A MODEL TRANSFORMATION APPROACH TO AUTOMATED MODEL EVOLUTION

by

YUEHUA LIN

JEFFREY G. GRAY, COMMITTEE CHAIR
BARRETT BRYANT
ANIRUDDHA GOKHALE
MARJAN MERNIK
CHENGCUI ZHANG

A DISSERTATION

Submitted to the graduate faculty of The University of Alabama at Birmingham,
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

BIRMINGHAM, ALABAMA

2007

A MODEL TRANSFORMATION APPROACH
TO AUTOMATED MODEL EVOLUTION

YUEHUA LIN

COMPUTER AND INFORMATION SCIENCES

ABSTRACT

It is well-known that the inherent complex nature of software systems adds to the challenges of software development. The most notable techniques for addressing the complexity of software development are based on the principles of abstraction, problem decomposition, separation of concerns and automation. As an emerging paradigm for developing complex software, Model-Driven Engineering (MDE) realizes these principles by raising the specification of software to models, which are at a higher level of abstraction than source code. As models are elevated to first-class artifacts within the software development lifecycle, there is an increasing need for frequent model evolution to explore design alternatives and to address system adaptation issues. However, a system model often grows in size when representing a large-scale real-world system, which makes the task of evolving system models a manually intensive effort that can be very time consuming and error prone. Model transformation is a core activity of MDE, which converts one or more source models to one or more target models in order to change model structures or translate models to other software artifacts. The main goal of model transformation is to provide automation in MDE. To reduce the human effort associated with model evolution while minimizing potential errors, the research described in this dissertation has contributed toward a model transformation approach to automated model evolution.

A pre-existing model transformation language, called the Embedded Constraint Language (ECL), has been evolved to specify tasks of model evolution, and a model transformation engine, called the Constraint-Specification Aspect Weaver (C-SAW), has been developed to perform model evolution tasks in an automated manner. Particularly, the model transformation approach described in this dissertation has been applied to the important issue of model scalability for exploring design alternatives and crosscutting modeling concerns for system adaptation.

Another important issue of model evolution is improving the correctness of model transformation. However, there execution-based testing has not been considered for model transformation testing in current modeling practice. As another contribution of this research, a model transformation testing approach has been investigated to assist in determining the correctness of model transformations by providing a testing engine called M2MUnit to facilitate the execution of model transformation tests. The model transformation testing approach requires a new type of test oracle to compare the actual and expected transformed models. To address the model comparison problem, model differentiation algorithms have been designed and implemented in a tool called DSMDiff to compute the differences between models and visualize the detected model differences.

The C-SAW transformation engine has been applied to support automated evolution of models on several different experimental platforms that represent various domains such as computational physics, middleware, and mission computing avionics. The research described in this dissertation contributes to the long-term goal of alleviating the increasing complexity of modeling large-scale, complex applications.

DEDICATION

To my husband Jun,

my parents, Jiafu and Jinying, and my sisters, Yuerong and Yueqin

for their love, support and sacrifice.

To Wendy and Cindy,

my connection to the future.

ACKNOWLEDGEMENTS

I am deeply grateful to all the people that helped me to complete this work. First and foremost, I wish to thank my advisor, Dr. Jeff Gray, who has offered to me much valuable advice during my Ph.D. study, as well as inspired me to pursue high quality research from an exemplary work ethic. Through constant support from his DARPA and NSF research grants, I was able to start my thesis research at a very early stage during the second semester of my first year of doctoral study, which allowed me to focus on this research topic without involving other non-research duties. With his expertise and research experiences in the modeling area, he has led me into the research area of model transformation and helped me make stable and significant research progress. Moreover, Dr. Gray provided unbounded opportunities and resources that enabled me to conduct collaborative research with several new colleagues. In addition, he encouraged me to participate in numerous professional activities (e.g., conference and journal reviews), and generously shared with me his experiences in proposal writing. Without his tireless advising efforts and constant support, I could not have matured into an independent researcher.

I also want to show my gratitude to Dr. Barrett Bryant. I still remember how I was impressed by his prompt replies to my questions during my application for graduate study in the CIS department. Since that time, he has offered many intelligent and insightful suggestions to help me adapt to the department policies and procedures, strategies and culture.

I would like to thank Dr. Chengcui Zhang for her continuous encouragement. As the only female faculty member in the department, she has been my role model as a successful female researcher. Thank you, Dr. Zhang, for sharing with me your experiences in research and strategies for job searching.

To Dr. Aniruddha Gokhale and Dr. Marjan Mernik, I greatly appreciate your precious time and effort in serving as my committee members. I am grateful to his willingness to assist me in improving this work.

To Janet Sims, Kathy Baier, and John Faulkner, who have been so friendly and helpful during my Ph.D. studies – they have helped to make the department a very pleasant place to work by making me feel at home and at ease with your kind spirit.

I am indebted to my collaborators at Vanderbilt University. Special thanks are due to Dr. Sandeep Neema, Dr. Ted Bapty, and Zoltan Molnar who helped me overcome technical difficulties during my tool implementation. I would like to thank Dr. Swapna Gokhale at the University of Connecticut, for sharing with me her expertise and knowledge in performance analysis and helping me to better understand Stochastic Reward Nets. I also thank Dario Correal at the University of Los Andes, Colombia for applying one of my research results (the C-SAW model transformation engine) to his thesis research. Moreover, thanks to Dr. Frédéric Jouault for offering a great course, which introduced me to many new topics and ideas in Model Engineering. My work on model differentiation has been improved greatly by addressing his constructive comments.

My student colleagues in the SoftCom lab and the department created a friendly and cheerful working atmosphere that I enjoyed and will surely miss. To Jing Zhang, I

cherish the time we were working together on Dr. Gray's research grants. To Hui Wu, Alex Liu, Faizan Javed and Robert Tairas, I appreciate your time and patience in listening to me and discussing my work, which helped me to overcome difficult moments and made my time here at UAB more fun.

To my best friends, Wenyan Gan and Shengjun Zheng, who I met with luck in my middle school, I appreciate your giving my parents and younger sisters long term help when I am out of hometown.

My strength to complete this work comes from my family. To my Mom and Dad, thank you for giving me the freedom to pursue my life in another country. To my sisters, thank you for taking care of our parents when I was far away from them. To my husband, Jun, thank you for making such a wonderful and sweet home for me and being such a great father to our two lovely girls. Without your unwavering love and support, I can not imagine how I would complete this task. The best way I know to show my gratitude is to give my love to you from the bottom of my heart as you have given to me.

Last, I am grateful to the National Science Foundation (NSF), under grant CSR-0509342, and the DARPA Program Composition for Embedded Systems (PCES), for providing funds to support my research assistantship while working on this dissertation.

TABLE OF CONTENTS

Page

TABLE OF CONTENTS (Continued)

Page

CHAPTER

TABLE OF CONTENTS (Continued)

Page

CHAPTER

TABLE OF CONTENTS (Continued)

Page

CHAPTER

LIST OF TABLES

Table *Page*

LIST OF FIGURES

LIST OF LISTINGS

LIST OF ABBREVIATIONS

AMMA	Atlas Model Management Architecture
ANTLR	Another Tool for Language Recognition
AOM	Aspect-Oriented Modeling
AOP	Aspect-Oriented Programming
AOSD	Aspect-Oriented Software Development
API	Application Program Interface
AST	Abstract Syntax Tree
ATL	Atlas Transformation Language
CASE	Computer-Aided Software Engineering
CIAO	Component-Integrated ACE ORB
CORBA	Common Object Request Broker Architecture
C-SAW	Constraint-Specification Aspect Weaver
CWM	Common Warehouse Metamodel
DSL	Domain-Specific Language
DSM	Domain-Specific Modeling
DSML	Domain-Specific Modeling Language
DRE	Distributed Real-Time and Embedded
EBNF	Extended Backus-Naur Form
ECL	Embedded Constraint Language

EMF	Eclipse Modeling Framework
EQAL	Event Quality Aspect Language
ESML	Embedded Systems Modeling Language
GME	Generic Modeling Environment
GPL	General Programming Language
GReAT	Graph rewriting and transformation
IP	Internet Protocol
LHS	Left-Hand Side
MDA	Model-Driven Architecture
MDE	Model-Driven Engineering
MDPT	Model-Driven Program Transformation
MCL	Multigraph Constraint Language
MIC	Model-Integrated Computing
MOF	Meta Object Facility
MTBE	Model Transformation by Example
NP	Non-deterministic Polynomial time
OCL	Object Constraint Language
OMG	Object Management Group
PBE	Programming by Example
PIM	Platform-Independent Model
PICML	Platform-Independent Component Modeling Language
PLA	Production-Line Architecture
PSM	Platform-Specific Model

QBE	Query by Example
QoS	Quality of Service
RHS	Right-Hand Side
RTES	Real-Time and Embedded Systems
SRN	Stochastic Reward Net
SRNML	Stochastic Reward Net Modeling Language
QVT	Query/View/Transformations
SIML	System Integrated Modeling Language
XMI	XML Metadata Interchange
XML	Extensible Markup Language
XSLT	Extensible Stylesheet Transformations
YATL	Yet Another Transformation Language
UAV	Unmanned Aerial Vehicle
UDP	User Datagram Protocol
UML	Unified Modeling Language
UUID	Universally Unique Identifier
VB	Visual Basic

CHAPTER 1

INTRODUCTION

It is well-known that the inherent complex nature of software systems increases the challenges of software development [Brooks, 95]. The most notable techniques for addressing the complexity of software development are based on the principles of abstraction, problem decomposition, information hiding, separation of concerns and automation [Dijkstra, 76], [Parnas, 72]. Since the inception of the software industry, various efforts in software research and practice have been made to provide abstractions to shield software developers from the complexity of software development.

Computer-Aided Software Engineering (CASE) was a prominent effort that focused on developing software methods and modeling tools that enabled developers to express their designs in terms of general-purpose graphical programming representations, such as state machines, structure diagrams, and dataflow diagrams [Schmidt, 06]. As the first software product sold independently of a hardware package, Autoflow was a flowchart modeling tool developed in 1964 by Martin Goetz of Applied Data Research [Johnson, 98]. Although CASE tools have historical relevance in terms of offering some productivity benefits, there are several limitations that have narrowed their potential [Gray et al., 07].

The primary drawback of most CASE tools was that they were constrained to work with a fixed notation, which forced the end-users to adopt a language prescribed by the tool vendors. Such a universal language may not be suitable in all cases for an end-user's distinct needs for solving problems in their domain. As observed by Schmidt, "As a result, CASE had relatively little impact on commercial software development during the 1980s and 1990s, focusing primarily on a few domains, such as telecom call processing, that mapped nicely onto state machine representations" [Schmidt, 06].

Another goal of CASE is to automate software development by synthesizing implementations from the graphical design representations. However, there exists a major hindrance to achieve such automation due to the lack of integrated transformation technologies to transform graphical representations at a high-level of abstraction (e.g., design model) to a low-level representation (e.g., implementation code). Consequently, many CASE systems were restricted to a few specific application domains and unable to satisfy the needs for developing production-scale systems across various application domains [Schmidt, 06].

There also has been significant effort toward raising the abstraction of programming languages to shield the developers from the complexity of both language and platform technologies. For example, early programming languages such as assembly language provide an abstraction over machine code. Today, Object-Oriented languages such as C++ and Java introduce additional abstractions (e.g., abstract data types and objects) [Hailpern and Tarr, 06]. However, the advances on programming languages still cannot cover the fast growing complexity of platforms. For example, popular middleware platforms, such as J2EE, .NET and CORBA, contain thousands of classes and methods

with many intricate dependencies. Such middleware evolves rapidly, which requires considerable manual effort to program and port application code to newer platforms when using programming languages [Schmidt, 06]. Also, programming languages are hard to describe system-wide, non-functional concerns such as system deployment, configuration and quality assurance because they primarily aim to specify functional aspects of a system.

To address the challenges in current software development such as the increased complexity of products, shortened development cycles and heightened expectations of quality [Hailpern and Tarr, 06], there is an increasing need for new languages and technologies that can express the concepts effectively for a specific domain. Also, new methodologies are needed for decomposing a system to various but consistent aspects, and enabling transformation and composition between various artifacts in the software development lifecycle within a unified infrastructure. To meet these challenges, Model-Driven Engineering (MDE) [Kent, 02] is an emerging approach to software development that centers on higher level specifications of programs in Domain-Specific Modeling Languages (DSMLs), offering greater degrees of automation in software development, and the increased use of standards [Schmidt, 06]. In practice, Domain-Specific modeling (DSM) is a methodology to realize the vision of MDE [Gray et al., 07].

1.1 Domain-Specific Modeling (DSM)

MDE represents a design approach that enables description of the essential characteristics of a problem in a manner that is decoupled from the details of a specific solution space (e.g., dependence on specific middleware or programming language). To

apply lessons learned from earlier efforts at developing higher level platform and language abstraction, a movement within the current MDE community is advancing the concept of customizable modeling languages, in opposition to a universal, general-purpose language that attempts to offer solutions for a broad category of users such as the Unified Modeling Language (UML) [Booch et al., 99]. This newer breed of tools enables DSM, an MDE methodology that generates customized modeling languages and environments for a narrow domain of interest.

In the past, abstraction was improved when programming languages evolved towards higher levels of specification. DSM takes a different approach, by raising the level of abstraction, while at the same time narrowing down the design space, often to a single range of products for a single domain [Gray et al., 07]. When applying DSM, the language follows the domain abstractions and semantics, allowing developers to perceive themselves as working directly with domain concepts of the problem space instead of code concepts of the solution space. Also, domain-specific models are subsequently transformed into executable code by a sequence of model transformations to provide automation support for software development. As shown in Figure 1-1, DSM technologies combine the following:

- **Domain-specific modeling languages** "whose type systems formalize the application structure, behavior, and requirements within particular domains" [Schmidt, 06]. A **metamodel** formally defines the abstract syntax and static semantics of a DSML by specifying a set of modeling elements and their valid relationships for that specific domain. A **model** is an instance of the metamodel that represents a particular part of a real system. Developers use DSMLs to build

domain-specific models to specify applications and their design intents [Gray et al., 07].

- **Model transformations** play a key role in MDE to convert models to other software artifacts. They are used for refining models to capture more system details or synthesizing various types of artifacts from models. For example, models can be synthesized to source code, simulation input and XML deployment descriptions. Model transformation can be automated to reduce human effort and potential errors during software development.

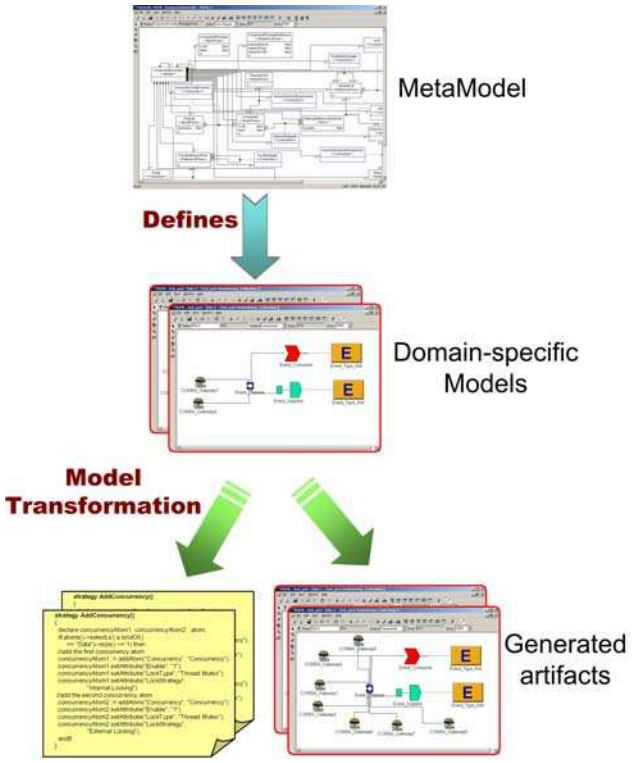


Figure 1-1 - *Metamodel, models and model transformation*

The DSM philosophy of narrowly defined modeling languages can be contrasted with larger standardized modeling languages, such as the UML, which are fixed and

whose size and complexity [Gîrba and Ducasse, 06] provide abstractions that may not be needed in every domain, adding to the confusion of domain experts. Moreover, using notations that relate directly to a familiar domain not only helps flatten learning curves but also facilitates the communication between a broader range of experts, such as domain experts, system engineers and software architects. In addition, the ability of DSM to synthesize artifacts from high-level models to low-level implementation artifacts, simplifies the activities in software development such as developing, testing and debugging. Most recently, the "ModelWare" principle (i.e., everything is a model) [Kurtev et al., 06] has been adopted in the MDE community to provide a unified infrastructure to integrate various artifacts and enable transformations between them during the software development lifecycle.

The key challenge in applying DSM is to define useful standards that enable tools and models to work together portably and effectively [Schmidt, 06]. Existing de facto standards include the Object Management Group's Model Driven Architecture (MDA) [MDA, 07], Query/View/Transformations (QVT) [QVT, 07] and the MetaObject Facilities (MOF) [MOF, 07]. These standards can also form the basis for domain-specific modeling tools. Existing metamodeling infrastructures and tools include the Generic Modeling Environment (GME) [Lédeczi et al., 01], ATLAS Model Management Architecture (AMMA) [Kurtev et al., 06], Microsoft's DSL tools [Microsoft, 05], [Cook et al., 07], MetaEdit+ [MetaCase, 07], and the Eclipse Modeling Framework (EMF) [Budinsky et al., 04]. Initial success stories from industry adoption of DSM have been reported, with perhaps the most noted being Saturn's multi-million dollar cost savings associated with timelier reconfiguration of an automotive assembly line driven by

domain-specific models [Long et al., 98]. The newly created DSM Forum [DSM Forum, 07] serves as a repository of several dozen successful projects (mostly from industry, such as Nokia, Dupont, Honeywell, and NASA) that have adopted DSM.

1.2 The Need for Frequent Model Evolution

The goal of MDE is to raise the level of abstraction in program specification and increase automation in software development in order to simplify and integrate the various activities and tasks that comprise the software development lifecycle. In MDE, models are elevated as the first-class artifacts in software development and used in various activities such as software design, implementation, testing and evolution.

A powerful justification for the use of models concerns the flexibility of system analysis, i.e., system analysis can be performed while exploring various design alternatives. This is particularly true for distributed real-time and embedded (DRE) systems, which have many properties that are often conflicting (e.g., battery consumption versus memory size), where the analysis of system properties is often best provided at higher levels of abstraction [Hatcliff et al., 03]. Also, when developers apply MDE tools to model large-scale systems containing thousands of elements, designers must be able to examine various design alternatives quickly and easily among myriad and diverse configuration possibilities. Ideally, a tool should simulate each new design configuration so that designers could rapidly determine how some configuration aspect, such as a communication protocol, affects an observed property, such as throughput. To provide support for that degree of design exploration, frequent change evolution is required within system models [Gray et al., 06].

Although various types of changes can be made to models, there are two categories of changes that designers often do manually—typically with poor results. The first category comprises changes that crosscut the model representation's hierarchy in order to adapt the modeled system to new requirements or environments. The second category of change evolution involves scaling up parts of the model—a particular concern in the design of large-scale distributed, real-time, embedded systems, which can have thousands of coarse-grained components. Model transformation provides automation support in MDE, not only for translating models into other artifacts (i.e., exogenous transformation) but also for changing model structures (i.e., endogenous transformation). Application of model transformation to automate model evolution can reduce human effort and potential errors. The research described in this dissertation concentrates on developing an automated model transformation approach to address two important system properties—system adaptability and scalability at the modeling level, each corresponding to one category of model evolution, as discussed in the following sections.

1.2.1 System Adaptability through Modeling

Adaptability is emerging as a critical enabling capability for many applications, particularly for environment monitoring, disaster management and other applications deployed in dynamically changing environments. Such applications have to reconfigure themselves according to fluctuations in their environment. A longstanding challenge of software development is to construct software that is easily adapted to changing requirements and new environments. Software production-line architectures (PLAs) are a

promising technology for the industrialization of software development by focusing on the automated assembly and customization of domain-specific component [Clements and Northrop, 01], which requires the ability to rapidly configure, adapt and assemble independent components to produce families of similar but distinct systems [Deng et al., 08]. As demand for software adaptability increases, novel strategies and methodologies are needed for supporting the requisite adaptations across different software artifacts (e.g., models, source code, test cases, documentation) [Batory et al., 04].

In modeling, many requirements changes must be made across a model hierarchy, which are called crosscutting modeling concerns [Gray et al., 01]. An example is the effect of fluctuating bandwidth on the quality of service across avionics components that must display a real-time video stream. To evaluate such a change, the designer must manually traverse the model hierarchy by recursively clicking on each submodel. Another example is the Quality of Service (QoS) constraints of Distributed Real-Time and Embedded (DRE) systems. The development of DRE systems is often a challenging task due to conflicting QoS constraints that must be explored as trade-offs among a series of alternative design decisions. The ability to model a set of possible design alternatives, and to analyze and simulate the execution of the representative model, offers great assistance toward arriving at the correct set of QoS parameters needed to satisfy the requirements for a specific DRE system. Typically, the QoS specifications are also distributed in DRE system models, which necessitate intensive effort to make changes manually.

1.2.2 System Scalability through Modeling

Scalability is a desirable property of a system, a network, or a process, which indicates its ability to either handle growing amounts of work in a graceful manner, or to be readily enlarged [Bondi, 00] (e.g., new system resources may be added or new types of objects the system needs to handle). The corresponding form of design exploration for system scalability involves experimenting with model structures by expanding different portions of models and analyzing the result on scalability. For example, a network engineer may create various models to study the effect on network performance when moving from two routers to eight routers, and then to several dozen routers. This requires the ability to build a complex model from a base model by replicating its elements or substructures and adding the necessary connections [Lin et al., 07-a].

This type of change requires creating model elements and connections. Obviously, scaling a base model of a few elements to thousands of new elements requires a staggering amount of clicking and typing within the modeling tool. The ad hoc nature of this process causes errors, such as forgetting to make a connection between two replicated elements. Thus, manual scaling affects not only modeling performance, but also the representation's correctness.

According to the above discussion, to support system adaptability and scalability requires extensive support from the host modeling tool to enable rapid change evolution within the model representation. There are several challenges that need to be addressed in order to improve the productivity and quality of model evolution. These challenges are discussed in the next section.

1.3 Key Challenges in Model Evolution

As discussed in Section 1.2, with the expanded focus of software and system models has come the urgent need to manage complex change evolution within the model representation [Sendall and Kozaczynski, 03]. In current MDE practice, as the size of system models expands, the limits of MDE practice are being pushed to address increasingly complex model management issues that pertain to change evolution within the model representation by providing new methodologies and best practices. Also, there is an increasing need to apply software engineering principles and processes into general modeling practice to assist in systematic development of models and model transformation. As a summary, the research described in this dissertation focuses on challenges in current modeling practice that are outlined in the following subsections.

1.3.1 The Increasing Complexity of Evolving Large-scale System Models

To support frequent model evolution, changes to models need to be made quickly and correctly. Model evolution tasks have become human intensive because of the growing size of system models and the deeply nested structures of models, inherently from the complexity of large-scale software systems.

From our personal experience, models can have multiple thousands of coarse grained components (others have reported similar experience, please see [Johann and Egyed, 04]). Modeling these components using traditional manual model creation techniques and tools can approach the limits of the effective capability of humans. Particularly, the process of modeling a large DRE system with a DSML, or a tool like MatLab [Matlab, 07], is different than traditional class-based UML modeling. In DRE

systems modeling, the models consist of instances of all entities in the system, which can number into several thousand instances from a set of types defined in a metamodel (e.g., thousands of individual instantiations of a sensor type in a large sensor network model). Traditional UML models (e.g., UML class diagrams) are typically not concerned with the same type of instance-level focus, but instead specify the entities and their relationship of a system at design time (such as classes). This is not to imply that UML-based models do not have change evolution issues such as scalability issues (in fact, the UML community has recognized the importance of specifying instance models at a large-scale [Cuccuru et al., 05]), but the problem is more acute with system models built with DSMLs. The main reason is that system models are usually sent to an analysis tool (e.g., simulation tool) to explore system properties such as performance and security. Such models need to capture a system by including the instances of all the entities (such as objects) that occur at run-time, which leads to their larger size and nested hierarchy [Lin et al., 07-a].

Due to the growing size and the complicated structures of a large-scale system model, a manual process for making correct changes can be laborious, error-prone and time consuming. For example, to examine the effect of scalability on a system, the size of a system model (e.g., the number of the participant model elements and connections) needs to be increased or decreased frequently. The challenges of scalability affect the productivity of the modeling process, as well as the correctness of the model representation. As an example, consider a base model consisting of a few modeling elements and their corresponding connections. To scale a base model to hundreds, or even thousands of duplicated elements would require a considerable amount of mouse clicking and typing within the associated modeling tool [Gray et al., 06]. Furthermore,

the tedious nature of manually replicating a base model may also be the source of many errors (e.g., forgetting to make a connection between two replicated modeling elements). Therefore, a manual process to model evolution significantly hampers the ability to explore design alternatives within a model (e.g., after scaling a model to 800 modeling elements, it may be desired to scale back to only 500 elements, and then back up to 700 elements, in order to understand the impact of system size). An observation from the research described in this dissertation is that the complexities of model evolution must be tackled at a higher level of abstraction through automation with a language tailored to the task of model transformation.

1.3.2 The Limited Use of Model Transformations

Model transformation has the potential to provide intuitive notations at a high-level of abstraction to define tasks of model evolution. However, this new role of model transformation has not been addressed fully by current modeling research and practice. For example, transformations in software modeling and design are mostly performed between modeling languages representing different domains. The role of transformation within the same language has not been fully explored as a new applications of stepwise refinement. Such potential roles for transformations may include the following:

1. model optimizations—transforming a given model to an equivalent one that is optimized, in the sense that a given metric or design rule is respected;
2. consistency checks—transforming different viewpoints of the same model into a common notation for purposes of comparison;

3. automation of parts of the design process—transformations are used in developing and managing design artifacts like models;
4. model scalability – automation of model changes that will scale a base model to a larger configuration [Lin et al., 07-a];
5. modularization of crosscutting modeling concerns – properties of a model may be scattered across the modeling hierarchy. Model transformations may assist in modularizing the specification of such properties in a manner that supports rapid exploration of design alternatives [Gray et al., 06].

Within a complex model evolution process, there are many issues that can be addressed by automated model transformation. The research described in this dissertation presents the benefits that model transformation offers in terms of capturing crosscutting model properties and other issues dealing with the difficulties of model scalability. Besides model scalability and modularization of crosscutting modeling concerns, another scenario is building implementation models (e.g., deployment models) based on design models (e.g., component models) [Balasubramanian et al., 06]. Such tasks can be performed rapidly and correctly in an automated fashion using the approach presented in Chapter 3.

1.3.3 The Lack of Model Transformation Testing for Improving Correctness

One of the key issues in software engineering is to ensure that the product delivered meets its specification. In traditional software development, testing [Gelperin and Hetzel, 88], [Zhu et al., 97] and debugging [Rosenberg, 96], [Zellweger, 84] have proven to be vital techniques toward improving quality and maintainability of software systems. However, such processes are heavily applied at source code levels and are less

integrated into modeling. In addition to formal methods (e.g., model checking and theorem proving for verifying models and transformations), testing, as a widely used technique serving as a best practice in software engineering, can serve as an engineering solution to validate model transformations.

Model transformation specifications are used to define tasks of model evolution. A transformation specification, like the source code in an implementation, is written by humans and susceptible to errors. Additionally, a transformation specification may be reusable across similar domains. Therefore, it is essential to test the correctness of the transformation specification (i.e., the consistency and completeness, as validated against model transformation requirements) before it is applied to a collection of source models. Consequently, within a model transformation infrastructure, it is vital to provide well-established software engineering techniques such as testing for validating model transformation [Küster, 06]. Otherwise, the correctness of the transformation may always be suspect, which hampers confidence in reusing the transformation. A contribution to model transformation testing is introduced in Chapter 5.

1.3.4 Inadequate Support for Model Differentiation

The algorithms and the supporting tools of model differentiation (i.e., finding mappings and differences between two models, also called model differencing or model comparison) may benefit various modeling practices, such as model consistency checking, model versioning and model refactoring. In the transformation testing framework, model differencing techniques are crucial to realize the vision of execution-

based testing of model transformations by assisting in comparing the expected result (i.e., the expected model) and the actual output (i.e., the output model).

Currently, there are many tools available for differentiating text files (e.g., code and documentation). However, these tools operate under a linear file-based paradigm that is purely textual, but models are often structurally rendered in a tree or graphical notation. Thus, there is an abstraction mismatch between currently available version control tools and the hierarchical nature of models. To address this problem, there have been only a few research efforts on UML model comparison [Ohst et al., 03], [Xing and Stroulia, 05] and metamodel independent comparison [Cicchetti et al., 07]. However, there has been no work reported on comparison of domain-specific models, aside from [Lin et al., 07-a].

Visualization of the result of model comparison (i.e., structural model differences) is also critical to assist in comprehending the mappings and differences between two models. To help communicate the comparison results, visualization techniques are needed to highlight model differences intuitively within a host modeling environment. For example, graphical symbols and colors can be used to indicate whether a model element is missing or redundant. Additionally, these symbols and colors are needed to decorate properties even inside models. Finally, a navigation system is needed to support browsing model differences efficiently. Such techniques are essential to understanding the results of model comparison. The details of a novel model differencing algorithm with visualization tool support are presented in Chapter 4.

1.4 Research Goals and Overview

To address the increasing complexity of modeling large-scale software systems and to improve the productivity and quality of model evolution, the main goal of the research described in this dissertation is to provide a high-level model transformation approach and associated tools for rapid evolution of large-scale systems in an automated manner. To assist in determining the correctness of model transformations, this research also investigates testing of model transformations. The model transformation testing project has led into an exploration of model comparison, which is needed to determine the differences between an expected model and the actual result. Figure 1-2 shows an integrated view of this research. The overview of the research is described in the following sections.

1.4.1 Model Transformation to Automate Model Evolution

To address the complexity of frequent model evolution, a contribution of this research is a model transformation approach to automated model evolution. A pre-existing model transformation language, called the Embedded Constraint Language (ECL), has been evolved to specify tasks of model evolution. The ECL has been re-implemented in a model transformation engine, called the Constraint-Specification Aspect Weaver (C-SAW), to perform model evolution tasks in an automated manner. Particularly, the model transformation approach described in this dissertation has been applied to the important issue of model scalability for exploring design alternatives and crosscutting modeling concerns for system adaptation.

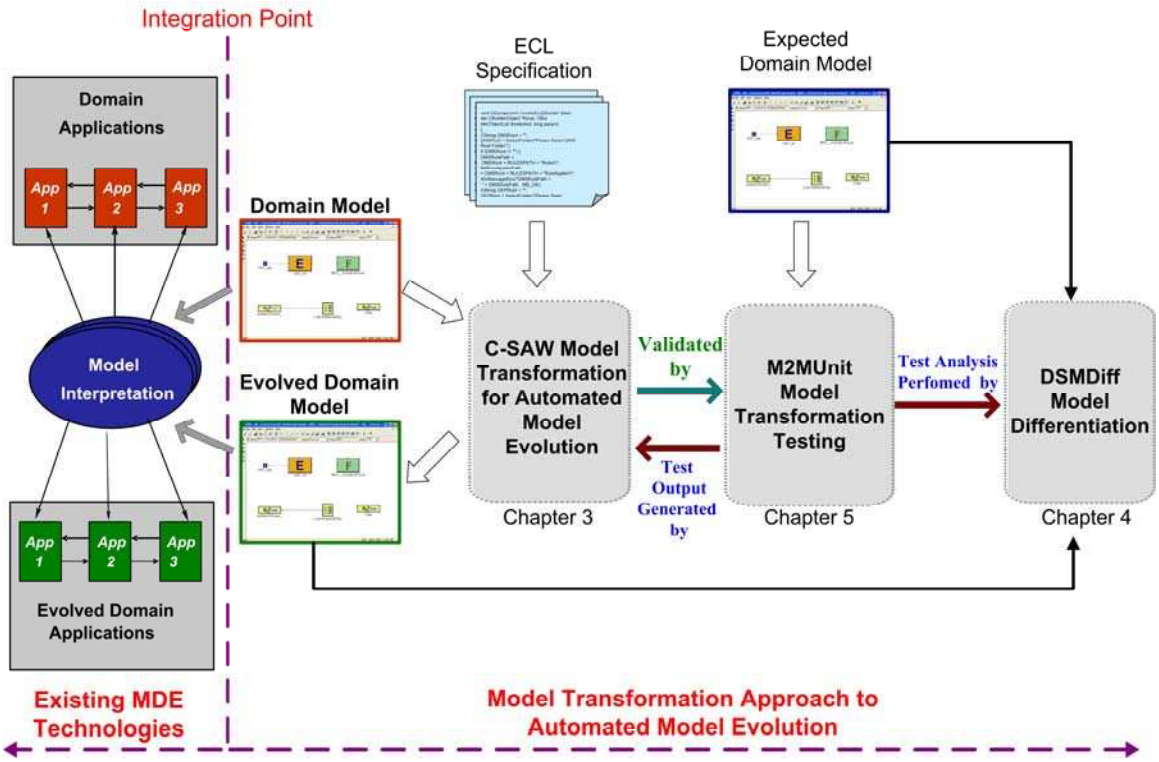


Figure 1-2 - *An overview of the topics discussed in this dissertation*

By enabling model developers to work at a higher level of abstraction, ECL serves as a small but powerful language to define tasks of model evolution. By providing automation to execute ECL specifications, C-SAW aims to reduce the complexity that is inherent in the challenge problems of model evolution.

1.4.2 Model Transformation Testing to Ensure the Correctness

Another important issue of model transformation is to ensure its correctness. To improve the quality of C-SAW transformations, a model transformation testing approach has been investigated to improve the accuracy of transformation results where a model transformation testing engine called M2MUnit provides support to execute test cases with the intent of revealing errors in the transformation specification.

The basic functionality includes execution of the transformations, comparison of the actual output model and the expected model, and visualization of the test results. Distinguished from classical software testing tools, to determine whether a model transformation test passes or fails requires comparison of the actual output model with the expected model, which necessitates model differencing algorithms and visualization. If there are no differences between the actual output and expected models, it can be inferred that the model transformation is correct with respect to the given test specification. If there are differences between the output and expected models, the errors in the transformation specification need to be isolated and removed.

By providing a unit testing approach to test the ECL transformations, M2MUnit aims to reduce the human effort in verifying the correctness of model evolution.

1.4.3 Model Differentiation Algorithms and Visualization Techniques

Driven by the need of model comparison for model transformation testing, model differencing algorithms and an associated tool called DSMDiff have been developed to compute differences between models. In addition to model transformation testing, model differencing techniques are essential to many model development and management practices such as model versioning.

Theoretically, the generic model comparison problem is similar to the graph isomorphism problem that can be defined as finding the correspondence between two given graphs, which is known to be in NP [Garey and Johnson, 79]. The computational complexity of graph matching algorithms is the major hindrance to applying them to practical applications in modeling. To provide efficient and reliable model differencing

algorithms, the research described in this dissertation provides a solution by using the syntax of modeling languages to help handle conflicts during model matching and combining structural comparison to determine whether the two models are equivalent. In general, DSMDiff takes two models as hierarchical graphs, starts from the top-level of the two containment models and then continues comparison to the child sub-models.

Visualization of the result of model differentiation (i.e., structural model differences) is critical to assist in comprehending the mappings and differences between two models. To help communicate the discovered model differences, a tree browser has been constructed to indicate the possible kinds of model differences (e.g., a missing element, or an extra element, or an element that has different values for some properties) with graphical symbols and colors.

1.4.4 Experimental Validation

This research provides a model transformation approach to automated model evolution that considers additional issues of testing to assist in determining the correctness of model transformations. The contribution has been evaluated to determine the degree to which the developed approach achieves a significant increase in productivity and accuracy in model evolution. The modeling artifacts available for experimental validation are primarily from two sources. One source is Vanderbilt University, a collaborator on much of the C-SAW research, who has provided multiple modeling artifacts as experimental artifacts. The other source is the Escher repository [Escher, 07], which makes modeling artifacts developed from DARPA and NSF projects available for experimentation.

C-SAW has been applied to several model evolution projects for experimental evaluation. The feedback from these case studies has been used to evaluate the modeling effectiveness of C-SAW (e.g., reduced time, increased accuracy and usability) and has demonstrated C-SAW as an effective tool to automate model evolution in various domains for specific types of transformations. Moreover, a case study is provided to highlight the benefit of the M2MUnit testing engine in detecting errors in model transformation. Analytical evaluation has been conducted to assess the performance and relative merit of the DSMDiff algorithms and tools.

1.5 The Structure of the Dissertation

To conclude, the major contributions of the thesis work include: 1) automated model evolution by offering ECL as a high-level transformation language to specify model evolution and providing the C-SAW model transformation engine to execute the ECL specifications; 2) apply software engineering practices such as testing to model transformations in order to ensure the correctness of model evolution; and 3) develop model differentiation algorithms and an associated tool (DSMDiff) for computing the mappings and differences between domain-specific models. The thesis research aims to address the difficult problems in modeling complex, large-scale software systems by providing support for evolving models rapidly and correctly.

The remainder of this dissertation is structured as follows: Chapter 2 provides further background information on MDE and DSM. Several modeling standards are introduced in Chapter 2, including MDA and the MOF metamodeling architecture. Furthermore, the concepts of metamodels and models are discussed in this background

chapter as well as the definitions and categories of model transformation are presented. DSM is further discussed in Chapter 2 by describing one of its paradigms - Model Integrated Computing (MIC) and its metamodeling tool GME, which is also the modeling environment used in the research described in this dissertation.

Chapter 3 details the model transformation approach to automate model evolution. The model transformation language ECL and the model transformation engine C-SAW are described as the initial work. The emphasis is given to describe how C-SAW has been used to address the important issues of model scalability for exploring alternative designs and model adaptability for adapting systems to new requirements and environments. Two case studies are presented to illustrate how C-SAW addresses the challenges. In addition, to demonstrate the benefits of this approach, experimental evaluation is discussed, including modeling artifacts, evaluation metrics and experimental results.

Chapter 4 describes the research contributions on model differentiation. This chapter begins with a brief discussion on the need for model differentiation, followed by detailed discussions on the limitations of current techniques. The problem of model differentiation is formally defined and the challenges for this problem are identified. The emphasis is placed on the model differentiation algorithms, including an analysis of non-structural and structural information of model elements, formal representation of models and details of the algorithms. The work representing visualization of model differences is also presented as necessary support to assist in comprehending the results of model differentiation. In addition, complexity analysis is given to evaluate the performance of the algorithms, followed by discussions on current limitations and future improvements.

Chapter 5 presents a model transformation testing approach. This chapter begins with a motivation of the specific need to ensure the correctness of model transformations, followed by a discussion on the limitations of current techniques. An overview of the model transformation testing approach is provided and an emphasis is given on the principles and the implementation of the model transformation testing engine M2MUnit. In addition, a case study is offered to illustrate this approach to assist in detecting the errors in ECL specifications.

Chapter 6 explores future extensions for this work and Chapter 7 presents concluding comments. Appendix A provides the grammar of ECL, and Appendix B lists the operations in ECL. There are two additional case studies provided in Appendix C to demonstrate the benefit of using C-SAW to scale domain-specific models.

CHAPTER 2

BACKGROUND

This chapter provides further background information on Model-Driven Engineering (MDE) and Domain-Specific Modeling (DSM). Several modeling standards are introduced, including Model-Driven Architecture (MDA) and the Meta Object Facility (MOF) metamodeling architecture. The concepts of metamodels and models are discussed, as well as the definitions and categories of model transformation. Domain-Specific Modeling is further discussed by describing one of its paradigms – Model Integrated Computing (MIC) and its metamodeling tool – the Generic Modeling Environment (GME), which is also the modeling environment used to conduct the research described in this dissertation.

2.1 Model-Driven Architecture (MDA)

MDA is a standard for model-driven software development promoted by the Object Management Group (OMG) in 2001, which aims to provide open standards to interoperate new platforms and applications with legacy systems [MDA, 07], [Frankel, 03], [Kleppe et al., 03]. MDA introduces a set of basic concepts such as model, metamodel, modeling language and model transformation and lays the foundation for MDE.

2.1.1 Objectives of MDA

To address the problem of the continual emergence of new technologies that forces organizations to frequently port their applications to new platforms, the primary goal of MDA is to provide cross-platform compatibility of application software despite any implementation, or platform-specific changes (to the hardware platform, the software execution platform, or the application software interface). In particular, MDA provides an architecture that assures portability, interoperability, reusability and productivity through architectural separation of concerns [Miller and Mukerji, 01]:

- **Portability**: "reducing the time, cost and complexity associated with retargeting applications to different platforms and systems that are built with new technologies;"
- **Reusability**: "enabling application and domain model reuse and reducing the cost and complexity of software development;"
- **Interoperability**: "using rigorous methods to guarantee that standards based on multiple implementation technologies all implement identical business functions;"
- **Productivity**: "allowing system designers and developers to use languages and concepts that are familiar to them, while allowing seamless communication and integration across the teams."

To meet the above objectives, OMG has established a number of modeling standards as the core infrastructure of the MDA:

- **The Unified Modeling Language (UML)** [UML, 07] is "a standard object-oriented modeling language and framework for specifying, visualizing, constructing, and documenting software systems;"

- **MetaObject Facility (MOF)** [MOF, 07] is “an extensible model driven integration framework for defining, manipulating and integrating metadata and data in a platform-independent manner;”
- **XML Metadata Interchange (XMI)** [XMI, 07] is “a model driven XML Integration framework for defining, interchanging, manipulating and integrating XML data and objects;”
- **Common Warehouse Metamodel (CWM)** [CWM, 07] is “standard interfaces that can be used to enable easy interchange of warehouse and business intelligence metadata between warehouse tools, warehouse platforms and warehouse metadata repositories in distributed heterogeneous environments.”

These standards form the basis for building platform-independent applications using any major open or proprietary platform, including CORBA, Java, .Net and Web-based platforms, and even future technologies.

2.1.2 The MDA Vision

OMG defines MDA as an approach to system development based on models. MDA classifies models into two categories: Platform-Independent Models (PIMs) and Platform-Specific Models (PSMs). These categories contain models at different levels of abstraction. PIM represents a view of a system without involving platform and technology details. PSM specifies a view of a system from a platform-specific viewpoint by containing platform and technology dependent information.

The development of a system according to the MDA approach starts by building a PIM with a high-level of abstraction that is independent of any implementation

technology. PIM describes the business functionality and behavior using UML, including constraints of services specified in the Object Constraint Language (OCL) [OCL, 07] and behavioral specification (dynamic semantics) specified in the Action Semantics (AS) language [AS, 01]. In the next phase the PIM is transformed to one or more PSMs. A PSM is tailored to specify a system in terms of the implementation constructs provided by the chosen platforms (e.g., CORBA, J2EE, and .NET). Finally, implementation code is generated from the PSMs in the code generation phase using model-to-code transformation tools. The MDA architecture is summarized in Figure 2-1.

The claimed advantages of MDA include increased quality and productivity of software development by isolating software developers from implementation details and allowing them to focus on a thorough analysis of the problem space. MDA, however, lacks the notion of a software development process. MDE is an enhancement of MDA that adds the notion of software development processes to operate and manage models by utilizing domain-specific technologies.

2.2 Basic Concepts of Metamodeling and Model Transformation

Metamodeling and model transformation are two important techniques used by model-driven approaches. However, there are no commonly agreed-upon definitions of these concepts in the literature and they may be analyzed from various perspectives. The remaining part of this chapter discusses some of these concepts to help the reader further understand the relationships among metamodels, models and model transformations.

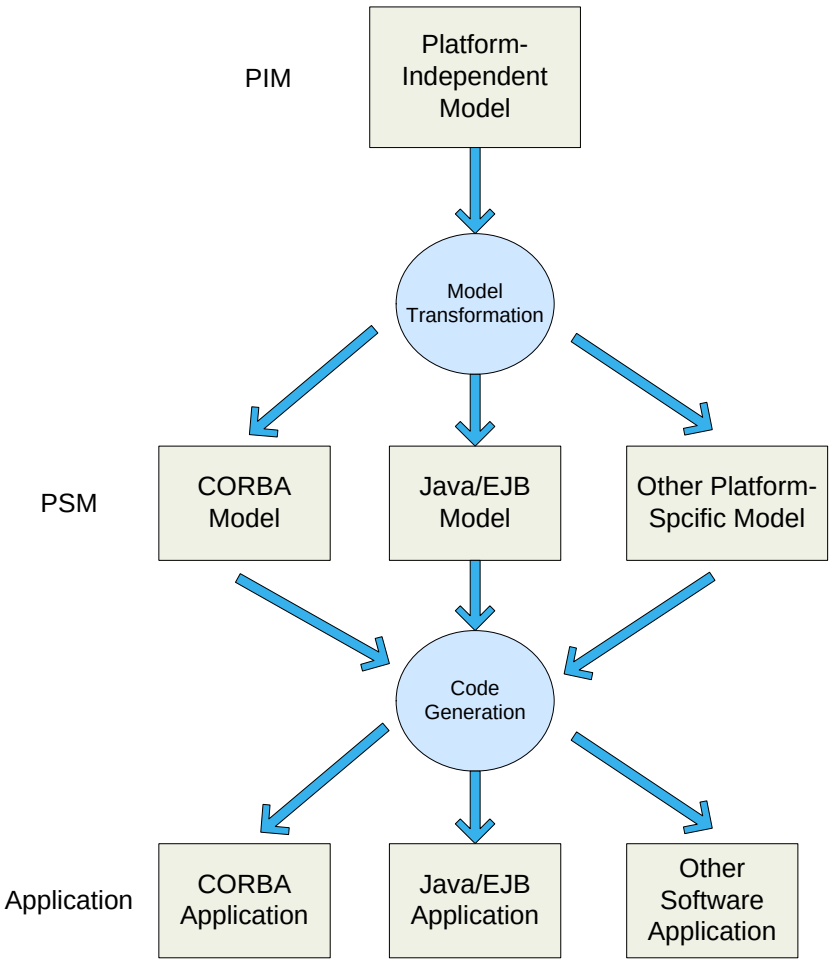


Figure 2-1 – *The key concepts of the MDA*

2.2.1 Metamodel, Model and System

MDE promotes models as primary artifacts in the software development lifecycle. There are various definitions that help to understand the relationship among modeling terms. For example, the MDA guide [MDA, 07] defines, “a model of a system is a description or specification of that system and its environment for some certain purpose. A model is often presented as a combination of drawings and text. The text may be in a modeling language or in a natural language.” Another model definition can be found in [Kleppe et al., 03], “A model is a description of a system written in a well-defined

language." In [Bézivin and Gerbé, 01] a model is defined as, "A model is a simplification of a system built with an intended goal in mind. The model should be able to answer questions in place of the actual system." In the context of this research, a model represents an abstraction of some real system, whose concrete syntax is rendered in a graphical iconic notation that assists domain experts in constructing a problem description using concepts familiar to them.

Metamodeling is a process for defining domain-specific modeling languages (DSMLs). A *metamodel* formally defines the abstract syntax and static semantics of a DSML by specifying a set of modeling elements and their valid relationships for that specific domain. A *model* is an instance of the metamodel that represents a particular part of a real system. *Conformance* is correspondence relationship between a metamodel and a model and *substitutability* defines a causal connection between a model and a system [Kurtev et al., 06]. As defined in [Kurtev et al., 06], a model is a *directed multigraph* that consists of a set of nodes, a set of edges and a mapping function between the nodes and the edges; a metamodel is a *reference model* of a model, which implies that there is a function associating the elements (nodes and edges) of the model to the nodes of the metamodel.

Based on the above definitions, the relation between a model and its metamodel is called conformance, which is denoted as *conformTo* or *c2*. Particularly, a metamodel is a model whose reference model is a metametamodel, and a metametamodel is a model whose reference model is itself [Kurtev et al., 06].

The substitutability principle is defined as, "a model *M* is said to be a representation of a system *S* for a given set of questions *Q* if, for each question of this set

Q, the model *M* will provide exactly the same answer that the system *S* would have provided in answering the same question" [Kurtev et al., 06]. Using this terminology, a model *M* is a representation of a given system *S*, satisfying the substitutability principle. The relation between a model and a system is called *representationOf*, which is also denoted as *repOf* [Kurtev et al., 06]. Figure 2-2 illustrates the relationship between a metamodel, a model and a system.

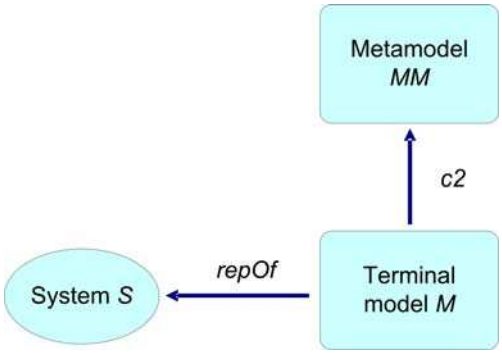


***Figure 2-2** - The relation between metamodel, model and system (adapted from [Kurtev et al., 06])*

2.2.2 The Four-Layer MOF Metamodeling Architecture

The MOF is an OMG standard for metamodeling. It defines a four-layer metamodeling architecture that a model engineer can use to define and manipulate a set of interoperable metamodels.

As shown in Figure 2-3, every model element on every layer strictly conforms to a model element on the layer above. For example, the MOF resides at the top (M3) level of the four-layer metamodel architecture, which is the meta-metamodel that conforms to itself. The MOF captures the structure or abstract syntax of the UML metamodel. The UML metamodel at the M2 level describes the major concepts and structures of UML

models. A UML model represents the properties of a real system (denoted as M1). MOF only provides a means to define the structure or abstract syntax of a language. For defining metamodels, MOF serves the same role that the extended Backus–Naur form (EBNF) [Aho et al., 07] plays for defining programming language grammars. For defining a DSML, a metamodel for that specific domain plays the role that a grammar plays for defining a specific language (e.g., Java).

There are other metamodeling techniques available for defining domain-specific modeling languages such as the Generic Modeling Environment (GME) [Lédeczi et al., 01], ATLAS Model Management Architecture (AMMA) [Kurtev et al., 06], Microsoft's DSL tools [Microsoft, 05], [Cook et al., 07], MetaEdit+ [MetaCase, 07], and the Eclipse Modeling Framework (EMF) [Budinsky et al., 04], which also follow the four-layer metamodeling architecture.

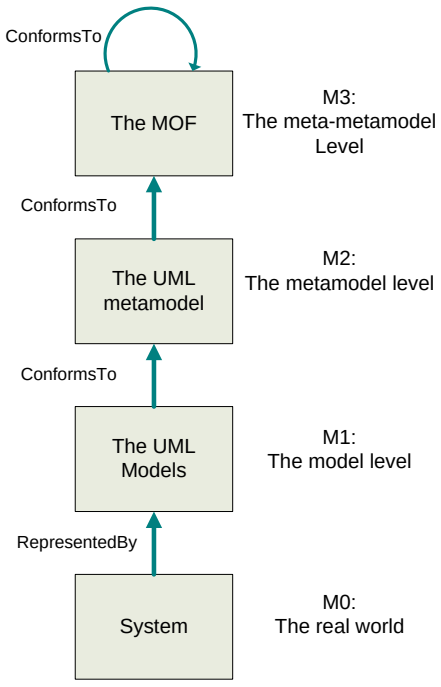


Figure 2-3 - *The MOF four-tier metamodeling architecture*

2.2.3 Model Transformation

Model transformation, a key component of model-driven approaches, represents the process of applying a set of transformation rules that take one or more source models as input to produce one or more target models as output [Sendall and Kozaczynski, 03], [Czarnecki and Helsen, 06], [Mens and Van Gorp, 05]. The source and target models may be defined either in the same modeling languages or in different modeling languages. Based on whether the source and target models conform to the same modeling language, model transformations can be categorized as *endogenous transformations* or *exogenous transformations*. Endogenous transformations are transformations between models expressed in the same language. Exogenous transformations are transformations between models expressed using different languages [Mens and Van Gorp, 05]. A typical example of endogenous transformations is model refactoring, where a change is made to the internal structure of models to improve certain qualities (e.g., understandability and modularity) without changing its observable behaviors [Zhang et al., 05-a]. An example of an exogenous transformation is model-to-code transformation that typically generates source code (e.g., Java or C++) from models.

Another standard to categorize model transformation is whether the source and target models reside at the same abstraction level. Based on this standard, model transformations can also be categorized as *horizontal transformations* and *vertical transformations* [Mens and Van Gorp, 05]. If the source and target models reside at the same abstraction level, such a model transformation is a horizontal transformation. Model refactoring is also a horizontal transformation. A vertical transformation is a transformation where the source and target models reside at different abstraction levels.

A typical example is model refinement, where a design model is gradually refined into a full-fledged implementation model, by means of successive refinement steps that add more concrete details [Batory et al., 04], [Greenfield et al., 04]. The dimensions of horizontal versus vertical transformations and endogenous versus exogenous transformations are orthogonal. For example, model migration translates models written in a language to another, but these two languages are at the same level of abstraction. A model migration is not only an exogenous transformation but also a horizontal transformation.

Although most existing MDE tools provide support for exogenous transformation to stepwise produce implementation code from designs, many modeling activities can be automated by endogenous transformations to increase the productivity of modeling and improve the quality of models. For example, such modeling activities include model refactoring [Zhang et al., 05-a] and model optimization [Mens and Van Gorp, 05]. Moreover, exogenous transformations are also useful for computing different views of a system model and synchronizing between them [Czarnecki and Helsen, 06]. This dissertation concentrates on applying endogenous transformations to automate model change evolution with an emphasis on addressing system adaptability and scalability, which is further discussed in Chapter 3. In the rest of the dissertation, the general term "model transformation" will refer to endogenous transformations (i.e., this research offers no contribution in the exogenous transformation form such as model-to-code transformation).

A model transformation is defined in a model transformation specification, which consists of a set of transformation rules. A transformation rule usually includes two parts:

a Left-Hand Side (LHS) and a Right-Hand Side (RHS). The LHS defines the configuration of objects in the source models to which the rule applies (i.e., filtering, which produces a subset of elements from the source model). The RHS defines the configuration of objects in the target models that will be created, updated or deleted by the rule. Both the LHS and RHS can be represented using any mixture of variables, patterns and logic.

A model transformation specification not only needs to define mapping rules, but also the scope of rule application. Additional parts of a transformation rule include the rule application strategy, and the rule application scheduling and organization [Czarnecki and Helsen, 06]. Rule application scoping includes the scope of source models and target models for rule application (in this case, scope refers to the portion or subset of a model to which the transformation is to be applied). The rule application strategy refers to how the model structure is traversed in terms of how selection matches are made with modeling elements when applying a transformation. A model transformation can also be applied as an in-place update where the source location becomes the target location. Rule application scheduling determines the order in which the rules are applied, and rule organization considers modularity mechanisms and organizational structure of the transformation specification [Czarnecki and Helsen, 06].

There exist various techniques to define and perform model transformations. Some of these techniques provide transformation languages to define transformation rules and their application, which can be either graphical or textual, either imperative or declarative. Although there exist different approaches to model transformation, the OMG has initiated a standardization process by adopting a specification on

Query/View/Transformation (QVT) [QVT, 07]. This process led to an OMG standard not only for defining model transformations, but also for defining views on models and synchronization between models. Typically, a QVT transformation definition describes the relationship between a source metamodel and a target metamodel defined by the MOF. It uses source patterns (e.g., the LHS part in a transformation rule) and target patterns (e.g., the RHS part in a transformation rule). In QVT, transformation languages are defined as MOF metamodels. A transformation is an instance of a transformation definition, and its source models and target models are instances of source patterns and target patterns, respectively. Such a generalized transformation pattern is shown partially in Figure 2-4, without indicating the transformation language level.

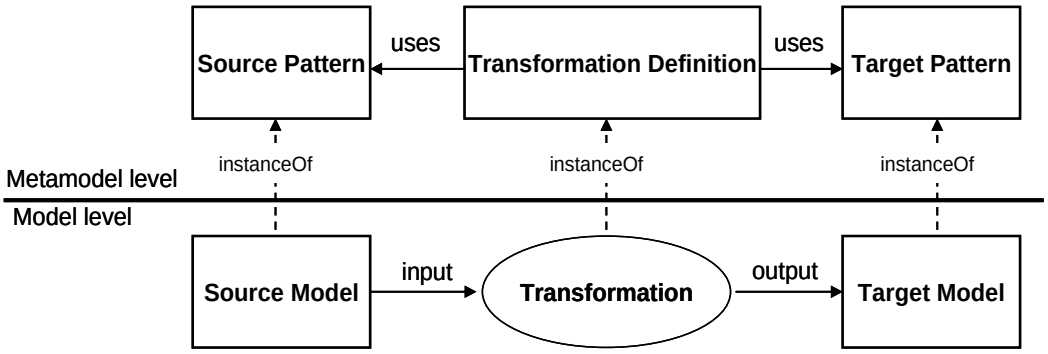


Figure 2-4 - *Generalized transformation pattern*

In MDE, model transformation is the core process to automate various activities in the software development life cycle. Exogenous transformation can be used to synthesize low-level software artifacts (e.g., source code) from high-level models, or to extract high-level models from lower level software artifacts such as reverse engineering. Endogenous transformation can be used to optimize and refactor models in order to improve the modeling productivity and the quality of models.

2.3 Supporting Technology and Tools

This research is tied to a specific form of MDE, called Model-Integrated Computing (MIC) [Sztipanovits and Karsai, 97], which has been refined at Vanderbilt University over the past decade to assist in the creation and synthesis of computer-based systems. The Generic Modeling Environment (GME) [GME, 07] is a metamodeling tool based on MIC principles, with which the dissertation research is conducted. The following sections provide further descriptions of MIC and GME.

2.3.1 Model-Integrated Computing (MIC)

MIC realized the vision of MDE a decade before the general concepts of MDE were enumerated in the modeling community. Similar to the MOF mechanism, MIC is also a four-layer metamodeling architecture that defines DSMLs for modeling real-world systems. Different from the MOF, MIC provides its own meta-metamodel called *GMEMeta* [Balasubramanian et al., 06-b] to define metamodels with notation similar to UML class diagrams and the OCL. In terms of MIC, the main concepts of model, metamodel, and other topics are defined as follows [Nordstrom, 99]:

- **Metamodeling Environment**: “a tool-based framework for creating, validating, and translating metamodels; ”
- **Metamodel**: also called the modeling paradigm, “formally defines a DSML for a particular domain, which captures the syntax, static semantics and visualization rules of the target domain;”
- **Modeling Environment**: “a system, based on a specific metamodel, for creating, analyzing, and translating domain-specific models;”

- **Model**: “an abstract representation of a computer-based system that is an instance of a specific metamodel.”

DSMLs are the backbone of MIC to capture the domain elements of various application areas. A DSML can be viewed as a five tuple [Karsai et al., 03]:

- **a concrete syntax** defines “the specific notation (textual or graphical) used to express domain elements;”
- **an abstract syntax** defines “the concepts, relationships, and integrity constraints available in the language;”
- **a semantic domain** defines “the formalism used to map the semantics of the models to a particular domain;”
- **a syntactic mapping** assigns “syntactic constructs (graphical or textual) to elements of the abstract syntax;”
- **a semantic mapping** relates “the syntactic concepts to the semantic domain.”

The key application domains of MIC range from embedded systems areas typified by automotive factories [Long et al., 98] to avionics systems [Gray et al., 04-b] that tightly integrate the computational structure of a system and its physical configuration. In such systems, MIC has been shown to be a powerful tool for providing adaptability in frequently changing environments [Sztipanovits and Kaisai, 97].

2.3.2 Generic Modeling Environment (GME)

GME is a metamodeling tool that realizes the principles of MIC [Lédeczi et al., 01]. As shown in Figure 2-5, GME provides a metamodeling interface to define

metamodels, a modeling environment to create and manipulate models and model interpretation to synthesize applications from models.

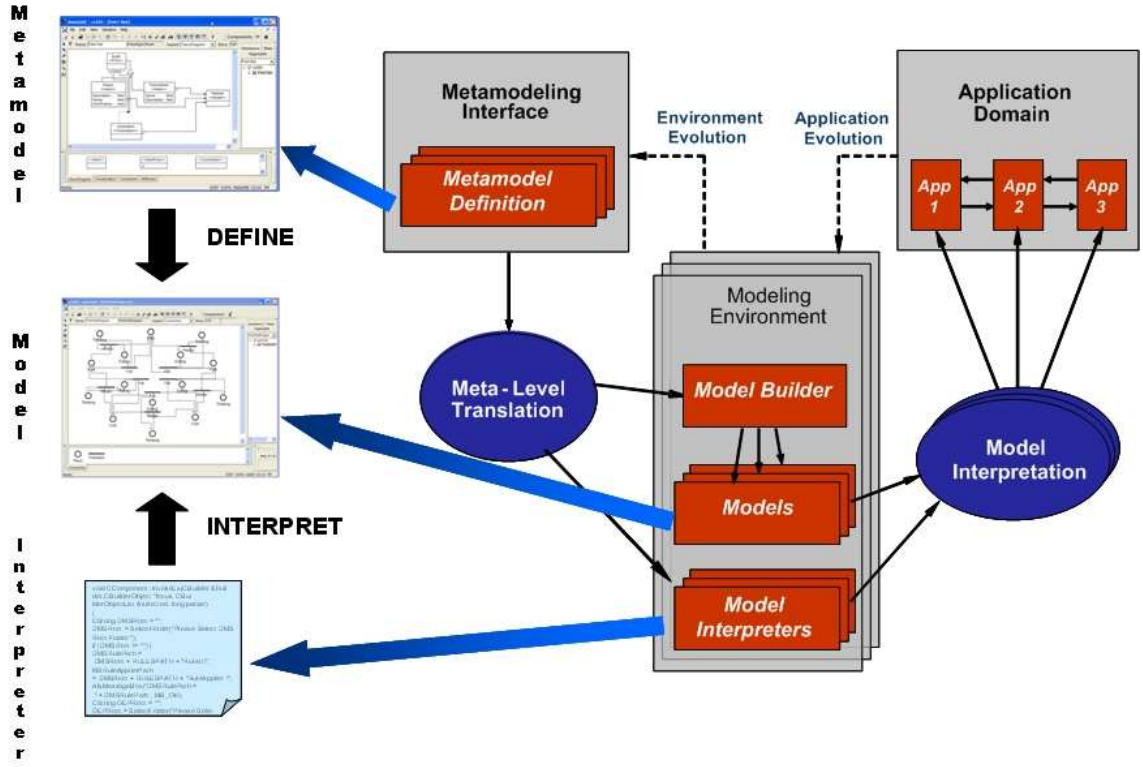


Figure 2-5 - *Metamodels, models and model interpreters (compilers) in GME (adapted from [Nordstrom et al., 99])*

When using the GME, a modeling paradigm is loaded into the tool by meta-level translation to define a modeling environment containing all the modeling elements and valid relationships that can be constructed in the target domain. Such an environment allows users to specify and edit visual models using notations common to their domain of expertise. GME also provides a mechanism for writing model compilers that translate models to different applications according to a user's various intentions. For example, such compilers can generate simulation applications or synthesize computer-based

systems. GME provides the following elements to define a DSML [Balasubramanian et al., 06-b]:

- **project:** "the top-level container of the elements of a DSML;"
- **folders:** "used to group similar elements;"
- **atoms:** "the atomic elements of a DSML, used to represent the leaf-level elements in a DSML;"
- **models:** "the compound objects in a DSML, used to contain different types of elements (e.g., references, sets, atoms, and connections);"
- **aspects:** "used to define different viewpoints of the same model;"
- **connections:** "used to represent relationships between elements of a DSML;"
- **references:** "used to refer to other elements in different portions of a DSML hierarchy;"
- **sets:** "containers whose elements are defined within the same aspect and have the same container as the owner."

The concepts of metamodel and model in GME are further illustrated by a state machine example, as shown in Figure 2-6 and Figure 2-7. Figure 2-6 shows a metamodel defined with the GME meta-metamodel, which is similar to a UML diagram class. It specifies the entities and their relationships needed for expressing a state machine, of which the instance models may be various state diagrams. As specified in the metamodel, a *StateDiagram* contains zero to multiple *StartState*, *State* or *EndState* elements, which are all inherited from *StateInheritance*, which is a first-class object (FCO) [1] . Thus,

[1]In GME, Atom, Model, Reference, Set and Connection are basic modeling elements called *first class objects* (FCOs).

StateInheritance may refer to *StartState*, *State* or *EndState* elements. Also, a *StateDiagram* contains zero to multiple *Transitions* between a pair of *StateInheritance* objects. Either *StateInheritance* or *Transition* is associated with an attribute definition such as a field. All of these *StateInheritance* or *Transition* elements are meta-elements of the elements in any of its instance models. Moreover, some rules can be defined as constraints within the metamodel by a language such as OCL. For example, a rule for the state machine may be "a state diagram contains only one *StartState*," which is specified in the OCL as "*parts("StartState")->size() = 1*".

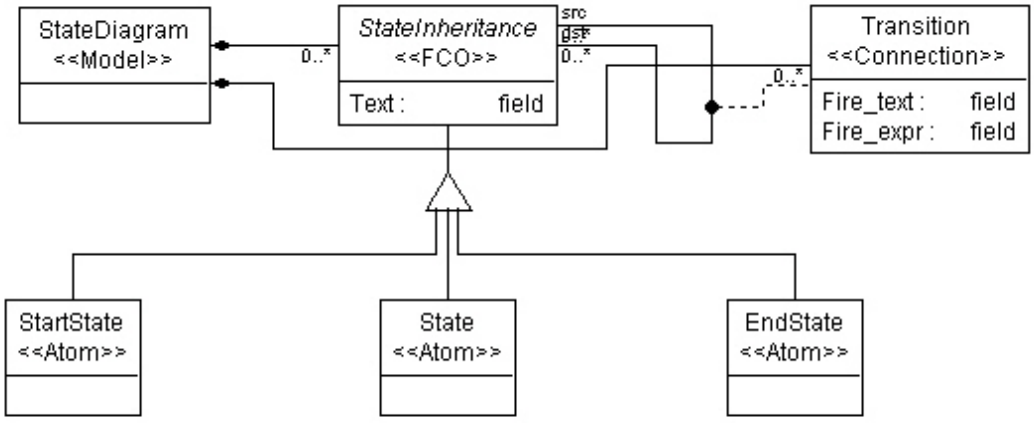


Figure 2-6 - *The state machine metamodel*

Figure 2-7 shows an Automated Teller Machine (ATM) model, which is an instance model of the state machine. Besides a *StartState* element and an *EndState* element, the ATM contains seven *State* elements and the necessary *Transition* elements between the *StartState*, *EndState* and *State* elements. The specification of this ATM model conforms to the state machine in several ways. For example, for any *State* element in the ATM (e.g., *CaredInserted* and *TakeReceipt*), its meta-element exists in the state

machine (i.e., *State*). Similarly, for all the links between *State* elements in the ATM, there exists an association in the state machine metamodel which leads from a *StateInheritance* to another *StateInheritance* or from a *StateInheritance* to itself. This represents type conformance within the metamodel. In other words, there exists a meta-element (i.e., type) in the metamodel for any element in the instance model. In addition to type conformance, the ATM needs to conform to the attribute and constraint definition in the state machine metamodel.

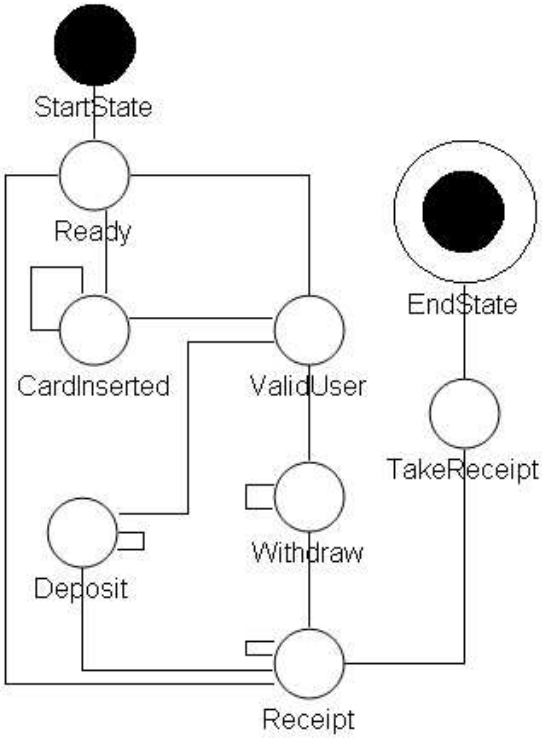


Figure 2-7 *- The ATM instance model*

To conclude, GME provides a framework for creating domain-specific modeling environments (DSMEs), which allow one to define DSMLs. The GME also provides a plug-in extension mechanism for writing model compilers that can be invoked from within the GME to synthesize a model into some other form (e.g., translation to code,

refinement to a different model, or simulation scripts). The tools developed to support the research have been implemented as GME plug-ins (i.e., the transformation engine C-SAW discussed in Chapter 3, model comparison tool DSMDiff discussed in Chapter 4 and transformation testing engine M2MUnit discussed in Chapter 5). All of the DSMLs presented in this thesis are also defined and developed within the GME.

CHAPTER 3

AUTOMATED MODEL EVOLUTION

This chapter presents a transformation approach to automate model change evolution. The specific challenges and the limitations of currently available techniques are discussed before a detailed introduction to the model transformation language ECL and the associated model transformation engine C-SAW. Particularly, this chapter concentrates on the role of C-SAW in addressing model evolution concerns related to system scalability and adaptability. Two case studies are offered to illustrate how these concerns are addressed by C-SAW. In addition, to demonstrate the benefits of this approach, experimental evaluation is discussed, including modeling artifacts, evaluation metrics and experimental results. Related work and a concluding discussion are presented at the end of this chapter.

3.1 Challenges and Current Limitations

One of the benefits of modeling is the ability to explore design alternatives. To provide support for efficient design exploration, frequent change evolution is required within system models. However, the escalating complexity of software and system models is making it difficult to rapidly explore the effects of a design decision and make

changes to models correctly. Automating such exploration with model transformation can improve both productivity and quality of model evolution [Gray et al., 06].

To support automated model evolution, a model transformation language should be able to specify various types of changes that are needed for common model evolution tasks. As discussed in Chapter 1, there are two categories of changes embodied in model evolution that this research addresses: one is changes for system adaptability that crosscut the model representation's hierarchy; the other is changes for system scalability that require replication of model elements and connections. To express these changes, a model transformation language should address the challenges discussed in the following subsections.

3.1.1 Navigation, Selection and Transformation of Models

Many model evolution tasks require traversing the model hierarchy, selecting model elements and changing the model structure and properties. For example, model scalability is a process that refines a simple base model to a more complex model by replicating model elements or substructures and adding necessary connections. Such replication usually emerges in multiple locations within a model hierarchy and requires that a model transformation language provide support for model navigation and selection. Particularly, model evolution is essentially a process to manipulate (e.g., create, delete, or change) model elements and connections dynamically. A model transformation language also needs to support basic transformation operations for creating and deleting model elements or changing their properties.

Model transformation is also a specific type of computation and may be performed in a procedural style. A model transformation may contain multiple individual and reusable procedures. In addition to the above model-oriented features, a model transformation language often needs to support sequential, conditional, repetitive and parameterized model manipulation for defining control flows and enabling data communication between transformation rules. Another challenge of model evolution is many modeling concerns are spread across a model hierarchy. New language constructs are needed to improve the modularization of such concerns, as discussed in the following section.

3.1.2 Modularization of Crosscutting Modeling Concerns

Many model evolution concerns are crosscutting within the model hierarchy [Zhang et al., 07], [Gray et al., 03]. An example is the widespread data logging mechanism embodied in a data communication system [Gray et al., 04-b]. Another example is the effect of fluctuating bandwidth on the quality of service across avionics components that must display a real-time video stream [Neema et al., 02]. To evaluate such system-wide changes inside a system model, the designer must manually traverse the model hierarchy by recursively clicking on each element and then make changes to the right elements. This process is tedious and error-prone, because system models often contain hierarchies several levels deep.

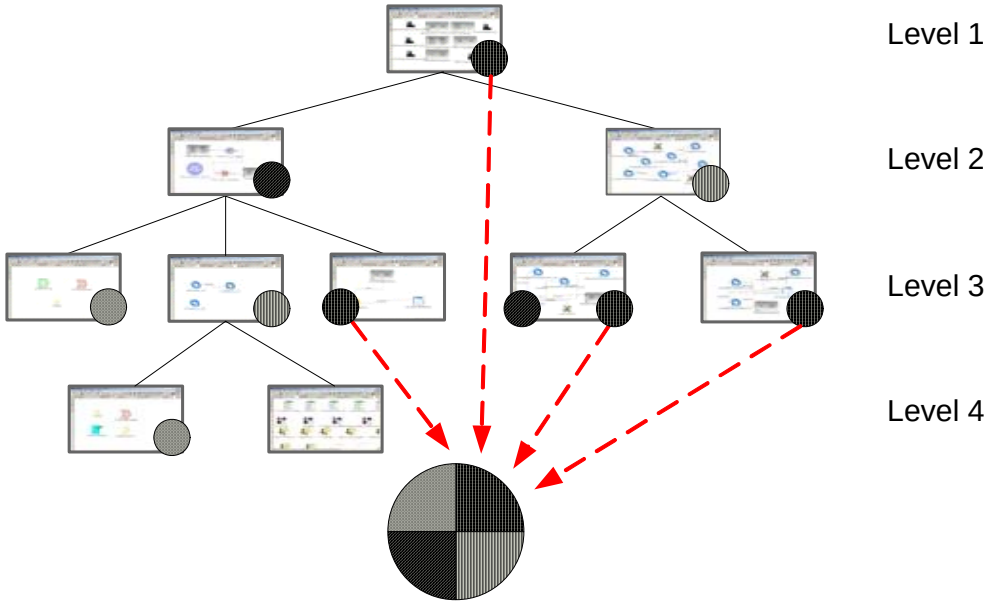


Figure 3-1 *- Modularization of crosscutting model evolution concerns*

Traditional programming languages such as Object-Oriented languages are not suitable for modularizing concerns that crosscut modules such as objects. Currently, Aspect-Oriented Software Development (AOSD) [Filman et al., 04] offers techniques to modularize concerns that crosscut system components. For example, Aspect-Oriented Programming (AOP) [Kiczales et al., 01] provides two new types of language constructs: advice, which is used to represent the crosscutting behavior; and pointcut expressions, which are used to specify the locations in the base program where the advice should be applied. An aspect is a modularization of a specific crosscutting concern with pointcut and advice definitions. Although the application of AOSD originally focused on programming languages, the community investigating aspect-oriented modeling is growing [AOM, 07]. To support modularity in specifying crosscutting modeling concerns, as shown in Figure 3-1, Aspect-Oriented constructs are needed in a model

transformation language (e.g., to specify a collection of model elements crosscutting within the model hierarchy). A desired result of such a model transformation language is to achieve modularization such that a change in a design decision is isolated to one location.

3.1.3 The Limitations of Current Techniques

For the purpose of automation, model evolution tasks can be programmed in either traditional programming languages or currently available model transformation languages. For example, many commercial and research toolsuites provide APIs to manipulate models directly. However, an API approach requires model developers to learn and use low-level tools (e.g., object-oriented languages and their frameworks) to program high-level model transformations. This emerges as a major hurdle when applying model-driven approaches in software development by end-users who are not familiar with general-purpose programming languages (GPLs).

There are a number of approaches to model transformation, such as graphical languages typified by graph grammars (e.g., GReAT [Agrawal, 03] and Fujaba [Fujaba, 07]), or a hybrid language (e.g., the ATLAS Transformation Language [Bézivin et al., 04], [Kurtev et al., 06] and Yet Another Transformation Language [Patrascoiu, 04]). However, these model transformation approaches aim to solve model transformation where the source model and target model belong to different metamodels, so their languages have complicated syntax and semantics, and additional mapping techniques between different metamodels are embodied in these approaches. Thus, a more lightweight language that aims to solve endogenous transformation where the source

model and target model belong to the same metamodel is more suitable to address the model evolution problems identified in this research.

3.2 The Embedded Constraint Language (ECL)

The model transformation approach developed in this research offers a textual transformation language called the Embedded Constraint Language (ECL). The earlier version of ECL was derived from Multigraph Constraint Language (MCL), which is supported in the GME to stipulate specific semantics within the domain during the creation of a domain's metamodeling paradigm [Gray, 02]. Similar to MCL, ECL is an extension of the OCL [Warmer and Kleppe, 99], which complements the industry standard UML by providing a language to write constraints and queries over object models. Thus, ECL has concepts and notations that are familiar to model engineers. The ECL also takes advantage of model navigation features from OCL to provide declarative constructs for automatic selection of model elements.

Originally, ECL was designed to address crosscutting modeling concerns where transformations were executed outside the GME by modifying the XML representation of models [Gray et al., 01], [Gray, 02]. A preliminary contribution of this dissertation was to enrich the language features of ECL and to adapt its interface to work as a plug-in within GME (i.e., the transformations are now performed within GME, not outside of GME through XML). Several criteria influenced the design of ECL, including:

- **The language should be small but powerful.** The primary goal for the design of a transformation specification language should allow users to describe a transformation using concepts from their own domain and modeling environment. Although there is

a tradeoff between conciseness and comprehension, the key to the design of a transformation language is a set of core abstractions that are intuitive and cover the largest possible range of situations implied by current modeling practice. To achieve this goal, a transformation language should consist of a small set (ideally a minimized set) of concepts and constructs, but be powerful enough to express a complete set of desired transformation activities such that the expressiveness of the language is maximized.

- **The language should be specific to model transformation.** A transformation-specific language needs to have full power to specify all types of modeling objects and transformation behaviors, including model navigation, model selection and model modification. Such a language should provide specific constructs and mechanisms for users to describe model transformations. This requires both a robust type system and a set of functionally rich operators. In other words, a transformation language needs to capture all the features of the model transformation domain.

As a result of these desiderata, ECL is implemented as a simple but highly expressive language for model engineers to write transformation specifications. The ECL provides a small but robust type system, and also provides features such as collection and model navigation. A set of operators are also available to support model aggregation and connections. In general, the ECL supports an imperative transformation style with numerous operations that can alter the state of a model.

3.2.1 ECL Type System

ECL currently provides a basic type system to describe values and model objects that appear in a transformation. The data types in ECL include the primitive data types (e.g., *boolean*, *integer*, *real* and *string*), the model object types (e.g., *atom, model, object* and *connection*) and the collection types (e.g., *atomList, modelList, objectList* and *connectionList*). The data types are explicitly used in parameter definition and variable declaration. These types are new additions to the earlier version of ECL described in [Gray, 02].

3.2.2 ECL Operations

ECL provides various operations to support model navigation, selection and transformation. These operations are described throughout this subsection.

Model collection

The manipulation of a collection of modeling elements is a common task for model transformation. In DSM, there exist modeling elements that have common features and can be grouped together. Model manipulations (e.g., navigations and evaluations) are often needed to be performed on such a collection of models. The concrete type of a collection in ECL is a bag, which can contain duplicate elements. An example operator for a collection is `size()`, which is similar to the OCL operator that returns the number of elements in a collection.

All operations on collections are denoted in an ECL expression by an arrow (`->`). This makes it easy to distinguish an operation of a model object type (denoted as a

period) from an operation on a collection. In the following statement, the select operation following the arrow is applied to the collection (i.e., the result of the atoms operation) before the arrow, and the size operator appears in the comparison expression.

```
atoms()->select(a | a.kindOf() == "Data")->size() >= 1
```

The above expression selects all the atoms, whose kind is “Data,” from the current model and determines whether the number of such atoms is equal to or greater than 1.

Model selection and aggregation

One common activity during model transformation is to find elements in a model. There are two different approaches to locating model elements. The first approach - querying - evaluates an expression over a model, returning those elements of the model for which the expression holds. The other common approach uses pattern matching where a term or a graph pattern containing free variables is matched against the model. Currently, ECL supports model queries primarily by providing the `select()` operator. Other operators include model aggregation operators to select a collection of objects (e.g. `atoms()`), and a set of operators to find a single object (e.g., `findModel(“aModelName”)`).

The `select(expression)` operator is frequently used in ECL to specify a selection from a source collection, which can be the result of previous operations and navigations. The result of the select operation is always a subset of the original collection. In addition, model aggregation operators can also be used to perform model querying. For example, `models(expression)` is used to select all the submodels that satisfy the constraint specified by the expression. Other aggregation operators include

`atoms(expression)`, `connections(expression)`, `source()` and `destination()`. Specifically, `source()` and `destination()` are used to return the source object and the destination object in a connection. Another set of operators are used to obtain a single object (e.g., `findAtom()` and `findModel()`). The following ECL uses a number of operators just mentioned:

```
rootFolder().findFolder("ComponentTypes").models()->
      select(m|m.name().endWith("Impl"))->AddConcurrency();
```

First, `rootFolder()` returns the root folder of a modeling project. Next, `findFolder()` is used to return a folder named “ComponentTypes” under the root folder. Then, `models()` is used to find all the models in the “ComponentTypes” folder. Finally, the `select()` operator is used to select all the models that match the predicate expression (i.e., those models whose names end with “Impl”). The `AddConcurrency` strategy is then applied to the resulting collection. The concept of a strategy is explained in Section 3.2.3.

Transformation operations

ECL provides basic transformation operations to add model elements, remove model elements and change the properties of model elements. Standard OCL does not provide such capabilities because it does not allow side effects on a model. However, a transformation language should be able to alter the state of a model. ECL extends the standard OCL by providing a series of operators for changing the structure or constraints of a model. To add new elements (e.g., a model, atom or connection) to a model, ECL provides such operators as `addModel()`, `addAtom()` and `addConnection()`.

Similarly, to remove a model, atom or connection, there are operators like `removeModel()`, `removeAtom()` and `removeConnection()`. To change the value of any attribute of a model element, `setAttribute()` can be used.

3.2.3 The Strategy and Aspect Constructs

There are two kinds of modular constructs in ECL: *strategy* and *aspect*, which are designed to provide aspect-oriented capabilities in specifying crosscutting modeling concerns. A strategy is used to specify elements of computation and the application of specific properties to the model entities (e.g., adding model elements). A modeling aspect is used to specify a crosscutting concern across a model hierarchy (e.g., a collection of model elements that cut across a model hierarchy).

In general, an ECL specification may consist of one or more strategies, and a strategy can be called by other strategies. A strategy call implements the binding and parameterization of the strategy to specific model entities. The context of a strategy call can be an entire project; a specific model, atom, or connection; or a collection of assembled modeling elements that satisfy a predicate. The aspect construct in ECL is used to specify such a context. Examples of ECL aspect and strategy are shown in Listing 3-1.

```
1  aspect FindData1(atomName, condName, condExpr : string)
2  {
3        atoms()->select(a | a.kind() == "Data" and a.name() == "data1")->
4              AddCond("Data1Cond", "value<200");
5  }
6
7  strategy AddCond(condName, condExpr : string)
8  {
9        declare p : model;
10       declare data, pre : atom;
11
12       data := self;
13       p := parent();
14
15       pre:=p.addAtom("Condition", condName);
16       pre.setAttribute("Kind", "PreCondition");
17       pre.setAttribute("Expression", condExpr);
18       p.addConnection("AddCondition", pre, data);
19 }
```

Listing 3-1 - *Examples of ECL aspect and strategy*

The `findData1` aspect selects a set of `data` atoms named “data1” from all the atoms and then a strategy called `AddCond` is applied to the selected atoms, which adds a `Condition` atom for each of the selected atoms (Line 15) and creates a connection between them (Line 18). The `AddCond` strategy also sets values to the attributes of each created `Condition` atom (Line 16 to 17).

With *strategy* and *aspect* constructs, ECL offers the ability to explore numerous modeling scenarios by considering crosscutting modeling concerns as aspects that can be rapidly inserted and removed from a model. This permits a model engineer to make changes more easily to the base model without manually visiting multiple locations.

3.2.4 The Constraint Specification Aspect Weaver (C-SAW)

The ECL is fully implemented within a model transformation engine called the Constraint-Specification Aspect Weaver (C-SAW)[2]. Originally, C-SAW was designed to address crosscutting modeling concerns [Gray, 02], but has evolved into a general model transformation engine to perform different modeling tasks. As shown in Figure 3-2, C-SAW executes the ECL transformation specification on a set of source models. One or more source models, together with transformation specifications, are taken as input to the underlying transformation engine. By executing the transformation specification, C-SAW weaves additive changes into source models to generate the target model as output.

Inside C-SAW there are two core components that relate to the ECL: the parser and the interpreter. The parser is responsible for generating an abstract syntax tree (AST) of the ECL specification. The interpreter then traverses this generated AST from top to bottom, and performs a transformation by using the modeling APIs provided by GME. Thus, the accidental complexities of using the low-level details of the API are made abstract in the ECL to provide a more intuitive representation for specifying model transformations. When model transformation is performed, additive changes can be applied to base models, leading to structural or behavioral changes that may crosscut multiple boundaries of the model. Section 3.3 provides an example where C-SAW serves as a model replicator to scale up models. The other example is presented in Section 3.4 to illustrate the capability of C-SAW to address crosscutting modeling concerns.

[2] The name C-SAW was selected due to its affinity to an aspect-oriented concept – a crosscutting saw, or csaw, is a carpenter's tool that cuts across the grain of wood.

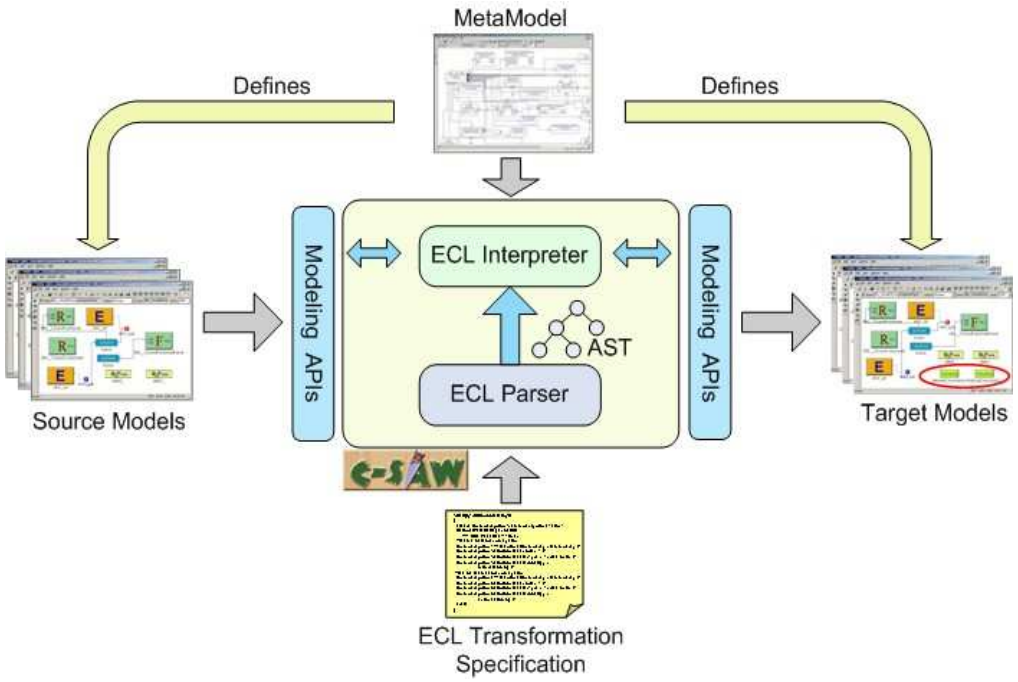


Figure 3-2 *- Overview of C-SAW*

3.2.5 Reducing the Complexities of Transforming GME Models

GME provides APIs in C++ and Java called the Builder Object Network (BON) to manipulate models as an extension mechanism for building model compilers (e.g., plug-in and add-on components) that supply an ability to alter the state of models or generate other software artifacts from the models. However, using BON to specify modeling concerns such as model navigation and querying introduces accidental complexities because users have to deal with the low-level details of the modeling APIs in C++ or Java. Listing 3-2 shows the code fragment for finding a model from the root folder using BON APIs in C++:

```
1  CbuilderFolder *rootFolder, string: modelName
2  CBuilderModel *result = null;
3  const CBuilderModelList *subModels;
4  subModels = ((CBuilderFolder *)rootFolder)->GetRootModels();
5  POSITION POS = subModels->GetHeadPosition();
6  while(POS){
7        CBuilderModel *subModel = subModels->GetNext(POS);
8        if(subModel->GetName() == modelName){
9              result = subModel;
10             return result;
11       }
12 }
13 return result;
```

Listing 3-2 – *Example C++ code to find a model from the root folder*

The ECL provides a more intuitive and high-level representation for specifying modeling concerns (e.g., model navigation, querying and transformation) such that the low-level details are made abstract. For example, to find a model from the root folder, the following ECL can be compared to the C++ code provided previously:

```
rootFolder().findModel("aModelName");
```

3.3 Model Scaling with C-SAW

In MDE, it is often desirable to evaluate different design alternatives as they relate to scalability issues of the modeled system. A typical approach to address scalability is model replication, which starts by creating base models that capture the key entities as model elements and their relationships as model connections. A collection of base models can be adorned with necessary information to characterize a specific scalability concern as it relates to how the base modeling elements are replicated and connected together. In current modeling practice, such model replication is usually accomplished by scaling the base model manually. This is a time-consuming process that represents a source of error, especially when there are deep interactions between model components. As an alternative

to the manual process, this section presents the idea of automated model replication through a C-SAW model transformation process that expands the number of elements from the base model and makes the correct connections among the generated modeling elements. The section motivates the need for model replication through a case study.

3.3.1 Model Scalability

One of the benefits of modeling is the ability to explore design alternatives. A typical form of design exploration involves experimenting with model structures by expanding different portions of models and analyzing the result on scalability [Lin et al., 07-a], [Gray et al., 05]. For example, a high-performance computing system may be evaluated when moving from a few computing nodes to hundreds of computing nodes. Model scalability is defined as the ability to build a complex model from a base model by replicating its elements or substructures and adding the necessary connections. To support model scalability requires extensive support from the host modeling tool to enable rapid change evolution within the model representation [Gray et al., 06]. However, it is difficult to achieve model scalability in current modeling practice due to the following challenges:

(1) Large-scale system models often contain many modeling elements: In practice, models can have multiple thousands of coarse-grained components. As discussed in Section 1.3.1, modeling these components using traditional manual model creation techniques and tools can approach the limits of the effective capability of an engineer. For example, the models of a DRE system consist of

several thousand instances from a set of types defined in a metamodel, which leads to their larger size and nested hierarchy.

(2) Manually scaling up models is laborious, time consuming and prone to errors: To examine the effect of scalability on a system, the size of a system model (e.g., the number of the participant model elements and connections) needs to be increased or decreased frequently. The challenges of scalability affect the productivity of the modeling process, as well as the correctness of the model representation. As an example, consider a base model consisting of a few modeling elements and their corresponding connections. To scale a base model to hundreds, or even thousands of duplicated elements would require a lot of mouse clicking and typing within the associated modeling tool [Gray et al., 06]. Furthermore, the tedious nature of manually replicating a base model may also be the source of many errors (e.g., forgetting to make a connection between two replicated modeling elements). A manual process to replication significantly hampers the ability to explore design alternatives within a model (e.g., after scaling a model to 800 modeling elements, it may be desired to scale back to only 500 elements, and then back up to 700 elements, in order to understand the impact of system size).

To address these challenges, the research described in this dissertation makes a contribution to model scalability by using a model transformation approach to automate replication[3] of base models. A transformation for model replication is called a replicator,

[3] The term "replication" has specific meaning in object replication of distributed systems and in database replication. In the context of this thesis, the term is used to refer to the repetition of modeling elements or structures among models to address scalability concerns.

which changes a model to address scalability concerns. In this approach, large-scale system models are automatically created from smaller, baseline specification models by applying model transformation rules that govern the scaling and replication behavior associated with stepwise refinement of models [Batory, 06].

3.3.2 Desired Characteristics of a Replication Approach

An approach that supports model scalability through replication should have the following desirable characteristics: 1) retains the benefits of modeling, 2) is general across multiple modeling languages, and 3) is flexible to support user extensions. Each of these characteristics (C1 through C3) is discussed further in this subsection.

C1. Retains the benefits of modeling: The power of modeling comes from the opportunity to explore various design alternatives and the ability to perform analysis (e.g., model checking and verification of system properties [Hatcliff et al., 03]) that would be difficult to achieve at the implementation level, but easier at the modeling level. Thus, a model replication technique should not perform scalability in such a way that analysis and design exploration is inhibited. This seems to be an obvious characteristic to desire, but we have observed replication approaches that remove these fundamental benefits of modeling.

C2. General across multiple modeling languages: A replication technique that is generally applicable across multiple modeling languages can leverage the effort expended in creating the underlying transformation mechanism. A side benefit of such generality is that a class of users can become familiar with a common replicator technique, which can be applied to many modeling languages.

C3. Flexible to support user extensions: Often, large-scale system models leverage architectures that are already well-suited toward scalability. Likewise, the modeling languages that specify such systems may embody similar patterns of scalability, and may lend themselves favorably toward a generative and reusable replication process. Further reuse can be realized if the replicator supports multiple types of scalability concerns in a templatized fashion (e.g., the name, type, and size of the elements to be scaled are parameters to the replicator). The most flexible type of replication would allow alteration of the semantics of the replication more directly using a language that can be manipulated easily by an end-user. In contrast, replicator techniques that are hard-coded restrict the impact for reuse.

3.3.3 Existing Approaches to Support Model Replication

As observed, there are two techniques that represent approaches to model replication used in common practice: 1) an intermediate phase of replication within a model compiler, 2) a domain-specific model compiler that performs replication for a particular modeling language.

A1. Intermediate stage of model compilation: A model compiler translates the representation of a model into some other artifacts (e.g., source code, configuration files, or simulation scripts). As a model compiler performs its translation, it typically traverses an internal representation of the model through data structures and APIs provided by the host modeling tool (e.g., the BON offered by GME). One of our earlier ideas for scaling large models considered performing the

replication as an intermediate stage of the model compiler. Prior to the generation phase of the compilation, the intermediate representation can be expanded to address the desired scalability. This idea is represented in Figure 3-3, which shows the model scaling as an internal task within the model compiler that directly precedes the artifact generation.

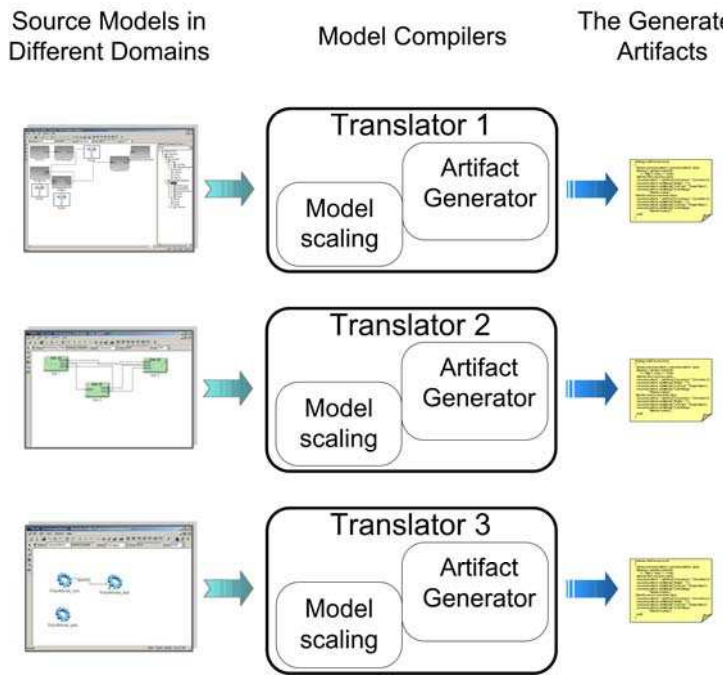


Figure 3-3 *- Replication as an intermediate stage of model compilation (A1)*

This approach is an inadequate solution to replication because it violates all three of the desired characteristics enumerated in Section 3.3.2. The most egregious violation is that the approach destroys the benefits of modeling. Because the replication is performed as a pre-processing phase in the model compiler, the replicated structures are never rendered back into the modeling tool itself to produce scaled models such that model engineers can further analyze the model scaling results. Thus, analysis and design alternatives are not made available to a model

engineer who wants to further evaluate the scaled models. Additionally, the pre-processing rules are hard-coded into the model compiler and intermixed with other concerns related to artifact generation. This coupling offers little opportunity for reuse in other modeling languages. In general, this is the least flexible of all approaches that we considered.

A2. Domain-specific model compiler to support replication: This approach to model scalability constructs a model compiler that is capable of replicating the models as they appear in the tool such that the result of model scaling is available to the end-user for further consideration and analysis. Such a model compiler has detailed knowledge of the specific modeling language, as well as the particular scalability concern. Unlike approach A1, this technique preserves the benefits of modeling because the end result of the replication provides visualization of the scaling, and the replicated models can be further analyzed and refined. Figure 3-4 illustrates the domain-specific model replicator approach, which separates the model scaling task from the artifact generator in order to provide end-users an opportunity to analyze the scaled models. However, this approach also has a few drawbacks. Because the replication rules are hard-coded into the domain-specific model replicator, the developed replicator has limited use outside of the intended modeling language. Thus, the generality across modeling languages is lost.

These first two approaches have drawbacks when compared against the desired characteristics of Section 3.3.2. The next section presents a more generalized solution based on C-SAW and ECL.

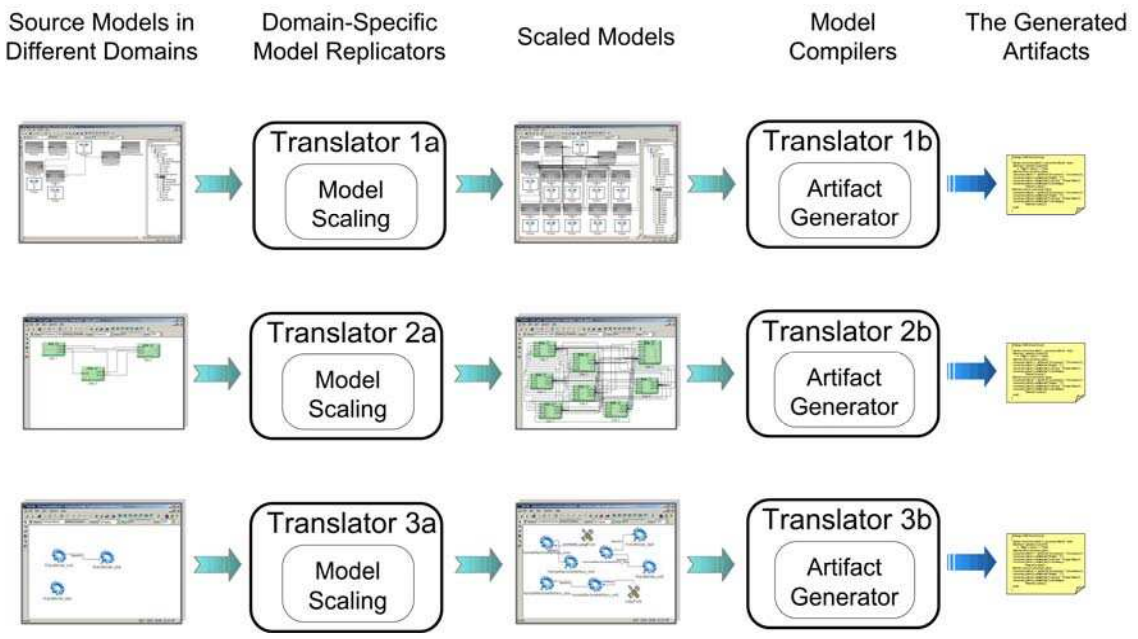


Figure 3-4 *- Replication as a domain-specific model compiler (A2)*

3.3.4 Replication with C-SAW

A special type of model compiler within the GME is a plug-in that can be applied to any metamodel (i.e., it is domain-independent). The C-SAW model transformation engine is an example of a plug-in that can be applied to any modeling language. The type of transformations that can be performed by C-SAW are endogenous transformations where the source and target models are defined by the same metamodel. C-SAW executes as a model compiler and renders all transformations (as specified in the ECL) back into the host modeling tool. A model transformation written in ECL can be altered very rapidly to analyze the effect of different degrees of scalability (e.g., the effect on performance when the model is scaled from 256 to 512 nodes).

This third approach to replication (designated as A3) advocates the use of a model transformation engine like C-SAW to perform the replication (please see Figure 3-5 for an overview of the technique). This technique satisfies all of the desirable characteristics of a replicator: by definition, the C-SAW tool is applicable across many different modeling languages, and the replication strategy is decoupled from other concerns (e.g., artifact generation) and specified in a way that can be easily modified through a higher level transformation language. These benefits improve the capabilities of hard-coded rules as observed in the approaches described in A1 and A2. With a model transformation engine, a second model compiler is still required for each domain as in A2 (see "Model Compilers" in Figure 3-4), but the scalability issue is addressed independently of the modeling language.

The key benefits of approach A3 can be seen by comparing it to A2. It can be observed that Figures 3-4 and 3-5 are common in the two-stage process of model replication followed by artifact generation with a model compiler. The difference between A2 and A3 can be found in the replication approach. In Figure 3-4, the replication is performed by three separate model compilers that are hard-coded to a specific domain (Translator 1a, Translator 2a, and Translator 3a), but the replication in Figure 3-5 is carried out by a single model transformation engine that is capable of performing replication on any modeling language. Approach A3 provides the benefit of a higher level scripting language that can be generalized through parameterization to capture the intent of the replication process. Our most recent efforts have explored this third technique for model replication on several existing modeling languages.

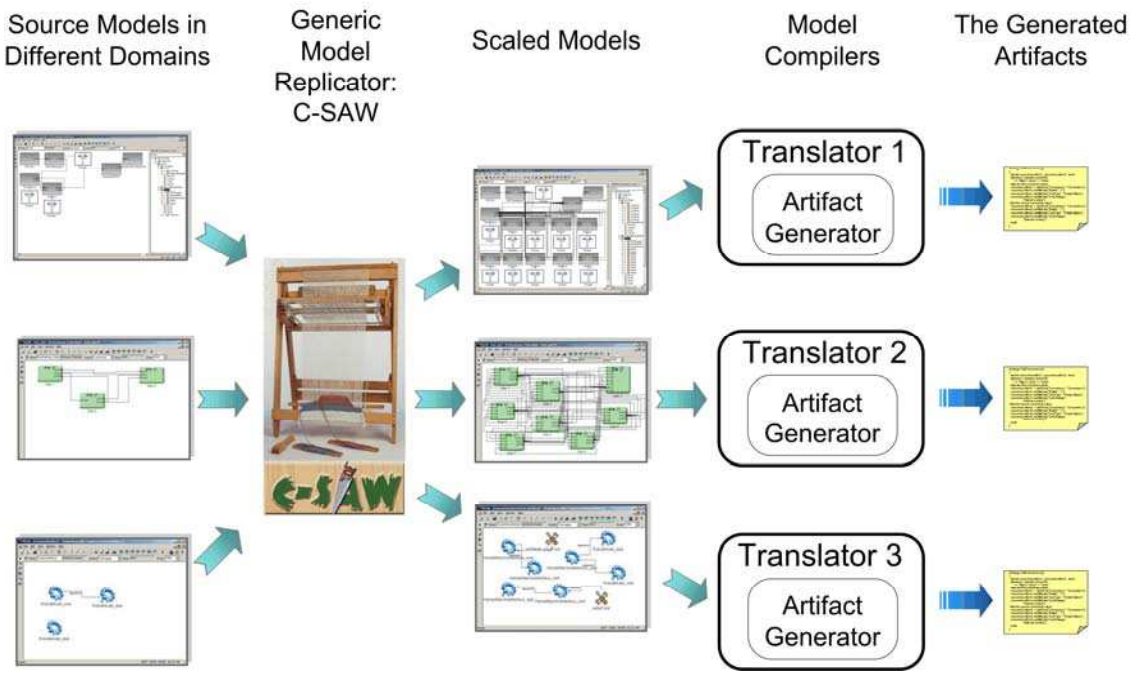


Figure 3-5 *- Replication using the model transformation engine C-SAW (A3)*

3.3.5 Scaling the System Integration Modeling Language (SIML)

In this section, the concept of model replication is demonstrated on an example modeling language that was created in the GME for the computational physics domain. The purpose of introducing this case study is to illustrate how our model transformation approach supports scalability among SIML models that contain multiple hierarchies.

The System Integration Modeling Language (SIML) is a modeling language developed to specify configurations of large-scale fault tolerant data processing systems used to conduct high-energy physics experiments [Shetty et al., 05]. SIML was developed by Shetty et al. from Vanderbilt University to model a large-scale real-time physics system developed at Fermi National Accelerator Laboratory (FermiLab) [Fermi, 07] for characterizing the subatomic particle interactions that take place in a high-energy physics

experiment. SIML models capture system structure, target system resources, and autonomic behavior. System generation technology is used to create the software from these models that implement communication between components with custom data type marshalling and demarshalling, system startup and configuration, fault tolerant behavior, and autonomic procedures for self-correction [Shetty et al., 05].

A system model expressed in SIML captures components and relationships at the systems engineering level. The features of SIML are hierarchical component decomposition and dataflow modeling with point-to-point and publish-subscribe communication between components. There are several rules defined by the SIML metamodel:

- A *system* model may be composed of several independent regions
- Each *region* model may be composed of several independent local process groups
- Each *local process group* model may include several primitive application models
- Each system, region, and local process group must have a representative *manager* that is responsible for mitigating failures in its area

A *local process group* is a set of processes that run the set of critical tasks to perform the system's overall function. In a data processing network, a local process group would include the set of processes that execute the algorithmic and signal processing tasks, as well as the data processing and transport tasks. A *region* is simply a collection of local process groups, and a *system* is defined as a collection of regions and possibly other supporting processes. These containment relationships lead to the multiple hierarchical structures of SIML models. A simple SIML base model is shown on the left side of Figure 3-6, which captures a system composed of one region and one local process group

in that region (shown as an expansion of the parent region), utilizing a total of 15 physical modeling elements (several elements are dedicated to supporting applications not included in any region).

Scalability Issues in SIML: In order to plan, deploy, and refine a high-energy physics data processing system, designers manually build a multitude of different SIML models that are subject to a variety of outside and changing constraints (e.g., the current availability of hardware, software, or human resources). An example of this process would be the manual creation of separate 16-, 32-, 64-, and 128-node versions of a baseline system used for bandwidth and latency testing purposes. This would later be followed by the creation of a set of significantly larger SIML models where the final system model could incorporate as many as 2,500 local processing groups. Each of these models would undergo a variety of analysis routines to determine several key properties of the system. These analyses include system throughput, network/resource utilization and worst-case managerial latency (the latency between managers and subordinates is crucial in evaluating the fault tolerance of the system). The results of these analyses may vary greatly as the structure of the system model is scaled in different ways.

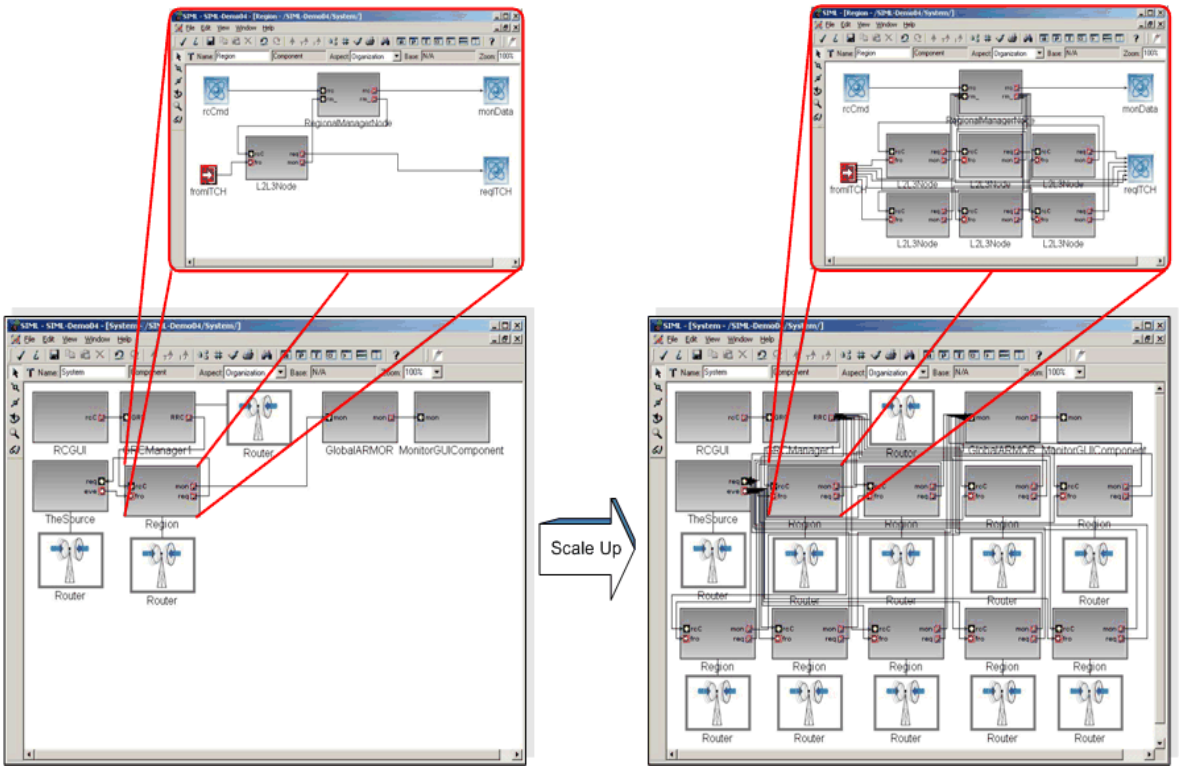


Figure 3-6 *- Visual Example of SIML Scalability*

The number of system configurations that are created using this process is directly proportional to the time and effort allowed by system designers to create valid system models using a manual approach. In practice, SIML models have been scaled to 32- and 64-node models. However, the initial scaling in these cases was performed manually. The ultimate goal of the manual process was to scale to 2,500 nodes. After 64 nodes, it was determined that scaling to further nodes would be too tedious to perform without proper automation through improved tool support. Even with just a small expansion, the manual application of the same process would require an extraordinary amount of manual effort (e.g., much mouse-clicking and typing) to bring about the requisite changes, and increase the potential for introducing error into the model (e.g., forgetting to add a required connection). If the design needs to be scaled forward or backward, a manual approach

would require additional effort that would make the exploration and analysis of design alternatives impractical. Therefore, a significant victory for design agility can be claimed if replicators can be shown to scale a base SIML model quickly and correctly into a variety of larger and more elaborate SIML models. This case study shows the benefits that result from scaling SIML models by applying an automated approach that exhibits the desired characteristics for replicators.

To decide what scaling behaviors such a replicator needs to perform, domain-specific knowledge and rules for creating a SIML model as embodied in its metamodel need to be captured. For example, there are one-to-many relationships between system and regional managers, and also one-to-many relationships between regional and local process group managers. These relationships are well-defined. Because the pattern of these relationships is known, it is feasible to write a replicator to perform automatic generation of additional local process groups and/or regions to create larger and more elaborate system models.

In general, scaling up a system configuration using SIML can involve: 1) an increase in the number of regions, 2) an increase in the number of local process groups per region, or 3) both 1 and 2. Considering the SIML model in Figure 3-6, the system (which originally has one region with one local process group) is increased to nine regions with six local process groups per region. Such replication involves the following tasks:

- Replication of the local process group models
- Replication of the entire region models and their contents

- Generation of communication connections between the regional managers and newly created local managers
- Generation of additional communication connections between the system manager and new regional manager processes

The scaled model is shown in the right side of Figure 3-6. This example scales to just 9 regions and 6 nodes per region simply because of the printed space to visualize the figure.

ECL Transformation to Scale SIML: The scalability shown in Figure 3-6 can be performed by a replicator, which is a model transformation specification in ECL as shown in Listing 3-3. As a point of support for the effectiveness of replicators as transformations, this ECL specification was written in less than an hour by a user who was very familiar with ECL, but had studied the SIML metamodel for less than a few hours.

The ECL transformation specification is composed of an aspect and several strategies. In Listing 3-3, the aspect `Start` (Line 1) invokes two strategies, `scaleUpNode` and `scaleUpRegion` in order to replicate the local process group node (i.e., `L2L3Node`) within the region model and the region itself. The strategy `scaleUpNode` (Line 7) discovers the "Region" model, sets up the context for the transformation, and calls the strategy `addNode` (Line 12) that will recursively increase the number of local process group nodes. The new node instance is created on Line 18, which is followed by the construction of the communication connections between ports, regional managers and the newly created nodes (Line 21 to Line 23). Some other connections are omitted here for the sake of keeping the listing concise. Two other

strategies, `scaleUpRegion` (Line 29) and `addRegion` (Line 34), follow a similar mechanism.

The process of modeling systems using SIML illustrates the benefits of replicators by providing an automated technique that uses transformations to scale models in a concise and flexible manner. Because of the multiple hierarchies of SIML models, replications usually need to be performed on all the elements associated with containment relationships within a model. To perform a scaling task across multiple model hierarchies, ECL supports model navigation through its model querying and selection operations. A model navigation concern can be specified concisely in the ECL. For example, Line 9 of Listing 3-3 is a declarative statement for finding all the region models by navigating from the root folder to the system model, which calls these three querying operations: `rootFolder()`, `findFolder()` and `findModel()`.

Also, flexibility of the replicator can be achieved in several ways. Lines 3 and 4 of Listing 3-3 specify the magnitude of the scaling operation, as well as the names of the specific nodes and regions that are to be replicated. In addition to these parametric changes that can be made easily, the semantics of the replication can be changed because the transformation specification can be modified directly by an end-user. This is not the case in approaches A1 and A2 from Section 3.3.3 because the replication semantics are hard-coded into the model compiler.

```
14 aspect Start()
15 {
16   scaleUpNode("L2L3Node", 5); //add 5 L2L3Nodes in the Region
17   scaleUpRegion("Region", 8); //add 8 Regions in the System
18 }
19
20 strategy scaleUpNode(node_name : string; max : integer)
21 {
22    rootFolder().findFolder("System").findModel("Region").addNode(node_name,max,1);
23 }
24
25 strategy addNode(node_name, max, idx : integer)           //recursively add nodes
26 {
27   declare node, new_node, input_port, node_input_port : object;
28
29   if (idx<=max) then
30      node := rootFolder().findFolder("System").findModel(node_name);
31      new_node := addInstance("Component", node_name, node);
32
33      //add connections to the new node; three similar connections are omitted here
34      input_port := findAtom("fromITCH");
35      node_input_port := new_node.findAtom("fromITCH");
36      addConnection("Interaction", input_port, node_input_port);
37
38      addNode(node_name, max, idx+1);
39   endif;
40 }
41
42 strategy scaleUpRegion(reg_name : string; max : integer)
43 {
44    rootFolder().findFolder("System").findModel("System").addRegion(reg_name,max,1);
45 }
46
47 strategy addRegion(region_name, max, idx : integer)      //recursively add regions
48 {
49   declare region, new_region, out_port, region_in_port, router, new_router
50      : object
51   if (idx<=max) then
52      region := rootFolder().findFolder("System").findModel(region_name);
53      new_region := addInstance("Component", region_name, region);
54
55      //add connections to the new region; four similar connections are omitted here
56      out_port := findModel("TheSource").findAtom("eventData");
57      region_in_port := new_region.findAtom("fromITCH");
58      addConnection("Interaction", out_port, region_in_port);
59
60      //add a new router and connect it to the new region
61      router := findAtom("Router");
62      new_router := copyAtom(router, "Router");
63      addConnection("Router2Component", new_router, new_region);
64
65      addRegion(region_name, max, idx+1);
66   endif;
67 }
```

Listing 3-3 *- ECL specification for SIML scalability*

In addition to the examples discussed in this section, replication strategies have also been developed for the Event Quality Aspect Language (EQAL) and Stochastic Reward Net Modeling Language (SRNML). EQAL has been used to configure a large

collection of federated event channels for mission computing avionics applications. Replication within EQAL was reported in [Gray et al., 06]. SRNML has been used to describe performability concerns of distributed systems built from middleware patterns-based building blocks. Replication within SRNML was reported in [Lin et al., 07-a]. These two case studies are presented in Appendix C.

To conclude, replicating a hierarchical model requires that a model transformation language like ECL provide the capability to traverse models, the flexibility to change the scale of replication, and the computational power to change the data attributes within a replicated structure.

3.4 Aspect Weaving with C-SAW

When a concern spreads across a model hierarchy, a model is difficult to comprehend and change. Currently, the most prominent work in aspect modeling concentrates on notational aspects for UML [France et al., 04], but tools could also provide automation using AOSD principles. Originally, one motivation for developing C-SAW was the need to specify constraints that crosscut the model of a distributed real-time embedded system [Gray et al., 01]. In the initial stage of this research, C-SAW was used in weaving crosscutting changes into the Embedded System Modeling Language (ESML), which is introduced in the next section.

3.4.1 The Embedded Systems Modeling Language

The Embedded Systems Modeling Language (ESML), designed by the Vanderbilt DARPA MoBIES team, is a domain-specific graphical modeling language for modeling

real-time mission computing embedded avionics applications [Shetty et al., 05]. ESML has been defined within the GME and provides the following modeling categories to allow representation of an embedded system: a) Components, b) Component Interactions, and c) Component Configurations.

Bold Stroke is a product-line architecture written in several million lines of C++ that was developed by Boeing to support families of mission computing avionics applications for a variety of military aircraft [Sharp, 00]. It is a very complex framework with several thousand components implemented in over three million lines of source code. There are over 20 representative ESML projects for all of the Bold Stroke usage scenarios that have been defined by Boeing. For each specific scenario within Bold Stroke, the components and their interactions are captured by an event channel that is specified by an ESML model.

There are a number of crosscutting model properties in ESML models, as shown in Figure 3-7. The top of Figure 3-7 shows the interaction among components in a mission-computing avionics application modeled in the ESML. The model illustrates a set of avionics components (Global Positioning Satellite and navigational display components, for example) that collaborate to process a video stream that provides a pilot with real-time navigational data. The middle of the figure shows the internal representation of two components, which reveals the data elements and other constituents intended to describe the infrastructure of component deployment and the distribution middleware. The infrastructure implements an event-driven model, in which components update and transfer data to each other through event notification and callback methods.

Among the components in Figure 3-7 are a concurrency atom and two data atoms (circled). Each of these atoms represents a system concern that spreads across the model hierarchy. The concurrency atom (in red circle with *) identifies a system property that corresponds to the synchronization strategy distributed across the components. The collection of atoms (in blue circle with #) defines the recording policy of a black-box flight data recorder. Some data elements also have an attached precondition (in green circle with @) to assert a set of valid values when a client invokes the component at run-time.

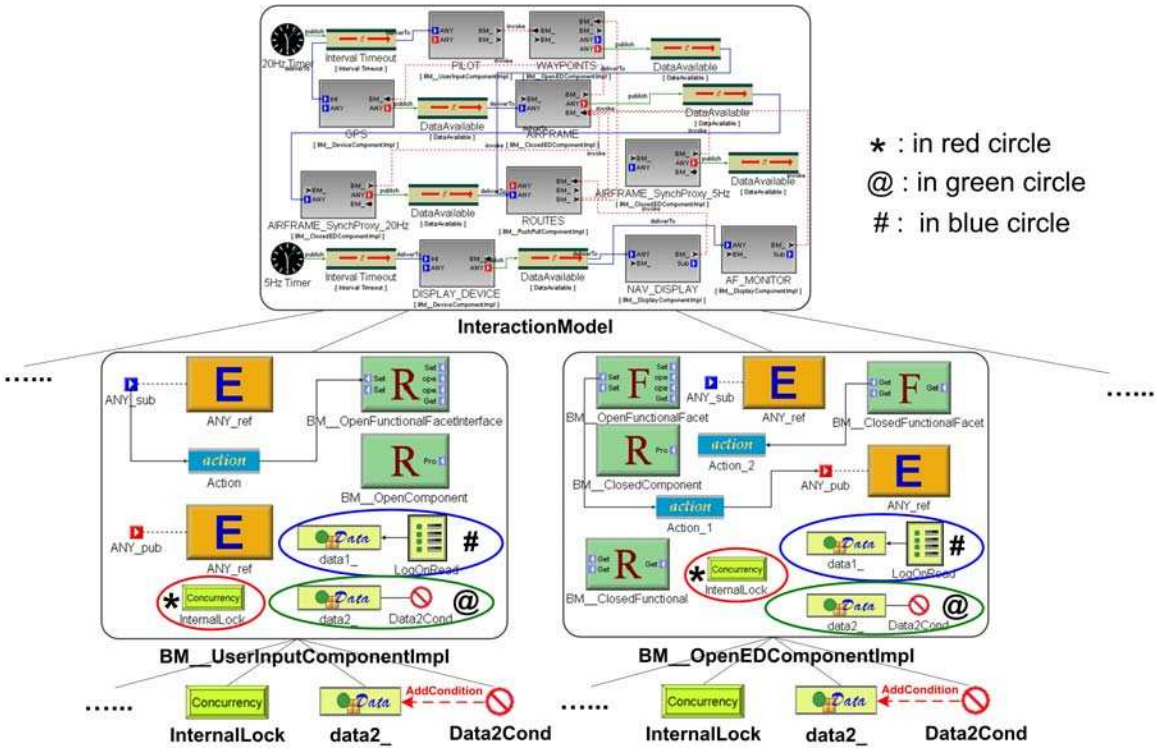


Figure 3-7 *- A subset of a model hierarchy with crosscutting model properties. Concerns related to synchronization (in red circle with *), black-box data recording (in blue circle with #), and preconditions (in green circle with @) are scattered across many submodels.*

To analyze the effect of an alternative design decision manually, model engineers must change the synchronization or flight data recorder policies, which requires making the change manually at each component's location. The partial system model in Figure 3-7 is a subset of an application with more than 6,000 components. Manually changing a policy will strain the limits of human ability in a system that large. With ECL, model engineers simply define a modeling aspect to specify the intention of a crosscutting concern. An example is given in the following subsection.

3.4.2 Weaving Concurrency Properties into ESML Models

There are several locking strategies available in Bold Stroke to address concurrency control (e.g., Internal Locking and External Locking). Internal Locking requires the component to lock itself when its data are modified, and External Locking requires the user to acquire the component's lock prior to any access of the component. However, existing Bold Stoke component models lack the ability to modularize and specify such concurrency strategies. Figure 3-8 illustrates the ESML modeling capabilities for specifying the internal configuration of a component. The "BM_ClosedEDComponentImpl" is shown in this figure. For this component, the facet/receptacle descriptors and event types are specified, as well as internal data elements, but it does not have any elements that represent the concurrency mechanisms. A model transformation may be used to weave concurrency representations into these ESML component models. This transformation task can be described as follows:

> *Insert two concurrency atoms (one internal and one external lock) to each model that has at least one data atom.*

To perform this transformation manually into over 20 existing component models will be time consuming and susceptible to errors. However, C-SAW can automate model evolution task based on an ECL transformation specification.

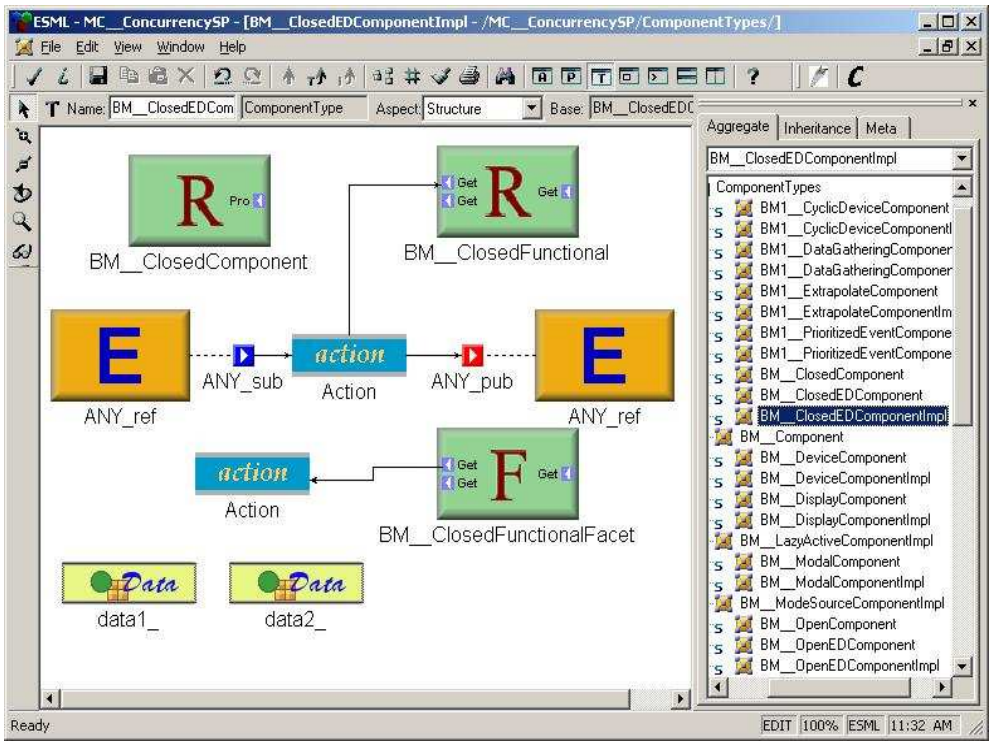


Figure 3-8 - *Internal representation of a Bold Stroke component*

To weave concurrency atoms into existing component models, an ECL transformation specification is defined as shown in Listing 3-4. The `Start` aspect defines a collection of component models whose names end with "Impl." The `AddConcurrency` strategy is then performed on the collection of models meeting the `Start` selection criteria. `AddConcurrency` is represented by the following tasks: for any component model that has at least one `Data` atom, create two `Concurrency` atoms within the model representing Internal Locking and External Locking.

```
1  strategy AddConcurrency()
2  {
3    declare concurrencyAtom1, concurrencyAtom2 : atom;
4
5    if(atoms()->select(a | a.kindOf() == "Data")->size() >= 1) then
6      //add the first concurrency atom
7      concurrencyAtom1 := addAtom("Concurrency", "InternalConcurrency");
8      concurrencyAtom1.setAttribute("Enable", "1");
9      concurrencyAtom1.setAttribute("LockType", "Thread Mutex");
10     concurrencyAtom1.setAttribute("LockStrategy", "Internal Locking");
11
12     //add the second concurrency atom
13     concurrencyAtom2 := addAtom("Concurrency", "ExternalConcurrency");
14     concurrencyAtom2.setAttribute("Enable", "1");
15     concurrencyAtom2.setAttribute("LockType", "Thread Mutex");
16     concurrencyAtom2.setAttribute("LockStrategy", "External Locking");
17   endif;
18 }
19 aspect Start( )
20 {
21    rootFolder().findFolder("ComponentTypes").models()
22                ->select(m|m.name().endWith("Impl"))->AddConcurrency();
23 }
```

Listing 3-4 - *ECL specification to add concurrency atoms to ESML models*

In the `Start` aspect, the `rootFolder()` function first returns the root folder of a modeling project. Next, `findFolder()` is used to return a folder named "ComponentTypes" under the root folder. Then, `models()` is used to find all the models in the "ComponentTypes" folder. Finally, the `select()` operator is used to select all the models whose names end with "Impl." The `AddConcurrency` strategy is then applied to the resulting collection. In the `AddConcurrency` strategy, line 5 of Listing 3-4 determines whether there exist any `Data` atoms in the current context model. If such atoms exist, the first `Concurrency` atom is created and its attributes are assigned appropriate values in line 7 through line 8 and the second `Concurrency` atom is created and defined in lines 13 through 16.

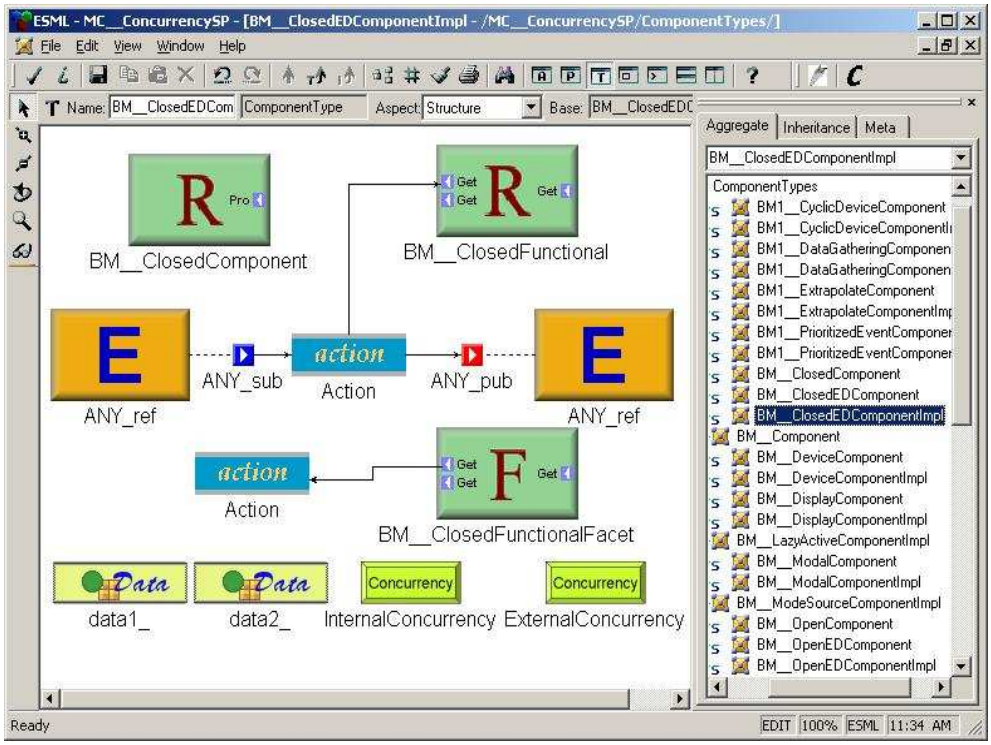


Figure 3-9 *- The transformed Bold Stroke component model*

Considering the component model shown in Figure 3-8 as one of the source models for this transformation, the resulting target model generated from the transformed source model is shown in Figure 3-9. After this transformation is accomplished, two concurrency atoms, representing internal locking and external locking, are inserted into the source model. As a result, all the existing models can be adapted to capture concurrency mechanisms rapidly without extensive manual operations. The ECL is essential to automate such a transformation process.

3.5 Experimental Validation

Experimental validation of the contributions described in this dissertation, in terms of the ability to enable model evolution, has been performed by evaluating the research on large-scale models in various domains such as computational physics,

middleware, and mission computing avionics. This section outlines the artifacts available for experimentation, as well as the assessment questions and measurement metrics that were used to evaluate the research results.

3.5.1 Modeling Artifacts Available for Experimental Validation

The modeling artifacts available for experimental validation are primarily from two sources. One source is Vanderbilt University, a collaborator on much of the C-SAW research, who has provided multiple modeling artifacts as experimental platforms. The other source is Escher [Escher, 07], which is an NSF sponsored repository of modeling artifacts developed from DARPA projects available for experimentation. As discussed in Section 3.4, C-SAW was used to support change evolution of Bold Stroke component models, which were defined in the Embedded Systems Modeling Language (ESML). This experiment assisted in transforming legacy codes [Gray et al., 04-b]. Another modeling artifact is the System Integration Modeling Language (SIML) as discussed in Section 3.3. C-SAW was used to support system scalability issues with SIML.

The Event QoS Aspect Language (EQAL) [Edwards, 04] is a modeling language from Vanderbilt that is used to graphically specify publisher-subscriber service configurations for large-scale DRE systems. The EQAL modeling environment consists of a GME metamodel that defines the concepts of publisher-subscriber systems, in addition to several model compilers that synthesize middleware configuration files from models. The EQAL model compilers automatically generate publisher-subscriber service configuration files and component property description files needed by the underlying

middleware. EQAL was also used for experimentation of model scalability with C-SAW [Gray et al., 05].

Other modeling artifacts include the Stochastic Reward Net Modeling Language (SRNML) [Lin et al., 07-a] and Platform-Independent Component Modeling Language (PICML) [Balasubramanian et al., 06-a]. SRNML has been used to describe performability concerns of distributed systems built from middleware patterns-based building blocks. PICML is a domain-specific modeling language for developing component-based systems. Case studies on using C-SAW to support model evolution of EQAL and SRNML are given in Appendix C.

3.5.2 Evaluation Metrics for Project Assessment

Experimental validation of this research has been based on various experimental evaluations. There are a set of metrics used in the research validation.

Domain generality is used to demonstrate that the C-SAW transformation engine is DSML-independent and able to perform a variety of modeling tasks. This can be assessed by determining how much customization effort is needed to adapt the C-SAW model transformation approach for each new modeling language.

There are also other metrics that were assessed to determine the amount of effort and cost required to apply C-SAW to evolve models, and the effect of using C-SAW. This set of metrics includes *productivity* and *accuracy*. Productivity assessment is used to determine the ability of C-SAW to reduce the efforts (represented by amount of time) in developing model transformations to perform model evolution tasks, compared to a manual model evolution process (i.e., using editing operations in a modeling environment

such as GME). Accuracy is an assessment of C-SAW's ability to reduce errors in performing model evolution tasks compared to a manual process. The expected benefits of the model transformation approach and its supporting tools are improved productivity and increased accuracy.

Experimental validation was conducted by observing the level of effort expended in applying the results of the research to evolve model artifacts as identified in Section 3.5.1, and the correctness of the result. These metrics provide an indication of the success of the research as it relates to the ability to evolve domain models.

3.5.3 Experimental Result

Experimental results for validating the research are from the feedback and observations during the applications of C-SAW to support automated evolution of models on several different experimental platforms.

As an initial result, this work has been experimentally applied to a mission computing avionics application provided from Boeing where C-SAW was used to evolve component models to transform the code base through an approach called Model-Driven Program Transformation (MDPT) [Gray et al., 04-b], [Zhang et al., 04] (Note: MDPT is not a contribution of this research, but illustrates an application of C-SAW). On this experimental platform, C-SAW was used to weave the concurrency mechanisms, synchronization and flight data recorder policies into component models specified in ESML in order to adapt the modeled systems to new requirements. These concerns are usually spread across the model hierarchy and it is hard to address using a manual approach. In addition, C-SAW was applied to component-based distributed system

development by weaving deployment aspects into component models specified in PICML. Using C-SAW to compose deployment specification from component models not only facilitates modifications to the model in the presence of a large number of components, but also gives assurance that changes to model elements keep the model in a consistent state [Balasubramanian et al., 06]. More recently, C-SAW has been used in addressing the important issue of model scalability [Gray et al., 05], [Lin et al., 07-a] in SIML (as mentioned in Section 3.3), EQAL and SRNML (as discussed in Appendix C). The feedback from these experiments provides two categories of results to demonstrate the benefits of using C-SAW:

- The result of the first category is related to domain generality. C-SAW has been designed and implemented as a modeling-language independent model transformation engine. Currently, C-SAW can be used in any modeling language that is conformant to a GME metamodel and is able to support any kind of model evolution when the source model and the target model belong to the same metamodel. Thus, C-SAW can be applied to various domains without any customization effort for performing a variety of modeling tasks such as aspect weaving [Gray et al., 06], model scalability [Lin et al., 07-a] and model refactoring [Zhang et al., 05-a]. This demonstrates that the C-SAW approach meets the quality measurement of the domain generality. Also, C-SAW is implemented as a GME plug-in. It is easy to install by registering the freely-available component (please see the conclusion of this chapter for the URL), without the need to install any other libraries or software. This installation process is the same for all modeling languages.

- The result of the second category was evaluated to determine the degree of how the C-SAW approach improves productivity and increases accuracy for model evolution. As observed, compared to a manual approach by making changes using the editing operations of GME, using C-SAW not only reduces the time significantly but also decreases the potential errors. For example, SIML models have been scaled by hand to 32 and 64 nodes. After 64 nodes, the manual process deteriorated taking several days with multiple errors. Using C-SAW, SIML models have been scaled up to 2500 nodes within a few minutes; flexibility for scaling up or down can also be achieved through parameterization. For a user familiar with ECL, the time to create a model transformation by a user unfamiliar with the domain is often less than 1.5 hours. Moreover, an ECL specification may be tested for correctness (e.g., using the testing engine described in Chapter 4) on a single model before it is applied to a large collection of models, which helps to reduce the potential errors of C-SAW transformations. In contrast, a manual process requires an overwhelming amount of ad hoc mouse clicking and typing, which makes it easy to make errors.

To conclude, these results have preliminarily demonstrated C-SAW as an effective tool to assist in model evolution in various domains for specific types of transformations.

3.6 Related Work

The general area **of** related work concerns model transformation approaches and applications, especially modeling research and practice that provide abilities to specify

model evolution concerns that address issues such as model scalability and evolution. This section provides an overview of work related to these issues.

3.6.1 Current Model Transformation Techniques and Languages

A large number of approaches to model transformation have been proposed by both academic and industrial researchers and there are many model transformation tools available (example surveys can be found in [Czarnecki and Helsen, 06], [Mens and Van Gorp, 05], [Sendall and Kozaczynski, 03]). In general, there are three different approaches for defining transformations, as summarized from [Sendall and Kozaczynski, 03]:

- **Direct Model Manipulation** – developers access an internal model representation and use a general-purpose programming language (GPL) to manipulate the representation from a set of procedural APIs provided by the host modeling tool. An example is Rational XDE, which exposes an extensive set of APIs to its model server that can be used from Java, VB or C++ [Rose, 07]. Another example is the GME, which offers the BON (Section 3.2.5) as a set of APIs in Java and C++ to manipulate models [GME, 07].
- **Intermediate Representation** – a modeling tool can export the model into an intermediate representation format (e.g., XML). Transformations can then be performed on the exported model by an external transformation tool (e.g., XSLT [XSLT, 99]), and the output models can be imported back into the host modeling tool. An example is OMG's Common Warehouse Metamodel (CWM) Specification [CMW, 07] and transformation implemented using XSLT. In fact, the original

implementation of the ECL performed a transformation using XSLT on GME models exported as XML [Gray et al., 01], [Gray, 02].

- **Specialized Transformation Language** – a specialized transformation language provides a set of constructs for explicitly specifying the behavior of the transformation. A transformation specification can typically be written more concisely than direct manipulation with a GPL.

The direct manipulation approach provides APIs that may be familiar to programmers using Java or C++ frameworks, but may not be familiar to end-users handling high-level modeling notations in specific domains. The disadvantage of APIs used by a GPL is that they lack high-level abstraction constructs to specify model transformations so that end-users have to deal with the accidental complexity of the low-level GPL.

The advantage of the XSLT-based approach is that XSLT is an industry standard for transforming XML where the XML Metadata Interchange (XMI) [XMI, 07] is used to represent models. However, XSLT requires experience and considerable effort to define even simple model transformation [Sendall and Kozaczynski, 03]. Manual implementation of model transformation in XSLT quickly leads to non-maintainable implementations because of the verbosity and poor readability of XMI and XSLT.

The specialized transformation language approach provides a domain-specific language (DSL) [Mernik et al., 05] for describing transformations, which offers the most potential expressive power for transformations. Currently, numerous model transformation languages have been proposed by both academic and industrial researchers. These languages are used to define transformation rules and rule application

strategies that can be either graphical or textual. Additionally, model transformation languages may be either imperative or declarative [Czarnecki and Helsen, 06].

There are two major kinds of model transformation specification languages: one represents a graphical language, typified by graph grammars (e.g., Graph Rewriting and Transformation Language (GReAT) [Agrawal, 03], AToM^3 [Vangheluwe and De Lara, 04] and Fujaba [Fujaba, 07]), the other is a hybrid language (e.g., Atlas Transformation Language (ATL) [Bézivin et al., 04], [Kurtev et al., 06] and Yet Another Transformation Language (YATL) [Patrascoiu, 04]). The distinguishing features of these two language categories are summarized as:

- **Graphical transformation language:** In this approach, models are treated as graphs and model transformations are specified as graph transformations. Graph transformations are realized by the application of transformation rules, which are rewriting rules for graphs specified as graph grammars. The left-hand side (LHS) of a transformation rule is a graph to match, and the right-hand side (RHS) is a replacement graph. If a match is found for the LHS graph, then the rule is fired, which results in the matched sub-graph of the graph under transformation being replaced by the RHS graph. In such a language, graphical notations are provided to specify graph patterns, model transformation rules and control flow of transformation execution. Compared to a textual language, a graphical language is efficient in communicating graph patterns. However, it can be tedious to use purely graphical notations to describe complicated computation algorithms. As a result, it may require generation to a separate language to apply and execute the transformations.

- **Hybrid transformation language:** combines declarative and imperative constructs. Declarative constructs are used to specify source and target patterns as transformation rules (e.g., filtering model elements), and imperative constructs are used to implement sequences of instructions (e.g., assignment, looping and conditional constructs). However, embedding predefined patterns renders complicated syntax and semantics for a hybrid language [Kurtev et al., 06].

Many existing model transformation languages (including those discussed above) allow transformation to be specified between two different domains (e.g., a transformation that converts a UML model into an entity-relationship model). The ECL can be distinguished from these approaches as a relatively simple and easy-to-learn language that focuses on specifying and executing endogenous transformation where the source and target models belong to the same domain. However, it has full expressive power for model replication and aspect modeling because these tasks can be specified as endogenous transformations.

3.6.2 Related Work on Model Scalability

Related work on model scalability contributes to the ability to specify and generate instance-based models with repetitive structures. The approach proposed by Milicev [Milicev, 02] uses extended UML Object Diagrams to specify the instances and links of a target model that is created during automatic translation; this target model is called the domain mapping specification. An automatically generated model transformer is used to produce intermediate models, which are refined to final output artifacts (e.g, C++ codes). Similar to the ideas presented in this dissertation, Milicev adopts a model transformation approach whereby users write and execute transformation specifications

to produce instance-based models. However, different from our approach, Milicev's work is domain-dependent because the domain mapping specifications and model transformers are domain-specific. Moreover, the target of his work is the reusability of code generators built in existing modeling tools, which introduces an intermediate model representation to bridge multiple abstraction levels (e.g., from model to code). The target of our approach is to use existing model transformation tools, which support model-to-model transformations at the same abstraction level. Model transformations can be written that perform model replication in a domain-independent manner without any effort toward extending existing model representations across different abstraction levels.

Several researchers have proposed standard notations to represent repetitive structures of modeling real-time and embedded systems, which is helpful in discovering possible model replication patterns. The MARTE (Modeling and Analysis of Real-Time and Embedded systems) request for proposals was issued by the OMG in February 2005, which solicits submissions for a UML profile that adds capabilities for modeling Real-Time and Embedded Systems (RTES), and for analyzing schedulability and performance properties of UML specifications. One of the particular requests of MARTE concerns the definition of common high-level modeling constructs for factoring repetitive structures for software and hardware. Cuccuru et al. [Cuccuru et al., 05] proposed multi-dimensional multiplicities and mechanisms for the description of regular connection patterns between model elements. However, these proposed patterns are in an initial stage and have not been used by any existing modeling tools. Their proposal mainly works with models containing repetitive elements that are identical, but may not specify all the model replication situations that were identified in this dissertation (e.g., to represent

models with a collection of model elements of the same type, which are slightly different in some properties, or have similar but not identical relationships with their neighbors). As such research matures, we believe significant results will be achieved toward representation of repetitive or even similar model structures. Such maturity will contribute to standardizing and advancing model replication capabilities.

The C-SAW approach advocates using existing model transformation techniques and tools to address model scalability, especially where modeling languages lack support for dynamic creation of model instances and links. This investigation on the application of model transformations to address scalability concerns extends the application area of model transformations. The practice and experiences illustrated in this chapter help to motivate the need for model scalability.

3.7 Conclusion

The goal of the research described in this chapter is to provide a model transformation approach to automate model evolution. This chapter presents the major extensions to the ECL model transformation language and its associated engine C-SAW to address model evolution concerns that relate to important system-wide issues such as scalability and adaptability. ECL is a transformation language to specify various types of evolution tasks in modeling, such as scalability concerns that allow a model engineer to explore design alternatives. C-SAW has been implemented as a GME plug-in to execute ECL specifications within the GME. This enables weaving changes into GME domain models automatically.

Experimental validation is also discussed in this chapter to assess the benefits and effectiveness of the C-SAW approach in automating model evolution. There are several large-scale models available from the Escher Institute [Escher, 07] that were used as experimental platforms. These models have several thousand modeling elements in various domains such as computational physics, middleware, and mission computing avionics. As an experimental result, the C-SAW transformation engine has been applied to support automated evolution of models on several of these different experimental platforms. Particularly, C-SAW has been used to address the important issue of model scalability for exploring design alternatives and crosscutting concerns for model adaptation and evolution. The observation and feedback from the usage of C-SAW has demonstrated that C-SAW not only helps to reduce the human effort in model evolution, but also helps to improve the correctness. Other benefits provided by C-SAW include modeling language independency and the capability to perform various types of model evolution tasks. The C-SAW plug-in downloads, publications, and video demonstrations are available at the project website: http://www.cis.uab.edu/gray/Research/C-SAW/

To improve the correctness of a model transformation specification, a model transformation testing approach as discussed in Chapter 5 provides support for testing model transformation specifications, which requires model comparison techniques. As another contribution of this research, the next chapter presents the algorithms and tool support called DSMDiff for model comparison.

CHAPTER 4

DSMDIFF: ALGORITHMS AND TOOL SUPPORT FOR MODEL DIFFERENTIATION

This chapter describes the contribution of this dissertation on model differentiation. It begins with a brief discussion on the need for model differentiation, followed by detailed discussions on the limitations of current techniques. The problem of model differentiation is formally defined and the key issues are identified. The core of this chapter is to present the developed model differentiation algorithms and the associated tool called DSMDiff, including an analysis of non-structural and structural information of model elements, formal representation of models and details of the algorithms. This chapter also motivates the importance of visualizing the model differences in a manner that can be comprehended by a model engineer. The chapter provides an evaluation of the algorithms and concludes with an overview of related work and a summary.

4.1 Motivation and Introduction

As MDE is emerging as a software development paradigm that promotes models as first-class artifacts to specify properties of software systems at a higher level of abstraction, the capability to identify mappings and differences between models, which

is called model differentiation, model differencing or model comparison, is essential to many model development and management practices [Cicchetti et al., 2007]. For example, model differentiation is needed in a version control system that is model-aware to trace the changes between different model versions to understand the evolution history of the models. Model comparison techniques and tools may help maintain consistency between different views of a modeled system. Particularly, model differentiation is needed in the model transformation testing research discussed in Chapter 5 to assist in testing the correctness of model transformations by comparing the expected model and the resulting model after applying a transformation ruleset.

Although there exist many techniques available for differentiating text files (e.g., source code and documentation) and for structured data (e.g., XML documents), such tools either operate under a linear file-based paradigm that is purely textual (e.g., the Unix diff tool [Hunt and McIlroy, 75]) or perform comparison on a tree structure (e.g., the XMLDiff tool [Wang et al., 03]). However, models are structurally represented as graphs and are often rendered in a graphical notation. Thus, there is a structural mismatch between currently available text-based differentiation tools and the graphical nature of models. Furthermore, from our experience, large models can contain several thousand modeling elements, which makes a manual approach to model differentiation infeasible. To address these problems, more research is needed to explore automated differentiation algorithms and supporting tools that may be applied to models with graphical structures.

4.2 Problem Definition and Challenges

Theoretically, generic model comparison is similar to the graph isomorphism problem that is known to be in NP [Garey and Johnson, 79]. Some research efforts aim to provide generic model comparison algorithms, such as the Bayesian approach, which initially provides diagram matching solutions to architectural models and data models [Mandelin et al., 06]. However, the computational complexity of general graph matching algorithms is the major hindrance to applying such algorithms to practical applications in modeling. Thus, it is necessary to loosen the constraints on graph matching to find solutions for model comparison. A typical solution is to provide differentiation techniques that are specific to a particular modeling language, where the syntax and semantics of this language help to handle conflicts during model matching.

Currently, there exist many types of modeling languages. Particularly, the UML is a popular object-oriented modeling language. The majority of investigations into model differentiation focus on UML diagrams [Ohst et al., 03], [Xing and Stroulia, 05]. Alternatively, DSM [Gray et al., 07] is an emerging MDE methodology that generates customized modeling languages and environments from metamodels that define a narrow domain of interest. Distinguished from UML, which is a general purpose modeling language, DSMLs aim to specify the solution directly using rules and concepts familiar to end-users of a particular application domain.

There are two main differences between domain-specific models and UML diagrams: 1) UML diagrams have a single definition for syntax and static semantics (i.e., a single metamodel), however, domain-specific models vary significantly in their structures and properties when their syntax and static semantics are defined in different

metamodels, which correspond to different DSMLs customized for specific end-users; 2) domain-specific models are usually considered as instance-based models (e.g., large domain-specific system models often have repetitive and nested hierarchical structures and may contain large quantities of objects of the same type), but traditional UML diagrams are primarily class-based models. Thus, domain-specific models and UML diagrams differ in structure, syntax and semantics. New approaches are therefore required to analyze differences among domain-specific models. However, there has been little work reported in the literature on computing differences between domain-specific models that are visualized in a graphical concrete syntax. To address the problem of computing the differences between domain-specific models, the following issues need to be explored:

- What are the essential characteristics of domain-specific models and how are they defined?
- What information within domain-specific models needs to be compared and what information is needed to support metamodel-independent model comparison?
- How is this information formalized within the model representation in a particular DSML?
- How are model mappings and differences defined to enable model comparison?
- What algorithms can be used to discover the mappings and differences between models?

- How to visualize the result of model comparison to assist in comprehending the mappings and differences between two models?

4.2.1 Information Analysis of Domain-Specific Models

To develop algorithms for model differentiation, one of the critical questions is whether to determine if the two models are syntactically equivalent or to determine if they are semantically equivalent. Because the semantics of most modeling languages are not formally defined, the developed algorithms only determine whether the two models are syntactically equivalent[4]. To achieve this, a model comparison algorithm must be informed by the syntax of a specific DSML. Thus, this section discusses how the syntax of a DSML is defined and what essential information is embodied in the syntax.

As discussed in Chapter 2, metamodeling is a common technique for conceptualizing a domain by defining the abstract syntax and static semantics of a DSML. A metamodel defines a set of modeling elements and their valid relationships that represent certain properties for a specific domain. The GME [Lédeczi et al., 01] is a meta-configurable tool that allows a DSML to be defined from a metamodel. Domain-specific models can be created using a DSML and may be translated into source code, or synthesized into data to be sent to a simulation tool. The algorithms presented in this chapter have been developed within the context of the GME, but we believe these algorithms can solve broader model comparison problems in other metamodeling tools such as the ATLAS Model Management Architecture (AMMA) [Kurtev et al., 06],

[4] Please note that this is not a serious limitation when compared to other differentiation methods. The large majority of differentiation techniques offer syntactic comparison only, especially those focused on detecting textual differences

Microsoft's DSL tools [Microsoft, 05], MetaEdit+ [MetaCase, 07], and the Eclipse Modeling Framework (EMF) [Budinsky et al., 04].

There are three basic types of entities used to define a DSML in GME: *atom*, *model* and *connection*. An *atom* is the most basic type of entity that cannot have any internal structures. A *model* is another type of entity that can contain other modeling entities such as child models and atoms. A *connection* represents the relationship between two entities. Generally, the constructs of a DSML defined in a metamodel consist of a set of model entities, a set of atom entities and a set of connections. However, these three types of entities are generic to any DSML and provide domain-independent type information (i.e., called the *type* in GME terminology). Each entity (e.g., model, atom or connection) in a metamodel is given a name to specify the role that it plays in the domain. Correspondingly, the name that is defined for each entity in a metamodel represents the domain-specific type (i.e., called the *kind* in GME terminology), which end-users see when creating an instance model. Moreover, attributes are used to record state information and are bound to atoms, models, and connections. Thus, without considering its relationships to other elements, a model element is defined syntactically by its type, kind, name and a set of attributes. Specifically, type provides certain meta information to help determine the essential structure of a model element for any DSML (e.g., model, atom or connection) and is needed in metamodel-independent model differentiation. Meanwhile, kind and name are specific to a given DSML and provide non-structural syntactical information to further assist in model comparison. Other syntactical information of a model element include its relationships to other elements (i.e.,

connections to its neighbours), which may also distinguish the identity of modeling elements.

In summary, to determine whether two models are syntactically equivalent, model differentiation algorithms need to compare all the syntactical information between them. Such a set of syntactical information of a model element includes: 1) its type, kind, name and attribute information; and 2) its connections to other model elements. In addition, if these two models are containment models, the algorithms need to compare all the elements at all the levels. There is other information associated with a model that either relates to the concrete syntax of a DSML (e.g., visualization specifications such as associated icon objects and their default layouts and positions) or to the static semantics of a DSML (e.g., constraints to define domain rules). The concrete syntax is not generally involved in model differentiation for the purpose of determining whether two models from the same DSML are syntactically equivalent (e.g., the associated icon of a model element is always determined by its kind information from the metamodel definition). Similarly, because the constraints are defined at the metamodel level in our case (i.e., models with the same kind hold the same constraints), they are not explicitly compared in model differentiation; instead, kind equivalence implies the equivalence of constraints.

4.2.2 Formalizing a Model Representation as a Graph

In order to design efficient algorithms to detect differences between two models, it is necessary to understand the structure of a model. Figure 4-1 shows a GME model and its hierarchical structure. According to its hierarchical containment structure, a model can be represented formally as a hierarchical graph that consists of a set of nodes and

edges, which are typed, named and attributed. There are four kinds of elements in such a graph:

- **Node.** A node is an element of a model, represented as a 4-tuple (*name*, *type, kind*, *attributes*), where *name* is the identifier of the node, *type* is the corresponding metamodeling element for the node, *kind* is the domain-specific type, and *attributes* is a set of attributes that are predefined by the metamodel. There are two kinds of nodes:
    - **Model node:** a containment node that can be expanded at a lower level as a graph that consists of a set of nodes and a set of edges (i.e., a container). This kind of node is used to represent submodels within a model, which leads to multiple-level hierarchies of a containment model.
    - **Atom node:** an atomic node that cannot contain any other nodes (i.e., a leaf). This kind of node is used to represent atomic elements of a model.
- **Edge.** An edge is a 5-tuple (*name*, *type*, *kind*, *src*, *dst*), where *name* is the identifier of the edge, *type* is the corresponding metamodeling element for the edge, *kind* is the domain-specific type, *src* is the source node, and *dst* is the destination node. A connection can be represented as an edge.
- **Graph.** A directed graph consists of a set of nodes and a set of edges where the source node and the destination node of each edge belong to the set of nodes. A graph is used to represent an expanded model node.
- **Root.** A root is the graph at the top level of a multiple-level hierarchy that represents the top of a hierarchical model.

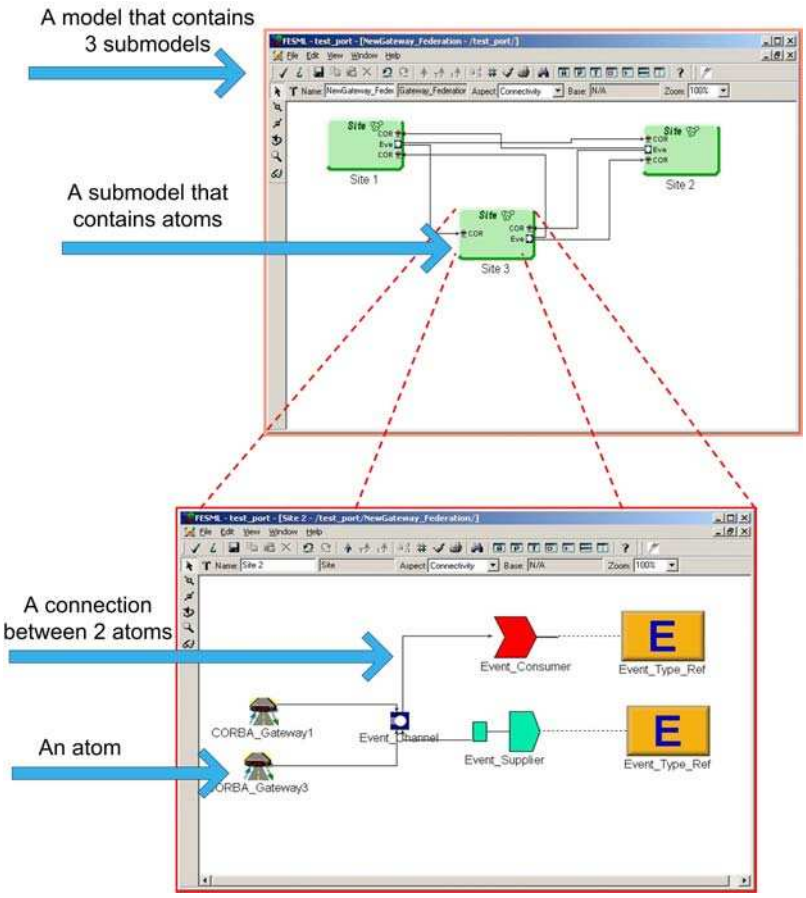


Figure 4-1 *- A GME model and its hierarchical structure*

4.2.3 Model Differences and Mappings

The task of model differentiation is to identify the mappings and differences between two containment models at all hierarchical levels. In general, the comparison starts from the top level of the two containment models and then continues to the child submodels. At each level, the comparison between two corresponding models (i.e., one is defined as the host model, denoted as M1, and the other is defined as the candidate model, denoted as M2), always produces two sets: the mapping set (denoted as MS) and the difference set (denoted as DS). The mapping set contains all pairs of model elements that are mapped to each other between two models. The difference set contains all

detected discrepancies between the two models. Before the details of the algorithms are presented, the definition of model mappings and differences is discussed.

A pair of mappings is denoted as Map ($elem^1$, $elem^2$), where $elem^1$ is in M1 and $elem^2$ is in M2, and may be a pair of nodes or a pair of edges. Map ($elem^1$, $elem^2$) is a bidirectional relationship that implies $elem^2$ is the only mapped correspondence in M2 for $elem^1$ in M1 based on certain matching metrics, and vice versa. The difference relationship between two models is more complicated than the mapping relationship. The notations used to represent the differences between two models are editing operational terms that are considered more intuitive [Alanen and Porres, 03]. For example, a *New* operation implies creating a model element, a *Delete* operation implies removing a model element and a *Change* operation implies changing the value of an attribute. We define DS = M2 – M1, where M2 is compared to M1. DS consists of a set of operations that yields M2 when applied to M1. The "-" operator is not commutative.

There are several situations that could cause two models to differ. The first situation of model difference occurs when some modeling elements (e.g., nodes or edges in the graph representation) are in M2, but not in M1. We denote this kind of difference as *New* (e^2) where e^2 is in M2, but not in M1. The converse is another situation that could cause a difference (i.e., elements in M1 are missing in M2). We denote this kind of difference as *Delete* (e^1) where e^1 is in M1, but not in M2. These two situations occur from structural differences between the two models. A third difference can occur when all of the structural elements are the same, but a particular value of an attribute is different. We denote this difference as *Change* (e^1, e^2, *f*, v^1, v^2), where e^1 in M1 and e^2 in M2 are a pair of mapping elements, f is the feature name (e.g., name of an attribute), v^1 is

the value of e^1.f, and v^2 is the value of e^2.f. Thus, the difference set actually includes three sets: DS = {N, D, C} where N is a set that contains all the New differences, D is a set that contains all the Delete differences, and C is a set that contains all the Change differences. This approach was initially defined in [Lin et al., 05] and extended in [Lin et al., 07-b].

4.3 Model Differentiation Algorithms

The model comparison algorithms developed as a part of the research described in this dissertation identify the mappings and differences between two containment models by comparing all the elements and their abstract syntactical information within these models. In general, the comparison starts from the two root models and then continues to the child submodels. At each level, two metrics (i.e., *signature matching* and *structural similarity*) are combined to detect the mapped nodes between a pair of models and the remaining nodes are examined to determine all the node differences. Based on the results of node comparison, all the edges are computed to discover all the edge mappings and differences.

To store the two models that need to be compared and the results of model comparison, a data structure called DiffModel is used. The structure of DiffModel contains a pair of models to be compared, a mapping set to store all the matched child pairs, and three difference sets to record New, Delete, and Change differences.

4.3.1 Detection of Model Mappings

It is well-known that some model comparison algorithms are greatly simplified by requiring that each element have a persistent identifier, such as a universally unique identifier (UUID), which is assigned to a newly created element and will not be changed

unless the element is removed [Ohst et al., 03]. However, such traceable links only apply to two models that are subsequent versions. In many modeling activities, model comparison is needed between two models that are not subsequent versions. A pair of corresponding model elements need to share a set of properties, which can be a subset of their syntactical information. Such properties may include type information, which can be used to select the model elements of the same type from the candidates to be matched because only model elements with the same type need to be compared. For example, in a Petri net model, a "place" node will not match a "transition" node. In addition to type information, identification information such as name is also important to determine mappings for domain-specific models. Therefore, a combination of syntactical properties for a node or an edge can be used to identify different model elements. Such properties are called the *signature* in DSMDiff, and are used as the first criterion to match model elements. Signature is a widely used term in much of the literature on structural data matching and may have different definitions [Wang et al., 03]. In our context, the signature of a node or an edge is a subset of its syntactical information, which is defined as follows:

- **Node Signature** is the concatenation of the type, kind and name of a node. Suppose v is a node in a graph. *Signature (v)* = /*Type (v)*/*Kind (v)*/*Name (v)*. If a node is nameless, its name is set as an empty string.

- **Edge Signature** is the concatenation of the type, kind and name of the edge as well as of the signatures of its source node and destination node. Suppose e is an edge in a graph, src is its source node and dst is its destination node. *Signature (e)*

= *Signature (src)*/*Type (e)*/*Kind (e)*/*Name (e)*/*Signature (dst)*. If an edge is nameless, its name is set as an empty string.

Signature Matching

Signature matching can be defined as:

- **Node Signature Matching:** Given two models, M1 and M2, suppose v^1 is a node in M1 and v^2 is a node in M2. There is a node signature matching between v^1 and v^2 if *Signature* (v^1) = *Signature* (v^2), which means the two strings (i.e., the signature of v^1 and the signature of v^2) are textually equivalent.
- **Edge Signature Matching:** Given two models, M1 and M2, suppose e^1 is an edge in M1 and e^2 is an edge in M2. There is an edge signature matching between e^1 and e^2 if *Signature* (e^1) = *Signature* (e^2), which means the two strings (i.e., the signature of e^1 and the signature of e^2) are textually equivalent.

A node v^1 in M1 mapping to a node v^2 in M2 implies their name, type and kind are matched. An edge e^1 in M1 mapping to an edge e^2 in M2 implies their name, type, kind, source node and destination node are all signature matched.

Usually, nodes are the most significant elements in a model and edge mappings also depend on whether their source and destination nodes match. Thus, DSMDiff first tries to match nodes that have the same signature. For example, to decide whether there is a node in M2 mapped to a node in M1 (denoted as v^1), the algorithm first needs to find all the candidate nodes in M2 that have the same signature as v^1 in M1. If there is only one candidate found in M2, the identified candidate is considered as a unique mapping for v^1 and they are considered as syntactically equivalent. If there is more than one candidate

that has been found, the signature cannot identify a node uniquely. Therefore, v^1 and its candidates in M2 will be sent for further analysis where structural matching is performed.

Structural Matching

In some cases, signature matching alone cannot find the exact mapping for a given model element. During signature matching, one node in M1 may have multiple candidates in M2. To find a unique mapping from these candidates, DSMDiff uses structural similarity as another criterion. The metric used for determining structural similarity between a node and its candidates is called edge similarity, which is defined as follows:

> **Edge Similarity:** Given two models, M1 and M2, suppose v^1 is a node in M1 and v^2 is one of its candidate nodes in M2. The edge similarity of v^2 to v^1 is the number of edges connecting to v^2, with each signature matched to one of the edges connecting to v^1.

During structural matching, if DSMDiff can find a candidate that has the maximal edge similarity, this candidate becomes the unique mapping for the given node. If it cannot find this unique mapping using edge similarity, one of the candidates will be selected as the host node's mapping, following the assumption that there may exist a set of identical model elements.

Listing 4-1 presents the algorithm to find the candidate node with maximal edge similarity for a given host node from a set of candidate nodes. It takes the host node (i.e., `hostNode`) and a set of candidate nodes of M2 (i.e., `candidateNodes`) as input, computes the edge similarity of every candidate node and returns a candidate with maximal edge similarity. Listing 4-2 gives the algorithm for computing edge similarity between a candidate node and a host node. It takes two maps as input – `hostConns`

stores all the incoming and outgoing edges of the host node indexed by their edge signature, and `candConns` stores all the incoming and outgoing edges of the candidate node indexed by their edge signature. By examining the mapped edge pairs between these two maps, the algorithm computes the edge similarity as output.

```
Name: findMaximalEdgeSimilarity
Input: hostNode, candidateNodes
Output: maximalCandidate

   1. Initialize three maps: hostConns, candConns and set
      maxSimilarity = 0, maximalCanidate = null;
   2. Store each edge signature and the number of associated
      edges of the hostNode in the map hostConns;
   3. For each candidate c in candidateNodes
         1) Store each of its edge signatures and the number of
            associated edges in the map candConns;
         2) Call computeEdgeSimilarity(hostConns, candConns) to
            compute the edge similarity of c to hostNode;
         3) If(the computed similarity > maxSimilarity)
                  maxSimilarity = the computed similarity;
                  maximalCandiate = c;
   4. Return maximalCandidate;
```

Listing 4-1 - *Finding the candidate of maximal edge similarity*

```
Name: computeEdgeSimilarity
Input: hostConns, candConns
Output: similarity

   1. Initialize similarity as zero;
   2. For each edge signature in the map hostConns
         1) Get the number of the edges associated with the
            edge signature as hostCount;
         2) Get the number of the edges from the map candConns
            associated with the edge signature as candCount;
         3) If candCount <= hostCount
               Similarity = similarity + candCount;
         4) Else
               Similarity = similarity + hostCount;
   3. Return similarity;
```

Listing 4-2 - *Computing edge similarity of a candidate*

The algorithm in Listing 4-1 determines that the unique correspondence found using the edge similarity has the most identical connections and neighbors to the host node when only one candidate has the maximal edge similarity. The algorithm also implies one candidate with the maximal edge similarity is selected as the unique correspondence when there are more than one candidates with the same maximal edge similarity; however, this selection may be incorrect in some cases and needs to be improved as discussed later in *Limitations and Improvement* (Section 4.5.2). DSMDiff only examines structural similarity within a specific local region where the host node is the center and its neighbor nodes form the border. In our experience, using signature matching and edge similarity to find model mappings not only speeds up the model differentiation process, but also generates accurate results in the experiments that have been conducted (one example is demonstrated in Chapter 5). After all the nodes in M1 have been examined by signature and structural matching, all the possible node mappings between M1 and M2 are found in general practice except for the cases discussed in Section 4.5.2.

4.3.2 Detection of Model Differences

As mentioned previously, there are three basic types of model differences: New, Delete and Change. To identify these various types of differences is another major task of DSMDiff. In order to increase the performance of DSMDiff, some of the procedures to detect model differences may be integrated into the previously discussed procedures for finding mappings.

```
Name: findSignatureMappingsAndDeleteDiffs
Input: diffModel
Output: hostSet, candMap, diffModel

    1. Initialize a set hostSet and a map candMap;
    2. Get M1 from diffModel and store all nodes of M1 in
       hostSet
    3. Get M2 from diffModel and store all nodes of M2 in
       candMap associated with their signature;
    4. For each node e1 in hostSet
         1) Get the count of the nodes from candMap that are
            signature matched to e1;
         2) If count == 1
              Get the candidate from candMap as e2;
              Add Map(e1, e2) to the mapping set of diffModel;
              Erase e1 from hostSet;
              Erase e2 from candMap;
         3) If count == 0
              Add e1 to the Delete set of diffModel;
              Erase e1 from hostSet;
         4) If count > 1
              Do nothing;
```

Listing 4-3 - *Finding signature mappings and the Delete differences*

To discover all the Delete differences, DSMDiff must find all the model elements in M1 that do not have any signature matched candidates in M2. In signature matching, DSMDiff examines how many candidates can be found in M2 that have the same signature as each element in M1. If only one is found, a pair of mappings is constructed and added to the mapping set. If more than one is found, the host element and the found candidates are sent to structural matching. If no candidate can be identified, the host element is considered as a Delete difference, which means it exists in M1 but does not exist in M2. Listing 4-3 summarizes the algorithm.

After all the mappings are discovered between M1 and M2, the mapped elements are filtered out. The remaining elements in M2 are then taken as the New differences (i.e., a New difference indicates that there is an element in M2 that is missing in M1).

The Change differences are used to indicate varying attributes between any pair of mappings. Both model nodes and atom nodes may have a set of attributes; thus, a pair of matched model nodes or atom nodes may have Change differences. DSMDiff compares the values of each attribute of each pair of model or atom mappings. If the values are different, the attribute name is added to the Change difference set.

After all the node mappings and differences are determined, DSMDiff then tries to find the edge mappings and differences between M1 and M2 using these strategies: 1) all the edges connecting to a Delete node are Delete edges; 2) all the edges connecting to a New node are New edges; 3) the edge signature matching is applied to find out the edge mappings; and 4) the remaining edges in M1 are taken as additional Delete edges and those in M2 are taken as additional New edges.

4.3.3 Depth-First Detection

The traversal strategy of DSMDiff is depth-first, which traverses from the root level of a model hierarchy and then walks down to the lower levels to compare all the child submodels until it reaches the bottom level, where there are no submodels that can be expanded. Supporting such depth-first detection requires that all the node mappings found at a current level be categorized into two groups: model node mappings and atom node mappings. DSMDiff then performs model comparison on each pair of model node mappings. Each atom node mapping is examined for attribute equivalence. If there are some attributes with different values, these represent Change differences between the models. If all the attributes are matched, it is inferred that two nodes are equivalent because there is no Change, Delete or New difference.

To summarize, Listing 4-4 presents the overall algorithm of DSMDiff to calculate the mappings and the differences between two models. It takes `diffModel` as input, which is a typed DiffModel and initially stores two models (M1 and M2). DSMDiff produces two sets: the mapping set (MS) and the difference set (DS) that consists of three types of differences (N: the set of New differences, D: the set of Delete differences, and C: the set of Change differences). All of these mapping and difference sets are stored in the `diffModel` during execution of DSMDiff.

```
Name: DSMDiff
Input: diffModel
Output: diffModel

   1. Initialize a set hostSet and a map candMap;
   2. Get the host model from diffModel as M1 and the
      candidate model as M2;
   3. Detect attribute differences between M1 and M2 and add
      them to the Change set of diffModel;
   4. //Find node mappings by signature matching
      findSignatureMappingsAndDeleteDiffs (diffModel,
                                           hostSet, candMap);
   5. If(hostSet is not empty && candMap is not empty)
            //Find node mappings by structural matching
            For each element e1 in hostSet
                 1) Get its candidates from candMap into a set
                    called candSet;
                 2) e2 = findMaximalEdgeSimilarity(e1,candSet);
                 3) Add Pair(e1, e2) to the Mapping set of
                    diffModel;
                 4) Erase e1 from hostSet;
                 5) Erase e2 from candMap;
   6. If(candMap is not empty)
            Add all the remained members of candMap to the New
            set of diffModel;
   7. For each mapped elements that are not submodels
            Detect attribute differences and add them to the
            Change set of diffModel;
   8. Compute edge mappings and differences
   9. //Walk into child submodels
      For each childDiffModel that stores a pair mapped
      submodels
            DSMDiff(childDiffModel);
```

Listing 4-4 *- DSMDiff Algorithm*

4.4 Visualization of Model Differences

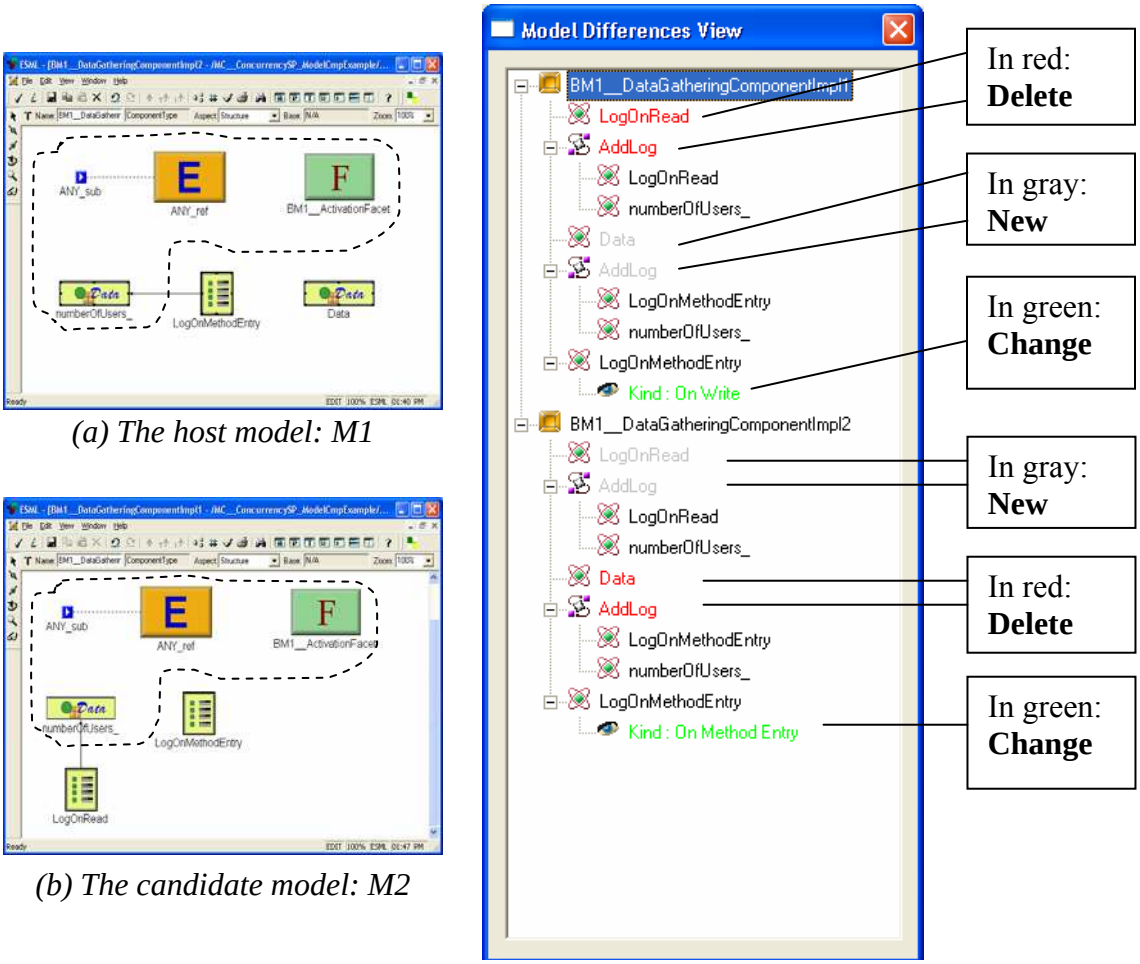
Visualization of the result of model differentiation (i.e., structural model differences) is critical to assist in comprehending the mappings and differences between two models. To help communicate the comparison results intuitively within a host modeling environment, a tree browser has been developed to visualize the structural differences and to support navigation among the analyzed model differences.

This browser looks similar to the model browser of GME, using the same graphical icons to represent items with types of model, atom and connection. To indicate the various types of differences, the browser uses three colors: red for a Delete difference, gray for a New difference, and green for a Change difference. The model difference browser displays two symmetric difference sets in two containment change trees: one indicates the difference set DS = M2 – M1 by annotating M1 with colors; and the other indicates the difference set DS' = M1 - M2 = -DS by annotating M2 with colors. If DS = {New = N, Delete = D, Change = C} then DS' = {New = D, Delete = N, Change = C}. For example, if there is a Delete difference in M1, correspondingly there is a New difference in M2. Such a symmetric visualization helps comprehend the corresponding relationships between two hierarchical models.

Figure 4-2 shows screenshots of two models and the detected differences in the model difference browser[5]. The host model M1 is shown in Figure 4-2a, and the candidate model M2 is shown in Figure 4-2b. The corresponding model elements within the encircled regions in these two models are the mappings, which are filtered and not

[5] Because the actual color shown in the browser can not be rendered in print, the figure has annotations that indicate the appropriate color.

displayed in the browser. The browser only visualizes the detected differences, as shown in Figure 4-2c.



(a) The host model: M1

(b) The candidate model: M2

(c) Model differences

***Figure 4-2** - Visualization of model differences*

The root of the upper tree is M1; its subtrees and leaf nodes are all the differences compared to M2, which is represented by the bottom tree. For example, the first child of the upper tree is a Delete difference, which is in red. This difference means the `LogOnRead` element is in M1, but is missing in M2. Correspondingly, there is a New

difference in the bottom tree, which is in gray. It indicates that M2 misses the `LogOnRead` element when it is compared to M1. A Change difference is detected for the `LogOnMethodEntry` element; although this element exists in both models, one of its attributes, called kind, has different values: "On Write" in M1 but "On Method Entry" in M2. Such a Change difference item is highlighted in green. When the two trees do not have any subtree or leaf node, we can infer there is no difference between these two models. To focus on any model element, a user can navigate across the tree and double-click an item of interest, and the corresponding model element is brought into focus within the editing window.

4.5 Evaluation and Discussion

This section first briefly analyzes the complexity of the algorithm and illustrates an example application. The current limitations and proposed improvements for DSMDiff are also discussed.

4.5.1 Algorithm Analysis

Generally, DSMDiff is a level-wise model differentiation approach. It begins with the two root models at the top-levels and then continues to their child models at the lower levels. At each level, node comparison is performed to detect the node mappings by using signature matching and edge similarity, followed by edge comparison to detect the edge mappings and differences. These steps are repeated on the mapped child models until the bottom-level is reached.

The core of the DSMDiff algorithms include signature matching (Step 4 in Listing 4-4) and edge similarity matching (Step 5 in Listing 4-4), which significantly influence the execution time. To estimate the complexity of signature matching and edge similarity matching, we assume the two models have similar structures and sizes. Given a model, L denotes the depth of the model hierarchy; N denotes the average number of nodes; and, M denotes the average number of the edges of a model node. The size of a model node is denoted as S, where S = N+M. Considering the case that every node at all levels except for the lowest level are model nodes, the total number of model nodes is denoted as T, where $T = \sum_{i=0}^{L-2} N^i \approx N^{L-1}$.

In the best case, all the mappings and differences between two model nodes can be found by signature matching, in which the complexity depends on the size of the model nodes. In `findSignatureMappingsAndDeleteDiffs` (Listing 4-3), where signature matching is performed to detect node mappings and differences, all the candidate nodes and their signatures are stored in a sorted map; the upper bound for the complexity of this step is O(N x logN). To find correspondences from this map for all the node elements of M1, the complexity is also O(N x logN). Later, similar computation is taken to compute the edge mappings and differences (i.e., Step 8 of Listing 4-4); such complexity is neglected here because the number of edges is less than the number of nodes. Overall, because all the model nodes within the model hierarchy need to be compared, the complexity for this best case is O(N x logN x T).

In the worst case, no exact mapping is found for a pair of model nodes during the signature matching. Thus, all the nodes need to be examined by edge similarity matching (i.e., Step 5 in Listing 4-4), which is the most complicated step in Listing 4-4. Assume

that there is an edge between any pair of nodes, then a node has N-1 edges, which is the worst case regarding the complexity. In edge similarity matching (i.e., Step 5 in Listing 4-4), the most complicated step is `findMaximalEdgeSimilarity` (Listing 4-1), which computes the edge similarity of all the candidate nodes for a host node, where all edge signatures of a candidate node and the number of the associated edges are stored in a map (i.e., Substep 3.1 in Listing 4-1). The complexity for building this map is O({N-1} x log{N-1}). To compute the edge similarity of every candidate node (i.e., Step 3 of Listing 1), the computation cost is bound by O(R x {N-1} x log{N-1}), where R is the number of candidate nodes with $R \leq N$. Because Step 3 is the most complicated step in Listing 4-1, the upper bound of `findMaximalEdgeSimilarity` is also O(R x {N-1} x log{N-1}). To find the candidate with maximal edge similarity for each host node (i.e., Step 5 in Listing 4-4), the cost is bounded by O(N x R x {N-1} x log{N-1}). To compute all the node mappings at all the levels in a model hierarchy using edge similarity matching, the upper bound of the complexity for this worst case is O(T x N x R x {N-1} x log{N-1}), which is in the polynomial class. For the same reason (i.e., the number of edges is less than the number of nodes), the complexity of detecting edge mappings and differences is neglected.

Although the complexity of constant-time signature comparison and associated string comparison is not counted here, the algorithm achieves polynomial time in complexity according to the above analysis.

4.5.2 Limitations and Improvement

DSMDiff is based on the assumption that domain-specific models are defined precisely and unambiguously. That is, domain-specific models are instances of a metamodel that can be distinguished from each other by comparing a set of properties of the elements and the connections to their neighbors. However, when there are several candidates with the same maximal edge similarity, DSMDiff may produce inaccurate results. A typical case occurs when there are nodes connected to each other but their mappings have not been determined yet. As shown in Figure 4-3, there is an A node connected to three nodes: B, C and D. In M2, the A' node connects to three other nodes: B', C' and D', and A'' is connected to B''. Given that nodes with the same letter label have the same signatures (e.g., all the A nodes have the same signature and all the B nodes have the same signature), then the connections between an A node and a B node have the same edge signature. According to the algorithm in Listing 4-1, suppose the A node is examined first for structural matching and the A' node in M2 is selected as the mapping of the A node in M1. When the B node is examined, the algorithm may select either B' or B'' in M2 as the mapping of the B node in M1 because both B nodes in M2 have the same edge similarity as the B node in M1. If the B'' node in M2 is selected as the mapping to the B node in M1, the result is incorrect because B' is the correct mapping. In such cases, DSMDiff needs to use new rules or criteria to help find the correct mapping. For example, a new rule needs to be added to the algorithm in Listing 1 to require selecting first the unmapped node in M1 that has maximal already-mapped neighbors. Another improvement will allow interaction between DSMDiff and users, who can select the mappings from multiple candidates manually.

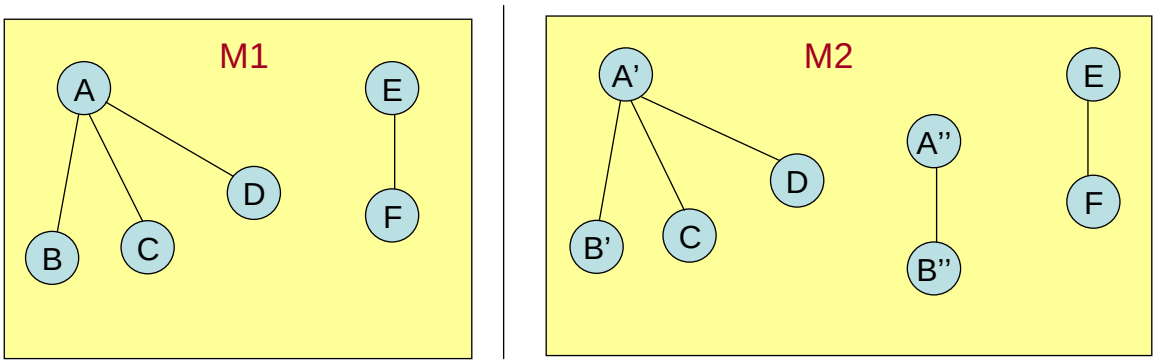


Figure 4-3 - *A nondeterministic case that DSMDiff may produce incorrect result*

Besides the performance and the correctness of the results, it is also important for model differentiation algorithms to produce a small set of model differences (ideally a minimal set) rather than providing a large set of model differences. In other words, the conciseness of the produced result is another metric contributing to the overall quality of model differentiation algorithms. Currently, DSMDiff compares two models M1 and M2 by traversing their composition trees in parallel. When an element from a model cannot be matched to an element of the other model at some level, the algorithm does not traverse the children of this element. One issue with this scheme is that DSMDiff is not able to detect when a subtree has been moved from one container to another between M1 and M2. The algorithm will only report that a whole subtree has been deleted from M1, and that a whole subtree has been added to M2, without noting that these are identical subtrees. This implies that the reported difference set is less concise than it could be. To solve this problem, a new type of model difference needs to be introduced: Move, which may reference the subtree in M1, and its new container in M2. An additional step is also required in the algorithms to compare all the elements of M1 and M2 that have not been matched when first traversing them. However, this step is expensive in the general case

because many elements may need to be compared. This cost is actually avoided in the current version of the algorithm by assuming a similar composition structure in M1 and M2.

DSMDiff visualizes all the possible differences as a containment tree in a browser, but does not directly highlight the differences upon the associated model elements within the editing window. To indicate the differences directly on the model diagrams and attribute panels within the modeling environment, a set of graphical decorators, which may be shapes or icons, could be attached to the corresponding model elements or attributes in order to change their look according to the type of model differences. In addition, our solution using coloring to highlight all possible types of model differences may fail to work when users are color-blind, or when a screenshot of the model difference tree view is printed in black-and-white (e.g., the need to add annotations to Figure 4-2c). A visualization mechanism to complement the coloring would indicate the Delete differences by striking through them, the Change ones by underlining them, and marking the New ones bold. This could be a complimentary solution that needs to be investigated in the future.

4.6 Related Work

This work is related to differentiation techniques for various software artifacts such as source code, documents, diagrams and models. There are two important categories of related work: 1) the algorithms to compute model differences, and 2) the visualization techniques to highlight those differences.

4.6.1 Model Differentiation Algorithms

There exist a number of general-purpose differentiation tools for comparing two or more text files (e.g., code or documentation). As an example, Unix diff [Hunt and McIlroy, 75] is a popular tool for comparing two text files. Diff compares files and indicates a set of additions and deletions. Many version control tools also provide functionality similar to diff to identify changes between versions of text documents [Eick et al., 01].

Although many tools are available for differentiating text documents, limited support is currently available for differentiating graphical objects such as UML diagrams and domain-specific models. As the importance of model differentiation techniques to system design and its evolution is well-recognized, there have been some research efforts focused on model difference calculation.

Several metamodel-independent algorithms regarding difference calculation between models are presented in [Alanen and Porres, 03] and [Ohst et al., 03], which are developed primarily based on existing algorithms for detecting changes in structured data [Chawathe et al., 96] or XML documents [Wang et al., 03]. In these approaches, a set of change operations such as “create” and “delete” are used to represent and calculate model differences, which is similar to our approach. However, they are based on the assumption that the model versions are manipulated through the editing tool that assigns persistent identifiers to all model elements. Such capability is not available when two models are developed separately (e.g., by different developers in a non-collaborative context, or by different editing tools) or generated by execution of a model transformation.

To provide algorithms independent of such identifiers, UMLDiff uses name similarity and structure similarity for detecting structural changes between the designs of subsequent versions of UML models [Xing and Stroulia, 05]. However, differentiation applied to domain-specific modeling is more challenging than difference analysis on UML diagrams. The main reason is that UML diagrams usually belong to a single common metamodel that can be represented formally as a containment-spanning tree starting at a virtual root and progressing down to packages, classes and interfaces. However, domain-specific models may belong to different metamodels according to their domains and are considered as hierarchical graphs. Also, a differentiation algorithm for domain-specific models needs to be metamodel-independent in order to work with multiple DSMLs. This required DSMDiff to consider the type information of instance models, as well as the type information of the corresponding metamodel.

A promising approach is to represent the result of model difference as a model itself. A recent work presented in [Cicchetti et al., 07] proposes a metamodel-independent approach to model difference representation. Within this approach, the detected model differences are represented as a difference model, which conforms to a metamodel that is automatically derived from the metamodel of the to-be-compared base models. Such a derivation process itself is a model transformation. Also, because the base models and the difference models are all model artifacts, other model-to-model transformations are induced to compose models (e.g., apply a difference model to a base model to produce the other base model). Thus, such an approach can be supported in a modeling platform and does not require other ad hoc tool support. A possible future improvement to

DSMDiff would be to integrate this approach to assist in representation of model differences.

4.6.2 Visualization Techniques for Model Differences

There has been some work toward visualizing model differences textually. IBM Rational Rose [Rose, 07] and Magic Draw UML [MagicDraw, 07] display model differences in a textual way. These tools convert the diagrams into hierarchical text and then perform differentiation on this hierarchy. Changes are shown using highlighting schemes on the text. Although this approach is relatively easy to implement, its main drawback is that changes are no longer visible in a graphical form within the actual modeling tool, which makes the difference results more difficult to comprehend.

Other researchers have shown that the use of color and graphical symbols (e.g., icons) are more efficient in highlighting model differences. An approach is proposed in [Ohst et al., 03] where coloring is used to highlight the model differences in two overlapping diagrams. A differentiation tool described in [Mehra et al., 05] presents graphical changes by developing a core set of highlighting schemes and an API for depicting changes in a visual diagram. UMLDiff presents a change-tree visualization technique. It reuses the visualization of Eclipse's Java DOM model for displaying different entities with diverse icons and separate visibility with various colors. Additionally, UMLDiff extends the visualization to use different icons to represent the differentiation results (e.g., "+" for add, "-" for remove).

Although it is intuitive to visualize model differences by coloring and iconic notations, these techniques are not specifically tied to modeling concepts and lack the

ability to be integrated into MDE processes. DSMDiff provides a model difference browser that displays the structural differences in a tree view, which is similar to the change-tree visualization technique of UMLDiff. To preserve the convention of the host modeling environment, many GME icons are used to represent the corresponding modeling types of the model difference items in the tree view. For example, a Delete atom or a New atom corresponds to an atom type. To avoid overuse of icons (e.g., "+" and "-" are commonly used for a collapsed folder and an expanded folder, respectively), DSMDiff uses colors to represent various types of model differences.

4.7 Conclusion

In this chapter, the model differentiation problem is defined in the context of Domain-Specific Modeling. The main points include: 1) domain-specific modeling is distinguished from traditional UML modeling because it is a variable-metamodel approach whereas UML is a fixed-metamodel approach; 2) the underlying metamodeling mechanism used to define a DSML determines the properties and structures of domain-specific models; 3) domain-specific models may be formalized as hierarchical graphs annotated with a set of syntactical information. Based on these characteristics, model differentiation algorithms and an associated tool called DSMDiff were developed to discover the mappings and differences between any two domain-specific models. The chapter also describes a visualization technique to display model differences structurally and highlight them using color and icons.

The applicability of DSMDiff has been demonstrated within the context of model transformation testing, as discussed in Chapter 5. To ensure the correctness of model transformation, executable testing can help detect errors in a model transformation

specification. To realize the vision of model transformation testing, a model differentiation technique is needed for comparison of the actual output model and the expected model, and visualization of the detected visualization. If there is no difference between the actual output and expected models, it can be inferred that the model transformation is correct with respect to the given test specification. If there are differences between the output and expected models, the errors in the transformation specification need to be isolated and removed. In this application, DSMDiff serves as a model comparator to perform the model comparison and visualize the produced differences.

CHAPTER 5

MODEL TRANSFORMATION TESTING

To ensure the correctness of model transformation, testing techniques can help detect errors in a model transformation specification. This chapter presents a model transformation testing approach. It begins with a discussion of the specific need to ensure the correctness of model transformation, followed by a discussion on the limitations of current techniques. An overview of the model transformation testing approach is provided and an emphasis is given on the principles and the implementation of the model transformation testing engine M2MUnit. In addition, a case study is offered to illustrate using this approach to assist in detecting the errors in ECL specifications. Related work and concluding remarks are presented in the rest of this chapter.

5.1 Motivation

Model transformation is the core process in MDE for providing automation in software development [Sendall and Kozaczynski, 03]. Particularly, model-to-model transformation is investigated in the dissertation research to facilitate change evolution within MDE. To improve the reliability of such automation, validation and verification techniques and tools are needed to ensure the correctness of model transformation, as discussed in the following section. Although there are various techniques that facilitate

quality assurance of model transformation, a testing approach is investigated in this dissertation to improve the correctness of model transformation.

5.1.1 The Need to Ensure the Correctness of Model Transformation

As discussed in Chapter 2, there are different types of model transformation. Examples of such transformation are exogenous transformation (e.g., model-to-code transformation for generating code from models) and endogenous transformation (e.g., model-to-model transformation for altering the internal structure of the model representation itself). The model transformation approach discussed in Chapter 3 supports endogenous transformation. To perform a model transformation, the source models and the ECL transformation specification are taken by the transformation engine C-SAW as input to generate the target model as output. In such a model transformation environment, assuming the model transformation engine works correctly and the source models are properly specified, the quality of the transformed results depends on the correctness of the model transformation specifications.

As defined in [Mens and Van Gorp, 05], there are two types of correctness. One is *syntactic correctness,* which is defined as, “Given a well-formed source model, can we guarantee that the target model produced by the transformation is well-formed?” The other is *semantic correctness*, which is a more significant and complex issue, “Does the produced target model have the expected semantic properties?” In this research, a model transformation specification is correct if the produced model meets its specified requirements with both the expected syntactic and semantic correctness. Specifically, the reasons for validating the correctness of a model transformation specification include:

- **Transformation specifications are error-prone:** like the code in an implementation, transformation specifications are written by humans and susceptible to errors. Also, transformation specifications need to define complicated model computation logic such as model navigation, selection and manipulation, which makes it hard to specify correctly.
- **Transformation specifications are usually applied to a collection of models:** the input of a model transformation is a single model or a collection of models. When MDE is applied to develop a large-scale and complex system, it is common to apply transformation specifications to a large set of models. Before a transformation specification is performed on a large quantity of models, it is prudent to first test its correctness on a small set of models.
- **Transformation specifications are reusable:** because it takes intensive effort to define model transformations, the reusability of a transformation specification is critical to reduce human effort. Before a model transformation is reused in the same domain or across similar domains, it is also necessary to ensure its correctness.

Thus, there is a need for verification and validation techniques and tools to assist in finding and correcting the errors in model transformation specifications. At the implementation level, traditional software engineering methods and practices such as testing have been widely used in ensuring the quality of software development. However, at the modeling level, research efforts and best practices are still needed to improve the quality of models and model transformations. The need for model transformation testing is discussed in the following section.

5.1.2 The Need for Model Transformation Testing

Verification and validation are well-established techniques for improving the quality of a software artifact within the overall software development lifecycle [Harrold, 00], [Adrion et al., 82]. These techniques can be divided into two forms: *static analysis* and *dynamic analysis*. Static analysis does not require execution of software artifacts; this form of verification includes model checking and proof of correctness. Execution-based testing is an important form of dynamic analysis that is performed to support quality assurance in traditional software development [Harrold, 00].

As a new emerging software development paradigm, MDE highlights the need for verification and validation techniques that are specific to model and model transformation artifacts. Currently, there are a variety of verification techniques proposed for model transformation (e.g., model checking [Hatcliff et al., 03], [Holzmann, 97], [Schmidt and Varró, 03], simulation [Yilmaz, 01] and theorem proving [Varró et al., 02]). Common to all of these verification techniques is that they rely on a formal semantics of the specification or programming language concerned.

Despite the relative maturity of formal verification within software engineering research, practical applications are limited to safety-critical and embedded systems [Clarke and Wing, 96]. Reasons for this include the complexity of formal specification techniques [Adrion et al., 82] and the lack of training of software engineers in applying them [Hinchey et al., 96]. Furthermore, there are also well-known limitations for formal verification such as the state-explosion problem within model checking [Hinchey et al., 96].

Execution-based testing is widely used in practice to provide confidence in the quality of software [Harrold, 00]. Compared to formal verification, testing has several advantages that make it a practical method to improve the quality of software. These advantages are: 1) the relative ease with which many of the testing activities can be performed; 2) the software artifacts being developed (e.g., model transformation specifications) can be executed in its expected environment; 3) much of the testing process can be automated [Harrold, 00]. Model transformation specifications are executable, which makes execution-based testing a feasible approach to finding transformation faults by executing specifications within the model transformation environment without the need to translate models and transformations to formal specifications and to develop analytic models for formal verification.

In contrast to formal verification, model transformation testing has been developed as a contribution of the dissertation to validate model transformation. It aims at improving the confidence that a model transformation specification meets its requirements, but cannot prove any property as a guarantee (i.e., model transformation testing cannot assert the absence of errors, but is useful in revealing their presence, as noted by Dijsktra in relation to general testing [Dijkstra, 72]). The following subsections discuss the investigated testing approach for assisting in improving the quality of model transformations.

5.2 A Framework for Model Transformation Testing

There are various levels of software testing such as unit testing [Zhu et al., 97], and system testing [Al Dallal and Sorenson, 02]. Unit testing is a procedure that aims at

validating individual software units or components. System testing is conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements. In the research described in this dissertation, model transformation testing is developed to support unit testing a model transformation as a modular unit (e.g., the ECL strategy). Theoretically, a complete verification of a program or a model transformation specification can only be obtained by performing exhaustive testing for every element of the domain. However, this technique is not practical because functional domains are sufficiently large to make the number of required test cases infeasible [Adrion et al., 82]. In practice, testing relies on the construction of a finite number of test cases and execution of parts or all of the system for the correctness of the test cases [Harrold, 00], [Zhu et al., 97]. A model transformation testing framework should facilitate the construction and execution of test cases.

5.2.1 Overview

Model transformation testing involves executing a specification with the intent of finding errors [Lin et al., 05]. A testing framework should assist in generating tests, running tests, and analyzing tests. Figure 5-1 shows the framework for model transformation testing.

There are three primary components to the testing framework: test case constructor, testing engine, and test analyzer. The *test case constructor* consumes the test specification and produces a suite of test cases that are necessary for testing a transformation specification. The generated test cases are passed to the *testing engine* to be executed. The *test analyzer* visualizes the results and provides a capability to navigate

among any differences. The research provides tool support for executing and analyzing tests, which is realized by a testing engine. An assumption is that test suites will be constructed manually by transformation testers.

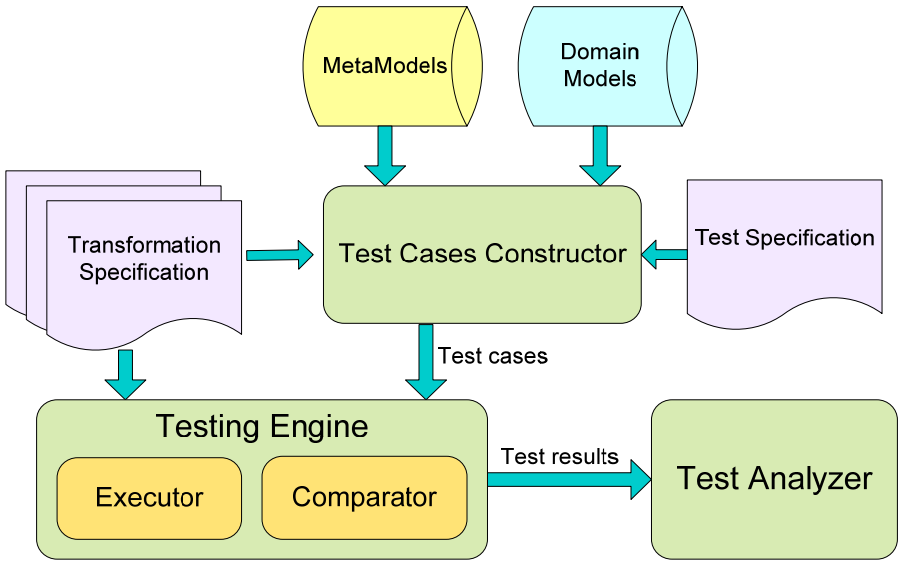


Figure 5-1 *- The model transformation testing framework*

5.2.2 Model Transformation Testing Engine: M2MUnit

To provide tool support for executing test cases that are needed for testing a model transformation specification, a model transformation testing engine called M2MUnit has been developed as a GME plug-in to run test cases and visualize the test results. A test case contains an input model, the to-be-tested model transformation specification and an expected model. Figure 5-2 shows an overview of the model transformation testing engine M2MUnit.

As shown in Figure 5-2, there are three major components within the testing engine: *an executor*, *a comparator* and *a test analyzer*. The executor is responsible for

executing the transformation specification on the input model to generate the output model. The comparator considers the output model to the expected model and collects the results of comparison. To assist in comprehending test results, basic visualization functionality of the test analyzer is also implemented within M2MUnit to structurally highlight the detected model differences. During the executor and comparator steps, the metamodel provides required information on types and constraints that are needed to assist in comparison of the expected and output models. Moreover, critical data that are included in a test case is indicated in Figure 5-2 such as input model, expected model and to-be-tested specification.

The correctness of a model transformation specification can be determined by checking if the output of the model transformation satisfies its intent (i.e., when there are no differences between the output model and the expected model). If there are no differences between the actual output and expected models, it can be inferred that the model transformation is correct with respect to the given test specification. If there are differences between the output and expected models, the errors in the transformation specification need to be isolated and removed.

The role of the executor is essentially a model transformation engine with functionality performed by C-SAW. Also, model comparison is performed between an expected model and an output model that are not subsequent versions. The output model is produced by the executor and the expected model is constructed by a tester. As discussed in Chapter 4, DSMDiff algorithms do not require two models to be subsequent versions. Thus, DSMDiff serves as the model comparator of M2MUnit to perform the model comparison and is also responsible for visualizing the produced differences as the

test analyzer. In fact, the development of DSMDiff was originally motivated by research on model transformation testing [Lin et al., 05].

To illustrate the feasibility and utility of the transformation testing framework, the next section describes a case study of testing a model transformation.

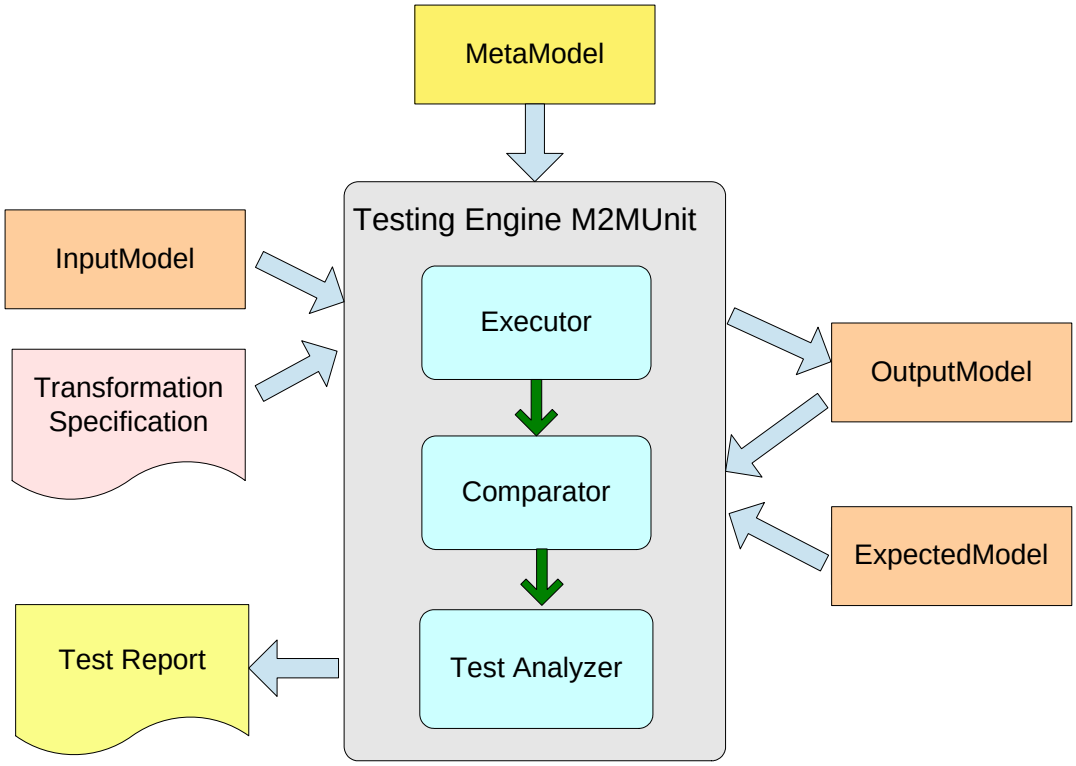


Figure 5-2 *- The model transformation testing engine M2MUnit*

5.3 Case Study

This case study is performed on an experimental platform, the Embedded Systems Modeling Language (ESML) introduced in Chapter 3, which is a freely available domain-specific graphical modeling language developed for modeling real-time mission computing embedded avionics applications [Sharp, 00]. There are over 50 ESML

component models used for this case study that communicate with each other via a real-time event-channel mechanism. An ESML component model may contain several data elements. This case study shows how M2MUnit can assist in finding errors in a transformation specification.

5.3.1 Overview of the Test Case

The test case is designed to validate an ECL specification developed for the following model transformation task: 1) find all the `Data` atoms in a component model, 2) create a `Log` atom for each `Data` atom, and then set its `Kind` attribute to "On Method Entry" and its `MethodList` attribute to "update," and 3) create a connection from the `Log` atom to its corresponding `Data` atom. Figure 5-3 and Figure 5-4 represent the input model and the expected model of this transformation task, respectively. The input model contains a `Data` atom called `numberOfUsers`. The expected model contains a `Log` atom called `LogOnMethodEntry`, which connects to the `Data` atom `numberOfUsers`. The `Kind` attribute of `LogOnMethodEntry` is set to "On Method Entry" and the `MethodList` attribute is set to "update." Such an expected model represents a correct transformation output.

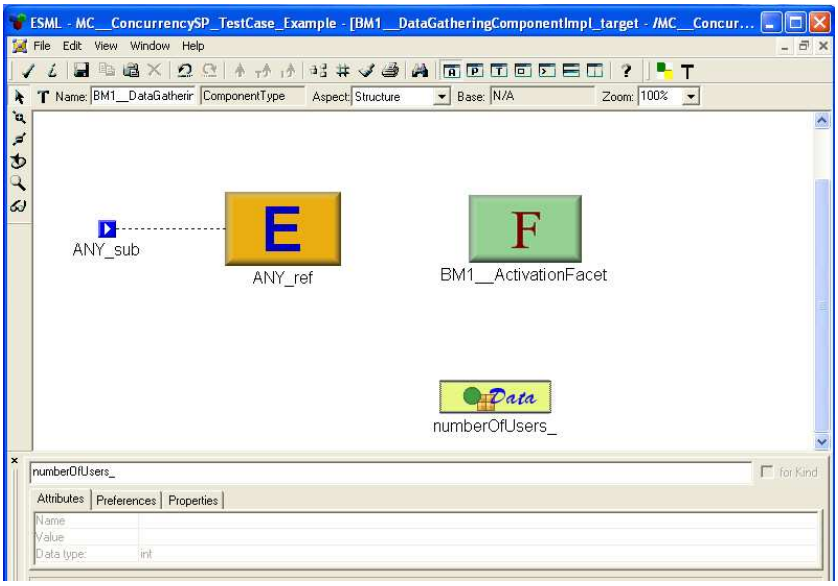


Figure 5-3 - *The input model prior to model transformation*

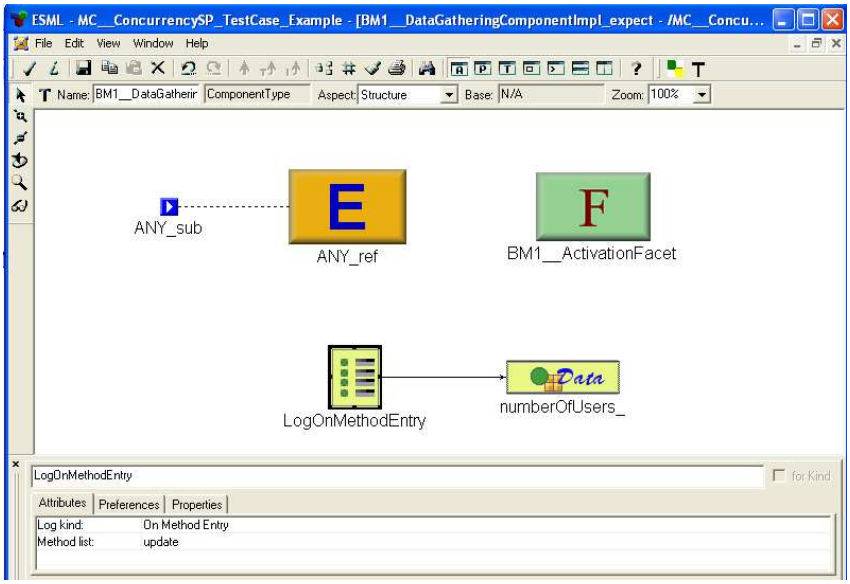


Figure 5-4 - *The expected model for model transformation testing*

To perform such a task by C-SAW, an ECL specification can be defined to transform the input model to the expected model. Listing 5-1 represents the initial ECL model transformation specification developed to accomplish the prescribed

transformation of the case study. This specification defines one aspect and two strategies. The `Start` aspect finds the input model and applies the `FindData` strategy. The `FindData` strategy specifies the search criteria to find all the `Data` atoms. The `AddLog` strategy is executed on those `Data` atoms identified by `FindData`. The `AddLog` strategy specifies the behavior to create the `Log` atom for each `Data` atom. Before this specification is applied to all component models and reused later, it is necessary to test its correctness.

```
1  strategy FindData()
2  {
3     atoms()->select(a | a.kindOf() == "Data")->AddLog();
4  }
5
6  strategy AddLog()
7  {
8    declare parentModel : model;
9    declare dataAtom, logAtom : atom;
10
11   dataAtom := self;
12   parentModel := parent();
13
14   logAtom := parentModel.addAtom("Log", "LogOnMethodEntry");
15   parentModel.addAtom("Log", "LogOnRead");
16   logAtom.setAttribute("Kind", "On Write");
17   logAtom.setAttribute("MethodList", "update");
18 }
19
20 aspect Start( )
21 {
22   rootFolder( ).findFolder("ComponentTypes").models()->
23     select(m|m.name().endWith( "DataGatheringComponentImpl_target"))->FindData();
24 }
```

Listing 5-1 - *The to-be-tested ECL specification*

5.3.2 Execution of the Test Case

The test case is constructed as a GME project, from which M2MUnit is invoked as a GME plugin. The execution of the test case includes two steps: first, the to-be-tested ECL specification is executed by the executor to produce an output target model; second,

the output target model and the expected model are sent to the comparator, which compares the models and passes the result to the test analyzer to be visualized.

Figure 5-5 shows the output model. When comparing it to the expected model, there are three differences, as shown in Figure 5-6. Figure 5-7 shows the visualization of the detected differences in a tree view.

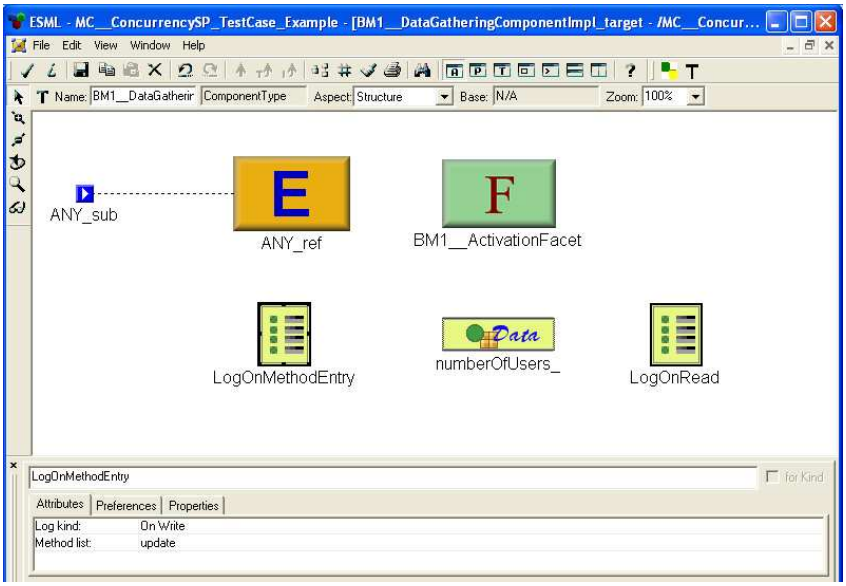


Figure 5-5 - *The output model after model transformation*

Because of these detected differences between the output model and the expected model, the ECL specification is suspected to have errors. To discover these errors, the following differences need to be examined:

- **Difference 1:** an extra atom `LogOnRead` is inserted in the output model, which needs to be deleted and is highlighted in red.
- **Difference 2:** there is a missing connection from `LogOnMethodEntry` to `numberOfUsers`, which needs to be created in the output model and is highlighted in gray.

- **Difference 3:** the `kind` attribute of the `LogOnMethodEntry` has a different value “On Write" from the expected value “On Method Entry,” which needs to be changed and is highlighted in green.

Figure 5-6 - *A summary of the detected differences*

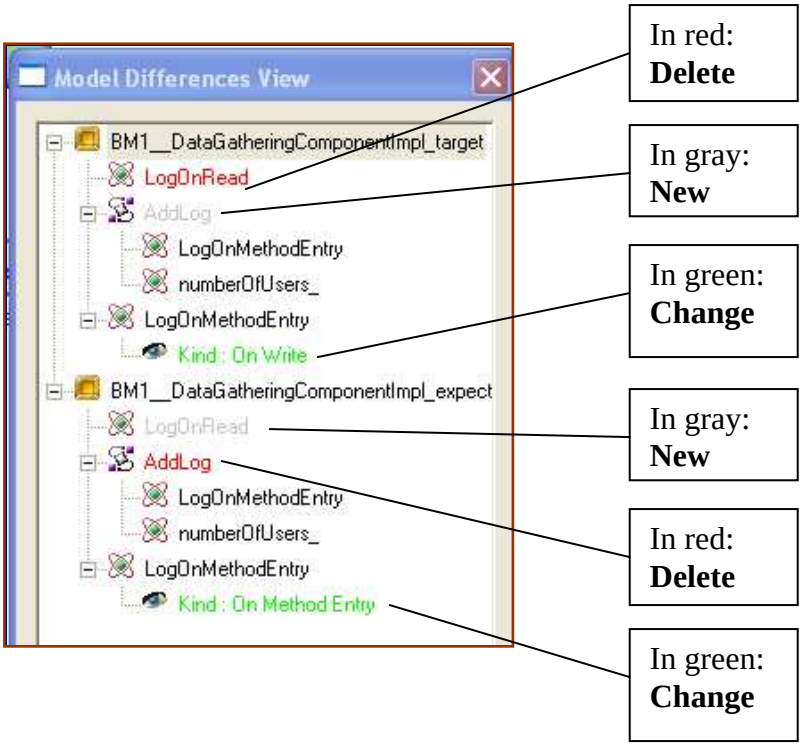


Figure 5-7 - *Visualization of the detected differences*

5.3.3 Correction of the Model Transformation Specification

According to the test results, it is obvious that there are three corrections that need to be made to initial the transformation specification. One correction is to add a statement

that will create the connection between `LogOnMethodEntry` and `numberOfUsers`. The second correction is to delete the line that adds `LogOnRead`: `parentModel.addAtom("Log", "LogOnRead")`. The third correction is to change the value of the `Kind` attribute from "On Write" to "On Method Entry." The modified transformation specification is shown in Listing 5-2 with the corrections underlined or marked by strikethrough. However, this does not imply that the correction is automated – the corrections need to be made manually after observing the test results.

```
1  strategy FindData()
2  {
3    atoms()->select(a | a.kindOf() == "Data")->AddLog();
4  }
5
6  strategy AddLog()
7  {
8    declare parentModel : model;
9    declare dataAtom, logAtom : atom;
10
11   dataAtom := self;
12   parentModel := parent();
13
14   logAtom := parentModel.addAtom("Log", "LogOnMethodEntry");
15   parentModel.addAtom("Log", "LogOnRead");
16   logAtom.setAttribute("Kind", "On Method Entry");
17   logAtom.setAttribute("MethodList", "update");
18
19   parentModel.addConnection("AddLog", logAtom, dataAtom);
20 }
21
22 aspect Start( )
23 {
24    rootFolder().findFolder("ComponentTypes").models()->
25      select(m|m.name().endWith("DataGatheringComponentImpl_target"))->FindData();
26    }
```

Listing 5-2 - *The corrected ECL specification*

5.4 Related Work

Regarding formal verification, model checking is a widely used technique for verification of model properties (e.g., the SPIN Model checker [Holzmann, 97], the Cadena model checking toolsuite [Hatcliff et al., 03], and the CheckVML tool [Schmidt

and Varró, 03]). SPIN is a verification system for models of distributed software, which has been used to detect design errors for a broad range of applications ranging from a high-level description of distributed algorithms to detailed code for controlling telephone exchanges. The main idea of SPIN is that system behaviors and requirements are specified as two aspects of the design by defining a verification or prototype in a specification language. The prototype is verified by checking the internal and mutual consistency of the requirements and behaviors. The Cadena model checking toolsuite extends SPIN to add support for objects, functions, and references. CheckVML is a tool for model checking dynamic consistency properties in arbitrary well-formed instance models. It first translates models into a tool-independent intermediate representation, then automatically generates the input language of the back-end model checker tool (e.g., SPIN). Generally, model checking is based on formal specification languages and automata theory.

A mathematically proven technique for validating model transformations is proposed in [Varró et al., 02], where the main idea is to perform mathematical model transformations in order to integrate UML-based system models and mathematical models of formal verification tools. However, such an approach requires a detailed mathematical description and analysis of models and transformations, which may limit the applicability for general use. Other formal methods include temporal logics [Manna and Pnueli, 92] and assertions [Hoare, 69], which have been proposed to verify the conformance of a program with its specification at the implementation level.

These techniques are based on formal specification and may be used for formal proof of correctness. However, proving correctness of model transformations formally is

difficult and requires formal verification techniques. Execution-based testing is a feasible alternative to finding transformation faults without the need to translate models and transformations to formal specifications. Using testing to determine the correctness of model and model transformation also provides opportunities to bring mature software engineering techniques to modeling practice.

There has been work on applying testing to validate design models. For example, [Pilskalns et al., 07] presents an approach for testing UML design models to uncover inconsistencies. This approach uses behavioral views such as sequence diagrams to simulate state change in an aggregate model, which is the artifact of merging information from behavioral and structural UML views. OCL pre-conditions, post-conditions and invariants are used as a test oracle. However, there are only a few reports in the literature regarding efforts that provide the facilities for model transformation testing. Our own work on model transformation testing published as [Lin et al., 04], [Lin et al., 05] is one of the earliest reports addressing this issue. Another initial work on model transformation testing is [Fleurey et al., 04], which presents a general view of the roles of testing in the different stages of model-driven engineering, and a more detailed exploration of approaches to testing model transformations. Based on this, Fleurey et al. highlight the particular issues for the different testing tasks, including adequacy criteria, test oracles and automatic test data generation. More recently, there have been a few works that expand research on model transformation testing by exploring additional testing issues. For example, a metamodel-based approach for test generation is proposed in [Brottier et al., 06] for model transformation testing. In [Mottu et al., 06], mutation analysis is investigated for model transformation to evaluate the quality of test data. Experiences in

validating model transformations using a white box approach is reported in [Kuster and Abd-El-Razik, 06]. Such research has focused on test coverage, test data analysis and test generation of source code from models, but this dissertation effort primarily aims at providing a testing engine to run tests and analyze the results.

5.5 Conclusion

This chapter presents another contribution of the dissertation on model transformation testing to improve the accuracy of transformation results. In addition to the developed model transformation testing framework and the unit testing approach for model transformation, the model transformation testing engine M2MUnit has been implemented to provide support to execute test cases with the intent of revealing errors in the model transformation specification. Distinguished from classical software testing tools, to determine whether a model transformation test passes or fails requires comparison of the actual output model with the expected model, which requires model differencing algorithms and visualization. The DSMDiff approach presented in Chapter 4 provides solutions for model differentiation and visualization.

The result presented in this chapter provides an initial solution to applying testing techniques at the modeling level. There are other fundamental issues that need to be explored deeply in order to provide mature testing solutions. In the next chapter, several critical issues are proposed for advancing the research on model transformation testing.

CHAPTER 6

FUTURE WORK

This chapter outlines research directions that will be investigated as future work. To alleviate the complexity of developing model transformations, research into the idea of Model Transformation by Example (MTBE) is proposed to assist users in constructing model transformation rules through interaction with modeling tools. Test generation from test specifications and metamodel-based coverage criteria to evaluate test adequacy are also discussed as future work for model transformation testing. To provide support to isolate the errors in model transformation specifications, model transformation debugging is another software engineering practice that needs to be investigated as an extension of the research described in this dissertation.

6.1 Model Transformation by Example (MTBE)

There are several dozen model transformation languages that have been proposed over the last five years, with each having a unique syntax and style [Sendall and Kozaczynski, 03], [Mens and Van Gorp, 05], [Czarnecki and Helsen, 06]. Because these model transformation languages are based on various techniques (e.g., relational approach or graph rewriting approach) and not all the language concepts are explicitly specified (e.g., transformation rule scheduling and model matching mechanism), it is

difficult for certain classes of users of the model transformation languages to write model transformation rules. To simplify the task of developing a model transformation specification, Model Transformation by Example (MTBE) is an approach that enables an end-user to record the type of transformation that they desire and then have the modeling tool infer the transformation rule corresponding to that example [Varró, 06], [Wimmer et al., 07].

The idea of MTBE has similar goals to Programming by Example (PBE) [Lieberman, 00] and Query by Example (QBE) [Zloof, 77] in that a user interacts with a modeling tool that records a set of actions and generates some representative script that can replay the recorded actions. The inferred script could represent a fragment of code, a database query, or in the case of MTBE, a model transformation rule. To realize MTBE within a modeling tool, an event tracing mechanism needs to be developed and algorithms are needed for inferring a transformation rule from the set of event traces.

Typically, there is a specific series of events that occur during a user modeling session; lower-level events are user interactions with the windowing system (e.g., “mouse click”) and higher-level events (composed from a sequence of lower-level events in a certain context) correspond to the core meaning of the user actions (e.g., “add attribute” or “delete connection”). To support event tracing of user interaction within most modeling tools requires a fair amount of manual customization by either modifying the source code of the modeling tool or hooking into the tool’s published event channel (if it exists).

With respect to programming languages, an event is a detectable action occurring during program execution [Auguston, 98] (e.g., expression evaluation, subroutine call,

statement execution, message sending/receiving). Within the context of MTBE, an event corresponds to some action made by a user during interaction with a modeling tool. An event is delimited by a time interval with a beginning and an end. Such a model to record the history of change events is defined as the event trace [Auguston et al., 03].

To support general model evolution analysis, a new language will be designed for expressing computations over event traces as a basis for inferring model transformation rules. Such a language will allow analysis of model changes to be generalized, rather than fixed within the modeling tool. This language will be defined by an event grammar, which represents all of the possible events of interest within a given modeling context. For example, an “add connection” event is represented as the following:

```
FOREACH C: atom-atom-connection
FROM A1: atom A2: atom
C.log( SAY( "added connection " C.conn_name
        "[" A1._name "-" A2._name"]" ) )
```

The tangible asset of this proposed work will be a language processor for event grammars, which will generate the requisite instrumentation of the modeling tool to perform the analysis specified in a query that is based on the event grammar. Integrating the language processor into a collection of modeling tools allows the specification of model evolution analysis in a tool-independent manner. This approach is distinguished from another MTBE approach proposed by Dániel Varró [Varró, 06], where transformation rules are derived semi-automatically from an initial prototypical set of interrelated source and target models. These initial model pairs describe critical cases of the model transformation problem in a purely declarative way. The derived transformation rules can be refined later by adding further source-target model pairs. The main advantage of the approach is also that transformation designers do not need to learn

a new model transformation language. Compared to Varró's approach and manual adaptation of a modeling tool for a desired model evolution analysis task, the proposed investigation of MTBE using event grammars has the following apparent advantages:

- The notion of an event grammar provides a general basis for model evolution tasks. This makes it possible to reason about the meaning of model evolution at an appropriate level of granularity.
- An event grammar provides a coordinated system to refer to any interesting event in the evolution history of the user modeling session. Assertions about different modeling events may be specified and checked in an event query to determine if any sequence of undesirable changes were made.
- Trace collection is implemented by instrumentation of a modeling tool. Because only a small projection of the entire event trace is used by any set of rules, it is possible to implement selective instrumentation, powerful event filtering and other optimizations to reduce dramatically the size of the collected trace. More importantly, special-purpose instrumentation will enable most analyses to be moved from post-mortem to live response during the actual modeling session.

Furthermore, it may be possible for the algorithms and techniques of PBE and QBE to be adapted to the context of MTBE. An extensive set of event queries will be created to correspond with the event grammar for model evolution analysis. That is, during the record phase of MTBE, all of the relevant modeling events will be logged as an event trace. The event trace will then serve as input to the MTBE algorithms that generate the corresponding transformation rule. By plugging different back-ends into the MTBE

algorithms, it is anticipated that various model transformation languages can be inferred from a common example.

For each new evolution analysis task (e.g., version control) that is needed, similar manual adaptation may be required with slight variation. The proposed work will produce an approach that generalizes the evolution analysis task and the underlying event channel of the modeling tool to allow the rapid addition of new analysis capabilities across a range of DSM tools.

6.2 Toward a Complete Model Transformation Testing Framework

The dissertation work on model transformation testing as discussed in Chapter 5 is an initial step towards partially automating test execution and assisting in test result analysis. To develop a more mature approach to model transformation testing, additional research efforts are needed to provide support for test generation and test adequacy analysis.

Test generation creates a set of test cases for testing a model transformation specification. A test case usually needs to contain general information, input data, test condition and action, and the expected result. Manually creating test cases is a human intensive task. To reduce the human effort in generating tests by improving the degree of automation, a test specification is required to define test cases, even test suites and a test execution sequence. Such a language may be an extension to the model transformation language that provides language constructs for defining tests. An envisioned test specification example for testing ECL specification is shown as the following.

Test test1
{
Specification file: *"C:\ESML\ModelComparison1\Strategies\addLog.spc"*
Start Strategy: *FindData*
GME Project: *"C:\ESML\ModelCompariosn1\modelComparison1.mga"*
Input model: *"ComponentTypes\DataGatheringComponentImpl"*
Output model: *"ComponentTypes\Output1"*
Expected model: *"ComponentTypes\Expected1"*
Pass: *Output1 = Expected1*
}

Such a test is composed of a *name* (e.g., "test1") and *body*. The test body defines the locations and identifiers of the model transformation specification, the start procedure to execute, a test project built for the testing purpose, the input source and output target models, the expected model, as well as the criteria for asserting a successful pass (i.e., the test oracle is a comparison between two models). Such a test specification can be written manually by a test developer or generated by the modeling environment with specific support to directly select the involved ECL specification, the input model and expected model. Thus, test developers can build and edit tests from within the modeling environment. An effective test specification language also needs to support the definition of test suites and a test execution sequence. In addition, it may also need to support various types of test oracles, which provide mechanisms for specifying expected behaviors and verifying that test executions meet the specification. Currently, the M2MUnit testing engine only supports one type of oracle, i.e., comparing the actual output and the expected output that are model type. However, it is possible for other types of test oracles to compare the actual output and the expected output that are primitive types such as integer, double or string or just a fragment of a model.

Currently, the M2MUnit testing approach has not investigated the concept of test coverage adequacy to formally ensure that the transformation specification has been fully

tested. Thus, more research efforts are needed to provide test criteria in the context of model transformation testing to ensure the test adequacy. Traditional software test coverage criteria such as statement coverage, branch coverage and path coverage [Schach, 07], [Adrion et al., 82] may be applied or adapted to a procedural style of model transformation such as used in ECL. In addition, other criteria specific to a particular modeling notation may be developed to help evaluate the test adequacy. Metamodel coverage is such a criterion to evaluate the adequacy of model transformation testing [Fleurey et al., 04]. Metamodeling provides a way to precisely define a domain. Test adequacy can be achieved by generating test cases that cover the domain entities and their relationships defined in a metamodel. For example, a MOF-based metamodel can reuse existing criteria defined for UML class diagrams [Fleurey et al., 04], [Andrews et al., 03]. It has been recognized that the input metamodel for a transformation is usually larger than the actual metamodel used by a transformation. Such an actual metamodel is a subset of the input metamodel and is called the *effective metamodel* [Brottier et al., 06]. An effective metamodel can be derived from the to-be-tested transformation and used for generating valid models for tests [Baudry et al., 06]. However, it is tedious to generate models manually. An automation technique can be applied to generate sufficient instance models from a metamodel for large scale testing [Ehrig et al., 06].

In summary, the future work for the M2MUnit testing framework include: 1) a test specification language and tool support for generation and execution of tests or test suite, and 2) coverage criteria for evaluating test adequacy, especially through metamodel-based analysis.

6.3 Model Transformation Debugging

Model transformation testing assists in determining the presence of errors in model transformation specifications. After determining that errors exist in a model transformation, the transformation specification must be investigated in order to ascertain the cause of the error. Model transformation debugging is a process to identify the specific location of the error in a model transformation specification.

Currently, C-SAW only supports transformation developers to write “print” statements for the purpose of debugging. A debugger is needed to offer support for tracing down why the transformation specifications do not work as expected. A model transformation debugger has many of the same functionalities as most debugging tools to support setting breakpoints, stepping through one statement at a time and reviewing the values of the local variables and status of affected models [Rosenberg, 96].

A model transformation debugger would allow the step-wise execution of a transformation to enable the viewing of properties of the transformed model as it is being changed in the modeling tool. A major technical problem of a model transformation debugger is to visualize the status of affected models during execution. For example, it may be a large set of models whose status has changed after executing a set of statements. A challenge of this future work is to represent the change status efficiently.

The testing toolsuite and the debugging facility together will offer a synergistic benefit for detecting errors in a transformation specification and isolating the specific cause of the error.

CHAPTER 7

CONCLUSIONS

With the expanded focus of software and system models has come the urgent need to manage complex change evolution within the model representation. Designers must be able to examine various design alternatives quickly and easily among myriad and diverse configuration possibilities. Existing approaches to exploring model change evolution include: 1) modifying the model by hand within the model editor, or 2) writing programs in C++ or Java to perform the change. Both of these approaches have limitations, which degrades the capability of modeling to explore system issues such as adaptability and scalability. A manual approach to evolving a large-scale model is often time consuming and error prone, especially if the size of a system model continues to grow. Meanwhile, many model change concerns crosscut the model hierarchy, which usually requires a considerable amount of typing and mouse clicking to navigate and manipulate a model in order to make a change. There is increasing accidental complexity when using low-level languages such as C++ or Java to define high-level model change evolution concerns such as model querying, navigation and transformation.

Despite recent advances in modeling tools, many modeling tasks can still benefit from increased automation. The overall goal of the research described in this dissertation is to provide an automated model transformation approach to model evolution. The key

contributions include: 1) investigating the new application of model transformation to address model evolution concerns, especially the system-wide adaptability and scalability issues, 2) applying a testing process to model transformations, which assists in improving the quality of a transformation; and 3) developing algorithms to compute and visualize differences between models. The main benefit of the research is reduced human effort and potential errors in model evolution. The following sections summarize the research contributions in each of these areas.

7.1 The C-SAW Model Transformation Approach

To assist in evolving models rapidly and correctly, the research described in this dissertation has developed a domain-independent model transformation approach. Evolved from an earlier aspect modeling language originally designed to address crosscutting modeling concerns [Gray et al., 01], the Embedded Constraint Language (ECL) has been developed to support additional modeling types and provide new operations for model transformation. ECL is a high-level textual language that supports an imperative model transformation style. Compared to other model transformation languages, ECL is a small but expressive language that aims at defining model transformation where the source and target models belong to the same metamodel (i.e., an endogenous transformation language). C-SAW serves as the model transformation engine associated with the new ECL extensions described in Chapter 3. Various types of model evolution tasks can be defined concisely in ECL and executed automatically by C-SAW.

The dissertation describes the use of C-SAW to address the accidental complexities associated with current modeling practice (e.g., manually evolving the deep

hierarchical structures of large system models can be error prone and labor intensive). Particularly, this dissertation focuses on addressing two system development issues through modeling: scalability and adaptability. At the modeling level, system scalability is correspondingly formulated as a model scalability problem. A transformation specified in ECL can serve as a model replicator that scales models up or down with flexibility in order to explore system-wide properties such as performance and resource allocation. Also, the research described in this dissertation has investigated using C-SAW to address system adaptability issues through modeling. For example, systems have to reconfigure themselves according to fluctuations in their environment. In most cases, such adaptation concerns crosscut the system model and are hard to specify. C-SAW permits the separation of crosscutting concerns at the modeling level, which assists end-users in rapidly exploring design alternatives that would be infeasible to perform manually. As an extension to the earlier investigation in [Gray et al., 01] for modularizing crosscutting modeling concerns, the research described in this dissertation applied C-SAW to address new concerns such as component deployment and synchronization that often spread across system components.

To simplify the development of model transformation, as a future extension to the C-SAW work, the dissertation proposes an approach called Model Transformation by Example (MTBE) to generate model transformation rules through a user's interaction with the modeling tool. An event trace mechanism and algorithms that infer model transformation rules form recorded events need to be developed to realize the vision of MTBE.

7.2 Model Transformation Testing

Another important issue of model transformation is to ensure its correctness. There are a variety of formal methods proposed for validation and verification for models and associated transformations (e.g., model checking). However, the applicability of formal methods is limited due to the complexity of formal techniques and the lack of training of many software engineers in applying them [Hinchey et al., 96], [Gogolla, 04]. Software engineering practices such as execution-based testing represent a feasible approach for finding transformation faults without the need to translate models and transformations to formal specifications. As one of the earliest research efforts to investigate model transformation testing, the dissertation has described a unit testing approach (M2MUnit) to help detect errors in model transformation specifications where a model transformation testing engine provides support to execute test cases with the intent of revealing errors in the transformation specification.

The basic functionality includes execution of the transformations, comparison of the actual output model and the expected model, and visualization of the test results. Distinguished from classical software testing tools, to determine whether a model transformation test passes or fails requires comparison of the actual output model with the expected model, which necessitates model differencing algorithms and visualization. If there are no differences between the actual output and expected models, it can be inferred that the model transformation is correct with respect to the given test specification. If there are differences between the output and expected models, the errors in the transformation specification need to be isolated and removed.

To further advance model transformation testing, the dissertation proposes several important issues for future investigation. These issues include a test specification language to support test generation and metamodel-based coverage criteria to evaluate test adequacy. Also, to provide a capability to locate errors in model transformation specification, model transformation debugging is proposed as another software engineering practice to improve the quality of model transformation.

7.3 Differencing Algorithms and Tools for Domain-Specific Models

Driven by the need for model comparison required by model transformation testing, model differencing algorithms and an associated tool called DSMDiff have been developed to compute differences between models.

Theoretically, the generic model comparison problem is similar to the graph isomorphism problem, which is known to belong to NP [Garey and Johnson, 79]. The computational complexity of graph matching algorithms is the major hindrance to applying them to practical applications in modeling. To provide efficient and reliable model differencing algorithms, the dissertation has developed a solution using the syntax of modeling languages to help handle conflicts during model matching and combine structural comparison to determine whether the two models are equivalent. In general, DSMDiff takes two models as hierarchical graphs, starts from the top-level of the two containment models and then continues comparison to the child submodels.

Compared to traditional UML model differentiation algorithms, comparison of domain-specific models is more challenging and characterized as: 1) domain-specific modeling is distinguished from traditional UML modeling because it is a variable-

metamodel approach whereas UML is a fixed-metamodel approach; 2) the underlying metamodeling mechanism used to define a DSML determines the properties and structures of domain-specific models; 3) domain-specific models may be formalized as hierarchical graphs annotated with a set of syntactical information. Based on these characteristics, DSMDiff has been developed as a metamodel-independent solution to discover the mappings and differences between any two domain-specific models.

Visualization of the result of model differentiation (i.e., structural model differences) is critical to assist in comprehending the mappings and differences between two models. To help communicate the discovered model differences, a research contribution has also investigated a visualization technique to display model differences structurally and highlight them using color and icons. For example, a tree browser has been developed to indicate the possible kinds of model differences (e.g., a missing element, or an extra element, or an element that has different values for some properties).

Based on complexity analysis, DSMDiff achieves polynomial time complexity. The applicability of DSMDiff has been discussed within the context of model transformation testing. In addition to model transformation testing, model differencing techniques are essential to many model development and management practices such as model versioning.

7.4 Validation of Research Results

The C-SAW transformation engine has been applied to support automated evolution of models on several different modeling languages over multiple domains. On different experimentation platforms, C-SAW was applied successfully to integrate

crosscutting concerns into system models automatically. For example, C-SAW was used to weave the concurrency mechanisms, synchronization and flight data recorder policies into component models of real-time control systems provided by Boeing [Gray et al., 04-b], [Gray et al., 06], [Zhang et al., 05-b]. More recently, C-SAW has been applied to improve the adaptability of component-based applications. For example, C-SAW was used to weave deployment concerns into PICML models that define component interfaces, along with their properties and system software building rules of component-based distributed systems [Balasubramanian et al., 06-a]. Using model replicators, four case studies [Gray et al., 05], [Lin et al., 07-a] were used to demonstrate C-SAW's ability to scale base models to large product-line instances. C-SAW was also used in Model-Driven Program Transformation (MDPT) and model refactoring [Zhang et al., 05-a]. In MDPT, the contribution was specific to a set of models and concerns (e.g., logging and concurrency concerns). Moreover, C-SAW has been used by several external researchers in their research. For example, Darío Correal from Colombia has applied C-SAW to address crosscutting concerns in workflow processes [Correal, 06]. The C-SAW web site contains software downloads, related papers, and several video demonstrations [C-SAW, 07].

The case studies have indicated the general applicability and flexibility of C-SAW to help evolve domain-specific models across various domains represented by different modeling languages (e.g., SIML and SRNML). These experimental results have also demonstrated that using C-SAW to automate model change evolution reduces the human effort and potential errors when compared to a corresponding manual technique.

To conclude, the escalating complexity of software and system models is making it difficult to rapidly explore the effects of a design decision. Automating such exploration with model transformation can improve both productivity and model quality.

LIST OF REFERENCES

[Adrion et al., 82] W. Richards Adrion, Martha A. Branstad and John C. Cherniavsky, "Validation, Verification, and Testing of Computer Software," *ACM Computing Surveys*, vol. 14 no. 2, June 1982, pp. 159-192.

[Agrawal, 03] Aditya Agrawal, Gábor Karsai, and Ákos Lédeczi, "An End-to-End Domain-Driven Software Development Framework," *18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) - Domain-driven Track,* Anaheim, California, October 2003, pp. 8-15.

[Aho et al., 07] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman, *Compilers Principles, Techniques, and Tools*, 2nd edition, Addison-Wesley, 2007.

[Al Dallal and Sorenson, 02] Jehad Al Dallal, Paul Sorenson, "System Testing for Object-Oriented Frameworks Using Hook Technology," *17th IEEE International Conference on Automated Software Engineering (ASE)*, Edinburgh, Scotland, September 2002, pp. 231.

[Alanen and Porres, 02] Marcus Alanen and Ivan Porres, "Difference and Union of Models," *6th International Conference on Unified Modeling Language (UML)*, Springer-Verlag LNCS 2863, San Francisco, California, October 2003, pp. 2-17.

[Andrews et al., 03] Anneliese Andrews, Robert France, Sudipto Ghosh and Gerald Craig, "Test Adequacy Criteria for UML Design Models," *Software Testing, Verification and Reliability*, vol. 13 no. 2, April-June 2003, pp. 95-127.

[ANTLR, 07] ANTLR website, 2007, http://www.antlr.org/

[AOM, 07] Aspect-Oriented Modeling, http://www.aspect-modeling.org/aosd07/

[AS, 01] Object Management Group**,** Action Semantics for the UML, 2001, http://www.omg.org.

[Auguston, 98] Mikhail Auguston, "Building Program Behavior Models," *ECAI Workshop on Spatial and Temporal Reasoning*, Brighton, England, August 1998, pp. 19-26.

[Auguston et al., 03] Mikhail Auguston, Clinton Jeffery, and Scott Underwood, “A Monitoring Language for Run Time and Post-Mortem Behavior Analysis and Visualization,” *AADEBUG Workshop on Automated and Algorithmic Debugging*, Ghent, Belgium, September 2003.

[Balasubramanian et al., 06-a] Krishnakumar Balasubramanian, Aniruddha Gokhale, Yuehua Lin, Jing Zhang, and Jeff Gray, “Weaving Deployment Aspects into Domain-Specific Models,” *International Journal on Software Engineering and Knowledge Engineering*, June 2006, vol. 16 no.3, pp. 403-424.

[Balasubramanian et al., 06-b] Krishnakumar Balasubramanian, Aniruddha Gokhale, Gabor Karsai, Janos Sztipanovits, and Sandeep Neema, “Developing Applications Using Model-Driven Design Environments,” *IEEE Computer* (Special Issue on Model-Driven Engineering), February 2006, vol. 39 no. 2, pp. 33-40.

[Batory, 06] Don Batory, “Multiple Models in Model-Driven Engineering, Product Lines, and Metaprogramming,” *IBM Systems Journal*, vol. 45 no. 3, July 2006, pp. 451–461.

[Batory et al., 04] Don Batory, Jacob Neal Sarvela, and Axel Rauschmeyer, “Scaling Step-Wise Refinement,” *IEEE Transactions on Software Engineering*, vol. 30 no. 6, June 2004, pp. 355-371.

[Baudry et al., 06] Benoit Baudry, Trung Dinh-Trong, Jean-Marie Mottu, Devon Simmonds, Robert France, Sudipto Ghosh, Franck Fleurey, and Yves Le Traon, "Challenges for Model Transformation Testing," Proceedings of workshop on Integration of Model Driven Development and Model Driven Testing (IMDT), Bilbao, Spain, July 2006.

[Baxter et al., 04] Ira Baxter, Christopher Pidgeon, and Michael Mehlich, “DMS: Program Transformation for Practical Scalable Software Evolution,” *International Conference on Software Engineering (ICSE)*, Edinburgh, Scotland, May 2004, pp. 625-634.

[Bernstein, 03] Philip A. Bernstein, “Applying Model Management to Classical Meta Data Problems,” *The Conference on Innovative Database Research (CIDR)*, Asilomar, California, January 2003, pp. 209-220.

[Bézivin, 03] Jean Bézivin, “On the Unification Power of Models,” *Journal of Software and System Modeling*, vol. 4 no. 2, May 2005, pp. 171-188.

[Bézivin and Gerbé, 01] Jean Bézivin and Olivier Gerbé, “Towards a Precise Definition of the OMG/MDA Framework,” *Automated Software Engineering (ASE)*, San Diego, California, November 2001, pp. 273-280.

[Bézivin et al., 04] Jean Bézivin, Frédéric Jouault, and Patrick Valduriez, "On the Need for MegaModels," *OOPSLA Workshop on Best Practices for Model-Driven Software Development*, Vancouver, Canada, October 2004.

[Bondi, 00] André B. Bondi, "Characteristics of Scalability and Their Impact on Performance," *2nd International Workshop on Software and Performance*, Ottawa, Ontario, Canada, 2000, pp. 195–203.

[Booch et al., 99] Grady Booch, James Rumbaugh and Ivar Jacobson, *The Unified Modeling Language User Guide*, Addison Wesley, 1999.

[Brooks, 95] Frederic P. Brooks, *Mythical Man-Month*, Addison-Wesley, 1995.

[Brottier et al., 06] Erwan Brottier, Franck Fleurey, Jim Steel, Benoit Baudry, Yves Le Traon, "Metamodel-based Test Generation for Model Transformations: An Algorithm and a Tool," *17th International Symposium on Software Reliability Engineering (ISSRE)*, Raleigh, North Carolina, November 2006, pp. 85–94.

[Budinsky et al., 04] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick and Timothy J. Grose, *Eclipse Modeling Framework*, Addison-Wesley, 2004.

[Chawathe, 96] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom, "Change Detection in Hierarchically Structured Information," *The ACM SIGMOD International Conference on Management of Data*, Montreal, Canada, June 1996, pp. 493-504.

[Cicchetti, 07] Antonio Cicchetti, Davide Di Rusico, and Alfonso Pierantonio, "Metamodel Independent Approach to Difference Representation," *Journal of Object Technology* (Special Issue from TOOLS Europe 2007), June 2007, 20 pages.

[Clarke and Wing, 96] E. M. Clarke and J. M. Wing, "Formal Methods: State of the Art and Future Directions," *ACM Computing Surveys*, vol. 28, 1996, pp. 626–643.

[Clements and Northrop, 01] Paul Clements and Linda Northrop, *Software Product-lines: Practices and Patterns*, Addison-Wesley, 2001.

[CMW, 07] Object Management Group, Common Warehouse Metamodel Specification, http://www.omg.org/technology/documents/modeling_spec_catalog.htm#CWM

[Correal, 06] Darío Correal, "Definition and Execution of Multiple Viewpoints in Workflow Processes," *Companion to the 21st ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, October 2006, Portland, Oregon, pp. 760-761.

[C-SAW, 07] C-SAW: The Constraint Specification Weaver Website, 2007, http://www.cis/uab.edu/gray/Research/C-SAW

[Czarnecki and Helsen, 06] Krzysztof Czarnecki and Simon Helsen, "Feature-based Survey of Model Transformation Approaches," *IBM Systems Journal*, 2006, vol. 45 no. 3, pp. 621-646.

[Cuccuru et al., 05] Arnaud Cuccuru, Jean-Luc Dekeyser, Philippe Marquet, Pierre Boulet, "Towards UML2 Extensions for Compact Modeling of Regular Complex Topologies," *Model-Driven Engineering Languages and Systems (MoDELS)*, Springer-Verlag LNCS 3713, Montego Bay, Jamaica, October 2005, pp. 445-459.

[Deng et al., 08] Gan Deng, Douglas C. Schmidt, Aniruddha Gokhale, Jeff Gray, Yuehua Lin, and Gunther Lenz, "Evolution in Model-Driven Software Product-line Architectures," *Designing Software-Intensive Systems: Methods and Principles*, (Pierre Tiako, ed.), Idea Group, 2008.

[Dijkstra, 76] Edsger Wybe Dijkstra, ed., *A Discipline of Programming*, Prentice Hall, 1976.

[Dijkstra, 72] Edsger Dijsktra, "The Humble Programmer," *Communications of the ACM*, October 1972, pp. 859-866

[DSM Forum, 07] Domain-Specific Modeling Forum, 2007, http://www.dsmforum.org/tools.html

[Edwards, 04] George Edwards, Gan Deng, Douglas Schmidt, Aniruddha S. Gokhale, and Bala Natarajan, "Model-Driven Configuration and Deployment of Component Middleware Publish/Subscribe Services," *Generative Programming and Component Engineering (GPCE)*, Springer-Verlag LNCS 3286, Vancouver, Canada, October 2004, pp. 337-360.

[Ehrig et al., 06] Karsten Ehrig, Jochen M. Kuster, Gabriele Taentzer, and Jessica Winkelmann, "Generating Instance Models from Meta Models," *8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, Springer-Verlag LNCS 4037, Bologna, Italy, June 2006, pp. 156-170.

[Eick et al., 01] Stephen G. Eick, Joseph L. Steffen, and Eric E. Sumner, "SeeSoft--A Tool for Visualizing Line-Oriented Software Statistics," *IEEE Transactions on Software Engineering*, vol. 18 no. 11, 2001, pp. 957-968.

[Engels and Groenewegen, 00] Gregor Engels and Luuk Groenewegen, "Object-Oriented Modeling: A Roadmap," *Future of Software Engineering, Special Volume Published in Conjunction with ICSE 2000*, (Finkelstein, A., ed.), May 2000, pp. 103-116.

[Escher, 07] The Escher Repository, 2007. http://escher.isis.vanderbilt.edu

[Evans, 03] Eric Evans, *Domain-Driven Design: Tackling Complexity at the Heart of Software*, Addison-Wesley, 2003.

[Fermi, 07] Fermi lab, 2007, http://www.fnal.gov/

[Filman et al., 04] Robert Filman, Tzilla Elrad, Siobhan Clarke, and Mehmet Aksit, editors. *Aspect-Oriented Software Development*, Addison-Wesley, 2004.

[Fleurey et al., 04] Franck Fleurey, Jim Steel, Benoit Baudry, "Validation in model-driven engineering: testing model transformations," *1st International Workshop on Model, Design and Validation*, Rennes, Bretagne, France, November 2004, pp. 29–40

[France et al., 04] Robert France, Indrakshi Ray, Geri Georg, and Sudipto Ghosh, "An Aspect-Oriented Approach to Design Modeling," *IEE Proceedings on Software*, vol. 4 no. 151, August 2004, pp. 173-185.

[Frankel, 03] David S. Frankel, *Model Driven Architecture: Applying MDA to Enterprise Computing*, John Wiley and Sons, 2003.

[Fujaba, 07] The FUJABA Toolsuite. http://wwwcs.uni-paderborn.de/cs/fujaba/

[Garey and Johnson, 79] Michael R. Garey, David S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W H Freeman and Co, 1979.

[Gîrba and Ducasse, 06] Tudor Gîrba and Stéphane Ducasse, "Modeling History to Analyze Software Evolution," *Journal of Software Maintenance and Evolution*, vol. 18 no. 3, May-June 2006, pp. 207-236.

[Gelperin and Hetzel, 88] David Gelperin and Bill Hetzel, "The Growth of Software Testing," *Communications of the ACM*, vol. 31 no. 6, June 1988, pp. 687-695.

[GME, 07] Generic Modeling Environment, 2007, http://escher.isis.vanderbilit.edu/tools/get_tool?GME

[Gogolla, 04] Martin Gogolla, "Benefits and Problems of Formal Methods," *Ada Europe*, Springer-Verlag LNCS 3063, Palma de Mallorca, Spain, June 2004, pp. 1-15.

[Gokhale et al., 04] Aniruddha Gokhale, Douglas Schmidt, Balachandran Natarajan, Jeff Gray, and Nanbor Wang, "Model-Driven Middleware," *Middleware for Communications*, (Qusay Mahmoud, editor), John Wiley and Sons, 2004.

[Gray et al., 01] Jeff Gray, Ted Bapty, Sandeep Neema, and James Tuck, "Handling Crosscutting Constraints in Domain-Specific Modeling," *Communications of the ACM*, vol. 44 no. 10, October 2001, pp. 87-93.

[Gray, 02] Jeff Gray, "Aspect-Oriented Domain-Specific Modeling: A Generative Approach Using a Metaweaver Framework," Ph.D. Thesis, Dept. of Electrical Engineering and Computer Science, Vanderbilt University, Nashville, Tennessee, 2002.

[Gray et al., 03] Jeff Gray, Yuehua Lin, and Jing Zhang, "Aspect Model Weavers: Levels of Supported Independence," *Middleware 2003: Workshop on Model-driven Approaches to Middleware Applications Development*, Rio de Janeiro, Brazil, June 2003.

[Gray et al., 04-a] Jeff Gray, Matti Rossi, and Juha Pekka Tolvanen, "Preface: Special Issue on Domain-Specific Modeling," *Journal of Visual Languages and Computing*, vol. 15 nos. 3-4, June/August 2004, pp. 207-209.

[Gray et al., 04-b] Jeff Gray, Jing Zhang, Yuehua Lin, Hui Wu, Suman Roychoudhury, Rajesh Sudarsan, Aniruddha Gokhale, Sandeep Neema, Feng Shi, and Ted Bapty, "Model-Driven Program Transformation of a Large Avionics Framework," *Generative Programming and Component Engineering (GPCE)*, Springer-Verlag LNCS 3286, Vancouver, Canada, October 2004, pp. 361-378.

[Gray et al., 05] Jeff Gray, Yuehua Lin, Jing Zhang, Steve Nordstrom, Aniruddha Gokhale, Sandeep Neema, and Swapna Gokhale, "Replicators: Transformations to Address Model Scalability," *8th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, Springer-Verlag LNCS 3713, Montego Bay, Jamaica, October 2005, pp. 295-308.

[Gray et al., 06] Jeff Gray, Yuehua Lin, Jing Zhang, "Automating Change Evolution in Model-Driven Engineering," *IEEE Computer* (Special Issue on Model-Driven Engineering), February 2006, vol. 39 no. 2, pp. 41-48.

[Gray et al., 07] Jeff Gray, Juha-Pekka Tolvanen, Steven Kelly, Aniruddha Gokhale, Sandeep Neema, and Jonathan Sprinkle, *Handbook of Dynamic System Modeling*, (Paul Fishwick, ed.), CRC Press, 2007.

[Greenfield et al., 04] Jack Greenfield, Keith Short, Steve Cook, and Stuart Kent, *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*, Wiley Publishing, Inc., 2004.

[Hailpern and Tarr, 06] Brent Hailpern and Peri Tarr, "Model-Driven Development: The Good, the Bad, and the Ugly", *IBM Systems Journal*, vol. 45 no. 3, July 2006, pp. 451–461.

[Harrold, 00] Mary J. Harrold, "Testing: A Road Map," *Future of Software Engineering, Special Volume Published in Conjunction with ICSE 2000*, (A. Finkelstein, ed.), May 2000, Limerick, Ireland, pp. 61-72.

[Hatcliff et al., 03] John Hatcliff, Xinghua Deng, Matthew B. Dwyer, Georg Jung, and Venkatesh P. Ranganath, "Cadena: An Integrated Development, Analysis, and Verification Environment for Component-based Systems," *International Conference on Software Engineering (ICSE)*, Portland, Oregon, May 2003, pp. 160-173.

[Hayashi et al., 04] Susumu Hayashi, Yibing Pan, Masami Sato, Kenji Mori, Sul Sejeon, and Shusuke Haruna, "Test Driven Development of UML Models with SMART Modeling System," *7th International Conference on Unified Modeling Language (UML)*, Springer-Verlag LNCS 3237, Lisbon, Portugal, October 2004, pp. 295-409.

[Hinchey et al., 96] Michael Hinchey, Jonathan Bowen, and Robert Glass, "Formal Methods: Point-Counterpoint," *IEEE Computer*, vol. 13 no. 2, April 1996, pp. 18-19.

[Hirel et al., 00] Christophe Hirel, Bruno Tuffin, and Kishor Trivedi, "SPNP: Stochastic Petri Nets. Version 6.0," *Computer Performance Evaluation: Modeling Tools and Techniques*, Springer Verlag LNCS 1786, Schaumburg, Illinois, March 2000, pp. 354-357.

[Hoare, 69] Charles A. R. Hoare, "An Axiomatic Basis for Computer Programming," *Communications of the ACM*, vol. 12 no. 10, October 1969, pp. 576-580.

[Holzmann, 97] Gerard J. Holzmann, "The Model Checker SPIN," *IEEE Transactions on Software Engineering*, vol. 23 no. 5, May 1997, pp. 279-295.

[Hunt and McIlroy, 75] J. W. Hunt and M. D. McIlroy, "An Algorithm for Differential File Comparison," *Computing Science Technical Report No. 41*, Bell Laboratories, 1975.

[Johann and Egyed, 04] Sven Johann and Alexander Egyed, "Instant and Incremental Transformation of Models," *19th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Linz, Austria, September 2004, pp. 362-365.

[Johnson, 98] Luanne J. Johnson, "A View from the 1960s: How the Software Industry Began," *IEEE Annals of the History of Computing*, vol. 20 no. 1, January-March 1998, pp. 36-42.

[Karsai et al., 03] Gábor Karsai, Janos Sztipanovits, Ákos Lédeczi and Ted Bapty, "Model-Integrated Development of Embedded Software," *Proceedings of IEEE*, vol. 91 no. 1, January 2003, pp. 145-164.

[Karsai et al., 04] Gábor Karsai, Miklos Maroti, Ákos Lédeczi, Jeff Gray, and Janos Sztipanovits, "Composition and Cloning in Modeling and Meta-Modeling," *IEEE Transactions on Control Systems Technology*, vol. 12 no. 2, March 2004, pp. 263-278.

[Kent, 02] Stuart Kent, "Model Driven Engineering," *3rd International Conference on Integrated Formal Methods (IFM'02)*, Springer-Verlag LNCS 2335, Turku, Finland, May 2002, pp. 286-298

[Kleppe et al., 03] Anneke Kleppe, Jos Warmer, and Wim Bast, *MDA Explained. The Model Driven Architecture: Practice and Promise*. Addison-Wesley, 2003

[Kiczales et al., 01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold, "Getting Started with AspectJ," *Communications of the ACM*, vol. 44 no. 10, October 2001, pp. 59-65.

[Kogekar et al., 06] Arundhati Kogekar, Dimple Kaul, Aniruddha Gokhale, Paul Vandal, Upsorn Praphamontripong, Swapna Gokhale, Jing Zhang, Yuehua Lin, and Jeff Gray, "Model-driven Generative Techniques for Scalable Performabality Analysis of Distributed Systems," *IPDPS Workshop on Next Generation Systems*, Rhodes, Greece, April 2006.

[Kuster and Abd-El-Razik, 06] Jochen M. Kuster and Mohamed Abd-El-Razik, "Validation of Model Transformations - First Experiences using a White Box Approach," *3rd International Workshop on Model Development, Validation and Verification (MoDeV^{2}a)*, Genova, Italy, October, 2006.

[Kurtev et al., 06] Ivan Kurtev, Jean Bézivin, Frédéric Jouault, and Patrick Valduriez, "Model-based DSL Frameworks," *Companion of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Portland, Oregon, October 2006, pp. 602-616.

[Küster, 06] Jochen M. Küster, "Definition and Validation of Model Transformations," *Software and Systems Modeling*, vol. 5 no. 3, 2006, pp. 233-259.

[Lédeczi et al., 01] Ákos Lédeczi, Arpad Bakay, Miklos Maroti, Peter Volgyesi, Greg Nordstrom, Jonathan Sprinkle, and Gábor Karsai, "Composing Domain-Specific Design Environments," *IEEE Computer*, vol. 34 no. 11, November 2001, pp. 44-51.

[Lieberman, 00] Henry Lieberman, "Programming by Example," *Communications of the ACM*, vol. 43, no. 3, March 2000, pp. 72-74.

[Lin et al., 04] Yuehua Lin, Jing Zhang, and Jeff Gray, "Model Comparison: A Key Challenge for Transformation Testing and Version Control in Model-Driven Software Development," *OOPSLA Workshop on Best Practices for Model-Driven Software Development*, Vancouver, Canada, October 2004.

[Lin et al., 05] Yuehua Lin, Jing Zhang, and Jeff Gray, "A Framework for Testing Model Transformations," in *Model-driven Software Development*, (Beydeda, S., Book, M. and Gruhn, V., eds.), Springer, 2005, Chapter 10, pp. 219-236.

[Lin et al., 07-a] Yuehua Lin, Jeff Gray, Jing Zhang, Steve Nordstrom, Aniruddha Gokhale, Sandeep Neema, and Swapna Gokhale, "Model Replication: Transformations to Address Model Scalability," conditionally accepted, *Software: Practice and Experience*.

[Lin et al., 07-b] Yuehua Lin, Jeff Gray and Frédéric Jouault, "DSMDiff: A Differencing Tool for Domain-Specific Models," *European Journal of Information Systems (*Special Issue on Model-Driven Systems Development), Fall 2007.

[Long et al., 98] Earl Long, Amit Misra, and Janos Sztipanovits, “Increasing Productivity at Saturn,” *IEEE Computer*, vol. 31 no. 8, August 1998, pp. 35-43.

[Manna and Pnueli, 92] Zohar Manna and Amir Pnueli, *The Temporal Logic of Reactive and Concurrent Systems, Specification*, Springer-Verlag, 1992.

[Mandelin et al., 06] David Mandelin, Doug Kimelman, and Daniel Yellin, “A Bayesian Approach to Diagram Matching with Application to Architectural Models,” *28th International Conference on Software Engineering (ICSE)*, Shanghai, China, May 2006, pp. 222-231.

[Marsan et al., 95] M.A. Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis, *Modelling with Generalized Stochastic Petri Nets*, Wiley Series in Parallel Computing, John Wiley and Sons, 1995.

[MDA, 07] Object Management Group, Model Driven Architecture, http://www.omg.org/mda/

[Mehra et al., 05] Akhil Mehra, John Grundy, and John Hosking, “A Generic Approach to Supporting Diagram Differencing and Merging for Collaborative Design,” *20th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Long Beach, California, November 2005, pp. 204-213.

[Mens and Van Gorp, 05] Tom Mens and Pieter Van Gorp, “A Taxonomy of Model Transformation,” *International Workshop on Graph and Model Transformation (GraMoT)*, Tallinn, Estonia, September, 2005.

[Mernik et al., 05] Marjan Mernik, Jan Heering, and Anthony M. Sloane, “When and How to Develop Domain-Specific Languages,” *ACM Computing Surveys*, December 2005, vol. 37 no. 4, pp. 316-344.

[MetaCase, 07] *MetaEdit+ 4.5 User’s Guide*. http://www.metacase.com

[Microsoft, 05] *Visual Studio Launch: Domain-Specific Language (DSL) Tools: Visual Studio 2005 Team System*. http://msdn.microsoft.com/vstudio/teamsystem/workshop/DSLTools

[Milicev, 02] Dragan Milicev, “Automatic Model Transformations Using Extended UML Object Diagrams in Modeling Environments,” *IEEE Transactions on Software Engineering*, April 2002, vol. 28 no. 4, pp. 413-431.

[Miller and Mukerji, 01] Joaquin Miller and Jishnu Mukerji, *MDA Guide Version 1.0.1*, http://www.omg.org/docs/omg/03-06-01.pdf

[MOF, 07] Object Management Group, Meta Object Facility specification, http://www.omg.org/technology/documents/modeling_spec_catalog.htm#MOF

[Mottu et al., 06] Jean-Marie Mottu, Benoit Baudry and Yves Le Traon, "Mutation Analysis Testing for Model Transformations," *2nd European conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA)*, Bilbao, Spain, July 2006. pp. 376-390.

[Muppala et al., 94] Jogesh K. Muppala, Gianfranco Ciardo, and Kishor S. Trivedi, "Stochastic Reward Nets for Reliability Prediction," *Communications in Reliability, Maintainability and Serviceability*, July 1994, pp. 9-20.

[Neema et al., 02] Sandeep Neema, Ted Bapty, Jeff Gray, and Aniruddha Gokhale, "Generators for Synthesis of QoS Adaptation in Distributed Real-Time Embedded Systems," *Generative Programming and Component Engineering (GPCE)*, Springer-Verlag LNCS 2487, Pittsburgh, Pennsylvania, October 2002, pp. 236-251.

[Nordstrom et al., 99] Gregory G. 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 (ECBS)*, Nashville, Tennessee, April 1999, pp. 68-74.

[Nordstrom, 99] Gregory G. Nordstrom, "MetaModeling - Rapid Design and Evolution of Domain-Specific Modeling Environments," Ph.D. Thesis, Dept. of Electrical Engineering and Computer Science, Vanderbilt University, Nashville, Tennessee, 1999.

[OCL, 07] Object Management Group, Object Constraint Language Specification, http://www.omg.org/technology/documents/modeling_spec_catalog.htm#OCL

[Ohst et al., 03] Dirk Ohst, Michael Welle, and Udo Kelter, "Differences Between Versions of UML Diagrams," *European Software Engineering Conference/Foundations of Software Engineering*, Helsinki, Finland, September 2003, pp. 227-236.

[Parnas, 72] David Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules," *Communications of the ACM*, December 1972, vol. 15 no. 12, pp. 1053-1058.

[Patrascoiu, 04] Octavian Patrascoiu, "Mapping EDOC to Web Services Using YATL," *8th International IEEE Enterprise Distributed Object Computing Conference (EDOC)*, Monterey, California, September 2004, pp. 286-297.

[Peterson, 77] James L. Peterson, "Petri Nets," *ACM Computing Surveys*, vol. 9 no. 3, September 1977, pp. 223-252.

[Petriu et al., 05] Dorina C. Petriu, Jinhua Zhang, Gordon Gu and Hui Shen, "Performance Analysis with the SPT Profile," in *Model-Driven Engineering for Distributed and Embedded Systems*, (S. Gerard, J.P. Babeau, J. Champeau, eds.), Hermes Science Publishing Ltd., London, England, 2005, pp. 205-224.

[Pilskalns et al., 07] Orest Pilskalns, Anneliese Andrews, Andrew Knight, Sudipto Ghosh, and Robert France, "Testing UML Designs," *Information and Software Technology*, vol. 49 no.8, August 2007, pp. 892-912.

[Pohjonen and Kelly, 02] Risto Pohjonen and Steven Kelly, "Domain-Specific Modeling," *Dr. Dobbs Journal*, August 2002, pp. 26-35.

[QVT, 07] MOF Query/Views/Transformations Specification, 2007. http://www.omg.org/cgi-bin/doc?ptc/2005-11-01.

[Rácz et al., 99] Sándor Rácz and Miklós Telek, "Performability Analysis of Markov Reward Models with Rate and Impulse Reward," *International Conference on Numerical Solution of Markov Chains*, Zaragoza, Spain, September 1999, pp. 169-180.

[Rose, 07] IBM Rational Rose, http://www-306.ibm.com/software/awdtools/developer/rose/

[Rosenberg, 96] Jonathan B. Rosenberg, *How Debuggers Work - Algorithms, Data Structures, and Architecture*, John Wiley and Sons, Inc, 1996.

[Schach, 07] Stephen R. Schach, *Object-Oriented and Classical Software Engineering*, 7th Edition, McGraw-Hill, 2007.

[Schmidt, 06] Douglas C. Schmidt, "Model-Driven Engineering," *IEEE Computer*, February 2006, vol. 39 no. 2, pp. 25-32.

[Schmidt et al., 00] Douglas C. Schmidt, Michael Stal, Hans Rohnert, Frank Buschman, *Pattern-Oriented Software Architecture – Volume 2: Patterns for Concurrent and Networked Objects*, John Wiley and Sons, 2000.

[Schmidt and Varró, 03] Akos Schmidt and Daniel Varró, "CheckVML: A Tool for Model Checking Visual Modeling Languages," *6th International Conference on the Unified Modeling Language (UML)*, Springer-Verlag LNCS 2863, San Francisco, California, October 2003, pp. 92-95.

[Sendall and Kozaczynski, 03] Shane Sendall and Wojtek Kozaczynski, "Model Transformation - the Heart and Soul of Model-Driven Software Development," *IEEE Software*, vol. 20 no. 5, September/October 2003, pp. 42-45.

[Sharp, 00] David C. Sharp, "Component-Based Product Line Development of Avionics Software," *First Software Product Lines Conference (SPLC-1)*, Denver, Colorado, August 2000, pp. 353-369.

[Shetty et al., 05] Shweta Shetty, Steven Nordstrom, Shikha Ahuja, Di Yao, Ted Bapty, and Sandeep Neema, "Integration of Large-Scale Autonomic Systems using Multiple Domain Specific Modeling Languages," *12th IEEE International Conference and Workshops on the Engineering of Autonomic Systems* (*ECBS)*, Greenbelt, Maryland, April 2005, pp. 481-489.

[Sztipanovits, 02] Janos Sztipanovits, "Generative Programming for Embedded Systems," *Keynote Address: Generative Programming and Component Engineering (GPCE),* Springer-Verlag LNCS 2487, Pittsburgh, Pennsylvania, October 2002, pp. 32-49.

[Sztipanovits and Karsai, 97] Janos Sztipanovits, Gábor Karsai, "Model-Integrated Computing," *IEEE Computer*, April 1997, pp. 10-12.

[UML, 07] Object Management Group, Unified Modeling Language Specification, http://www.omg.org/technology/documents/modeling_spec_catalog.htm#UML

[Vangheluwe and De Lara, 04] Hans Vangheluwe and Juan de Lara, "Domain-Specific Modelling with AToM3," *4th OOPSLA Workshop on Domain-Specific Modeling*, Vancouver, Canada, October 2004.

[Varró, 06] Dániel Varró, "Model Transformation by Example," *9th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, Genova, Italy, October 2006, pp. 410-424.

[Varró et al., 02] Dániel Varró, Gergely Varró, and András Pataricza, "Designing the Automatic Transformation of Visual Languages," *Science of Computer Programming*, vol. 44 no. 2, 2002, pp. 205-227.

[Wang et al., 03] Yuan Wang, David J. DeWitt, and Jin-Yi Cai, "X -Diff: An Effective Change Detection Algorithm for XML Documents," *19th International Conference on Data Engineering*, Bangalore, India, March 2003, pp. 519-530.

[Warmer and Kleppe, 99] Jos Warmer and Anneke Kleppe, *The Object Constraint Language: Precise Modeling with UML*, Addison-Wesley, 1999.

[Whittle, 02] Jon Whittle, "Transformations and Software Modeling Languages: Automating Transformations in UML," *5th International Conference on the Unified Modeling Language (UML),* Springer-Verlag LNCS 2460, September-October 2002, Dresden, Germany, pp. 227-242.

[Wimmer et al., 07] Manuel Wimmer, Michael Strommer, Horst Kargl, and Gerhard Kramler, "Towards Model Transformation Generation By-Example," *40th Hawaii International Conference on System Sciences (HICSS)*, Big Island, Hawaii, January 2007.

[Xing and Stroulia, 05] Zhenchang Xing and Eleni Stroulia, "UMLDiff: An Algorithm for Object-Oriented Design Differencing," *20th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Long Beach, California, November 2005, pp. 54-65.

[XMI, 07] Object Management Group, XMI specification, http://www.omg.org/technology/documents/modeling_spec_catalog.htm#XMI

[XSLT, 99] W3C, XSLT Transformation version 1.0, 1999, http://www.w3.org/TR/xslt

[Yilmaz, 01] Levent Yilmaz, "Verification and Validation: Automated Object-Flow Testing of Dynamic Process Interaction Models," *33rd Winter Conference on Simulation*, Arlington, Virginia, December 2001, pp. 586-594.

[Zellweger, 84] Polle T. Zellweger, "Interactive Source-Level Debugging of Optimized Programs," Ph.D. Thesis, Department of Computer Science, University of California, Berkeley, California, May 1984.

[Zhang et al., 04] Jing Zhang, Jeff Gray, and Yuehua Lin, "A Generative Approach to Model Interpreter Evolution," *4th OOPSLA Workshop on Domain-Specific Modeling*, Vancouver, Canada, October 2004, pp. 121-129.

[Zhang et al., 05-a] Jing Zhang, Yuehua Lin, and Jeff Gray, "Generic and Domain-Specific Model Refactoring using a Model Transformation Engine," *Model-Driven Software Development*, (Beydeda, S., Book, M. and Gruhn, V., eds.), Springer, 2005, Chapter 9, pp. 199-218.

[Zhang et al., 05-b] Jing Zhang, Jeff Gray, and Yuehua Lin, "A Model-Driven Approach to Enforce Crosscutting Assertion Checking," *27th ICSE Workshop on the Modeling and Analysis of Concerns in Software (MACS)*, St. Louis, Missouri, May 2005.

[Zhang et al., 07] Jing Zhang, Jeff Gray, Yuehua Lin, and Robert Tairas, "Aspect Mining from a Modeling Perspective," *International Journal of Computer Applications in Technology (*Special Issue on Concern Oriented Software), Fall 2007.

[Zhu et al., 97] Hong Zhu, Patrick Hall, and John May, "Software Unit Test Coverage and Adequacy," *ACM Computing Surveys*, vol. 29 no. 4, December 1997, pp. 367-427.

[Zloof, 77] Moshé Zloof, "Query By Example," *IBM Systems Journal*, vol. 16 no. 4, 1977, pp. 324-343.

APPENDIX A

EMBEDDED CONSTRAINT LANGUAGE GRAMMAR

The Embedded Constraint Language (ECL) extensions described in Chapter 3 are based on an earlier ECL description presented in [Gray, 02]. Furthermore, this earlier ECL definition was an extension of the Multigraph Constraint Language (MCL), which was an early OCL-like constraint language for the first public release of the GME [GME, 07]. The ECL grammar is defined in an ANTLR [ANTLR, 07] grammar (ecl.g), which is presented in the remainder of this appendix. Much of this grammar has legacy productions from the original MCL. This section does not claim a major contribution of this dissertation, but is provided for completeness to those desiring a more formal description of the ECL syntax.

```
//begin ecl.g

        class ECLParser {
        exception

        default: //Print error messages
        name :  id : IDENT ;
        defines : { DEFINES defs  SEMI } ;
        defs  :  def ( COMMA def )* ;
        def  :  name     ;
        cpars : cpar ( SEMI cpar )* ;
        cpar : name ( COMMA name )* ":" name ;
```

```
cdef : { INN foldername { DOT modelname { DOT aspectname }}}
        (STRATEGY name | ASPECT name )
         LPAR  { cpars } RPAR
         priority
         description
         LBRACE cexprs  { action  } RBRACE
        | FUNCTION name LPAR { cpars } RPAR cexprs  { action } ;
priority : PRIORITY "=" id:INTEGER |  ;
description :    id:STR | ;
foldername : name ;
modelname :  name ;
aspectname : name ;
lval : name ;
astring  : str:ACTION ;
action  : astring;
cexprs : cexpr SEMI ( cexpr SEMI )* ;
cexpr : { action }
        ( assign | DECLARE STATIC cpar
                 | DECLARE cpar
                 | lexpr) ;
assign : lval ASSIGN lexpr ;
```

```
ifexpr : IF cexpr
         THEN cexprs { action }
         { ELSE cexprs  { action } }
         ENDIF ;
lexpr :  relexpr lexpr_r ;
lexpr_r: { lop  relexpr  lexpr_r } ;
relexpr : addexpr  { rop addexpr } ;
addexpr : mulexpr  addexpr_r ;
addexpr_r :{   addop  mulexpr  addexpr_r } ;
mulexpr : unaexpr  mulexpr_r ;
mulexpr_r :{  mulop unaexpr  mulexpr_r } ;
unaexpr : ( unaryop  postfixexpr  ) | postfixexpr ;
postfixexpr : primexpr  postfixcall ;
postfixcall : { (  (   DOT | ARROW | CARET )  call )  postfixcall } ;
primexpr : litcoll   |  lit | objname callpars | LPAR cexpr RPAR | ifexpr ;
callpars : {  LPAR  ( ( decl ) ? decl  { actparlist  }  |  { actparlist  }  )   RPAR } ;
lit : string | number ;
tname : name ;
litcoll   : coll  LBRACE { cexprlrange } RBRACE ;
cexprlrange :   cexpr   {   (   (COMMA cexpr )+   )  |  ( ".." cexpr ) } ;
call  :   name  callpars ;
decl :  name     ( COMMA name  )*     { ":" tname } "\|" ;
objname : name | SELF ;
```

```
actparlist : cexpr     ( COMMA cexpr  )* ;

lop : AND | OR | XOR | IMPLIES ;

coll  :   SET     | BAG  | SEQUENCE | COLLECTION ;

rop :   "==" | "<" | ">" | ">=" | "<=" |  "<>" ;

addop > :    PLUS | MINUS ;

mulop >  :    MUL | DIV ;

unaryop :    MINUS | NOT ;

string :   str:STR ;

number :    r:REAL | n:INTEGER;

}


// Token definitions

#token STR              "\"~[\"]*\""

#token INTEGER       "[0-9]+"

#token  REAL            "([0-9]+.[0-9]* | [0-9]*.[0-9]+) {[eE]{[\-\+]}[0-9]+}"

#token IDENT          "[a-zA-Z][a-zA-Z0-9_]*"


// end of ecl.g
```

APPENDIX B

OPERATIONS OF THE EMBEDDED CONSTRAINT LANGUAGE

addAtom
purpose: add an atom based on its partName(i.e., kindName) that belongs to a model and assign a new name to it
caller: a model or an object that represents a model
usage: caller.addAtom(string partName, string newName)
result type: atom

addConnection
purpose: add a connection with a specific kindName from a source object to a destination object within a caller
caller: a model or an object that represents a model
usage: caller.addConnection(string kindName, object source, object destination)
result type: connection

addFolder
purpose: add a folder based on its kindName and assign a new name to it
caller: a folder
usage: caller.addFolder(string kindName, string newName)
result type: folder

addMember
purpose: add an object as a member of a set
caller: an object that represents a set
usage: caller.addMember(object anObj)
result type: void

addModel
purpose: add a model based on its partName(i.e., kindName) that belongs to a model and assign a new name to it
caller: a model, a folder, an object that represents a model/folder or a list of models.
usage:
1) if caller is a single object, caller.addModel(string partName, string newName)
2) if caller is a list, caller-> addModel(string partName, string newName)
result type: model

addReference
purpose: add a reference with a specific kindName that refers to an object and assign a new name to it within a caller
caller: a model
usage: caller.addReference(string kindName, object refTo)
result type: reference

addSet
purpose: add a set based on its kindName and assign a new name to it within a caller or a list of callers
caller: a model or a list of models
usage: caller.addSet(string kindName, string newName) or caller->addSet(string kindName, string newName)
result type: set

atoms
purpose: return all the atoms or the atoms with specific kindName within a caller.
caller: a model or an object that represents a model
usage: caller.atoms() or caller.atoms(string kindName)
result type: atomList

connections
purpose: return all the connections with a specific connection kindname within a model
caller: a model or an object that represents a model
usage: caller.connections(string connName)
result type: objectList that represents a list of connections

destination
purpose: return the destination point of a connection
caller: a connection
usage: caller.destination()
result type: model/atom/reference/object

endWith
purpose: check if a string ends with a substring
caller: a string
usage: caller.endWith(string aSubString)
result type: boolean

findAtom
purpose: return an atom based on its name within a caller
caller: a model or an object that represents a model
usage: caller.findAtom(string atomName)
result type: atom

findConnection
purpose: find a connection with a specific kindName from a source object to a destination object within a caller
caller: a model or an object that represents a model
usage: caller.findConnection(string kindName, object source, object destination)
result type: connection

findFolder
purpose: return a folder with a specific name
caller: a folder
usage: caller.findFolder(string folderName)
result type: folder

findModel
purpose: return a model based on its name
caller: a folder or a model or an object that represents a model
usage: caller.findModel(string modelName)
result type: model

findObject
purpose: return an object based on its name within a caller
caller: a model or a folder, or an object that represents a model/folder
usage: caller.findObject(string objName)
result type: object

getAttribute
purpose: return the value of an attribute of a caller which type is int, bool, double or string
caller: an atom, a model or an object
usage: caller.getAttribute(string attrName)
result type: int, bool, double or string

intToString
purpose: convert an integer value to string
caller: none
usage: intToString(int val)
result type: string

isNull
purpose: determine if the caller is null
caller: an atom, a model or an object
usage: caller.isNull()
result type: boolean

kindOf
purpose: return an caller's kindname
caller: an atom, a model or an object
usage: caller.kindOf()
result type: string

models
purpose: return all the models or the models with specific kindName within a caller
caller: a model or a folder
usage: caller.models() or caller.models(string kindName).
result type: modelList

modelRefs
purpose: return all the models within a caller that are referred by the model references with the specific kindName
caller: a model or an object that represents a model
usage: caller.modelRefs(string kindName)
result type: modelList

name
purpose: return a caller's name
caller: an atom, a model or an object
usage: caller.name()
result type: string

parent
purpose: return the parent model of a caller
caller: a model, an atom or an object that represents a model/an atom.
usage: caller.parent()
result type: model

refersTo
purpose: return a model/an atom/an object that the caller refers to
caller: a modelRef or an object that represents a model reference
usage: caller.referesTo()
result type: model or atom or object

removeAtom
purpose: remove an atom based on its name
caller: a model or an object that represents a model, which is the parent model of the to-be-removed atom
usage: caller.removeAtom(string atomName)
result type: void

removeConnection
purpose: remove a connection with a specific kindName from a source object to a destination object within a caller
caller: a model or an object that represents a model
usage: caller.removeConnection(string kindName, object source, object destination)
result type: void

removeModel
purpose: remove a model based on its name
caller: a model or an object that represents a model, which is the parent model of the to-be-removed model
usage: caller.removeModel(string modelName)
result type: void

rootFolder
purpose: return the root folder of an open GME project
caller: none
usage: rootFolder()
result type: folder

select
purpose: select the atoms or the models within a caller according to the specified condition
caller: atomList, modelList or an objectList
usage: caller.select(logic expression)
result type: atomList or modelList

show
purpose: display string message
caller: none
usage: show(any expression that returns a string)
result type: void

size
purpose: return the size of the caller list
caller: atomList, modelList or an objectList
usage: caller.size()
result type: int

source
purpose: return the source point of a connection
caller: a connection
usage: caller.source()
result type: model/atom/reference/object

setAttribute
purpose: set a new value to an attribute of a caller
caller: an atom, a model or an object
usage: caller.setAttribute(string attrName, anyType value)
result type: void

APPENDIX C

ADDITIONAL CASE STUDIES ON MODEL SCALABILITY

In this appendix, the concept of model replicators is further demonstrated on two separate example modeling languages that were created in GME for different domains. The two DSMLs are:

- Stochastic Reward Net Modeling Language (SRNML), which has been used to describe performability concerns of distributed systems built from middleware patterns-based building blocks.
- Event QoS Aspect Language (EQAL), which has been used to configure a large collection of federated event channels for mission computing avionics applications.

C.1 Scaling Stochastic Reward Net Modeling Language (SRNML)

Stochastic Reward Nets (SRNs) [Muppala et al., 94] represent a powerful modeling technique that is concise in its specification and whose form is closer to a designer's intuition about what a performance model should look like. Because an SRN specification is closer to a designer's intuition of system behavior, it is also easier to transfer the results obtained from solving the models and interpret them in terms of the entities that exist in the system being modeled. SRNs have been used extensively for performance, reliability and performability modeling of different types of systems. SRNs are the result of a chain of evolution starting with Petri nets [Peterson, 77]. More discussion on SRNs can be found in [Marsan et al., 95], [Rácz et al., 99].

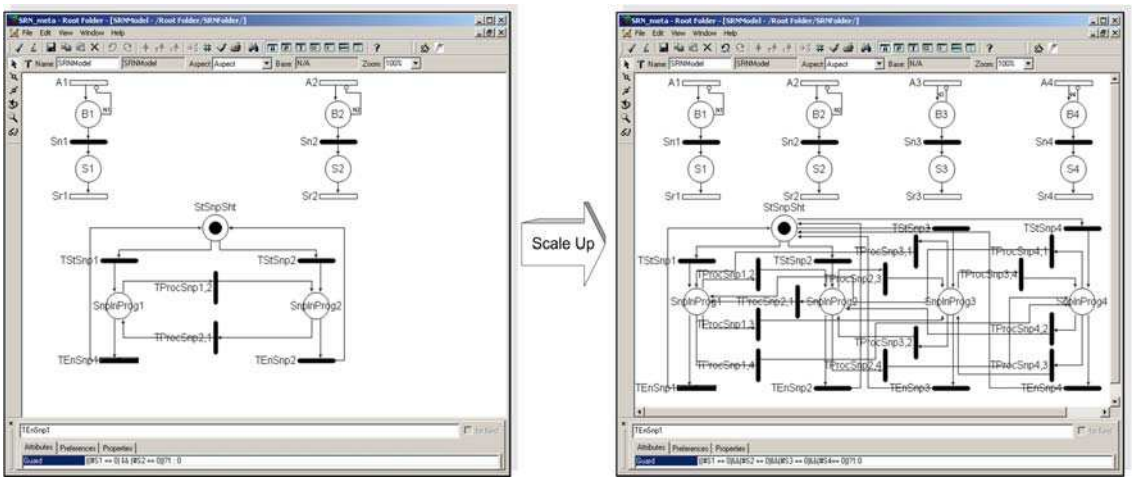
The Stochastic Reward Net Modeling Language (SRNML) is a DSML developed in GME to describe SRN models of large distributed systems [Kogekar et al., 06]. The SRNML is similar to the goals of performance-based modeling extensions for the UML,

such as the Schedulability, Performance, and Time profile [Petriu et al., 05]. The model compilers developed for SRNML can synthesize artifacts required for the SPNP tool [Hirel et al., 00], which is a model solver based on SRN semantics.

The SRN models, which are specified in SRNML, depict the Reactor pattern [Schmidt et al., 00] in middleware for network services, which provides synchronous event demultiplexing and dispatching mechanisms. In the Reactor pattern, an application registers an event handler with the event demultiplexer and delegates to it the responsibility of listening for incoming events. On the occurrence of an event, the demultiplexer dispatches the event by making a callback to its associated application-supplied event handler. As shown in Figure C-1a, an SRN model usually consists of two parts: the top half represents the event types handled by a reactor and the bottom half defines the associated execution snapshot. The execution snapshot needs to represent the underlying mechanism for handling the event types included in the top part (e.g., non-deterministic handling of events). Thus, there are implied dependent relations between the top and bottom parts. Any change made to the top will require corresponding changes to the bottom.

Figure C-1a shows the SRN model for the reactor pattern for two event handlers. The top of Figure C-1a models the arrival, queuing and service of the two event types. Transitions *A1* and *A2* represent the arrivals of the events of types one and two, respectively. Places *B1* and *B2* represent the queue for the two types of events. Transitions Sn1 and *Sn2* are immediate transitions that are enabled when a snapshot is taken. Places *S1* and *S2* represent the enabled handles of the two types of events, whereas transitions *Sr1* and *Sr2* represent the execution of the enabled event handlers of the two

types of events. An inhibitor arc from place *B1* to transition *A1* with multiplicity *N1* prevents the firing of transition *A1* when there are *N1* tokens in place *B1*. The presence of *N1* tokens in place *B1* indicates that the buffer space to hold the incoming input events of the first type is full, and no additional incoming events can be accepted. The inhibitor arc from place *B2* to transition *A2* achieves the same purpose for type two events.



a) *base model with 2 event handlers* **b)** *scaled model with 4 event handlers*

Figure C-1 *- Replication of Reactor Event Types (from 2 to 4 event types)*

The bottom of Figure C-1a models the process of taking successive snapshots and non-deterministic service of event handles in each snapshot. Transition *Sn1* is enabled when there are one or more tokens in place *B1*, a token in place *StSnpSht*, and no token in place *S1*. Similarly, transition *Sn2* is enabled when there are one or more tokens in place *B2*, a token in place *StSnpSht* and no token in place *S2*. Transitions *TStSnp1* and *TStSnp2* are enabled when there is a token in place *S1*, place *S2*, or both. Transitions *TEnSnp1* and *TEnSnp2* are enabled when there are no tokens in both places *S1* and *S2*. Transition *TProcSnp1,2* is enabled when there is no token in place *S1* and a token in place *S2*. Similarly, transition *TProcSnp2,1* is enabled when there is no token in place *S2* and a

token in place *S1*. Transition *Sr1* is enabled when there is a token in place *SnpInProg1*, and transition *Sr2* is enabled when there is a token in place *SnpInProg2*. All the transitions have their own guard functions, as shown in Table C -1.

Table C-1 - *Enabling guard equations for Figure C-1*

Transition	Guard Function
Sn_1	((#*StSnpShot* == 1) && (#B_1 >= 1) && (#S_1 == 0)) ? 1 : 0
⋮	⋮
Sn_m	((#*StSnpShot* == 1) && (#B_m >= 1) && (#S_m == 0)) ? 1 : 0
$TStSnp_1$	(#S_1 == 1) ? 1 : 0
⋮	⋮
$TStSnp_m$	(#S_m == 1) ? 1 : 0
$TEnSnp_1$	((#S_1 == 0) && (#S_2 == 0) && **…** (#S_m == 0)) ? 1 : 0
⋮	⋮
$TEnSnp_m$	((#S_1 == 0) && (#S_2 == 0) && **…** (#S_m == 0)) ? 1 : 0
$TProcSnp_{1,2}$	((#S_1 == 0) && (#S_2 == 1)) ? 1 : 0
⋮	⋮
$TProcSnp_{1,m}$	((#S_1 == 0) && (#S_m == 1)) ? 1 : 0
⋮	⋮
$TProcSnp_{m,m-1}$	((#S_m == 0) && (#S_{m-1} == 1)) ? 1 : 0
⋮	⋮
$TProcSnp_{m,1}$	((#S_m == 0) && (#S_1 == 1)) ? 1 : 0
Sr_1	(#$SnpInProg_1$ == 1) ? 1 : 0
⋮	⋮
Sr_m	(#$SnpInProg_m$ == 1) ? 1 : 0

C1.1 Scalability Issues in SRNML

The scalability challenges of SRN models arise from the addition of new event types and connections between their corresponding event handlers. For example, the top of the SRN model must scale to represent the event handling for every event type that is available. A problem emerges when there could be non-deterministic handling of events, which leads to the complicated connections between the elements within the execution

snapshot of an SRN model. Due to the implied dependencies between the top and bottom parts, the bottom part of the model (i.e., the snapshot) should incorporate appropriate non-deterministic handling depicted in the scaled number of event types. The inherent structural complexity and the complicated dependent relations within an SRN model make it difficult and impractical to scale up SRN models manually, which requires a computer-aided method such as a replicator to perform the replication automatically.

The replication behaviors for scaling up an SRN model can be formalized as computation logic and specified in a model transformation language such as ECL [Lin et al., 07]. Figure C-1a describes a base SRN model for two event types, and Figure C-1b represents the result of scaling this base model from two event types to four event types. Such scalability in SRN models can be performed with two model transformation steps. The first step scales the reactor event types (i.e., the upper part of the SRN model) from two to four, which involves creating the *B* and *S* places, the *A*, *Sn* and *Sr* transitions and associated connection arcs and renaming them, as well as setting appropriate guard functions for each new event type. The second step scales the snapshot (i.e., the bottom part of the SRN model) according to the newly added event types. Inside a snapshot, the model elements can be divided into three categories. The first category is a group of elements that are independent of each event type; the second category is a group of model elements that are associated with every two new event types; and the third category is a group of elements that are associated to one old event type and one new event type. Briefly, these three groups of elements can be built by three subtasks:

- Create the *TStSnp* and *TEnSnp* transitions and the *SnpInProg* place, as well as required connection arcs among them for each newly added event type; assign

the correct guard function for each created transition; this task builds the first group.

- For each pair of new event types, create two *TProcSnp* transitions and connect their *SnpInProg* places to these *TProcSnp* transitions; assign the correct guard function for each created transition; this task builds the second group.
- For each pair of <*old event type, new event type*>, create two *TProcSnp* transitions and connect their *SnpInProg* places to these *TProcSnp* transitions; assign the correct guard function for each created transition; this task builds the third group.

C1.2 ECL Transformation to Scale SRNML

In this example, only the model transformation for scaling the snapshot is illustrated. The ECL specification shown in Listing C-1 performs subtask one. It is composed of several strategies. The `computeTEnSnpGuard` (Line 1) strategy is used to re-compute the guard functions of the *TEnSnp* transitions when new event types are added. The ECL code on Lines 3 and 4 recursively concatenate the string that represents the guard function. After this string is created, it is passed to the `addEventsWithGuard` strategy (Line 41 to 47), which adds the new guard function and event to the snapshot. The `addEvents` strategy (Line 12) recursively calls the `addNewEvent` strategy to create necessary transitions, places and connections in the snapshot for the new event types with identity numbers from `min_new` to `max_new`. The `addNewEvent` strategy (Line 20) creates snapshot elements for a single new event type with identity number `event_num`. The `findAtom` operation on Line 25 is used

to discover the *StSnpSht* place in the snapshot. The *TStSnp* transition is created on Line 26 and its guard function is created on Lines 27 and 28. Next, the *SnpInProg* place and the *TEnSnp* transition are created on Lines 30 and 31, respectively. The guard function of the *TEnSnp* transition is set on Line 32. Finally, four connection arcs are created among the *StSnpSht* place, the *TStSnp* transition, the *SnpInProg* place and the *TEnSnp* transition (Lines 34 to 37).

Subtask one actually creates independent snapshot elements for each new event type. In collaboration, subtasks two and three build the necessary relationships between each pair of new event types, and each pair consisting of a new event type and an old event type. Listing C-2 shows the ECL specification to perform subtask two (i.e., to build the relationship between every two new event types). The `connectTwoEvents` strategy (Line 17) creates the *TProcSnp* transition and its associated connections between two events. Then, the `connectOneNewEventToOtherNewEvents` strategy (Line 9) recursively calls the `connectTwoEvent` strategy to build relationships between two new events. Finally, the `connectNewEvents` strategy (Line 1) builds the relationships between each pair of new event types by recursively calling the `connectOneNewEventToOtherNewEvents` strategy. Inside the `connectTwoEvents` strategy, the S*npInProg* places of the two event types are discovered on Lines 28 and 29, respectively. Then, two new *TProcSnp* transitions are created and their guard functions are set (Lines 30 through 33), followed by the construction of the connections between the *SnpInProg* places and the *TProcSnp* transitions (Lines 35 through 38).

```
1  strategy computeTEnSnpGuard(min_old, min_new, max_new : integer; TEnSnpGuardStr : string)
2  {
3       if (min_old < max_new) then
4          computeTEnSnpGuard(min_old + 1, min_new, max_new, TEnSnpGuardStr +
                              "(#S" + intToString(min_old) +  " == 0)&&");
5       else
6          addEventswithGuard(min_new, max_new, TEnSnpGuardStr + "(#S" + intToString(min_old) +
                              "== 0))?1:0");
7       endif;
8  }
9
10 ...  // several strategies not show here
11
12 strategy addEvents(min_new, max_new : integer; TEnSnpGuardStr : string)
13 {
14      if (min_new <= max_new) then
15             addNewEvent(min_new, TEnSnpGuardStr);
16             addEvents(min_new+1, max_new, TEnSnpGuardStr);
17      endif;
18 }
19
20 strategy addNewEvent(event_num : integer; TEnSnpGuardStr : string)
21 {
22      declare start, stTran, inProg, endTran : atom;
23      declare TStSnp_guard : string;
24
25      start := findAtom("StSnpSht");
26      stTran := addAtom("ImmTransition", "TStSnp" + intToString(event_num));
27      TStSnp_guard := "(#S" + intToString(event_num) + " == 1)?1 : 0";
28      stTran.setAttribute("Guard", TStSnp_guard);
29
30      inProg := addAtom("Place", "SnpInProg" + intToString(event_num));
31      endTran := addAtom("ImmTransition", "TEnSnp" + intToString(event_num));
32      endTran.setAttribute("Guard", TEnSnpGuardStr);
33
34      addConnection("InpImmedArc", start, stTran);
35      addConnection("OutImmedArc", stTran, inProg);
36      addConnection("InpImmedArc", inProg, endTran);
37      addConnection("OutImmedArc", endTran, start);
38 }
39
40 //recursively calls "addEvents" and "modifyOldGuards"
41 strategy addEventswithGuard(min_new, max_new : integer; TEnSnpGuardStr : string)
42 {
43      rootFolder().findFolder("SRNFolder").findModel("SRNModel").
44          addEvents(min_new, max_new, TEnSnpGuardStr);
45      rootFolder().findFolder("SRNFolder").findModel("SRNModel").
46          modifyOldGuards(1, min_new-1, TEnSnpGuardStr);
47 }
48 ...
```

Listing C-1 *- ECL transformation to perform first subtask of scaling snapshot*

```
1  strategy connectNewEvents(min_new, max_new: interger)
2  {
3      if(min_new < max_new) then
4              connectOneNewEventToOtherNewEvents(min_new, max_new);
5              connectNewEvents(min_new+1, max_new);
6      endif;
7  }
8
9  strategy connectOneNewEventToOtherNewEvents(event_num, max_new: integer)
10 {
11     if(event_num < max_new) then
12             connectTwoEvents(event_num, max_new);
13             connectNewEvents(event_num, max_new-1);
14     endif;
15 }
16
17 strategy connectTwoEvents(first_num, second_num : integer)
18 {
19     declare firstinProg, secondinProg : atom;
20     declare secondTProc1, secondTProc2 : atom;
21     declare first_numStr, second_numStr, TProcSnp_guard1, TProcSnp_guard2 : string;
22
23     first_numStr := intToString(first_num);
24     second_numStr := intToString(second_num);
25     TProcSnp_guard1 := "((#S" + first_numStr + " == 0) && (#S" + second_numStr +
                           "  == 1))?1 : 0";
26     TProcSnp_guard2 := "((#S" + second_numStr + " == 0) && (#S" + first_numStr +
                           " == 1))?1 : 0";
27
28     firstinProg := findAtom("SnpInProg" + first_numStr);
29     secondinProg := findAtom("SnpInProg" + second_numStr);
30     secondTProc1 := addAtom("ImmTransition", "TProcSnp" + first_numStr +
                              "," + second_numStr);
31     secondTProc1.setAttribute("Guard", TProcSnp_guard1);
32     secondTProc2 := addAtom("ImmTransition", "TProcSnp" + second_numStr +
                              "," + first_numStr);
33     secondTProc2.setAttribute("Guard", TProcSnp_guard2);
34
35     addConnection("InpImmedArc", firstinProg, secondTProc1);
36     addConnection("OutImmedArc", secondTProc1, secondinProg);
37     addConnection("InpImmedArc", secondinProg, secondTProc2);
38     addConnection("OutImmedArc", secondTProc2, firstinProg);
39 }
40 ...
```

Listing C-2 - *ECL transformation to perform second subtask of scaling snapshot*

To conclude, the introduction of new event types into an SRN model requires changes in several locations of the model. For example, new event types need to be inserted; some properties of model elements such as the guard functions need to be computed; and the execution snapshot needs to be expanded accordingly. The difficulties of scaling up an SRN model manually is due to the complicated dependencies among its model elements and parts, which can be addressed by a replicator using C-SAW and its

model transformation language ECL. With the expressive power of ECL, it is possible to specify reusable complicated transformation logic in a templatized fashion. The replication task is also simplified by distinguishing independent and dependent elements, and building up a larger SRN model in a stepwise manner. The result of the model replication preserves the benefits of modeling because the result of the replication can be persistently exported to XML and sent to a Petri net analysis tool.

C.2 Scaling Event QoS Aspect Language (EQAL)

The Event QoS Aspect Language (EQAL) [Edwards, 04] is a DSML for graphically specifying publisher-subscriber service configurations for large-scale DRE systems. Publisher-subscriber mechanisms, such as event-based communication models, are particularly relevant for large-scale DRE systems (e.g., avionics mission computing, distributed audio/video processing, and distributed interactive simulations) because they help reduce software dependencies and enhance system composability and evolution. In particular, the publisher-subscriber architecture of event-based communication allows application components to communicate anonymously and asynchronously. The publisher-subscriber communication model defines three software roles:

- *Publishers* generate events to be transmitted
- *Subscribers* receive events via hook operations
- *Event channels* accept events from publishers and deliver events to subscribers

The EQAL modeling environment consists of a GME metamodel that defines the concepts of publisher-subscriber systems, in addition to several model compilers that

synthesize middleware configuration files from models. The EQAL model compilers automatically generate publisher-subscriber service configuration files and component property description files needed by the underlying middleware.

The EQAL metamodel defines a modeling paradigm for publisher-subscriber service configuration models, which specify QoS configurations, parameters, and constraints. For example, the EQAL metamodel contains a distinct set of modeling constructs for building a federation of real-time event services supported by the Component-Integrated ACE ORB (CIAO) [Gokhale et al., 04], which is a component middleware platform targeted by EQAL. A federated event service allows sharing filtering information to minimize or eliminate the transmission of unwanted events to a remote entity. Moreover, a federated event service allows events that are being communicated in one channel to be made available on another channel. The channels typically communicate through CORBA Gateways, User Datagram Protocol (UDP), or Internet Protocol (IP) Multicast. In Figure C-2, to model a federation of event channels in different sites, EQAL provides modeling concepts including CORBA Gateways and other entities of the publish-subscribe paradigm (e.g., event consumers, event suppliers, and event channels).

C.2.1 Scalability Issues in EQAL

The scalability issues in EQAL arise when a small federation of event services must be scaled to a very large system, which usually accommodates a large number of publishers and subscribers [Gray et al., 06]. It is conceivable that EQAL modeling features, such as the event channel, the associated QoS attributes, connections and event

correlations must be applied repeatedly to build a large scale federation of event services. Figure C-2 shows a federated event service with three sites, which is then scaled up to federated event services with eight sites. This scaling process includes three steps:

- Add five `CORBA_Gateways` to each original site
- Repeatedly replicate one site instance to add five more extra sites, each with eight `CORBA_Gateways`
- Create the connections between all of the eight sites

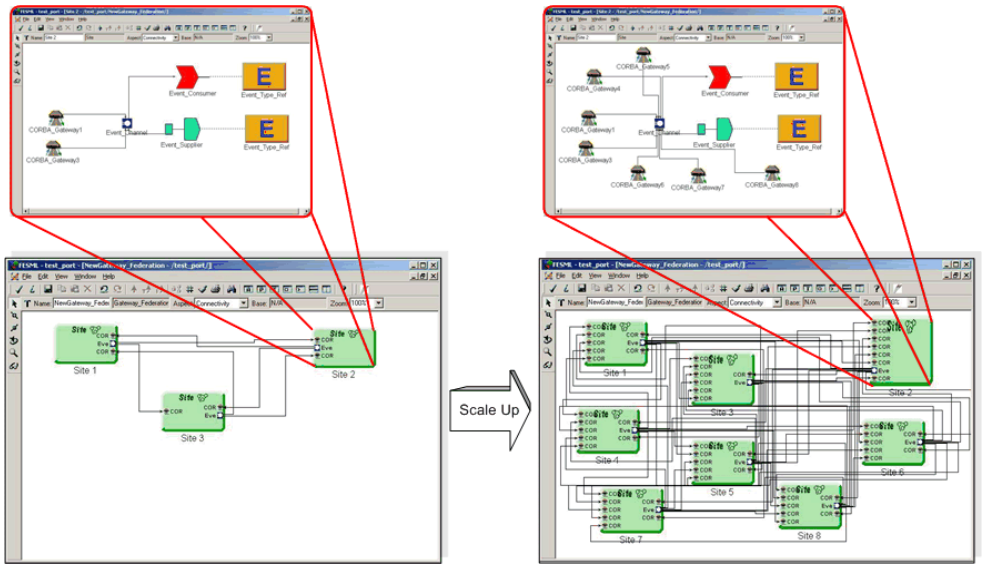


Figure C-2 *- Illustration of replication in EQAL*

C.2.2 ECL Transformation to Scale EQAL

The process discussed above can be automated with an ECL transformation that is applied to a base model with C-SAW. Listing C-3 shows a fragment of the ECL specification for the first step, which adds more Gateways to the original sites. The other steps would follow similarly using ECL. The size of the replication in this example was

kept to five sites so that the visualization could be rendered appropriately in Figure C-2. The approach could be extended to scale to hundreds or thousands of sites and gateways.

```
1  //traverse the original sites to add CORBA_Gateways
2  //n is the number of the original sites
3  //m is the total number of sites after scaling
4  strategy traverseSites(n, i, m, j : integer)
5  {
6    declare id_str : string;
7    if (i <= n) then
8      id_str := intToString(i);
9      rootFolder().findModel("NewGateway_Federation").
               findModel("Site " + id_str).addGateWay_r(m, j);
10     traverseSites(n, i+1, m, j);
11   endif;
12 }
13
14 //recursively add CORBA_Gateways to each existing site
15 strategy addGateWay_r(m, j: integer)
16 {
17   if (j<=m) then
18     addGateWay(j);
19     addGateWay_r(m, j+1);
20   endif;
21 }
22
23 //add one CORBA_Gateway and connect it to Event_Channel
24 strategy addGateWay(j: integer)
25 {
26   declare id_str : string;
27   declare ec, site_gw : object;
28   id_str := intToString(j);
29
30   addAtom("CORBA_Gateway", "CORBA_Gateway" + id_str);
31   ec := findModel("Event_Channel");
32   site_gw := findAtom("CORBA_Gateway" + id_str);
33   addConnection("LocalGateway_EC", site_gw, ec);
34 }
```

Listing C-3 *- ECL fragment to perform the first step of replication in EQAL*

To conclude, scaling EQAL from a small federation of events to a very large system requires creation of a large number of new publishers and subscribers. Also, the associated EQAL modeling features, such as the event channel, the associated QoS attributes, connections and event correlations must be built accordingly. Such model scalability can be achieved through ECL model transformation with the flexibility to control the scaling size and the reusability to repeat any desired scaling task.