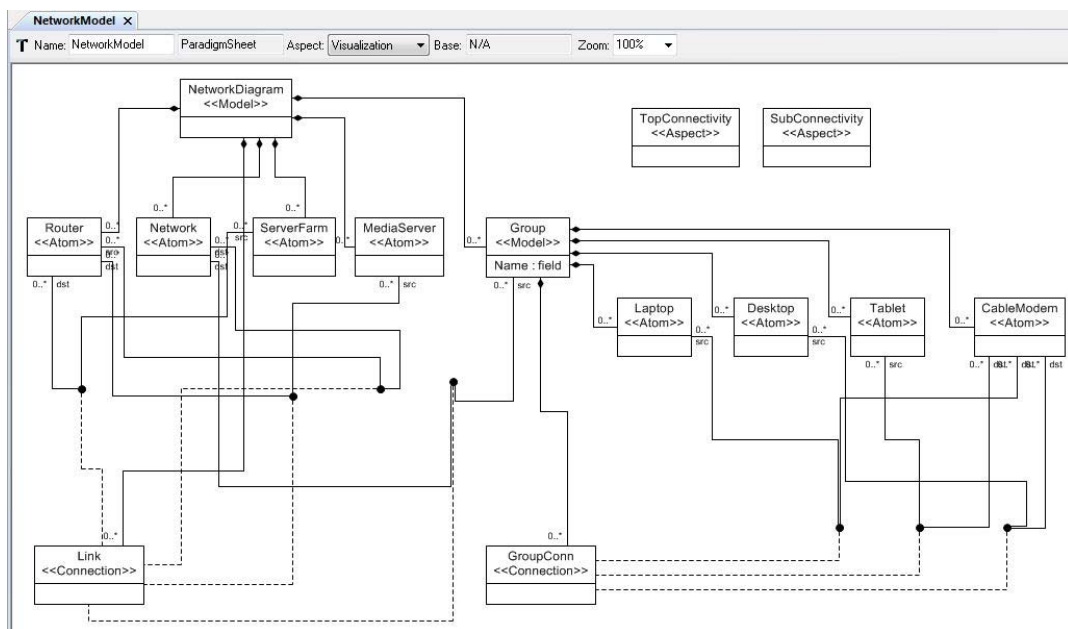
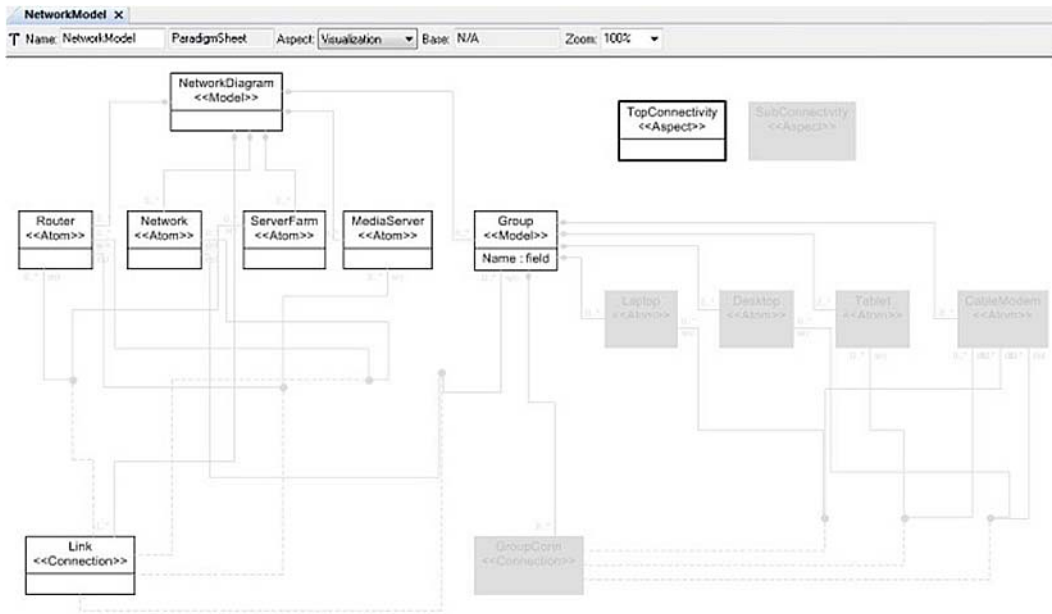


Figure 7.16 Metamodel for Network Diagramming

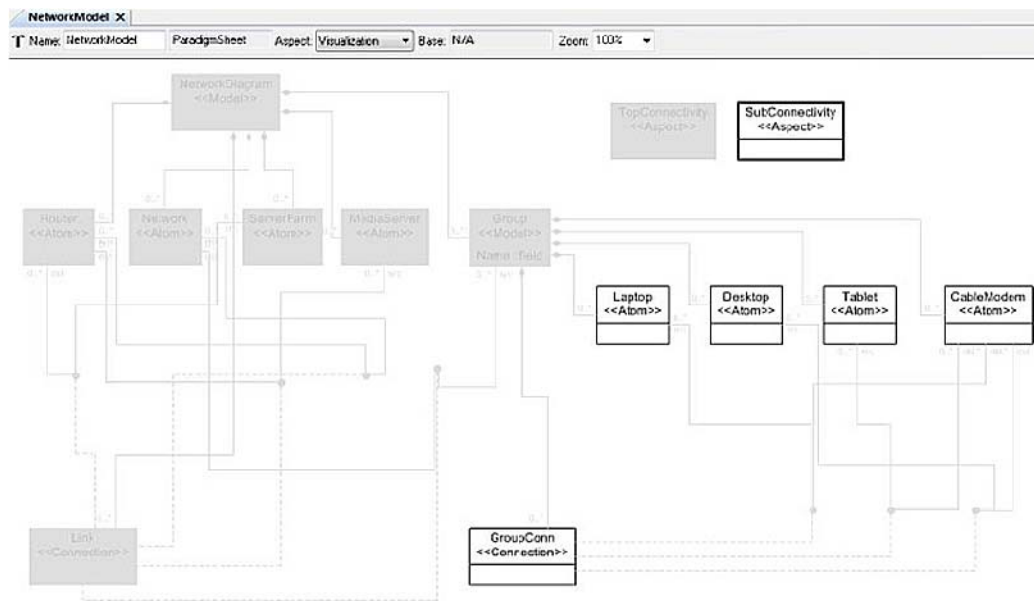
The metamodel includes two models, eight atoms, two connections, two aspects, and seven connectors. Model *NetworkDiagram* is defined as the root folder, and all metamodel elements except atoms, which are used in the sub-network layer, are linked to model *NetworkDiagram* with an aggregation relationship. Another model, called *Group*, is a container and manages the network elements in the sub-network domain. Atoms are used to define eight network hardware elements (e.g., *Network*, *Router*, *Media Server*, *Server Farm*, *Desktop*, *Tablet*, *Laptop*, and *Cable Model*). *Connection Link* links hardware elements in the top layer, and connection *GroupConn* is for connecting hardware elements in model *Group*. Seven connectors are defined to specify how atoms (or models) are linked each other and which connections are associated to the link. For example, atom *ServerFarm* is the source of a link and linked to atom *Router* through connection *Link*.

In addition, the metamodel has two aspects: *TopConnectivity* and *SubConnectivity*. As mentioned previously, an aspect is used to provide different structural views of a metamodel. In Figure 7.17(a), when aspect *TopConnectivity* is selected, a language designer can see metamodel elements only for the top layer (e.g., *Router*, *Network*, *Server Farm*, *Media Farm*, *Group*, and *Link*). Metamodel elements that are not related to the top layer are grayed. On the other hand, aspect *SubConnectivity* is selected, shown in Figure 7.17(b), a language designer can see all metamodel elements for the sub-network domain.

To define concrete syntax, a language designer must specify a value of the attribute *Icon*. The attribute *Icon* only accepts a string representing a file name to a graphics file. As shown in Figure 7.18, the Option box is opened to specify the icon path, and then the icon name and displayed name are specified in the attribute browser. If concrete syntax is not specified as icons, a default symbol is associated to each atom.



(a) Aspect for the top layer



(b) Aspect for the sub network layer

Figure 7.17 Aspects of Network Diagramming Metamodel

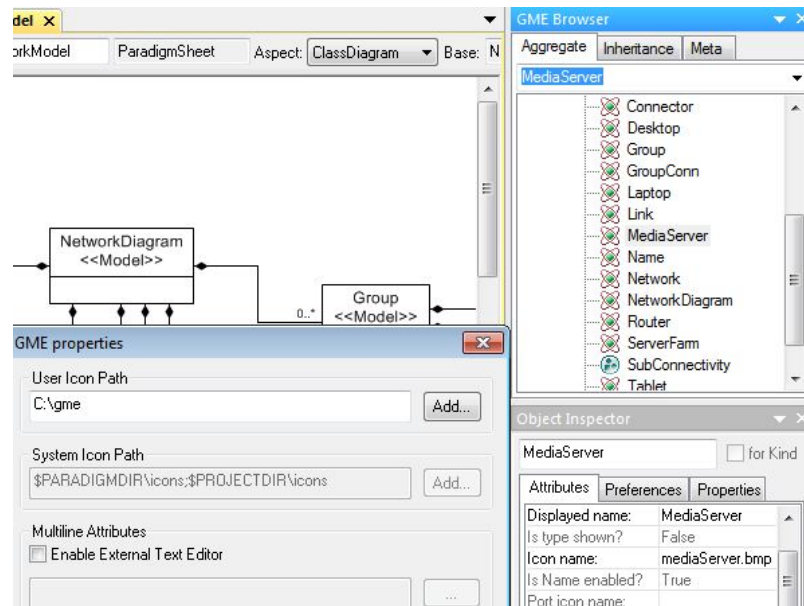


Figure 7.18 Concrete Syntax Specification

The last step for network diagramming tool development in GME is to generate and validate the metamodel by the GME environment generator. If no errors are found in the metamodel, the network diagramming tool is created and registered for domain experts to then use. The generated environment contains all the syntactic and semantic information about the domain to support the creation of a family of models in that domain.

Figure 7.19 depicts the creation of a new network diagram from the generated metamodel in GME. To create a new network diagram, a domain expert selects the *NetworkDiagram* metamodel from a new project dialog. The dialog box lists all available registered metamodels that have been created.

Figure 7.20 illustrates examples of network diagrams created from the *NetworkDiagram* metamodel. As shown in Figure 7.20, a network model can be created using the same user interface that was used for the network diagramming tool development (thus, showing the meta

nature of the GME and other similar metamodeling tools – this is the concept introduced in Chapter 2 that showed the relationship between a metamodel and a domain instance model).

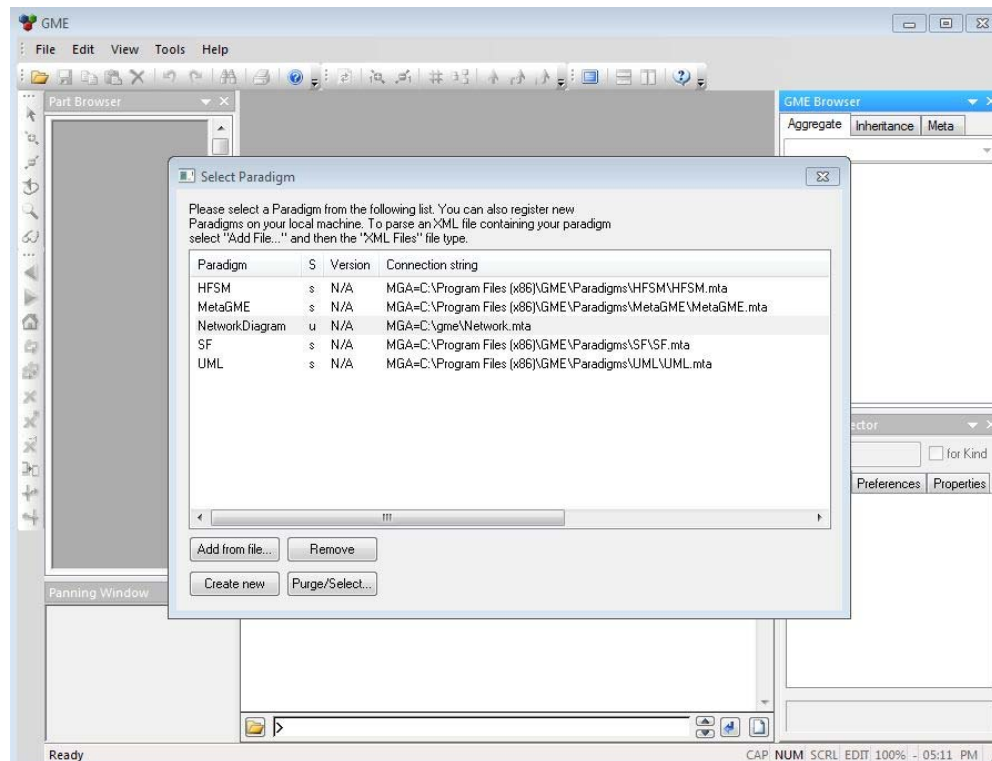
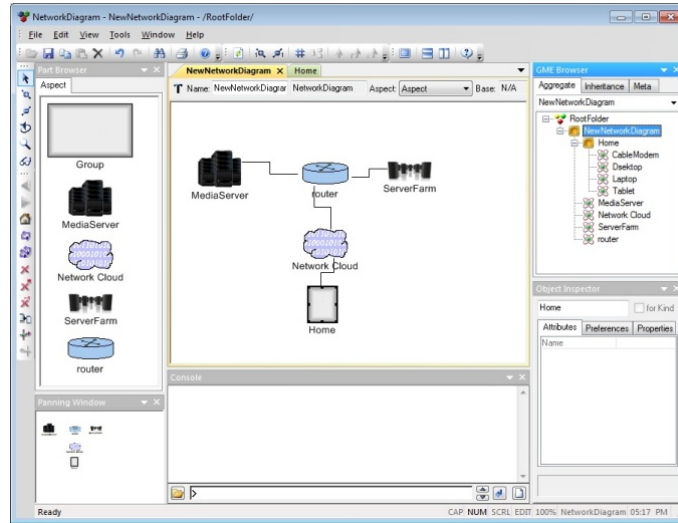


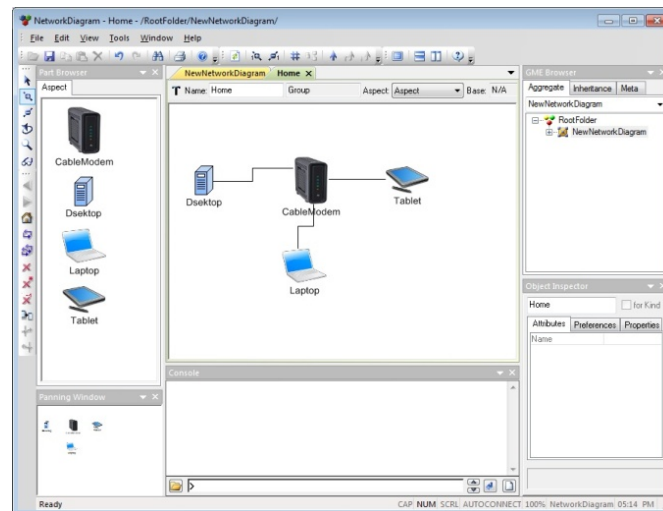
Figure 7.19 Creating a New Network Model

To create a network diagram, the network hardware components are listed in the Part Browser (during the previous step for defining the metamodel, the same area instead contained metamodeling elements). The Part Browser contains different hardware components depending on the model. In Figure 7.20(a), the Part Browser contains *Group*, *Network*, *Media Server*, and *Router*, which are aggregated parts of model *NetworkDiagram* and defined for modeling the top layer. On the other hand, the Part Browser in Figure 7.20(b) contains hardware components (e.g.,

*Cable Modem, Laptop, Desktop, and Tablet*), which are linked to model *Group* with the aggregation to model the sub-network domain.



(a) Top-Layer



(b) Sub-Layer

Figure 7.20 Network Models in GME

### 7.2.3 Development of Network Modeling Tool using MLCBD

In this section, we describe the development of the same network diagramming tool using MLCBD. Because network hardware components are not provided as predefined symbols in MLCBD, domain experts need to gather all of the images of the network hardware components used for modeling their network in a folder prior to demonstrating the network model examples, as shown in Figure 7.21.

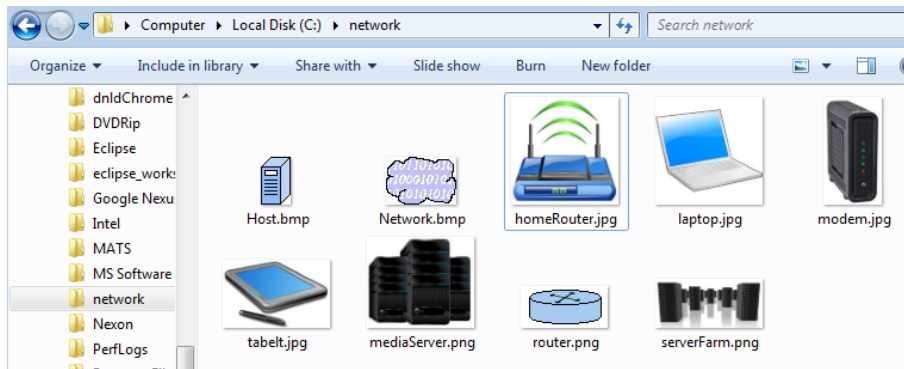
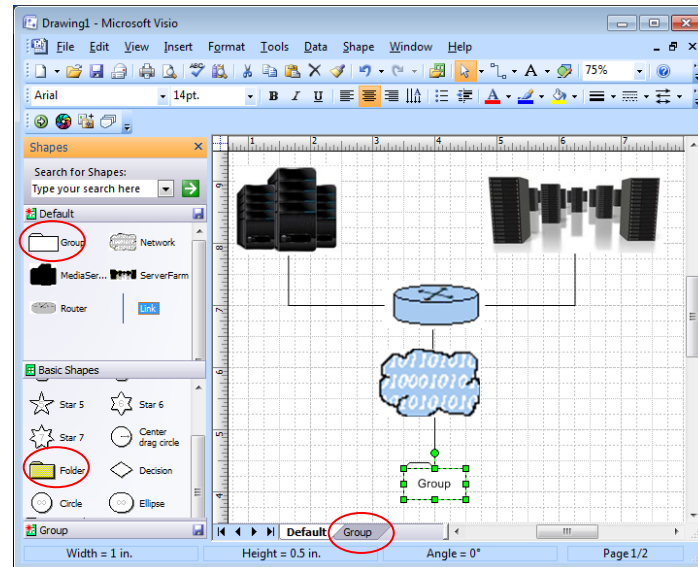


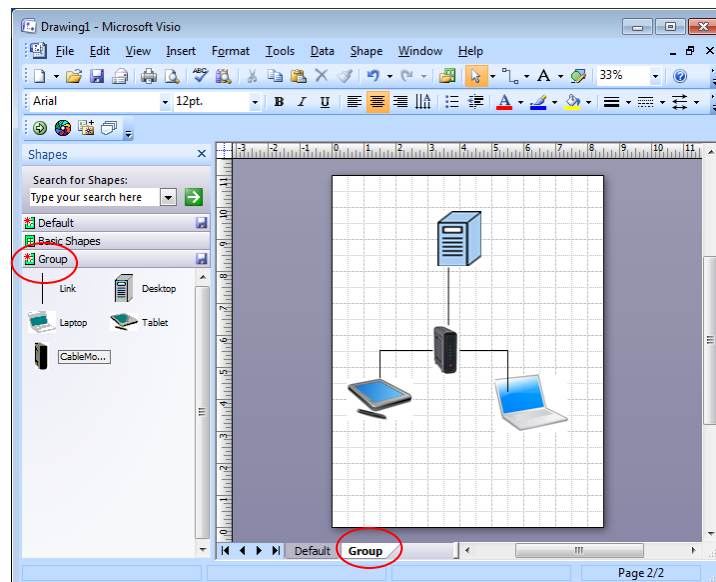
Figure 7.21 Image Files of Network Hardware Components

To create a network diagramming tool in Visio using MLCBD, a domain expert drags the network image files and drops them into an MS Visio page after clicking the Recording button (see Figure 7.10). Then, MLCBD records all the actions and behaviors of the domain expert, and then stores the unique symbols into the *Default* stencil to represent the concrete syntax. Figure 7.22 illustrates two network models demonstrated by domain experts using MLCBD in Visio.

Figure 7.22(a) depicts the demonstration of the top-layer network model. A symbol *Folder* is a special symbol representing a container. If *Folder* is dragged on to the Visio canvas area, a new page named *Folder* is created.



(a) Demonstration of Top-layer Network Model



(a) Demonstration of Sub-Network Layer Model

Figure 7.22 Demonstration of the Network Model in MLCBD

In addition, if the *Folder* name is changed, the name of the newly created page is also changed. In Figure 7.22(a), a new page *Group* is located beside the Default. A *Folder* is added and then renamed from *Folder* to *Group*. In addition, a new stencil named *Group* is created to manage the symbols that are required to model the sub-network domain (see Figure 7.22(b)).

Figure 7.23 shows some examples of Model Space Exploration during network diagramming tool creation. All three model instances are negative models, which do not conform to the inferred metamodel and static constraints. During Model Space Exploration, a domain expert should respond either *Yes* or *No* without guided information when model instances are presented.

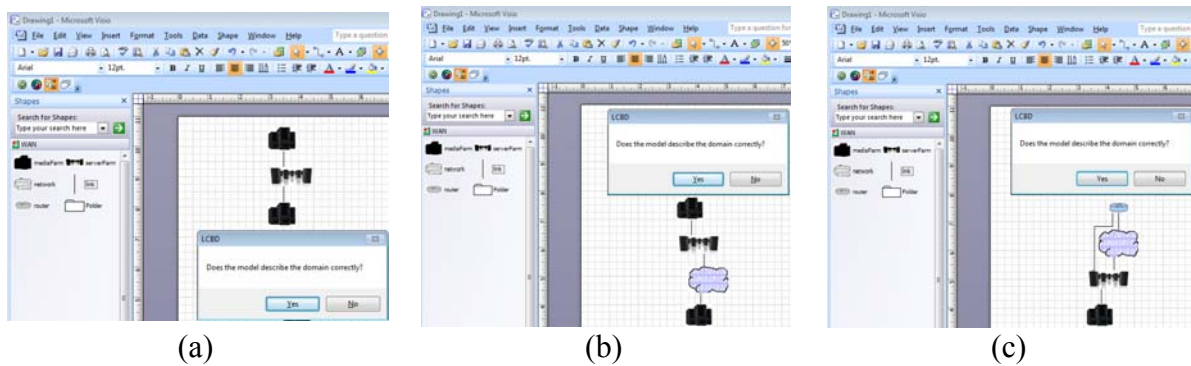
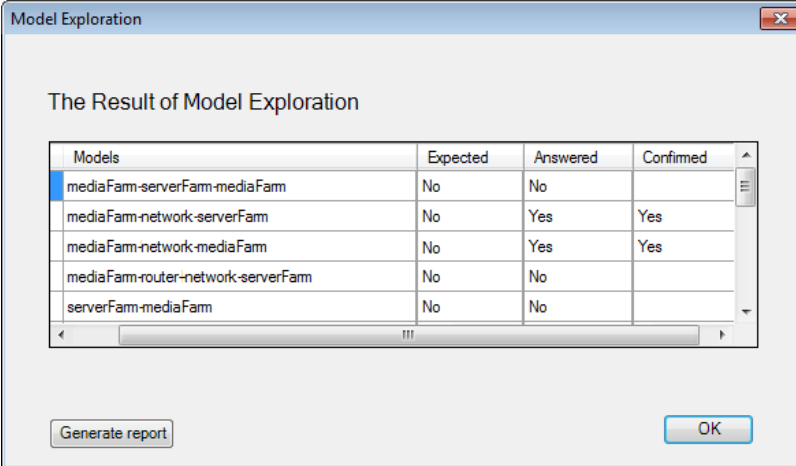


Figure 7.23 Examples of Model Space Exploration

If a domain expert responds *Yes* to one or more of the example models shown in Figure 7.23, that implies those models correctly describe notions of network modeling even though they are instantiated from a negative graph and do not conform to the inferred metamodel and static constraints. Based on the collective responses from the domain expert, the Model Space Exploration presents models, which may contain conflicted feedback, to the domain expert again to confirm whether the expert responded to the conflicted answer mistakenly or not.

After all conflicts are confirmed by domain experts, the result of Model Space Exploration is summarized as shown in Figure 7.24. The summary includes categories for *Models*, *Expected*, *Answered*, and *Confirmed*. Column *Models* lists all model elements used for a model instantiation. Column *Expected* specifies the expected answer of the model instance. To specify the expected answer, each model instance is checked against the inferred metamodel and static constraints. If a model instance conforms to the inferred metamodel and static constraints, *Expected* is set to *Yes*. Otherwise, it is set to *No*. Column *Answered* is the response from the domain expert when the model instance is presented, and column *Confirmed* is the decision of the domain expert for the conflicted model instances.



The screenshot shows a window titled "Model Exploration" with a subtitle "The Result of Model Exploration". Inside the window is a table with four columns: "Models", "Expected", "Answered", and "Confirmed". The table contains five rows of model instances. The first row is highlighted in blue. Below the table, there are two buttons: "Generate report" and "OK".

Models	Expected	Answered	Confirmed
mediaFam-serverFam-mediaFam	No	No	
mediaFam-network-serverFam	No	Yes	Yes
mediaFam-network-mediaFam	No	Yes	Yes
mediaFam-router-network-serverFam	No	No	
serverFam-mediaFam	No	No	

Figure 7.24 Summary of Model Space Exploration

The first model instance is a negative model because two *Media Servers* are connected with a *Server Farm* without a *Router*, which is not allowed. The second and third model instances are also negative models because *Media Servers* and *Server Farms* are connected without a *Router*. However, they were answered and confirmed *Yes* by the domain expert. This

means that the two model instances illustrate notions of the network model correctly, but were not initially inferred as correct from the information provided in the demonstration. The fourth case is similar to the second and third cases regarding confirmation of an expected negative model. The information about these models from the domain expert interaction is passed back to the Intermediate Model Space to update the previously inferred metamodel and static constraints.

#### 7.2.4 *Comparison between MLCBD and GME*

The development of a network diagramming tool is briefly described in previous sections. Section 7.2.2 presents a network diagramming language based on GME and Section 7.2.3 describes the same language based on MLCBD. In the following, we summarize the advantages and disadvantages of MLCBD-based language development by comparing it to the GME-based approach.

- *Complexity of development:* Similar to GMF, which is used to develop the FSM modeling language in Section 7.1, a domain expert must invest effort and time to learn important metamodeling notions in order to develop a DSML using GME. For example, atom, model, paradigm, and aspect are unique concepts in GME that enrich the syntax and semantics of a defined DSML. In GME, at least one *Model* should be defined in the metamodel to manage the entire metamodel elements. For this reason, designers tend to forget to link *Model* and other atoms with an aggregation relationship when they focus on domain entities and their structural relationships. The missing connections between model and atoms cause errors

when generating the language environment, which often takes time and effort to resolve.

- *Completeness of graphical representation:* Both tools support a set of predefined symbols to represent notions of a domain. In addition, both tools also support the use of image files for graphical completeness. However, as shown in Figure 7.18, GME can only point to an image file in an atom's attribute and binds the file at build time. One advantage of this approach is that concrete syntax can be changed by copying the image file to a folder where GME is directed. However, if the image file is replaced with the wrong file mistakenly, the error cannot be detected until domain experts create a new model using the DSML.
- *Analysis and Debug Capability:* When a language designer executes a model interpreter, the model interpreter checks the syntactic and semantics errors of the metamodel. However, even though GME can analyze the metamodel to check its completeness and correctness using a model interpreter, a language designer must have profound knowledge and experience to resolve the errors because GME cannot directly locate the problem to specific metamodel elements that cause those errors. Thus, the language designer must be able to analyze errors and find appropriate resolutions within the specific tooling features in GME.
- *Language Evolution:* GME supports backward compatibility by applying its own version control mechanism. When the metamodel is changed or evolved, GME checks between the versions of the past and current models and asks the domain expert whether to upgrade to a new metamodel. The overall evolution process is relatively easy in GME.

### 7.3 Summary

In this chapter, MLCBD was applied to develop two DSMLs, a FSM and network diagramming DSML. GMF and GME, which are well-known DSML metamodeling environments, were used to develop the same FSM and network diagramming DSML in order to evaluate the comparative capabilities of MLCBD.

Table 7.1 provides a quick overview of the strengths and weakness of each approach.

Table 7.1 Overview of Comparison

		GMF	GME	MLCBD
Complexity of Development		High	High	Low
Completeness of the Graphical Representation		High	High	High
Analysis and Debug	Metamodel	Support	Support	N/A
	Model	N/A	N/A	Model Space Exploration
Language Evolution		Medium	High	Support

MLCBD required less effort to develop a DSML than GMF and GME because the MLCBD framework does not require language development expertise such as syntax definition and semantics specification. MLCBD needs only domain knowledge (or a set of domain model examples) to develop a DSML. However, GMF needs modeling language development expertise

to define and manage four different models (e.g., domain, graphical, tooling, and mapping) as well as domain knowledge. Similarly, GME also needs knowledge and experience about metamodeling.

All three approaches offer a high-level of graphical representation completeness. However, GMF needs to use a menu-driven approach to redefine the graphical model, and GME may need to use a decorator mechanism to support complex graphical representation.

GMF and GME have a feature to analyze and debug (meta) models before generating the language environment. However, to resolve errors, software engineers should have substantial knowledge about either GMF or GME. On the contrary, MLCBD does support model analysis and debug features because it targets the generation of syntactic and semantics elements of a DSML from a set of domain model examples. Model Space Exploration is used to validate the correctness of the inferred metamodel and static constraints by presenting the user with a set of model instances generated from the inferred metamodel and static constraints. Model Space Exploration is also used to find the missing notions of a domain, which are accidentally not demonstrated domain experts.

Finally, language evolution is the weakest area of MLCBD. When change requests are submitted, GMF and GME can evolve a DSML by applying the change request to (meta) models and rebuilding the (meta) models. However, MLCBD requires iteration of the same DSML development process (e.g., demonstration of a domain, inferring metamodel and semantics, and model space exploration) to apply change requests.

## CHAPTER 8

### FUTURE WORK

This chapter outlines future research directions in the area of demonstration-based language development. To further enhance the expressiveness and functionality of MLCBD, several new features are proposed that may assist domain experts in describing precise and complete DSMLs. Section 8.1 lists several features that may improve the Intermediate Design Space capability, and Section 8.2 describes extensions of Model Space Exploration. Finally, formalization and tool support are described as improvements of the Syntax Map in Section 8.3.

#### 8.1 Enhancements to the Intermediate Design Space Capability

This section describes extensions to the capabilities of the Intermediate Design Space, which plays a vital role for DSML creation from a set of domain model examples. A set of model examples are transformed into a set of attributed graphs and then passed into the inference engine to generate a metamodel and the static constraints representing the envisioned DSML. Some domain-independent models are created from Metamodel Design Patterns to complement the lack of a complete set of domain model examples. This section describes extensions of metamodel design patterns and the semantic inference process in order to widen the application areas of MLCBD.

### *8.1.1 Improve Metamodel Design Patterns*

Design patterns have been widely adopted to promote software design reuse because they reflect the experience and knowledge of designers who have successfully solved recurring problems in different domains. Design Patterns were adopted in MLCBD in order to improve the quality of a metamodel. Specifically, metamodel design patterns guide the metamodel inference process by complementing the lack of domain model examples that can be provided by the domain expert. Thus, the more metamodel design patterns that can be identified from existing metamodels can help to guide the metamodel inference in order to build DSMLs that are closer to the notions of the domain.

In addition, further research is needed for the formal specification of metamodel design patterns. Similar to traditional software design patterns, metamodel design patterns can be applied to encourage the reuse of metamodel design. For example, metamodel design patterns can be used to create a new metamodel by composing them instead of designing each metamodel from scratch. Lédeczi et al. [Lédeczi et al., 2001a; Lédeczi et al., 2001] proposed metamodel composition to build a new language by applying composition rules to two existing metamodels. Their approach was implemented successfully in GME. Similarly, if metamodel design patterns are formally specified in a similar manner as formal component specification, then a metamodel can be created by applying simple composition (or weaving) rules to metamodel design patterns.

### 8.1.2 *Consideration about Dynamic Semantics*

Semantics describe the meaning of a language, and syntax is concerned with the form of a language's expression. Semantics, thus, plays the role of a bridge between the concepts of the language and the representations used to express the domain expert's intention. Specifying the semantics of a language using formal techniques can often be tedious and error-prone. In addition, formal specification of behavioral semantics requires much time and effort for even well-trained computer scientists. Currently, the MLCBD framework captures only static semantics, specifically static constraints, that are inferred when a domain expert demonstrates a set of domain model examples. The static constraints are associated to the corresponding metamodel and are used to verify if any structural pattern violation occurred when the domain expert demonstrated an example model. However, to extend MLCBD's features, an ability to capture dynamic semantics is needed from either a set of domain model examples or from a series of actions produced while a domain expert operationally demonstrates the concepts in their domain.

## 8.2 Enhancements to the Model Space Exploration Functionality

Model Space Exploration provides a set of models instantiated from the inferred metamodel with static constraints that can be used to verify and improve previously inferred DSML constructs. There are several limitations in the current approach and this section points out these issues as areas for future work to improve design space exploration within the MLCBD context.

### 8.2.1 *Enhance Model Space Exploration Algorithm*

One improvement that can be investigated further to improve Model Space Exploration is an approach to reduce significantly the traversed model space. Because the model space is created based on the graph search algorithm, some of the short paths actually could be sub-paths of the long paths. For example, model instances, shown earlier in Table 4.5(a) and (c), consist of two modeling elements, which are adjacent to each other. However, these two model instances also can be obtained from the model in Table 4.5(b) by breaking the model into two sub-paths. Therefore, when presenting these three model instances to domain experts, the expert may lose attention easily because the model instances are too trivial and may cause the domain expert to lose focus. Thus, the algorithm needs to consider path reduction in order to minimize the production of model instances that are identical in terms of semantics.

The adaptation of graph transformation is another option for improving the algorithm of Model Space Exploration. As mentioned in Chapter 2, the model transformation technique is a key technology of MDE. The graph grammar and transformations have been recognized as promising techniques for specifying complex transformation. For example, the ideas of the Graph Rewriting and Transformation Language (GReAT) [Agrawal, 2003; Karsai et al., 2003; Balasubramanian et al., 2006-a] can be applied to model space creation. GReAT is a graphical model transformation language that can be used to specify graph transformations between DSMLs. GReAT consists of a Pattern Specification Language, Graph Rewriting/Transformation Language, and Control Flow Language. The Pattern Specification Language specifies patterns of

objects and their links over graphs, where the vertices and edges map to specific classes and associations. In addition, a pattern matching algorithm is used to represent complex graphs concisely. The Graph Rewriting/Transformation language defines graph transformation steps by embellishing pattern graphs as well as specifying pre-conditions and post-conditions. The Control Flow Language is used to define the control structures of rules and provides features (e.g., rule sequencing, modularization and branching) in order to process control structures hierarchically.

The model exploration algorithm can be improved by adopting the ideas of the three major parts of GReAT (e.g., pattern specification, graph rewriting/transformation, and control flow) to handle a set of domain model examples used to infer a metamodel and static constraints.

### *8.2.2 Improve Layout Management*

To arrange the models instantiated for Model Space Exploration, the implementation uses the layout management algorithm provided by Microsoft Visio. Because Microsoft Visio does not provide broader APIs to manage shape layout, programmers cannot fully control the layout of modeling elements, and a clumsy layout may hinder the domain experts from understanding the model instances.

The most commonly used approach to arrange the layout of models automatically is to develop a language-specific algorithm. A number of modeling tools (e.g., GMF, GME, and MetaCase+ [MetaCase+, 2011]) provide an automatic layout feature in their model editors using their own fixed algorithms. The algorithms rearrange the layout of the models and make models more readable by avoiding the overlaps of model elements and connections. However, most of these algorithms do not consider the implicit semantics of the model elements and their

connections. In addition, fixed layout algorithms usually do not consider the underlying mental map of domain users [Misue et al., 1995].

Thus, a new layout management algorithm is needed to address these issues. For example, we may apply By-Demonstration techniques to address the ignorance of the underlying mental map. Currently, a By-Demonstration technique is applied to identify the concrete syntax by capturing a series of domain expert actions, but it can be extended for capturing the layout of each domain model example [Sun et al., 2011].

### 8.3 Improvements for the Syntax Map

The Syntax Map was introduced to assist in defining the requirements of DSMLs. The Syntax Map mainly uses the graphical representations in order to describe the syntactic and semantics requirements of a DSML visually. In addition, the Syntax Map assists domain experts in describing the requirements of the syntax according to the anticipated usages. Thus, the two characteristics of the Syntax Map (i.e., scenario-based and graphical representation-based requirements specification) help domain experts to describe the requirements of DSMLs precisely and completely. However, the Syntax Map needs to be improved to model additional DSML requirements. This section identifies these needs issues as future work to improve the Syntax Map.

#### 8.3.1 *Formalization of Syntax Map*

The Syntax Map does not apply a formal technique to specify each requirements modeling element. The specification of modeling elements is provided in English prose. Thus, the modeling elements can be understood differently and inconsistently by requirements

modelers. For example, the concept of containment could be specified more precisely using formal specification techniques.

### *8.3.2 Tool Support*

Additional tool support could encourage the use of the Syntax Map for DSML requirement modeling and management. If a DSML does not have complex structural patterns, the Syntax Map can be drawn using basic diagramming tools. However, if a DSML has many constructs that are deeply related to each other, domain users may be challenged in managing the Syntax Map without tool support. Thus, tool support is necessary to deal with the volume and complexity of anticipated DSML requirements.

## CHAPTER 9

### CONCLUSION

MDE has been proposed to address the challenges in software development by raising the level of abstraction. DSMLs and model transformation are key technologies enabling MDE. Unlike GPMLs, DSMLs offers precise and concise syntax and semantics to satisfy specific domain needs. Thus, DSMLs can improve productivity and quality while minimizing the learning curve [Kelly and Tolvanen, 2008]. However, DSML development requires much time and effort.

The overall goal of the research described in this dissertation was to provide a systematic and end-user (or domain expert) centered approach for building DSMLs. The key contributions include: 1) applying a demonstration-based approach to capture domain knowledge, specifically capturing concrete syntax, 2) designing and implementing an inference engine to generate a metamodel and static semantics from domain model examples, which are demonstrated by domain experts, 3) applying Model Space Exploration to investigate the correctness of the inferred metamodel and static constraints, as well as identify missing elements of the domain that need to be present in the inferred DSML. In addition, the concept of a Syntax Map was introduced to model the requirements of a DSML semi-formally using a graphical representation. The core contributions are summarized in the following sections of this chapter.

## 9.1 Syntax Map

Requirements describe the features of a system as demanded by end-users. A good set of requirements are critical for project success and play a primary role for communication among stakeholders. In addition, they are used to measure software quality. By its nature, elaborating a set of complete and precise requirements is challenging. Many researchers and practitioners have proposed and developed various approaches and tools to identify and manage requirements formally and systematically - examples include Use Case diagrams, Behavior Trees, CCS and CSP for modeling requirements (semi)-formally. However, little attention has been given to the issue of specifying and managing the requirements of DSMLs. As part of our contribution toward specifying DSML requirements, the concept of the Syntax Map is introduced in this dissertation. The Syntax Map is intended to model the requirements of graphical DSMLs and offers its own language to model the syntactic and semantics requirements of a DSML visually. With a Syntax Map, the requirements of the syntax are modeled by placing and linking the graphical symbols according to the usage of the syntax by domain experts. The goals of the Syntax Map are: 1) to minimize miscommunication between end users and programming language development experts, 2) to reason about correctness and completeness of the DSML requirements.

## 9.2 Intermediate Design Space

DSMLs can express a specific domain more precisely and concisely than GPMLs and they offer several advantages such as productivity improvement and shorter learning time [Kelly

and Tolvanen, 2008]. However, DSMLs can be a challenge to design using current practice because: 1) domain experts may need programming (or modeling) language development expertise, as well as domain knowledge, 2) lack of tool support hampers the progress of DSML development and can be an error prone process, and 3) the static semantics of a DSML is often hard to define by a domain expert.

The MLCBD approach described in this dissertation addresses the challenges of DSML development by providing an active role for the domain expert. The Intermediate Design Space is a core part of MLCBD and allows domain experts who do not have programming language development expertise to create their own DSMLs by themselves. To build a DSML, domain experts need to demonstrate a set of model examples using the MLCBD tool, which is a plugin to Microsoft Visio. The MLCBD tool captures the core parts of the concrete syntax as a domain expert demonstrates concepts from a specific domain. A set of model examples provided by the domain expert are transformed into a set of attribute graphs and then passed into the metamodel inference engine to infer a metamodel and its static constraints. Model instances are generated from metamodel design patterns, which reflect the experience and knowledge of designers who have successfully solved the recurring problems in metamodeling. The metamodel design patterns are provided to the metamodel inference engine as supplementary inputs.

Using MLCBD, domain experts can build their own DSML environment without the help of programming language experts. One current limitation of DSML creation with MLCBD is the level of detail that can be captured with respect to the semantics of the DSML. Although semantics offers a bridge between the concepts of the language and the representations used to express the concepts, it is difficult to capture and specify the behavioral semantics from a set of static model examples. Although some static semantics can be inferred with MLCBD, further

research is needed to extend the level of support to allow domain experts to express behavioral semantics more deeply.

### 9.3 Model Space Exploration

Model Space Exploration is used in MLCBD to improve the correctness of the inferred metamodel and its static constraints. Although domain experts have profound knowledge and deep experiences in their domain, modeling every possible domain concept may not be possible. Because of this, the metamodel and static constraints inferred from a set of user-supplied model examples may not have all of required syntax and semantics to model the domain. This has a detrimental effect on the completeness and correctness of the inferred DSML.

To address the need to improve the completeness of the demonstration-based approach of MLCBD, Model Space Exploration was investigated. In Model Space Exploration, a set of domain models is instantiated from the inferred metamodel and static constraints. The set of domain models includes both positive and negative models. The generated model examples are presented randomly to the domain expert, who is asked to confirm whether each example correctly represents an example in the envisioned DSML. The domain expert's feedback is sent back to the Intermediate Design Space inference engine to update the inferred metamodel and static constraints. Thus, Model Space Exploration allows domain users to verify the correctness of the inferred metamodel and static constraints by exploring a set of domain models that are generated from the current representation of the DSML from past model examples.

## 9.4 Dissertation Conclusion

Although DSMLs offer many benefits, developing a DSML is challenging for domain experts who have in-depth domain knowledge, but do not have programming language development expertise. In addition, the lack of tool support hampers the development and usage of DSMLs in everyday software development activities.

To address these issues, the research described in this dissertation provides a systematic and end-user (or domain expert) centered approach for building DSMLs. The approach consists of two major activities: Intermediate Design Space and Model Space Exploration. The Intermediate Design Space offers flexible modeling environments where domain experts can demonstrate notions of a domain with a graphical notation. In addition, the Intermediate Design Space infers a metamodel and its static constraints based on a set of domain model examples, which are demonstrated by domain experts. Metamodel design patterns were also investigated to assist with the inference. Model Space Exploration assists domain experts in reasoning about the correctness of the inferred metamodel and finding language concepts that were missed during the original demonstration. By iterating between the Intermediate Design Space and Model Space Exploration, domain experts can define their own DSML.

In addition, the Syntax Map helps domain experts to describe the requirements of a DSML semi-formally. A Syntax Map can visualize what notions of a DSML (i.e., abstract syntax) are required for describing a domain and how those notions are related with concrete entities (i.e., concrete syntax).

## LIST OF REFERENCES

- [Abrial, 2005] Jean-Raymond Abrial, *The B-Book: Assigning Programs to Meanings*, Cambridge University Press, 2005.
- [Agrawal et al., 2003] Aditya Agrawal, Gábor Karsai, and Ákos Lédeczi, “An End-to-End Domain-Driven Software Development Framework,” *In Proceedings of Object-Oriented Programming, Systems, Languages, and Applications*, Anaheim, CA, October 2003, pages 8-15.
- [Aichenig, 1999] Bernhard K. Aichenig, “Automated Black-Box Testing with Abstract VDM Oracles,” *In Proceedings of the 18th International Conference on Computer Safety, Reliability, and Security*, September 1999, Toulouse, France, pages 250-259.
- [Albert et al, 2003] Manoli Albert, Vicente Pelechano, Joan Fons, Marta Ruiz, and Oscar Pastor, “Implementing UML Association, Aggregation, and Composition: A Particular Interpretation Based on a Multidimensional Framework,” *In Proceedings of the International Conference on Advanced Information Systems Engineering*, Klagenfurt/Velden, Austria, June 2003, pages 143-158.
- [Amyot and Logrippo, 2000] Daniel Amyot and Luigi Logrippo, “Use Case Maps and Lotos for the Prototyping and Validation of a Mobile Group Call System,” *Computer Communications*, vol. 23, no. 12, July 2000, pages 1135-1157.
- [Amyot et al., 2006] Daniel Amyot, Hanna Farah, and Jean-François Roy, “Evaluation of Development Tools for Domain-Specific Modeling Languages,” *System Analysis and Modeling: Language Profiles*, 2006, pages 183-197.
- [Andries et al., 1999] Marc Andries, Gregor Engels, Annegret Habel, Berthold Hoffmann, Hans-Jörg Kreowski, Sabine Kuske, Detlef Plump, Andy Schürr, and Gabriele Taentzer, “Graph Transformation for Specification and Programming,” *Journal of Science of Computer Programming*, vol. 34, no. 1, April 1999, pages 1-54.
- [Angluin, 1980] Dana Angluin, “Inductive Inference of Formal Languages from Positive Data,” *Information and Computation/Information and Control*, vol. 45, no. 2, 1980, pages 117-135.
- [Armstrong, 2006] Deborah J. Armstrong, “The Quarks of Object-Oriented Development,” *Communications of the ACM*, vol. 49, no. 2, February 2006, pages 123-128.
- [Atkinson and Kuhne, 2003] Colin Atkinson and Thomas Kuhne, “Model-Driven Development: A Metamodeling Foundation,” *IEEE Software*, vol. 20, no. 5, May 2003, pages 36-41.

- [Backus 1978] John Backus, “Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs,” *Communications of the ACM*, vol. 21, no. 8, August 1978, pages 613-641.
- [Balakrishnan and Reps, 2010] Gogul Balakrishnan and Thomas Reps, “WYSINWYX: What You See Is Not What You eXecute,” *ACM Transactions on Programming Languages and Systems*, vol. 32, no. 6, Article 23, August 2010, pages 23:1-23:84.
- [Balasubramanian et al., 2006] Daniel Balasubramanian, Anantha Narayanan, Chris van Buskirk, and Gabor Karsai, “The Graph Rewriting and Transformation Language: GReAT,” *Electronic Communication of the European Association of Software Science and Technology*, vol. 1, 2006, 8 pages.
- [Barbier et al., 2003] Franck Barbier, Brian Henderson-Sellers, Annig Le Parc-Lacayrelle, and Jean-Michel Bruel, “Formalization of the Whole-Part Relationship in the Unified Modeling Language,” *IEEE Transactions on Software Engineering*, vol. 29, no. 5, May 2003, pages 459-470.
- [Bass et al., 2012] Len Bass, Paul Clements, and Rick Kazman, *Software Architecture in Practice (3rd ed.)*, Addison-Wesley Professional, 2012.
- [Bentley et al., 1987] Jon L. Bentley, Lynn W. Jelinski, and Brian W. Kernighan, “Chem: A Program for Phototypesetting Chemical Structure Diagrams,” *Computers and Chemistry*, vol. 11, no. 4, 1987, pages 281-297.
- [Bergstra and Klop, 1982] Jan A. Bergstra and Jan Willem Klop, “Algebraic Specifications for Parameterized Data Types with Minimal Parameter and Target Algebras,” *In Proceedings of the 9th Colloquium on Automata, Languages and Programming*, Aarhus, Denmark, July 1982, pages 23-34.
- [Berwick and Pilato, 1987] Robert C. Berwick and Sam Pilato, “Learning Syntax by Automata Induction,” *Machine Learning*, vol. 2, no. 1, March 1987, pages 9-38.
- [Bézivin, 2005] Jean Bézivin, “Model Driven Engineering: An Emerging Technical Space,” *Generative and Transformational Techniques in Software Engineering*, Braga, Portugal, July 2005, pages 36-64.
- [Bjørner and Jones, 1978] Dines Bjørner and Cliff B. Jones, *The Vienna Development Method: The Meta-Language*, Springer, 1978.
- [Blickle et al, 1998] Tobias Blickle, Jürgen Teich, and Lothar Thiele, “System-Level Synthesis Using Evolutionary Algorithms,” *Design Automation for Embedded Systems*, vol. 3, no. 1, 1998, pages 23-58.
- [Blostein and Schürr, 1999] Dorothea Blostein and Andy Schürr, “Computing with Graphs and Graph Transformations,” *Software: Practice and Experience*, vol. 29, no. 3, March 1999, pages 197-217.
- [Boehm, 1988] Barry W. Boehm, “A Spiral Model of Software Development and Enhancement,” *IEEE Computer*, vol. 21, no. 5, May 1988, pages 61-72.

- [Bondé et al., 2005] Lossan Bondé, Cédric Dumoulin, and Jean-Luc Dekeyser, “Metamodels and MDA Transformations for Embedded Systems,” *Advances in Design and Specification Languages for SoCs*, September 2005, pages 89-105.
- [Booch, 1997] Grady Booch, *Object-Oriented Analysis and Design with Applications*, Addison-Wesley, 1997.
- [Brand et al., 1996] Mark Brand, Arie van Deursen, Paul Klint, A. S. Klusener, and Emma Meulen, “Industrial Applications of ASF+SDF,” *Algebraic Methodology and Software Technology*, 1996, pages 9-18.
- [Brooks, 1987] Frederick Brooks, “No Silver Bullet - Essence and Accident in Software Engineering,” *IEEE Computer*, vol. 20, no. 4, April 1987, pages 10-19.
- [Bruns, 1997] Glenn Bruns, *Distributed Systems Analysis with CCS*, Prentice Hall, 1997.
- [Buchmann, 2012] Thomas Buchmann, “Towards Tool Support for Agile Modeling: Sketching Equals Modeling,” *In Proceedings of Extreme Modelling Workshop*, Innsbruck, Austria, October 2012.
- [Budinsky et al., 2004] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, and Timothy J. Grose, *Eclipse Modeling Framework*, Addison-Wesley, 2004.
- [Buhr and Casselman, 1995] R. J. A. Buhr and R. S. Casselman, *Use Case Maps for Object-Oriented Systems*, Prentice Hall, 1995.
- [Burnett et al., 2004] Margaret Burnett, Curtis Cook, and Gregg Rothermel, “End-user Software Engineering,” *Communications of the ACM*, vol. 47, no. 9, January 2004, pages 53-58.
- [Buschmann et al., 1996] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal, *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*, Wiley, 1996.
- [Chen et al., 2008] Qi Chen, John Grundy, and John Hosking, “SUMLOW: Early Design-Stage Sketching of UML Diagrams on an E-Whiteboard,” *Software Practice and Experience*, vol. 38, no. 9, July 2008, pages 961-994.
- [Chen et al., 2009] Kai Chen, Joseph Porter, Janos Sztipanovits, and Sandeep Neema, “Compositional Specification of Behavioral Semantics for Domain-Specific Modeling Languages,” *International Journal of Semantic Computing*, vol. 3, no. 1, March 2009, pages 31-56.
- [Cheng and Atlee, 2007] Betty H. C. Cheng and Joanne M. Atlee, “Research Directions in Requirements Engineering,” *In Proceedings of the ICSE 2007 Workshop on the Future of Software Engineering*, Minneapolis, MN, May 2007, pages 285-303.
- [Cho et al., 2011] Hyun Cho, Yu Sun, Jeff Gray, and Jules White, “Key Challenges for Modeling Language Creation By Demonstration,” *In Proceedings of the ICSE 2011 Workshop on Flexible Modeling Tools*, Honolulu, HI, May 2011.

- [Cho and Gray, 2011] Hyun Cho and Jeff Gray, “Design Patterns for Metamodels,” *11th Workshop on Domain-Specific Modeling*, Portland, OR, October 2011, pages 25-32.
- [Cho et al., 2012] Hyun Cho, Jeff Gray, and Eugene Syriani, “Creating Visual Domain-Specific Modeling Languages from End-User Demonstration,” *In Proceedings of the ICSE 2012 Workshop on Modeling in Software*, Zurich, Switzerland, June 2012, pages 22-28.
- [Cho et al., 2012] Hyun Cho, Jeff Gray, and Eugene Syriani, “Syntax Map: A Modeling Language for Capturing Requirements of Graphical DSML,” *The 19th Asia-Pacific Software Engineering Conference*, Hong Kong, Hong Kong, December 2012, pages 705-708.
- [Claßen, 1989] Ingo Claßen, “Revised ACT ONE: Categorical Constructions for an Algebraic Specification Language,” *In Proceeding of Workshop on Categorical Methods in Computer Science with Aspects from Topology*, 1989, pages 124-141.
- [Claßen et al., 1993] Ingo Claßen, Hartmut Ehrig, and Dietmar Wolz, *Algebraic Specification Techniques and Tools for Software Development*, World Scientific Pub Co. Inc., 1993.
- [Clarke and Baniassad, 2005] Siobhán Clarke and Elisa Baniassad, *Aspect-Oriented Analysis and Design*, Addison-Wesley Professional, 2005.
- [Clements et al., 2010] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Paulo Merson, Robert Nord, and Judith Stafford, *Documenting Software Architectures: Views and Beyond (2nd ed.)*, Addison-Wesley Professional, 2010.
- [Cockburn, 2000] Alistair Cockburn, *Writing Effective Use Cases*, Addison-Wesley Professional, 2000.
- [Consel and Danvy, 1991] Charles Consel and Olivier Danvy, “Static and Dynamic Semantics Processing,” *In Proceedings of the Symposium on Principles of Programming Languages*, Orlando, FL, 1991, pages 14-24.
- [Constantine and Yourdon, 1979] Larry Constantine and Edward Yourdon, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Prentice Hall, 1979.
- [Crosby, 1979] Philip B. Crosby, *Quality is Free: The Art of Making Quality Certain*, Penguin Books, 1979.
- [Czarnecki and Eisenecker, 2000] Krzysztof Czarnecki and Ulrich Eisenecker, *Generative Programming: Methods, Tools, and Applications*, Addison Wesley, 2000.
- [Czarnecki and Helsen, 2006] Krzysztof Czarnecki and Simon Helsen, “Feature-Based Survey of Model Transformation Approaches,” *IBM Systems Journal*, vol. 45, no. 3, 2006, pages 621-645.
- [Cypher, 1993] Allen Cypher, *Watch What I Do: Programming by Demonstration*, MIT Press, 1993.
- [Deursen and Klint, 1998] Arie van Deursen and Paul Klint, “Little Languages: Little Maintenance,” *Journal of Software Maintenance*, vol. 10, no. 2, March 1998, pages 75-92.

- [Deursen et al., 2000] Arie van Deursen, Paul Klint, and Joost Visser, "Domain-Specific Languages: An Annotated Bibliography," *ACM SIGPLAN Notices*, vol. 35, no. 6, June 2000, pages 26-36.
- [DeMarco, 1979] Tom DeMarco, *Structured Analysis and System Specification*, Prentice Hall, 1979.
- [Dijkstra, 1972] Edsger W. Dijkstra, *Structured Programming*, Academic Press Ltd., 1972.
- [Dijkstra, 1976] Edsger W. Dijkstra, *A Discipline of Programming*, Prentice Hall, 1976.
- [Dimitrov, 2010] Vladimir Dimitrov, "Finite State Automata Semantics in Communicating Sequential Processes," *Labour On Scientific*, vol. 49, no. 6.1, 2010, pages 66-71.
- [Dromey, 2003] Geoff R. Dromey, "From Requirements to Design: Formalizing the Key Steps," *In Proceedings of First International Conference on Software Engineering and Formal Methods*, Brisbane, Australia, September 2003, pages 2-11.
- [Ehrig et al., 2004] Hartmut Ehrig, Ulrike Prange, and Gabriele Taentzer, "Fundamental Theory for Typed Attributed Graphs and Graph Transformation," *In Proceeding of 2nd International Conference on Graph Transformation*, Roma, Italy, September 2004, pages 161-177.
- [Eker et al., 2003] Johan Eker, Joern W. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Stephen Neuendorffer, Sonia Sachs, and Yuhong Xiong, "Taming Heterogeneity-The Ptolemy Approach," *Proceedings of the IEEE*, vol. 91, no. 1, January 2003, pages 127-144.
- [Elaasar et al., 2006] Maged Elaasar, Lionel C. Briand, and Yvan Labiche, "A Metamodeling Approach to Pattern Specification and Detection," *Technical Report SCE-06-08*, Carleton University, March 2006.
- [Elrad et al., 2002] Tzilla Elrad, Omar Aldawud, and Atef Bader, "Aspect-Oriented Modeling: Bridging the Gap between Implementation and Design," *International Conference on Generative Programming and Component Engineering*, Pittsburgh, PA, October 2002, pages 189-201.
- [Emerson and Sztipanovits, 2006] Matthew Emerson and Janos Sztipanovits, "Techniques for Metamodel Composition," *In The 6th OOPSLA Workshop on Domain-Specific Modeling*, Portland, OR, October 2006, pages 123-139.
- [Erbas et al., 2003] Cagkan Erbas, Selin C. Erbas, and Andy D. Pimentel, "A Multiobjective Optimization Model for Exploring Multiprocessor Mappings of Process Networks," *In Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, Newport Beach, CA, October 2003, pages 182-187.
- [Erwig, 1998] Martin Erwig, "Abstract Syntax and Semantics of Visual Languages," *Journal of Visual Languages and Computing*, vol. 9, no. 5, 1998, pages 461-483.
- [Fantechi et al., 2002] Alessandro Fantechi, Stefania Gnesi, Giuseppe Lami, and Alessandro Maccari, "Application of Linguistic Techniques for Use Case Analysis," *In Proceedings of the 10th Anniversary IEEE Joint International Conference on Requirements Engineering*, September 2002, Essen, Germany, pages 157-164.

- [Favre, 2004] Jean-Marie Favre, “CacOphoNy: Metamodel Driven Architecture Reconstruction,” *In Proceedings of the 11th Working Conference on Reverse Engineering*, Delft, The Netherlands, 2004, pages 204-213.
- [Fey, 1988] Werner Fey, “Pragmatics, Concepts, Syntax, Semantics, and Correctness Notions of ACT TWO: An Algebraic Module Specification and Interconnection Language,” *Ph.D. Thesis, Report 88/26*, Technische Universität Berlin, 1988.
- [Flood and Carson, 1993] Robert L. Flood and Ewart R. Carson, *Dealing with Complexity: An Introduction to the Theory and Applications of Systems Science*, Plenum Press, 1993.
- [Fokkink, 2009] Wan Fokkink, “Process Algebra: An Algebraic Theory of Concurrency,” *In Proceedings of the 3rd International Conference on Algebraic Informatics*, Thessaloniki, Greece, May 2009, pages 47-77.
- [Fowler, 2003] Martin Fowler, *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, Addison-Wesley Professional, 2003.
- [France and Rumpe, 2007] Robert France and Bernhard Rumpe, “Model-Driven Development of Complex Software: A Research Roadmap,” *In Proceedings of the ICSE 2007 Workshop on the Future of Software Engineering*, Minneapolis, MN, May 2007, pages 37-54.
- [Fu and Booth, 1986] King-Sun Fu and Taylor L. Booth, “Grammatical Inference: Introduction and Survey-Part I,” *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-5, no. 1, January 1975, pages 95-111.
- [Gamma et al., 1995] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1995.
- [Gane and Sarson, 1979] Chris Gane and Trish Sarson, *Structured Systems Analysis: Tools and Techniques*, Prentice Hall, 1979.
- [Garlan and Shaw, 1994] David Garlan and Mary Shaw, “An Introduction to Software Architecture,” *Technical Report CMU/SEI-94-TR-021*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1994.
- [Gerber et al., 2002] Anna Gerber, Michael Lawley, Kerry Raymond, Jim Steel, and Andrew Wood, “Transformation: The Missing Link of MDA,” *In First International Conference of Graph Transformation*, Barcelona, Spain, October, 2002, pages 90 -105.
- [Goan et al., 1996] Terrance Goan, Nels Benson, and Oren Etzioni, “A Grammar Inference Algorithm for the World Wide Web,” *In Proceedings of AAAI Spring Symposium on Machine Learning in Information Access*, Stanford, CA, 1996, pages 41-48.
- [Gold, 1967] E. Mark Gold, “Language Identification in the Limit,” *Information and Control*, vol. 10, no. 6, May 1967, pages 447-474.
- [Goodman, 1968] Nelson Goodman, *Languages of Art: An Approach to a Theory of Symbols*, Bobbs-Merrill Co., 1968.

- [Gray et al., 2007] Jeff Gray, Juha-Pekka Tolvanen, Steven Kelly, Aniruddha Gokhale, Sandeep Neema, and Jonathan Sprinkle, *Handbook of Dynamic System Modeling*, CRC Press, 2007.
- [Gries, 2004] Matthias Gries, “Methods for Evaluating and Covering the Design Space During Early Design Development,” *Integration, the VLSI Journal*, vol. 38, no. 2, December 2004, pages 131-183.
- [Grönniger et al., 2007] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel, “Text-Based Modeling,” *In 4th International Workshop on Software Language Engineering*, Nashville, TN, September/October 2007.
- [Grose et al., 2002] Timothy J. Grose, Gary C. Doney, and Stephen A. Brodsky, *Mastering XMI: Java Programming with XMI, XML, and UML*, Wiley, 2002.
- [Grunske et al., 2005] Lars Grunske, Peter Lindsay, Nisansala Yatapanage, and Kirsten Winter, “An Automated Failure Mode and Effect Analysis Based on High-Level Design Specification with Behavior Trees,” *In Proceeding of 5th International Conference of Integrated Formal Methods*, Eindhoven, The Netherlands, December 2005, pages 129-149.
- [Grunske et al., 2008] Lars Grunske, Kirsten Winter, and Nisansala Yatapanage, “Defining the Abstract Syntax of Visual Languages with Advanced Graph Grammars-A Case Study Based on Behavior Trees,” *Journal of Visual Languages and Computing*, vol. 19, no. 3, June 2008, pages 343-379.
- [Günther, 2011] Sebastian Günther, “Development of Internal Domain-Specific Languages: Design Principles and Design Patterns,” *In Proceedings of Pattern Languages of Programs*, Portland, OR, October 2011.
- [Hagenbuchner et al., 2009] Markus Hagenbuchner, Alessandro Sperduti, and Ah-Chung Tsoi, “Graph Self-Organizing Maps for Cyclic and Unbounded Graphs,” *Neurocomputing*, vol. 72, no. 7-9, March 2009, pages 1419-1430.
- [Halambi et al., 1999] Ashok Halambi, Peter Grun, Vijay Ganesh, Asheesh Khare, Nikil Dutt, and Alex Nicolau, “EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability,” *In Proceedings of the Conference on Design, Automation, and Test in Europe*, Dresden, Germany, March 1999, pages 485-490.
- [Harel and Rumpe, 2004] David Harel and Bernhard Rumpe, “Meaningful Modeling: What’s the Semantics of “Semantics”?” *IEEE Computer*, vol. 37, no. 10, October 2004, pages 64-72.
- [He et al., 2003] Yong He, Daniel Amyot, and Alan W. Williams, “Synthesizing SDL from Use Case Maps: An Experiment,” *In Proceedings of the 11th International Conference on System Design*, Stuttgart, Germany, July 2003, pages 117-136.
- [Hemingway et al., 2007] Graham Hemingway, Hang Su, Kai Chen, and T. John Koo, “A Semantic Anchoring Infrastructure for the Design of Embedded Systems,” *In Proceedings of the Computer Software and Applications Conference*, vol. 1, Beijing, China, 2007, pages 287-294.

- [Herndon and Berzins, 1988] Robert M. Herndon Jr. and Valdis A. Berzins, "The Realizable Benefits of a Language Prototyping Language," *IEEE Transactions on Software Engineering*, vol. 14, no. 6, June 1988, pages 803-809.
- [Higuera, 2005] Colin De La Higuera, "A Bibliographical Study of Grammatical Inference," *Pattern Recognition*, vol. 38, no. 9, September 2005, pages 1332-1348.
- [Hoare, 1969] Charles Antony Richard Hoare, "An Axiomatic Basis for Computer Programming," *Communications of the ACM*, vol. 12, no. 10, October 1969, pages 576-580.
- [Hoare, 1973] Charles Antony Richard Hoare, "Hints on Programming Language Design," *Technical Report STAN-CS-73-403*, Stanford University, Stanford, CA, October 1973.
- [Hoare, 1978] Charles Antony Richard Hoare, "Communicating Sequential Processes," *Communications of the ACM*, vol. 21, no. 8, August 1978, pages 666-677.
- [Hopcroft et al., 2006] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman, *Introduction to Automata Theory, Languages, and Computation (3rd ed.)*, Prentice Hall, 2006.
- [Hsia et al., 1993] Pei Hsia, Alan Davis, and David Kung, "Status Report: Requirements Engineering," *IEEE Software*, vol. 10, no. 6, November 1993, pages 75-79.
- [Javed et al., 2008] Faizan Javed, Marjan Mernik, Jeff Gray, and Barrett R. Bryant, "MARS: A Metamodel Recovery System Using Grammar Inference," *Information and Software Technology*, vol. 50, no. 9-10, August 2008, pages 948-968.
- [Jouault and Bézivin, 2006] Frédéric Jouault and Jean Bézivin, "KM3: A DSL for Metamodel Specification," *International Conference on Formal Methods for Open Object-based Distributed Systems*, Bologna, Italy, June 2006, pages 171-185.
- [Julliand and Kouchnarenko, 2007] Jacques Julliand and Olga Kouchnarenko, "B 2007: Formal Specification and Development in B," *In Proceedings of 7th International Conference of B Users*, Besançon, France, January 2007.
- [Kaindl et al., 2002] Hermann Kaindl, Sjaak Brinkkemper, Janis A. Bubenko Jr., Barbara Farbey, Sol J. Greenspan, Constance L. Heitmeyer, Julio Cesar Sampaio do Prado Leite, Nancy R. Mead, John Mylopoulos, and Jawed Siddiqi, "Requirements Engineering and Technology Transfer: Obstacles, Incentives, and Improvement Agenda," *Requirements Engineering*, vol. 7, no. 3, 2002, pages 113-123.
- [Kang et al., 1990] Kyo C. Kang, Sholom Cohen, James Hess, William Novak, and Spencer Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," *Technical Report CMU/SEI-90-TR-021*, SEI, Carnegie Mellon University, Pittsburgh, PA, November 1990.
- [Kangas et al., 2006] Tero Kangas, Petri Kukkala, Heikki Orsila, Erno Salminen, Marko Hännikäinen, Timo D. Hämmäläinen, Jouni Riihimäki, and Kimmo Kuusilinna, "UML-Based Multiprocessor SoC Design Framework," *ACM Transactions on Embedded Computing Systems*, vol. 5, no. 2, May 2006, pages 281-320.
- [Karagiannis and Höfferer, 2008] Dimitris Karagiannis and Peter Höfferer, "Metamodeling as an Integration Concept," *Software and Data Technologies*, vol. 10, 2008, pages 37-50.

[Karsai et al., 2004] Gabor Karsai, Miklos Maroti, Akos Lédeczi, Jeff Gray, and Janos Sztipanovits, “Composition and Cloning in Modeling and Meta-Modeling,” *IEEE Transactions on Control Systems Technology*, vol. 12, no. 2, pages 263-278.

[Karsai et al., 2009] Gabor Karsai, Holger Krah, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel, “Design Guidelines for Domain Specific Languages,” *In Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling*, Orlando, FL, October 2009.

[Kelly and Tolvanen, 2000] Steven Kelly and Juha-Pekka Tolvanen, “Visual Domain-Specific Modeling: Benefits and Experiences of Using MetaCASE Tools,” *International Workshop on Model Engineering*, Sophia and Cannes, France, June 2000.

[Kelly and Tolvanen, 2008] Steven Kelly and Juha-Pekka Tolvanen, *Domain-Specific Modeling: Enabling Full Code Generation*, Wiley-IEEE Computer Society Press, 2008.

[Kieburtz et al., 1996] Richard B. Kieburtz, Laura McKinney, Jeffrey M. Bell, James Hook, Alex Kotov, Jeffrey Lewis, Dino P. Oliva, Tim Sheard, Ira Smith, and Lisa Walton, “A Software Engineering Experiment in Software Component Generation,” *In Proceedings of the 18th International Conference on Software Engineering*, Berlin, Germany, March 1996, pages 542-552.

[Kienhuis et al., 1997] Bart Kienhuis, Ed Deprettere, Kees Vissers, and Pieter van der Wolf, “An Approach for Quantitative Analysis of Application-Specific Dataflow Architectures,” *In Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors*, Zurich, Switzerland, July 1997, pages 338-349.

[Kirsopp and Shepperd, 2002] Colin Kirsopp and Martin J. Shepperd, “Making Inferences with Small Numbers of Training Sets,” *Software, IEE Proceedings*, vol. 149, no. 5, pages 123-130.

[Kolovos et al., 2006] Dimitrios S. Kolovos, Richard F. Paige, Tim Kelly, and Fiona A.C. Polack, “Requirements for Domain-Specific Languages,” *In Proceedings of ECOOP Workshop on Domain-Specific Program Development*, Nantes, France, July 2006.

[Kopp et al., 2009] Oliver Kopp, Daniel Martin, Daniel Wutke, and Frank Leymann, “The Difference Between Graph-Based and Block-Structured Business Process Modelling Language,” *Enterprise Modelling and Information Systems*, vol. 4, no. 1, 2009, pages 3-13.

[Krueger, 1992] Charles W. Krueger, “Software Reuse,” *ACM Computing Surveys*, vol. 24, no. 2, June 1992, pages 131-183.

[Kurlander and Feiner, 1993] David Kurlander and Steven Feiner, “Inferring Constraints from Multiple Snapshots,” *ACM Transactions on Graphics*, vol. 12, no. 4, October 1993, pages 277-304.

[Kurtev et al., 2006] Ivan Kurtev, Jean Bezivin, Frederic Jouault, and Patrick Valduriez, “Model-based DSL Frameworks,” *ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages and Applications*, Portland, OR, October 2006, pages 602-616.

- [Larman and Basili, 2003] Craig Larman and Victor R. Basili, "Iterative and Incremental Development: A Brief History," *IEEE Computer*, vol. 36, no. 6, June 2003, pages 47-56.
- [Lédeczi et al., 2001] Ákos Lédeczi, Árpád Bakay, Miklós Maróti, Péter Völgyesi, Greg Nordstrom, Jonathan Sprinkle, and Gábor Karsai, "Composing Domain-Specific Design Environments," *IEEE Computer*, vol. 34, no. 11, November 2001, pages 44-51.
- [Lédeczi et al., 2001a] Ákos Lédeczi, Greg Nordstrom, Gábor Karsai, Péter Völgyesi, and Miklós Maróti, "On Metamodel Composition," *In Proceedings of the International Conference on Control Applications*, Mexico City, Mexico, September 2001, pages 756-760.
- [Lenz and Wienands, 2006] Gunther Lenz and Christoph Wienands, *Practical Software Factories in .NET*, Apress, 2006.
- [Liskov and Zilles, 1975] Barbara Liskov and Stephen Zilles, "Specification Techniques for Data Abstractions", *SIGPLAN Notices*, vol. 10, no. 6, April 1975, pages 72-87.
- [Liu et al., 2010] Qichao Liu, Barrett R. Bryant, and Marjan Mernik, "Metamodel Recovery from Multi-Tiered Domains using Extended MARS," *In Proceedings of the 34th Annual International Computer Software and Applications Conference*, Seoul, South Korea, July 2010, pages 279-288.
- [Lucas et al., 1994] Simon M. Lucas, Enrique Vidal, A. Amiri, S. Hanlon, and Juan-Carlos Amengual, "A Comparison of Syntactic and Statistical Techniques for Offline OCR," *In Proceedings of the International Colloquium of Grammatical Inference*, Alicante, Spain, September 1994, pages 168-179.
- [Luisa et al., 2004] Mich Luisa, Franch Mariangela, and Inverardi Pierluigi, "Market Research for Requirements Analysis Using Linguistic Tools," *Requirements Engineering*, vol. 9, no. 1, February 2004, pages 40-56.
- [Maiden and Robertson, 2005] Neil Maiden and Suzanne Robertson, "Developing Use Cases and Scenarios in the Requirements Process," *In Proceedings of the 27th International Conference on Software Engineering*, May 2005, St. Louis, MO, pages 561-570.
- [Mapelsden et al., 2002] David Mapelsden, John Hosking, and John Grundy, "Design Pattern Modelling and Instantiation using DPML," *In Proceedings of the Fortieth International Conference on Tools Pacific: Objects for Internet, Mobile and Embedded Applications*, Sydney, Australia, February 2002, pages 3-11.
- [Martin, 1967] James Martin, *Design of Real-Time Computer Systems*, Prentice-Hall, 1967.
- [Martin, 1996] Robert C. Martin, "The Open-Closed Principle," *More C++ Gems*, vol. 8, January 1996, pages 97-112.
- [Mazanek and Minas, 2009] Steffen Mazanek and Mark Minas, "Business Process Models as a Showcase for Syntax-Based Assistance in Diagram Editors," *International Conference on Model Driven Engineering Languages and Systems*, Denver, CO, October, 2009, pages 322-336.

- [Medvidovic and Rosenblum, 1997] Nenad Medvidovic and David S. Rosenblum, "Domains of Concern in Software Architectures and Architecture Description Languages," *In Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages*, Berkeley, CA, October 1997, pages 199-212.
- [Mens and Gorp, 2006] Tom Mens and Pieter Van Gorp, "A Taxonomy of Model Transformation," *Electronic Notes in Theoretical Computer Science*, vol. 152, March 2006, pages 125-142.
- [Mernik et al., 2005] Marjan Mernik, Jan Heering, and Anthony M. Sloane, "When and How to Develop Domain-Specific Languages," *ACM Computing Surveys*, vol. 37, no. 4, December 2005, pages 316-344.
- [Messmer and Bunke, 1999] Bruno T. Messmer and Horst Bunke, "A Decision Tree Approach to Graph and Subgraph Isomorphism Detection," *Pattern Recognition*, vol. 32, no. 12, December 1999, pages 1979-1998.
- [Michalski, 1983] Ryszard S. Michalski, "A Theory and Methodology of Inductive Learning," *Artificial Intelligence*, vol. 20, no. 2, February 1983, pages 111-161.
- [Milanović et al., 2009] Milan Milanović, Dragan Gašević, Adrian Giurca, Gerd Wagner, and Vladan Devedžić, "Bridging Concrete and Abstract Syntaxes in Model-Driven Engineering: A Case of Rule Languages," *Software Practice & Experience*, vol. 39, no. 16, November 2009, pages 1313-1346.
- [Milner, 1989] Robin Milner, *Communication and Concurrency*, Prentice Hall, 1989.
- [Misue et al., 1995] Kazuo Misue, Peter Eades, Wei Lai, and Kozo Sugiyama, "Layout Adjustment and the Mental Map," *Journal of Visual Languages and Computing*, vol. 6, no. 2, June 1995, pages 183-210.
- [Mockus, 2007] Audris Mockus, "Large-Scale Code Reuse in Open Source Software," *In Proceeding of First International Workshop on Emerging Trends in FLOSS Research and Development*, Minneapolis, MN, May 2007, p.7.
- [Mohagheghi and Conradi, 2008] Parastoo Mohagheghi and Reidar Conradi, "An Empirical Investigation of Software Reuse Benefits in a Large Telecom Product," *ACM Transactions on Software Engineering and Methodology*, vol. 17, no. 3, Article 13, June 2008.
- [Moody, 2009] Daniel L. Moody, "The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering," *IEEE Transactions on Software Engineering*, November/December 2009, pages 756-779.
- [Morris et al., 1999] Michael G. Morris, Cheri Speier, and Jeffrey A. Hoffer, "An Examination of Procedural and Object-oriented Systems Analysis Methods: Does Prior Experience Help or Hinder Performance?" *Decision Sciences*, vol. 30, no. 1, January 1999, pages 107-136.
- [Murata, 1989] Tadao Murata, "Petri Nets: Properties, Analysis, and Applications," *Proceedings of the IEEE*, vol. 77, no. 4, April 1989, pages 541-580.

- [Nakatani et al., 1999] Lloyd H. Nakatani, Mark A. Ardis, Robert G. Olsen, and Paul M. Pontrelli, "Jargons for Domain Engineering," *In Proceedings of the 2nd Conference on Domain-Specific Languages*, Austin, TX, October 1999, pages 15-24.
- [Narayanan et al., 2010] Krishna Kumar Narayanan, Luis Felipe Posada, Frank Hoffmann, and Torsten Bertram, "Robot Programming by Demonstration," *Simulation, Modeling, and Programming for Autonomous Robots*, Darmstadt, Germany, November 2010, pages 288-299.
- [Nardi, 1993] Bonnie A. Nardi, *A Small Matter of Programming: Perspectives on End User Computing*, MIT Press, 1993.
- [Naumann and Jenkins, 1982] Justus D. Naumann and A. Milton Jenkins, "Prototyping: The New Paradigm for Systems Development," *MIS Quarterly*, vol. 6, no. 3, September 1982, pages 29-44.
- [Naur and Randell, 1969] Peter Naur and Brain Randell, (Eds.), "Software Engineering: Report of a Conference Sponsored by the NATO Science Committee," Garmisch, Germany, October 1968, Brussels, Scientific Affairs Division, NATO (1969).
- [Neema et al., 2003] Sandeep Neema, Janos Sztipanovits, Gabor Karsai, and Ken Butts, "Constraint-Based Design-Space Exploration and Model Synthesis," *In Third International Conference on Embedded Software*, Philadelphia, PA, October 2003, pages 290-305.
- [Ney, 1992] Hermann Ney, "Stochastic Grammars and Pattern Recognition," *Speech Recognition and Understanding*, vol. 75, 1992, pages 313-344.
- [Nordstrom et al., 1999] Greg Nordstrom, Janos Sztipanovits, Gábor Karsai, and Ákos Lédeczi, "Metamodeling - Rapid Design and Evolution of Domain-Specific Modeling Environments," *International Conference on Engineering of Computer-Based Systems*, Nashville, TN, April 1999, pages 68-74.
- [Nuseibeh and Easterbrook, 2000] Bashar Nuseibeh and Steve Easterbrook, "Requirements Engineering: A Roadmap," *In Proceedings of the Conference on The Future of Software Engineering*, Limerick, Ireland, June 2000, pages 35-46.
- [Offen, 1999] Ray J. Offen, "CASE Tools and Constraint, CADPRO - Experiments in CASE Tool Use and Constraint Conditions," *North Ryde: Macquarie University Joint Research Centre for Advanced Systems Engineering*, September 1999, pages 3-12
- [Oliveira et al., 2010] Marcio FS Oliveira, Francisco A. Nascimento, Wolfgang Mueller, and Flávio R. Wagner, "Design Space Abstraction and Metamodeling for Embedded Systems Design Space Exploration," *In Proceedings of the 7th International Workshop on Model-Based Methodologies for Pervasive and Embedded Software*, Antwerp, Belgium, September 2010, pages 29-36.
- [Ossher et al., 2010] Harold Ossher, Rachel Bellamy, Ian Simmonds, David Amid, Ateret Anaby-Tavor, Matthew Callery, Michael Desmond, Jacqueline de Vries, Amit Fisher, and Sophia Krasikov, "Flexible Modeling Tools for Pre-requirements Analysis: Conceptual Architecture and Research Challenges," *In Proceedings of Object-Oriented Programming, Systems, Languages, and Applications*, Reno/Tahoe, NV, October 2010, pages 848-864.

- [Ouardani et al., 2006] Adel Ouardani, Philippe Esteban, Mario Paludetto, and Jean-Claude Pascal, "A Meta-Modeling Approach for Sequence Diagrams to Petri Net Transformation within the Requirements Validation Process," *In Proceedings of the European Simulation and Modeling Conference*, Toulouse, France, October 2006, pages 345-349.
- [Paakki, 1995] Jukka Paakki, "Attribute Grammar Paradigms - A High-Level Methodology in Language Implementation," *ACM Computing Surveys*, vol. 27, no. 2, June 1995, pages 196-255.
- [Paige et al., 2000] Richard F. Paige, Jonathan S. Ostroff, and Phillip J. Brooke, "Principles for Modeling Language Design," *Information and Software Technology*, vol. 42, no. 10, July 2000, pages 665-675.
- [Parnas, 1999] David Lorge Parnas, "Software Engineering Programmes are not Computer Science Programmes," *IEEE Software*, vol. 6, no. 1, November/December 1999, pages 19-37.
- [Pedersen and Klein, 1988] Jan S. Pedersen and Mark H. Klein, "Using the Vienna Development Method (VDM) to Formalize a Communication Protocol," *Technical Report CMU/SEI-88-TR-026*, SEI, Carnegie Mellon University, Pittsburgh, PA, November 1988.
- [Petre, 1995] Marian Petre, "Why Looking isn't Always Seeing: Readership Skills and Graphical Programming," *Communications of the ACM*, vol. 38, no. 6, June 1995, pages 33-44.
- [Pohl et al., 2005] Klaus Pohl, Günter Böckle, and Frank J. van der Linden, *Software Product Line Engineering: Foundations, Principles, and Techniques*, Springer-Verlag Inc., 2005.
- [Potter et al., 1996] Ben Potter, David Till, and Jane Sinclair, *An Introduction to Formal Specification and Z (2nd ed.)*, Prentice Hall, 1996.
- [Pu et al., 1997] Calton Pu, Andrew Black, Crispin Cowan, Jonathan Walpole, and Charles Consel, "Microlanguages for Operating System Specialization," *In Proceedings of the Conference on Domain-Specific Language*, Paris, France, January 1997, pages 49-57.
- [Qattous et al., 2010] Hazem Qattous, Philip Gray, and Ray Welland, "An Empirical Study of Specification By Example in a Software Engineering Tool," *In Proceedings of the International Symposium on Empirical Software Engineering and Measurement*, Bolzano-Bozen, Italy, 2010, article 16.
- [Repenning and Perrone, 2000] Alexander Repenning and Corrina Perrone, "Programming By Example: Programming By Analogous Examples," *Communications of the ACM*, vol. 43, no. 3, March 2000, pages 90-97.
- [Rieger, 1995] Anke Rieger, "Inferring Probabilistic Automata from Sensor Data for Robot Navigation," *In Proceedings of the MLnet Familiarization Workshop and Third European Workshop on Learning Robots*, Crete, Greece, 1995, pages 65-74.
- [Rolland et al., 2003] Colette Rolland, Carine Souveyet, and Mohamed Ben Ayed, "Guiding Lyee User Requirements Capture," *Intention and Software Process*, vol. 16, no. 7-8, November 2003, pages 351-359.
- [Roscoe, 1997] Andrew William Roscoe, Charles Antony Richard Hoare, and Richard Bird, *The Theory and Practice of Concurrency*, Prentice Hall, 1997.

- [Rose et al., 2012] Louis M. Rose, Dimitrios S. Kolovos, and Richard F. Paige, “EuGENia Live: A Flexible Graphical Modelling Tool,” *In Proceedings of Extreme Modelling Workshop*, Innsbruck, Austria, October 2012.
- [Rosson and Alpert, 1990] Mary Beth Rosson and Sherman R. Alpert, “The Cognitive Consequences of Object-Oriented Design,” *Human Computer Interaction*, vol. 5, no. 4, December 1990, pages 345-379.
- [Royce, 1987] Winston W. Royce, “Managing the Development of Large Software Systems: Concepts and Techniques,” *In Proceedings of the 9th International Conference on Software Engineering*, Monterey, CA, April 1987, pages 328-339.
- [Rubel et al., 2011] Dan Rubel, Jaime Wren, and Eric Clayberg, *The Eclipse Graphical Editing Framework (GEF)*, Addison-Wesley Professional, 2011.
- [Saxena and Karsai, 2010] Tripti Saxena and Gabor Karsai, “Towards a Generic Design Space Exploration Framework,” *In Proceedings of the 10th International Conference on Computer and Information Technology*, Bradford, UK, 2010, pages 1940-1947.
- [Schäfer et al., 2011] Christian Schäfer, Thomas Kuhn, and Mario Trapp, “A Pattern-Based Approach to DSL Development,” *In Proceedings of Workshops on Domain-Specific Modeling*, Portland, OR, October 2011, pages 39-46.
- [Schätz et al., 2010] Bernhard Schätz, Florian Hölzl, and Torbjörn Lundkvist, “Design-Space Exploration through Constraint-Based Model-Transformation,” *In Proceedings of the 17th International Conference and Workshops on the Engineering of Computer-Based Systems*, Oxford, England, UK, March 2010, pages 173-182.
- [Schmidt, 1988] David A. Schmidt, *Denotational Semantics: A Methodology for Language Development*, William C. Brown Pub, 1988.
- [Schmidt, 2006] Douglas C. Schmidt, “Model-Driven Engineering,” *IEEE Computer*, vol. 39, no. 2, 2006, pages 25-31.
- [Selic, 2009] Bran Selic, “The Theory and Practice of Modeling Language Design for Model-Based Software Engineering: A Personal Perspective,” *In Proceedings of the 3rd International Summer School Conference on Generative and Transformational Techniques in Software Engineering III*, Braga, Portugal, July 2009, pages 290-321.
- [Sendall and Kozaczynski, 2003] Shane Sendall and Wojtek Kozaczynski, “Model Transformation - The Heart and Soul of Model-Driven Software Development,” *IEEE Software*, vol. 20, no. 5, September 2003, pages 42-45.
- [Smith, 1975] David Canfield, “Pygmalion: A Creative Programming Environment,” *Stanford Technical Report STAN-CS-75-499*, Stanford University, June 1975.
- [Snyder, 1986] Alan Snyder, “Encapsulation and Inheritance in Object-Oriented Programming Languages,” *In Proceedings of Object-Oriented Programming, Systems, Languages, and Applications*, Portland, OR, 1986, pages 38-45.

- [Solomonoff, 1959] Ray Solomonoff, “A New Method for Discovering the Grammars of Phrase Structure Languages,” *In Proceedings of the International Conference on Information Processing*, Paris, France, 1959, pages 285-289.
- [Steinberg et al., 2008] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks, *EMF: Eclipse Modeling Framework (2nd ed.)*, Addison-Wesley Professional, 2008.
- [Stevson and Fleck, 1997] Daniel E. Stevenson and Margaret M. Fleck, “Programming Language Support for Digitized Images or, The Monsters in the Closet,” *In Proceedings of the Conference on Domain-Specific Languages*, Paris, France, January 1997, pages 271-284.
- [Sun et al., 2009] Yu Sun, Jules White, and Jeff Gray, “Model Transformation By Demonstration,” *International Conference on Model Driven Engineering Languages and Systems*, Denver, CO, October 2009, pages 712-726.
- [Sun et al., 2011] Yu Sun, Jeff Gray, Gerti Kappel, Philip Langer, Manuel Wimmer, and Jules White, “A WYSIWYG Approach to Support Layout Configuration in Model Evolutions,” *Emerging Technologies for the Evolution and Maintenance of Software Models*, IGI Global, 2012, pages 92-120.
- [Sutton, 2005] Stanley M. Sutton Jr., “Aspect-Oriented Software Development and Software Process,” *In Proceeding of International Software Process Workshop*, Beijing, China, May 2005, pages 177-191.
- [Taylor et al., 1961] Warren Taylor, Lloyd Turner, and Richard Waychoff, “A Syntactical Chart of ALGOL 60,” *Communications of the ACM*, vol. 4, no. 9, September 1961, pages 393-396.
- [Thollard et al., 2000] Franck Thollard, Pierre Dupont, and Colin De La Higuera, “Probabilistic DFA Inference using Kullback-Leibler Divergence and Minimality,” *In Proceedings of the 17th International Conference on Machine Learning*, Stanford, CA, July 2000, pages 975-982.
- [Uchitel et al., 2002] Sebastian Uchitel, Jeff Kramer, and Jeff Magee, “Negative Scenarios for Implied Scenario Elicitation,” *ACM SIGSOFT Software Engineering Notes*, vol. 27, no. 6, November 2002, pages 109-118.
- [Valiant, 1979] Leslie G. Valiant, “The Complexity of Enumeration and Reliability Problems,” *SIAM Journal on Computing*, vol. 8, no. 3, 1979, pages 410-421.
- [Visser, 2007] Eelco Visser, “WebDSL: A Case Study in Domain-Specific Language Engineering,” *In Generative and Transformational Techniques in Software Engineering II*, Braga, Portugal, July 2007, pages 291-373.
- [Wang and Acero, 2001] Ye-Yi Wang and Alex Acero, “Grammar Learning for Spoken Language Understanding,” *In Proceedings of the Workshop on Automatic Speech Recognition and Understanding*, Madonna di Campiglio, Italy, 2001, pages 292-295.
- [Watson, 2008] Andrew Watson, “A Brief History of MDA,” *Upgrade-the European Journal for the Informatics Professional*, vol. 9, no. 2, 2008, pages 7-11.
- [Wegner, 1987] Peter Wegner, “Dimensions of Object-Based Language Design,” *ACM SIGPLAN Notices*, vol. 22, no. 12, December 1987, pages 168-182.

[Wen and Dromey, 2004] Lian Wen and Geoff R. Dromey, "From Requirements Change to Design Change: A Formal Path," *In Proceeding of International Conference on Software Engineering and Formal Methods*, September 2004, Beijing, China, pages 104-113.

[Wiegers, 1999] Karl E. Wiegers, "Writing Quality Requirements," *Software Development*, vol. 7, no. 5, May 1999.

[Wile, 1997] David S. Wile, "Abstract Syntax from Concrete Syntax," *In Proceedings of the 19th International Conference on Software Engineering*, Boston, MA, May 1997, pages 472-480.

[Woodbury et al., 2000] Robert Francis Woodbury, Sambit Datta, and Andrew Lincoln Burrow, "Erasure in Design Space Exploration," *In Proceedings of the 6th International Conference on Artificial Intelligence in Design*, Worcester, MA, June 2000, pages 521-543.

[Woodcock and Davies, 1996] Jim Woodcock and Jim Davies, *Using Z: Specification, Refinement, and Proof*, Prentice-Hall, 1996.

[Zhang and Xu, 2004] Yingzhou Zhang and Baowen Xu, "A Survey of Semantic Description Frameworks for Programming Languages," *ACM SIGPLAN Notices*, vol. 39, no. 3, March 2004, pages 14-30.

[Zloof, 1975] Moshé M. Zloof, "Query-By-Example: The Invocation and Definition of Tables and Terms," *International Conference on Very Large Data Bases*, Framingham, MA, September 1975, pages 1-24.

[EMF, 2011] Eclipse Modeling Project, <http://www.eclipse.org/modeling/>, 2011.

[GEF, 2011] Graphical Editing Framework, <http://www.eclipse.org/gef/>, 2011.

[GEMS, 2011] Generic Eclipse Modeling System (GEMS), <http://www.eclipse.org/gmt/gems/>, 2011.

[GME, 2012] Generic Modeling Environment, <http://www.isis.vanderbilt.edu/Projects/gme/>, 2012.

[IBM Database Fundamental] IBM Database Fundamentals, <http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/index.jsp?topic=%2Fcom.ibm.db2.luw.sql.ref.doc%2Fdoc%2Fr0006726.html>.

[IBM Rational DOORS] IBM Rational DOORS, Get it Right the First Time, IBM Rational DOORS Manual.

[IEEE 1471, 2000] Rich Hilliard, "IEEE-std-1471-2000: Recommended Practice for Architectural Description of Software Intensive Systems," *The Architecture Working Group of the Software Engineering Committee*, Standards Department, IEEE.

[Kermeta] Kermeta, <http://www.kermeta.org/docs/fr.irisa.triskell.kermeta.samples.fsm.documentation/build/html.chunked/KerMeta-The-FSM-example/index.html>.

[MDA, 2011] MDA Specifications, <http://www.omg.org/mda/specs.htm#MDAGuide>

[MetaCase+, 2011] MetaCase+, <http://www.metacase.com/>, 2011.

[MOF, 2011] Object Management Group Meta Object Facility Specification, [http://www.omg.org/technology/documents/modeling\\_spec\\_catalog.htm#MOF](http://www.omg.org/technology/documents/modeling_spec_catalog.htm#MOF).

[OCL, 2011] Object Management Group Object Constraint Language Specification, [http://www.omg.org/technology/documents/modeling\\_spec\\_catalog.htm#OCL](http://www.omg.org/technology/documents/modeling_spec_catalog.htm#OCL), 2011.

[Oracle Syntax Diagram] Oracle Syntax Diagram, [http://docs.oracle.com/cd/B28359\\_01/server.111/b28286/ap\\_syntax.htm#i624534](http://docs.oracle.com/cd/B28359_01/server.111/b28286/ap_syntax.htm#i624534).

[OMG BPMN] OMG Business Process Model And Notation (BPMN) Ver. 2.0, <http://www.omg.org/spec/BPMN/2.0/>.

[QImPrESS] QImPrESS Service Architecture Metamodel, [http://www.q-impress.eu/wordpress/wp-content/uploads/2009/05/d21-service\\_architecture\\_meta-model.pdf](http://www.q-impress.eu/wordpress/wp-content/uploads/2009/05/d21-service_architecture_meta-model.pdf).

[SharePoint 2010 REST] SharePoint 2010 REST Service Syntax Diagram, <http://blogs.msdn.com/b/sharepointpictures/archive/2011/03/30/sharepoint-2010-rest-service-syntax-diagram.aspx>.

[Simulink] Simulink Tutorial, [http://www.colorado.edu/mechanical/programs/undergraduate/matlab\\_tutorials/simulink/simulink.htm](http://www.colorado.edu/mechanical/programs/undergraduate/matlab_tutorials/simulink/simulink.htm).

[UML Infrastructure] OMG Unified Modeling Language (OMG UML) Infrastructure Version 2.4.1, <http://www.omg.org/spec/UML/2.4.1/Infrastructure/PDF>.

[UML Superstructure] OMG Unified Modeling Language (OMG UML) Superstructure Version 2.4.1, <http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF>.

[OMG XMI, 2011] Object Management Group XMI Specification, [http://www.omg.org/technology/documents/modeling\\_spec\\_catalog.htm#XMI](http://www.omg.org/technology/documents/modeling_spec_catalog.htm#XMI).

[UseCase Maps] Use Case Maps, <http://www.usecasemaps.org>.