A SEMANTIC ANCHORING INFRASTRUCTURE FOR MODEL-INTEGRATED COMPUTING

By

Kai Chen

Dissertation

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

August, 2006

Nashville, Tennessee

Approved:

Janos Sztipanovits

Stephen R. Schach

Gabor Karsai

Gautam Biswas

Sherif Abdelwahed

Benoit Dawant

献给我的父母和我的妻子

To my parents

&

To my beloved wife, Yongjie

ACKNOWLEDGMENTS

I am greatly appreciative and thankful to my advisor, Dr. Janos Sztipanovits, for his scientific vision, detailed guidance, great patience and tremendous encouragement throughout these years. Dr. Sztipanovits has guided me through my transition from an incoming graduate student to an outgoing researcher. I also thank Dr. Stephen R. Schach, Dr. Gabor Karsai, Dr. Gautam Biswas, Dr. Sherif Abdelwahed and Dr. Benoit Dawant for serving on my dissertation committee. I appreciate their guidance and advice. Dr. Schach, who advised me while I was in the Master's program, introduced me to the Software Engineering research area, which has direct impact on my thesis.

I would also like to thank the members of the Institute for Software Integrated Systems, especially Dr. Sandeep Neema, Dr. John Koo, Dr. Douglas Schmidt, Dr. Aniruddha Gokhale, Ethan Jackson, Matthew Emerson, Graham Hemingway, Gabor Madl, Andrew Dixon, Brian Williams and Christopher Buskirk. Our many discussions stimulated my research.

Most importantly, I want to express my deepest gratitude to my loving family. I am indebted to my mom and dad for their support, encouragement, and belief in me. Special thanks to my lovely wife, Yongjie, who makes all my work worthwhile. This work is dedicated to them.

Finally, thanks for the support and sponsorship given by the National Science Foundation through the Information Technology Research project, under contract number CCR-0225610.

TABLE OF CONTENTS

Page

ACKNOWLEDGMENTS iii

LIST OF TABLES vii

LIST OF FIGURES viii

LIST OF ABBREVIATIONS xi

Chapter

I. INTRODUCTION 1

II. BACKGROUND 7

Model-Driven Software Engineering 7
Model-Driven Architecture 8
Model-Integrated Computing 15
Model-Driven Software Development 26
Modeling Languages 34
The Unified Modeling Language 35
Hybrid System Interchange Format 39
Domain-Specific Modeling Language 40
Formal Methods 42
Semantic Specification Methods 42
Formal Specification Languages 44
Abstract State Machine 46

III. A SEMANTIC ANCHORING INFRASTRUCTURE 57

Formal DSML Specification 57
Semantic Anchoring Methodology 59
Semantic Anchoring Tool Suite 61
Abstract Syntax Modeling 62
Set-Valued Structural Semantics for Metamodels 63
A Formal Framework for Specifying Semantic Units 65
A Formal Framework for Model Transformation 67
The Abstract State Machine Language 69

IV. SEMANTIC ANCHORING CASE STUDY 74

The FSM domain in Ptolemy 74
The Basic FSM 75
The Hierarchical FSM 76
The Syntax Definition for FML 78
The Semantic Unit Specification for FML 79
AsmL Abstract Data Model for FML 80
Behavioral Semantics for FML 83
Semantic Anchoring Specification 87

V. A SEMANTIC UNIT FOR TIMED AUTOMATA BASED MODELING LANGUAGES 95

Semantic Units 95
Timed Automata 97
Timed Büchi Automata 98
Timed Safety Automata 98
Timed Automata Semantic Unit 97
Overview of TASU 100
Abstract Data Model 105
Operational Semantics 109
TASU Metamodel Specification 116
Semantic Anchoring to TASU 120
Semantic Anchoring for the UPPAAL Language 121
Semantic Anchoring for the IF Language 127

VI. SEMANTIC UNIT COMPOSITION 137

Compositional Specification of Behavioral Semantics 137
SEFSM Overview 140
Primary Semantic Units Used 146
Finite State Machine Semantic Unit 146
Synchronous Dataflow Semantic Unit 149
Compositional Semantics Specification for SEFSM Components 153
Structural Composition 153
Behavioral Composition 156
Compositional Semantics Specification for SEFSM Systems 160
Structural Composition 160
Behavioral Composition 162

VII. RESULTS, CONCLUTIONS AND FUTURE WORK 165

Results 165
Precision of the Semantics Specification 165
Validation Support 167
Satisfaction of the EFSM Designers 167
Efficiency of the Compositional Semantic Specification Approach 167

Conclusion 169
Future Work 171

REFERENCES 173

LIST OF TABLES

Table Page

7-1. Comparison between the CSSA and the traditional approach..........................168

LIST OF FIGURES

Figure Page

2-1. Model transformation 12

2-2. Overview of the MIC architecture 16

2-3. The metaprogrammable MIC tool suite 18

2-4. The GME architecture 19

2-5. The GReAT architecture 21

2-6. The DESERT design flow 22

2-7. The UDM framework architecture 24

2-8. The OTIF architecture 26

2-9. The AMDD approach 32

2-10. Layout of the HSIF tool chain implementation 39

2-11. Bold Stroke multi-threaded component interaction in ESML 41

3-1. Formal DSML specification 57

3-2. The semantic anchoring infrastructure 60

3-3. The semantic anchoring tool suite 62

3-4. Metamodel for a simple Automaton Language 64

3-5. Metamodel for a set of AsmL Data Structures 68

4-1. A basic FSM 76

4-2. A hierarchical FSM 77

4-3. A UML class diagram for the FML metamodel 78

4-4. Metamodel capturing AsmL Abstract Data Structures for FML 88

4-5. Top-level model transformation rule for the FML
semantic anchoring specifications 89

4-6. Model Transformation Rule: *SetAttributes* 90

4-7. Model Transition Rule: *SetInitialState* 91

4-8. Model Transition Rule: *CreateChildStateObject* 92

4-9. A hierarchical FSM model for ComputerStatus 93

5-1. A timed automaton example 97

5-2. A timed safety automaton with location invariants 98

5-3. The *System* and *Declaration* paradigms of the TASU metamodel 117

5-4. The *TimedAutomaton* paradigm of the TASU metamodel 118

5-5. A TASU timed automaton (*ComponentKindA*) 119

5-6. Top-level model transformation rule for the UPPAAL
semantic anchoring specification 122

5-7. A pattern graph in the GReAT specification of the transformational rule
for UPPAAL locations with Location Invariants 124

5-8. Semantic anchoring for a UPPAAL automaton with location invariants 125

5-9. Semantic anchoring for a UPPAAL automaton with urgent locations 125

5-10. Semantic anchoring for a UPPAAL automaton with committed locations 126

5-11. Semantic anchoring for a UPPAAL automaton with urgent synchronization 127

5-12. Semantic anchoring for an IF automaton with delayable transitions 129

5-13. An IF asynchronous model with policies #reliable, #urgent and #multicast 130

5-14. The semantic anchoring model for the IF asynchronous model
in Figure 5-13 132

5-15. The semantic anchoring model for the IF asynchronous model in Figure 5-13
with the delaying policies changed from #urgent to #delay[a, b] 134

5-16. The semantic anchoring model for the IF asynchronous model in Figure 5-13

with the delaying policies changed from #urgent to #rate[a, b] 136

6-1. A graphical representation for semantic unit composition...............................139

6-2. A simple SEFSM component model 142

6-3. A simple SEFSM system model...143

6-4. A paradigm in the SEFSM metamodel defining the system structure 144

6-5. A paradigm in the SEFSM metamodel defining the component structure.......145

6-6. A compositional structure of the SEFSM component
originally shown in Figure 6-2..154

6-7. The compositional structure of the SEFSM system
originally shown in Figure 6-3...161

LIST OF ABBREVIATIONS

AA-SU — Action Automaton Semantic Unit

AMDD — Agile Model Driven Development

AM — Agile Modeling

ASM — Abstract State Machine

AsmL — Abstract State Machine Language

CASE — Computer Aided System Engineering

CIM — Computation Independent Model

CCS — Calculus of Communicating Systems

CSP — Communicating Sequential Processes

CSSA — Compositional Semantics Specification Approach

DESERT — Design Space Exploration Tool

SDF — Synchronous Dataflow

SDF-SU — Synchronous Dataflow Semantic Unit

DSL — Domain Specific Language

DSML — Domain-Specific Modeling Language

EFSM — Extended Finite State Machine Language

EMF — Eclipse Modeling Framework

ESML — Embedded System Modeling Language

FML — FSM Modeling Language

FSM — Finite State Machine

FSM-SU — Finite State Machine Semantic Unit

GEF — Graphical Editor Framework

GME — Generic Modeling Environment

GReAT — Graph Rewriting and Transformation

HSIF — Hybrid System Interchange Format

ISIS — Institute for Software Integrated Systems

ITU — Telecommunication Sector of the International Telecommunication Union

LSL — Larch Shared Language

LOS — Lines of Specification

MDA — Model-Driven Architecture

MDSE — Model-Driven Software Engineering

MDSD — Model-Driven Software Development

MIC — Model-Integrated Computing

MIPS — Model-Integrated Program Synthesis

MoC — Model of Computation

MOF — Meta Object Facility

OCL — Object Constraint Language

OMG — Object Management Group

OMT — Object Modeling Technique

OOSE — Object-Oriented Software Engineering

OTIF — Open Tool Integration Framework

PIM — Platform Independent Model

PSM — Platform Specific Model

SEFSM — Simple Extended Finite State Machine Language

SDL — Specification and Description Language

TAML — Timed Automata Based Modeling Language

TASU — Timed Automata Semantic Unit

TDD — Test Driven Development

UDM — Universal Data Model

UML — Unified Modeling Language

UML-SPT — UML Profile for Schedulability, Performance and Time

VMC — Vehicle Motion Control

XMI — XML Metadata Interchange

CHAPTER I

INTRODUCTION

Model-driven software engineering (MDSE) is a software engineering technology in which modeling techniques are applied in the software development process. The objective of MDSE is to raise the level of abstraction by applying modeling techniques during the software development process. Throughout the history of software engineering, raising the level of abstraction has been the main goal that has driven the significant advances in developer productivity. It is this goal that has driven the evolution of software design from assembly languages to Third Generation Languages (3GLs), such as FORTRAN and C, to the object-orient languages, such as Java and C++. Today, visual modeling languages are the state of the art computation languages adopted by MDSE.

Currently the three most successful incarnations of MDSE are: Model-Driven Architecture (MDA) [2], Model-Integrated Computing (MIC) [7] and Model-Driven Software Development (MDSD) [28]. The MDA approach, proposed by OMG, allows a single model to specify the system functionality for multiple platforms through model transformation. The MIC approach, proposed and developed by ISIS at Vanderbilt University, emphasizes the adoption of domain-specific modeling languages (DSMLs) [13] in the software and system design processes. The MDSD refers to technologies that use models to automate the software development process in general. The term MDSD is

used when one does not wish to be associated with the OMG-only technology, vocabulary and vision.

Modeling languages are essential tools for MDSE, just as objected-oriented languages are the keys to objected-oriented design. In general, modeling languages fall into three categories: unified (or universal) modeling languages (such as UML [48]), interchange languages (such as the Hybrid System Interchange Format [50]) and DSMLs. Unified modeling languages are intended to act as all-encompassing frameworks, capable of handling any domain or system category. Interchange languages are optimized to providing specific quantitative analysis capabilities in design flows by facilitating the integration of a group of tools. DSMLs are tailored to the particular concepts, constraints and assumptions of application domains. A well-made DSML captures the concepts, relationships among the concepts, well-formedness rules, and semantics of the application domain and allows users to program imperatively and declaratively through model construction.

This research primarily focuses on MIC and DSMLs though some of the results are also applicable to the general MDSE approaches and modeling languages.

In many industrial applications, MIC is required to support the specification, analysis, design, verification and validation of large, complex systems in a broad range of heterogeneous domains, including hardware, software, information, process, personnel and facilities. In order to satisfy the complicated industrial requirements for MIC applications, instead of designing a single monolithic DSML that can capture all domains, multiple DSMLs are designed to address different domains. For example, one might define a DSML for high-level system design, a DSML for simulation, and a DSML

for property verification. These DSMLs are integrated together to create a MIC tool chain and to support a highly domain-specific design flow. However, there are a few concerns that may jeopardize adoption of DSMLs and MIC tool chains:

- The use of DSMLs with tightly integrated analysis tool chains leads to the accumulation of design assets as models defined in a DSML. Consequently, users run high risk of being "locked-in" to a particular tool chain.
- Incomplete and informal specification of DSMLs makes precise understanding of their syntax and semantics difficult. While a tightly integrated tool chain seems to relieve users from the need to fully understand the syntax and semantics of DSMLs, the cost may be high: the lack of in-depth understanding of models and analysis methods may prevent the organization from adopting new modeling and model analysis methods.
- The lack of formally specified semantics of DSMLs and analysis tools create major risk in safety critical applications. Semantic mismatch between design models and modeling languages of analysis tools may result in ambiguity in safety analysis or may produce conflicting results across different tools.

On the other hand, formal methods [63] [64] are mathematically based techniques for describing system properties. They are used to reveal ambiguity, incompleteness, and inconsistency in a system. When used early in the system development process, they can reveal design flaws that otherwise might be discovered only during the costly testing and debugging phases. When used later, they can help determine the correctness of a system implementation and the equivalence of different implementations. In recent years, formal methods have been widely employed to specify the semantics of modeling languages.

The MIC approach will be superficial and risky if the DSMLs it depends on have no precise semantics definition. However, many formal methods are too complicated and expensive for industrial applications. The **challenge** is to propose a lightweight DSML semantics specification approach without compromising the precision of the semantics.

This research is to address this challenge in DSML design. The following research statement outlines the objectives of this research.

The goal of this research is to propose and implement an affordable technology that can facilitate formal DSML design with precise syntax and semantics definition and to build a solid semantic infrastructure for the MIC approach.

The main **contribution** of this research is the proposal and development of a semantic anchoring infrastructure that facilitates the transformational specification of DSML semantics. It is based on the observation that, in the embedded software and systems domain, there is a finite set of basic behavioral categories, such as Finite State Machine, Timed Automata, Discrete Event Systems and Synchronous Dataflow, each of which captures the behavioral pattern of a class of systems. The semantic anchoring infrastructure includes a set of semantic units that capture the behavioral semantics of basic behavioral categories using a formal method, Abstract State Machines [70], as the underlying semantic framework.

If the semantics of a DSML can be directly mapped onto one of these basic categories, its semantics can be defined by simply specifying the model transformation

rules between the DSML and the Abstract Data Model of the semantic unit. However, in heterogeneous systems, the semantics is not always fully captured by a predefined semantic unit. If the semantics is specified from scratch (which is the typical solution if it is done at all) it is not only expensive but we lose the advantages of anchoring the semantics to a set of common and well-established semantic units. This is not only losing reusability of previous efforts, but has negative consequences on our ability to relate semantics of DSMLs to each other and to guide language designers to use well understood and safe behavioral and interaction semantic "building blocks" as well. We propose a compositional semantics specification approach to define semantics for heterogeneous DSMLs.

This paper is organized as follows:

Chapter I gives a short survey and summarizes the challenge, the goal and the main contribution of this research.

Chapter II investigates the state of the art research in model-driven software engineering, modeling languages and formal methods. A particular formal method, Abstract State Machine, which is adopted as the semantic framework in the semantic anchoring infrastructure, is also introduced in this chapter. The purpose of the survey is to develop a deep understanding of the key issues pertaining to the modeling techniques.

Chapter III presents the semantic anchoring infrastructure and the tool suite that supports the DSML semantics specification through semantic anchoring. It also discusses the advantages and disadvantages of two metamodeling strategies in specifying semantic units.

Chapter IV uses the FSM domain in Ptolemy as a case study to explain the semantic anchoring methodology and to illustrate how the semantic anchoring tool suite is applied to design DSMLs.

Chapter V illustrates the semantic unit specification by defining a Timed Automata Semantic Unit (TASU). The precise semantics of a wide range of Timed Automata based modeling languages (TAMLs) can then be defined by specifying semantic anchoring rules between a domain-specific TAML and the TASU.

Chapter VI introduces the compositional semantics specification approach, which defines the semantics of a DSML as the composition of multiple semantic units. An industrial-strength modeling language, EFSM (Extended Finite State Machine Language), is employed as a case study to illustrate compositional semantics specification.

Chapter VII examines the results of the research and provides recommendations for future work.

CHAPTER II

BACKGROUND

This chapter presents an overview of the state of the art research in Model-Driven Software Engineering, modeling languages and formal methods. The purpose of the survey is to develop a deep understanding of the key issues pertaining to the modeling techniques.

The first part of this chapter surveys the most successful approaches for Model-Driven Software Engineering, including Model-Driven Architecture, Model-Integrated Computing and Model-Driven Software Development. The second part introduces modeling languages employed in Model-Driven Software Engineering. The last part of this chapter presents formal methods.

Model-Driven Software Engineering

Today, software systems have become very complex. In many applications, they may be integrated with the physical systems that support safety critical functions, making them necessarily complex to develop, test and maintain. Models provide a good solution for the software developers to manage this complexity and to understand the design and associated risks by abstracting away irrelevant details while highlighting the relevant. More specifically, by modeling software, developers can: [1]

- Create and communicate software designs before committing additional resources.
- Trace the design back to the requirements, helping to ensure that they are building the right systems.
- Practice iterative development, in which models and other higher levels of abstraction facilitate quick and frequent changes.

Three most successful approaches for Model-Driven Software Engineering are Model-Driven Architecture, Model-Integrated Computing and Model-Driven Software Development.

Model-Driven Architecture

The Model-Driven Architecture (MDA) [2], proposed by the Object Management Group (OMG), starts with the long established idea of separating the specification of the operation of a system from the details of the way that the system uses the capabilities of its platform. The primary goal of MDA is to realize portability, interoperability and reusability through architectural separation of concerns. The MDA defines an approach to IT system specification that separates the specification of system functionality from the specification of the implementation of that functionality on a specific technology platform. To that end, the MDA defines an architecture for models that provides a set of guidelines for structuring specifications expressed as models.

The MDA approach and the standards that support it allow the model that specifies system functionality to be realized on multiple platforms through auxiliary mapping standards, or through point mappings to specific platforms. The approach also

allows different applications to be integrated by explicitly relating their models, enabling integration, interoperability and system evolution as platform technologies come and go [3].

MDA concepts are presented in terms of some existing or planned systems. A *system* may include anything: a program, a single computer system, some combination of parts of different systems, a federation of systems. Much of the discussion focuses on software within the system.

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 term *model-driven* means that MDA provides a means for using models to direct the course of understanding, design, construction, deployment, operation, maintenance and modification. The *architecture* of a system is a specification of the parts and connectors of the systems and the rules for the interactions of the parts using connectors.

A *platform* is a set of subsystem and technologies that provide a coherent set of functionality through interfaces and specified usage patterns, which any application supported by that platform can use without concern for the details of how the functionality provided by the platform is implemented. For example, *CORBA™* [4], *CORBA Components* [5] and *Java 2 Enterprise Edition (J2EE™)* [6] are some technology specific platform types. *Platform independence* is a quality, which a model may exhibit. A model is platform independent if its features can be interpreted on any platform.

A *viewpoint* of a system is a technique for abstraction using a selected set of architectural concepts and structuring rules, in order to focus on particular concerns within that system. The MDA specifies three viewpoints on a system, a computation independent viewpoint, a platform independent viewpoint and a platform specific viewpoint.

The *computation independent viewpoint* focuses on the environment of the system, and the requirements for the system; the details of the structure and processing of the system are hidden or as yet undetermined. The *platform independent viewpoint* focuses on the operation of a system while hiding the details necessary for a particular platform. A platform independent view shows the part of the specification that does not change from one platform to another. The *platform specific viewpoint* combines the platform independent viewpoint with an additional focus on the detail of the use of a specific platform by a system.

A *computation independent model* (CIM) is a view of a system from the computation independent viewpoint. The requirements for the system are modeled in a CIM by describing the situations in which the system will be used. A CIM is a model of a system that shows the system in the environment in which it will operate, and thus it helps in presenting exactly what the system is expected to do. It is useful, not only as an aid to understanding a problem, but also as a source of a shared vocabulary for use in other models. A CIM is sometimes called a business model. It may hide much or all information about the use of automated data processing systems. Typically such a model is independent of how the system is implemented.

A *platform independent model* (PIM) is a view of a system from the platform independent viewpoint. A PIM might consist of enterprise, information and computational Open Distributed Processing viewpoint specifications. A PIM exhibits a specific degree of platform independence so as to be suitable for use with a number of different platforms of similar type. A very common technique for achieving platform independence is to target a system model for a technology-neutral virtual machine. A virtual machine is defined as a set of parts and services (communication, scheduling, naming, etc.), which are defined independently of any specific platform and which are realized in platform-specific ways on different platforms. A virtual machine is a platform, and such a model is specific to that platform. But that model is platform independent with respect to the class of different platforms on which that virtual machine has been implemented.

A *platform specific model* (PSM) is a view of a system from the platform specific viewpoint. A PSM combines the specification in the PIM with the details that specify how that system uses a particular type of platform. PSMs have to use the platform concepts, such as exception mechanisms, parameter types (including platform-specific rules about objects references, value types, semantics of call by value, etc.), and component models.

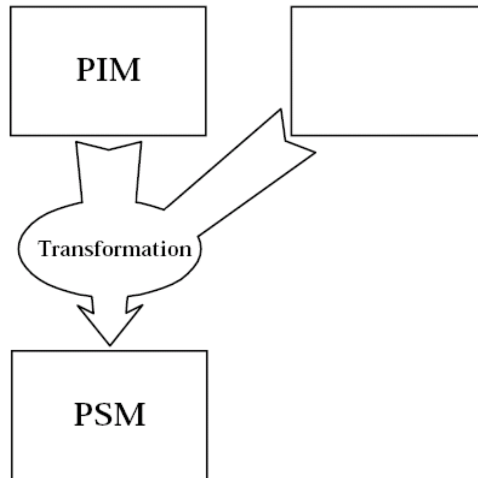

Figure 2-1 Model transformation

Model transformation is a process of converting one model to another model of the same system. The transformation from PIM to PSM is a key part of MDA. Figure 2-1 illustrates the MDA pattern, by which a PIM is transformed to a PSM. The PSM and other information are combined by the transformation to produce a PSM. The MDA proposes five approaches to realize the transformation from PIM to PSM, including marking, metamodel transformation, model transformation, pattern matching and model merging.

Among many tools that support MDA, IBM Eclipse-based modeling tools [30] may be one of the most popular tools. The Eclipse Modeling Framework (EMF) allows model designers to input a data model, and generate simple table-based editors and XMI schema for such models. The Graphical Editor Framework (GEF) supplies functions and classes useful for specifying graphical editors for Eclipse data models.

Eclipse Modeling Framework (EMF)

EMF [33] [34] is an open source framework and toolkit targeting MDA. It is the current implementation of a portion of MDA in the Eclipse family of tools. EMF extends Eclipse's Java Development Tool into the world of model-driven development and provides an easy way for users to define models, from which many common code-generation patterns are generated.

EMF provides a metamodel, called "Ecore", for describing EMF models. Ecore is based on a core subset of the OMG's Meta Object Facility (MOF) API [35]. In the current proposal for MOF 2.0, a similar subset of the MOF model, which it calls EMOF (Essential MOF) is separated out. There are small, mostly syntactical, differences between Ecore and EMOF. However, EMF can transparently read and write serializations of EMOF.

EMF uses XMI (XML Metadata Interchange) as its canonical form for a model representation [36]. Users have several ways of defining models in that form: (1) create the XMI document directly, using an XML or text editor; (2) export the XMI document from a modeling tool, such as Rational Rose; (3) annotate Java interfaces with model properties; (4) use XML Schema to describe the form of a serialization of the model.

Once an EMF model is created, the EMF generator can create a corresponding set of Java implementation classes. Every generated EMF class extends the framework base class, EObject, which enables the objects to integrate and work in the EMF runtime environment. EObject provides an efficient reflective API for accessing the object's properties. In addition, change notification is an intrinsic property of every EObject and an adapter framework can be used to support open-ended extension of the objects. The

runtime framework also manages bidirectional reference handshaking, cross-document referencing including demand-load, and arbitrary persistent forms with a default generic XMI serialization that can be used for any EMF model. EMF also provides support for dynamic models (Ecore models are created in memory and then instantiated without generating code).

These generated classes can be edited by adding methods and instance variables. If the EMF model is changed, the regenerated code will still keep the added code. If the code added depends on some changes in the model, the code is needed to be updated to reflect those changes; otherwise, the code is completely unaffected by model changes and regeneration.

EMF consists of two fundamental frameworks: the core framework and EMF.Edit. The core framework provides basic generation and runtime support to create Java implementation classes for a model. EMF.Edit extends and builds on the core framework, adding support for generating adapter classes that enable viewing and command-based (undoable) editing of a model, and even a basic working model editor.

Graphical Editor Framework (GEF)

EMF only provides part of the solution for models: data storage, property sheets, tree or table-based browsing, and code generation framework. GEF provides the graphical support needed for building a diagram editor on top of the EMF framework [37] [38]. Note that GEF and EMF can be used separately and there is no dependency between the two frameworks. The only thing they share is their integration with Eclipse change notification.

Every GEF application uses a model to represent the state of the diagram being created and edited. GEF employs a Model-View-Controller architecture which relies on controllers that listen for model changes and update the view in response. As all EMF model objects notify change via EMF's notification framework, notification to an EMF model is already in place.

The GEF Model can be an EMF model, or something totally different. The GEF Controller is called an EditPart, and for each EMF model element class, a corresponding EditPart class must be created. EditParts have figures, which are their graphical view, implemented in the lower-level Draw2D graphical framework [39]. EditParts respond to events by way of an EditPolicy. The job of the EditPolicy is to turn the event request into a Command. GEF uses the Command pattern to implement an undo stack.

Model-Integrated Computing

Model-Integrated Computing (MIC) [7] [8] [9] has been developed by the Institute for Software Integrated System (ISIS) at Vanderbilt University for embedded software and system design. MIC focuses on the formal representation, composition, analysis and manipulation of models during the design process. It employs domain-specific models to represent the software, its environment, and their relationships. With Model-Integrated Program Synthesis (MIPS), these models are then used to automatically synthesize the embedded applications. The MIC approach places models at the center of the entire life-cycle of systems, including specification, design, development, verification, integration, and maintenance.

Using MIC techniques, one can capture the requirements, actual architecture, and the environment of a system in the form of high-level models. The requirement models allow the explicit representation of desired functionalities and/or non-functional properties. The architecture models represent the actual structure of the system to be built, while the environment models capture the system environment. These models act as a repository of information that is needed for analyzing and generating the system.

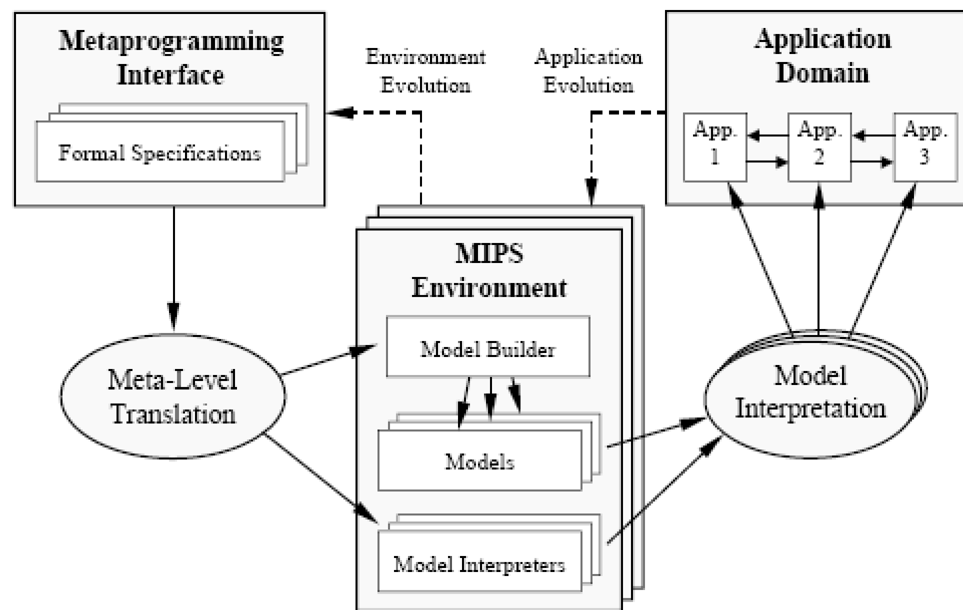

Figure 2-2 Overview of the MIC architecture [10]

As shown in Figure 2-2, the MIC architecture has the following three levels of abstraction:

Application Level represents the synthesized, adaptable software applications. The executable programs are specified in terms of a computation platform.

MIPS Level comprises generic, customizable, domain-specific tools for model building, model analysis and application synthesis. The generic components in MIPS level include

a customizable graphical model builder, database for storing and accessing models and model interpreters.

Meta-Level is a metaprogrammable interface, which provides metamodeling languages, metamodels, a metamodeling environment and meta-generators for creating domain specific tool chains on the MIPS level.

A MIC application starts with the formal specification of a new application domain through a metamodeling process. The metamodels capture the modeling concepts, the relationship among these concepts, integrity constraints and visualization rules of the application domain. The visualization rules determines how domain models are to be visualized and manipulated in a visual modeling environment. Through the step called "Meta-Level Translation", a MIPS environment can be generated from this application domain specification. The domain users create domain-specific models within the MIPS environment. The next step is to do model interpretation through model interpreters. Mode interpreters synthesize applications (application models), or translate models into input data structures for analysis tools. Internal tools are designed for specific MIPS environments, and typically include a model interpreter, an analysis algorithm and user interface. External tools are research tools that perform some static or dynamic analysis based on a domain independent abstract model.

The MIC development is also an evolutionary process supporting both the application evolution and the environment evolution. The application evolution is the evolution of the computer-based system execution models. Since both the behavior and structure of the system is specified using models, any changes to that behavior are made to models and the updated system is generated from the models. The environment

evolution is the evolution of the domain specification. When the domain environment is evolved, the metamodels need to be modified to capture the changes. A new MIPS environment is generated from the updated metamodel.

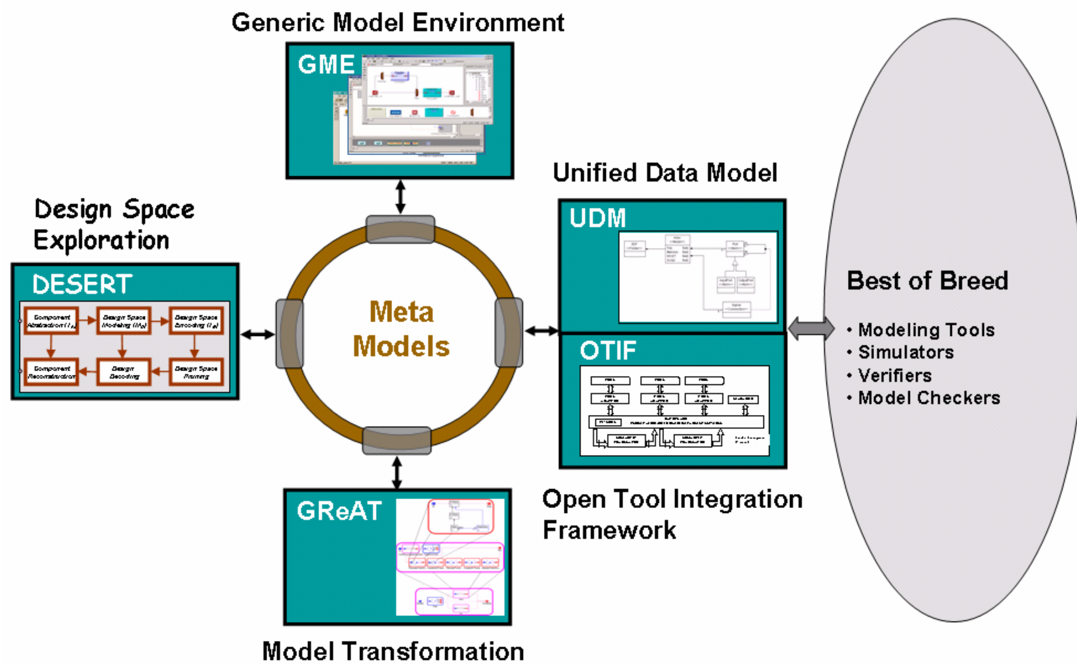

Figure 2-3 The metaprogrammable MIC tool suite

Figure 2-3 presents the overview of the fully integrated metaprogrammable MIC tool suite developed by ISIS, Vanderbilt University. The tool suite includes the Generic Modeling Environment (GME), the Graph Rewriting and Transformation (GReAT) Tool Suite, the Design Space Exploration Tool (DESERT), the Universal Data Model (UDM) Framework and the Open Tool Integration Framework (OTIF).

Generic Modeling Environment (GME)

GME [11] [12] is a configurable toolkit for creating domain-specific modeling and program synthesis environments. The configuration is accomplished through metamodels specifying the modeling paradigm of the application domain. The modeling paradigm contains the modeling concepts, the relationships among these modeling concepts, well-formedness rules, and concrete syntax of the domain. The modeling concepts will be used to construct models. The concrete syntax determines the organization and visualization of modeling concepts in domain models.

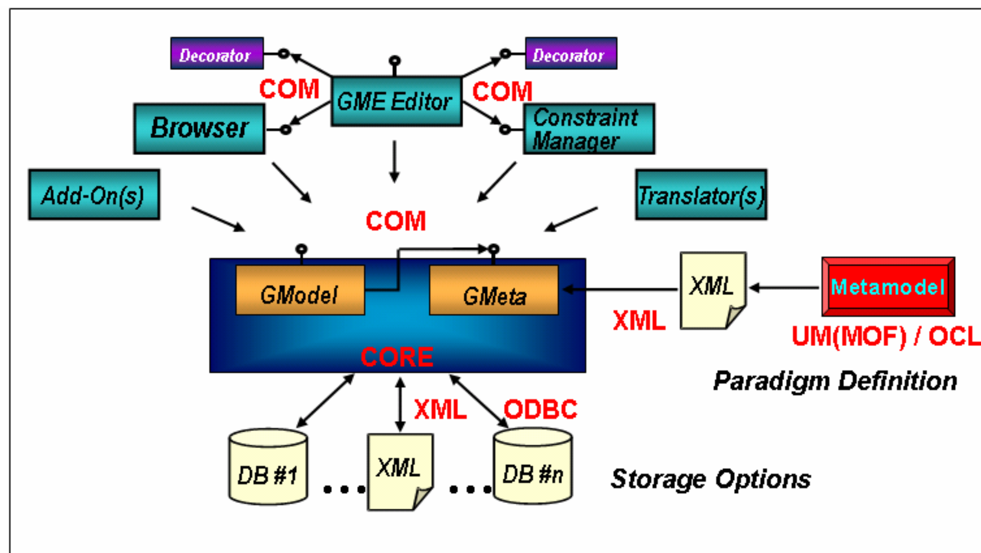

Figure 2-4 The GME architecture

Figure 2-4 presents the GME software architecture. The metamodeling language for GME is based on the UML class diagram and OCL constraints. The metamodels provide information that is used to automatically generate the target domain-specific modeling environment [13]. The generated domain-specific environment is then used to build domain models that are stored in a model database or in the XML format. These

domain models are used to automatically generate the applications or to synthesize input to different analysis tools.

GME has a modular, extensible architecture that uses Microsoft COM for integration. GME is easily extensible; external components can be written in many languages, including C++, Java, Visual Basic, C# and Python. GME has many advanced features. A built-in constraint manager enforces all domain constraints during model building. GME supports multiple-aspect modeling. It provides metamodel composition for reusing and combining existing modeling languages and language concepts [14] [15]. It supports model libraries for reuse at the model level. Model visualization is customizable through decorator interfaces.

The Graph Rewriting and Transformation (GReAT) Tool Suite

GReAT [16] [17] is designed for the specification and implementation of model to model transformations. The GReAT tool suite, as shown in Figure 2-5, includes the GReAT Modeling Tool for constructing model transformation algorithms, the GReAT Engine that directly interprets and executes GReAT programs, and the GReAT Debugger that compiles GReAT programs into efficient code. The entire GReAT tool suite is fully integrated with GME. The MIC model transformation technology is based on graph transformation semantics and can be applied to integrate models by extracting information from separate model databases, translating domain models for simulation and analysis tools [18] and realizing domain model evolution [19].

The GReAT language [20] is a graph-transformation based language that supports the high-level specification of complex model transformation programs. In this language, one describes the transformations as sequenced graph rewriting rules that operate on the

input models and construct an output model. The rules specify complex rewriting operations in a concise, yet precise manner, in the form of a matching pattern and a subgraph to be created as the result of the application of the rule. The rules always operate in a context: a specific subgraph of the input, and are explicitly sequenced for efficient execution. The rules are specified visually using a graphical model builder tool.

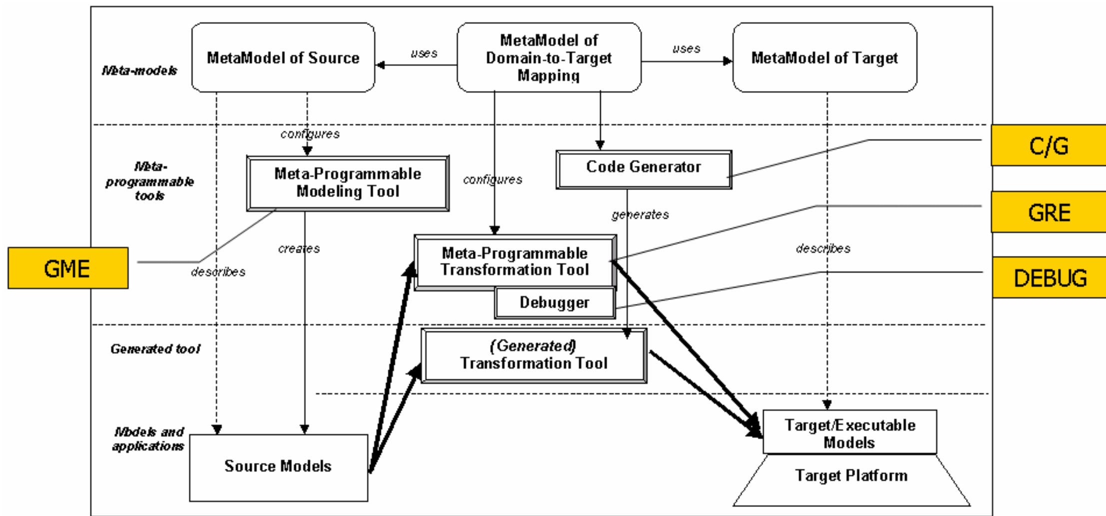

Figure 2-5 The GReAT architecture

The Design Space Exploration Tool (DESERT)

DESERT [21] [22] is a meta-programmable tool for navigation and pruning of large design spaces using constraints. It provides a model synthesis tool suite that can represent large design space and can manipulate them by means of structural constraints. An expressive constraint language based on a subset of OCL allows expression of compositional, resources, and performance (time, energy, size, weight, cost) constraints. Internally, DESERT employs a powerful and highly scalable symbolic representation based on Ordered Binary Decision Diagrams [23], that allows for rapid, and efficient

manipulation of very large design spaces with constraints. In order to solve constraints that involve complex mathematical operations, DESERT interfaces with Mozart, a powerful environment for constraint logic programming based on the Oz constraint language [24]. An XML based input and output interfaces accompanied with a programmatic API, allows easy and semantically correct integration of DESERT with custom DSMLs.

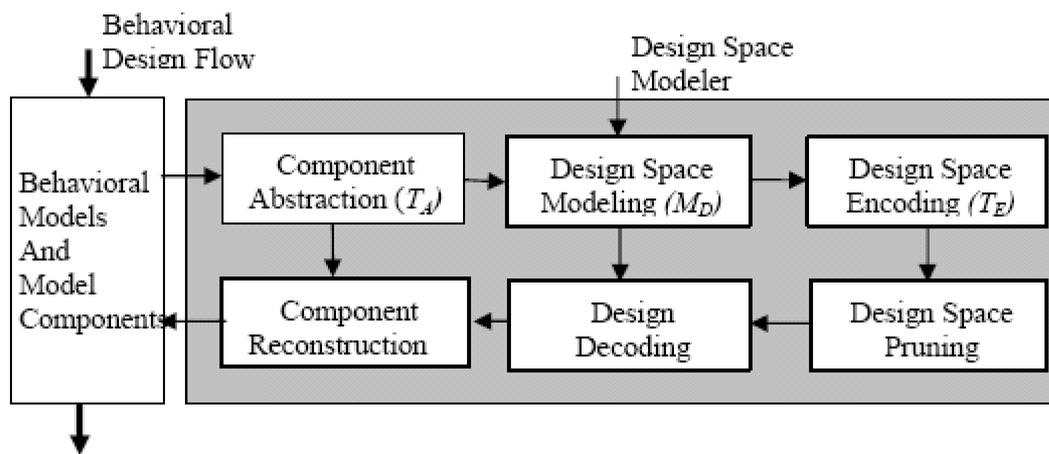

Figure 2-6 The DESERT design flow [21]

The inputs to the DESERT are models and model components used for behavioral modeling and analysis. The overall DESERT design flow, as shown in Figure 2-6, includes the following steps:

1. The *Component Abstraction* tool maps input model components into Abstract Components through the T_A model transformation.
2. The *Design Space Modeling* tool supports the construction of M_D design spaces using Abstract Components.

3. The *Design Space Encoding* tool maps the M_D design space into a representation form, which is suitable for manipulating/changing the space by restricting it with various design constraints.
4. The *Design Space Pruning* tool performs the design space manipulation and enables the user to select a design, which meets all structural design constraints.
5. The *Design Decoding* tool reconstructs the selected design from its encoded version.

The Universal Data Model (UDM)

UDM [25] is a metaprogrammable tool for providing uniform access to data structures that could be persistent. The tool uses UML class diagram as the language for defining the data structures and it generates C++ or Java class definitions for implementing the classes. Each attribute and association will have a corresponding setter/getter method in the generated code. The generated class implementations are handles pointing to generic objects that provide the real implementation: these generic objects can be persistent and be mapped into: XML files, GME project files, database tables, or CORBA structures. For each kind of generic object there is a separate back-end library that implements the objects in terms of the underlying technology (XML structures, GME objects, database tables, CORBA structs).

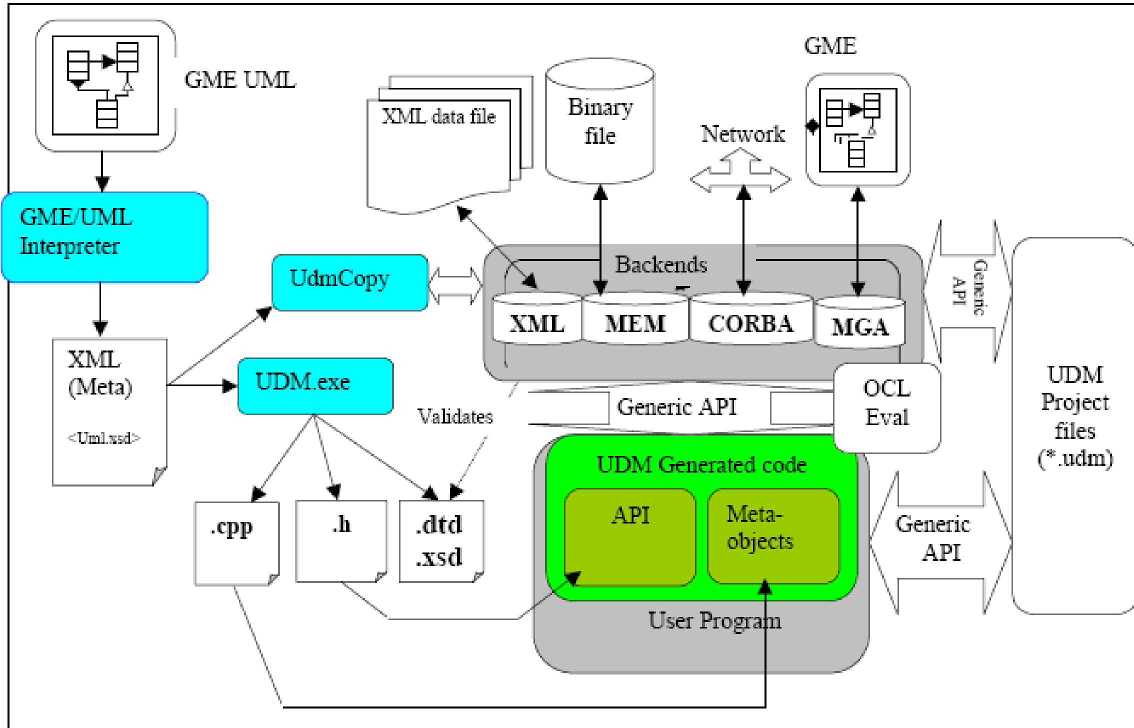

Figure 2-7 The UDM framework architecture [25]

The UDM framework architecture is shown in Figure 2-7. The current storage technologies support XML with automatically generated DTD files, MGA (the native interface of GME) and memory-based storage. The framework consists of the following modules:

- The GME UML environment with the GME UML paradigm and interpreter, which generates equivalent XML files from GME UML models.
- THE Udm program, which reads in the XML files and generates the metamodel-dependent portion of the UDM API: a C++ source file, a C++ header file, and an XML document type description (DTD) or XML schema definition (XSD).

- The generic Udm included headers and libraries to be linked to the user's program.
- Utility programs to manipulate and query Udm data (UdmCopy, UdmPat).

The Open Tool Integration Framework (OTIF)

OTIF [26] [27] provides a reusable software framework and integration technology for composing tool chains to form specific tool integration solutions. Development of large-scale engineering systems (including the software development for distributed, real-time embedded systems) often necessitates the integration of various engineering tools.

OTIF provides a set of reusable components and libraries, as well as a process to construct integrated tool chains. OTIF is based on a backplane-based architecture where tools can interchange data with each other. During interchange, the OTIF backplane schedules and executes appropriate transformations on the data (using translator elements), in a manner that is compliant with a modeled workflow across the tools.

The OTIF architecture for a 3-tool integration solution is illustrated in Figure 2-8. In general, elements of OTIF include:

- Backplane: a generic server component that includes the workflow engine, host tool metadata, and orchestrates the execution of the tool chain. The workflow engine enacts a workflow specified in visual model.
- Manager: a generic component for configuring the backplane.
- Tool adaptors: specific client components that reads and writes tool specific data, convert that data into/from a generic OTIF format.

- Semantic translator: specific client components that perform semantic translation on the data received from the backplane and send the results back to the backplane.

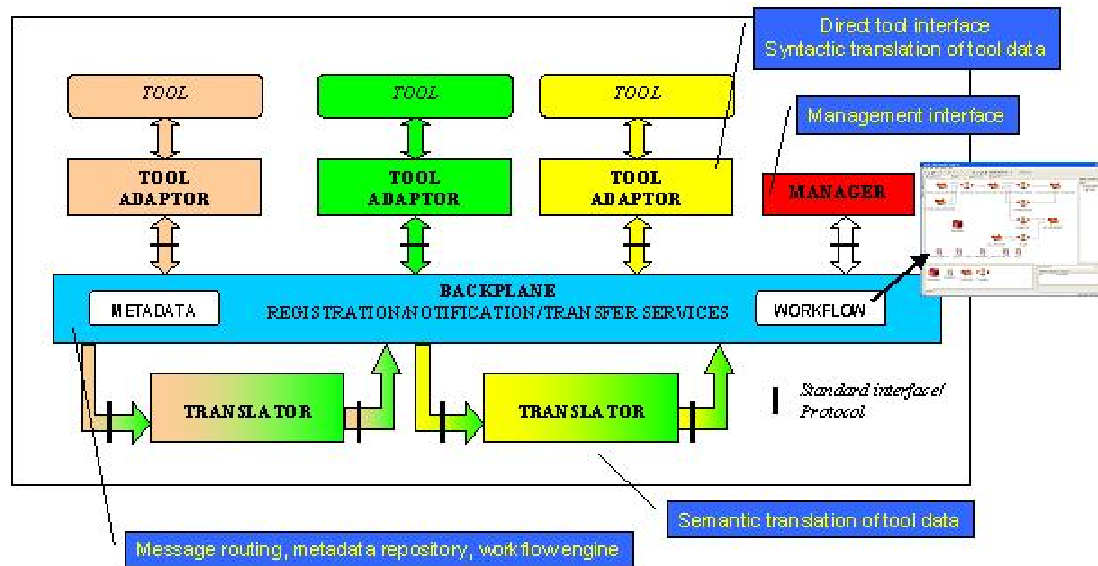

Figure 2-8 The OTIF architecture [26]

Model-Driven Software Development

Model-Driven Software Development (MDSD) [28] [29] refers to technologies that apply information captured by models to automate software development process in general. The term MDSD is used when one does not wish to be associated with the OMG-only technology, vocabulary and vision. So, different tool vendors and organizations propose different approaches and tools to support MDSD. In this section, Microsoft Software Factories [31] and Agile Model Driven Development [32] are used as two examples to illustrate different MDSD versions.

Microsoft Software Factories

The Microsoft approach to support MDSD is called Software Factories [40] [42], which is hosted by Microsoft Visual Studio .NET. The purpose of Software Factories is to provide a faster, less expensive and more reliable approach to application development by increasing the level of automation in application development, using modeling languages to enable rapid assembly and configuration of framework based components. Software Factories automate the packaging and delivery of the reusable assets, including models and model-driven tools, other types of tools, such as wizards, templates and utilities, development processes, implementation components, such as class libraries, frameworks and services, and content assets, such as patterns, style sheets, help files, configuration files, and documentation. With Software Factories, models are used not only for analysis and design, but to support many varied types of computation across the entire software life cycle.

In a brief definition, a software factory is a software product line that configures extensible tools, processes and content using a *software factory template* based on a *software factory schema* to automate the development and maintenance a variants of an archetypical product by adapting, assembling and configuring framework based components [41].

A software factory schema is a document that categorizes and summarizes the artifacts used to build and maintain a system, such as XML documents, models, configuration files, build scripts, source code files, SQL files, localization files, deployment manifests and test case definitions, in an orderly way, and that defines relationships between them. A software factory schema defines a recipe for building

members of a software product family. The viewpoints describe the ingredients and the tools used to prepare them. A process framework is constructed by attaching a micro process to each viewpoint, describing the development of conforming views, and by defining constraints like preconditions that must be satisfied before a view is produced, post-conditions that must be satisfied after it is produced, and invariants that must hold when the views have stabilized.

For any given project, the software factory schema is customized to create a recipe for building that specific member of the product family. A software factory schema contains both fixed and variable parts. The fixed parts remain the same for all members of the product family, while the variable parts change to accommodate the unique requirements of each specific member. Different parts of the customization can be performed at different times, according to the needs of the project. Some parts, such as adding or dropping whole viewpoints, and relationships between them, might be performed up front, based on major variations in requirements, such as dropping personalization. Other parts, such as modifying viewpoints and the relationships between them, might be performed incrementally, as the project progresses, based on more fine grained variations in requirements, such as deciding what mechanisms to use to drive the user interface.

A software factory template is an implementation of the software factory schema. With the software factory schema, the assets used to build family members can be described, but the concrete implementation of these assets is still missing. The software factory schema needs to be concretized by defining the domain specific languages (DSLs), patterns, frameworks and tools it describes, packaging them, and making them

available to product developers. Collectively, these assets form a software factory template. So, a software factory template includes code and metadata that can be loaded into extensible tools, like an Interactive Development Environment, or an enterprise life cycle tool suite, to automate the development and maintenance of family members. It is called a template because it configures the tools to produce a specific type of software, just as a document template loaded into a tool like Microsoft Word or Excel configures it to produce of a specific type of document.

A software factory template also needs to be customized for a specific family member. While customizing a software factory schema customizes the description of the software factory for the family member, however, customizing a template customizes the assets used to build the family member. The software factory template is generally customized at the same time as the software by adding, dropping or modifying the assets associated with those viewpoints. Examples of software factory template customization include creating projects for the subsystems and components to be developed, populating a palette with patterns to be applied, setting up references to libraries to be used, and configuring builds.

Assets for a software product may include the requirements, development process, architecture, components, deployment configuration and tests. They are specified, reused, managed and organized from the viewpoints defined by the customized software factory schema. Building a product using a Software Factory involves the following activities [41]:

- *Problem Analysis* determines whether or not the product is in scope for the Software Factory. Depending on the fit, some or all of product may be built

outside the Software Factory. In some cases, the software factory schema and template need to be changed so that they better accommodate the parts that did not fit well in future products.

- *Product Specification* defines the product requirements in terms of differences from the product line requirements. A range of product specification mechanisms can be used, depending on the extent of the differences in requirements, including property sheets, wizards, feature models, visual models and structured prose.
- *Product Design* maps the differences in requirements to differences in the product line architecture and the product development process, producing a product architecture, and a customized product development process.
- *Product Implementation* involves familiar activities, such as component and unit test development, builds, unit test execution, and component assembly. A range of mechanisms can be used to develop the implementation, depending on the extent of the differences, such as property sheets, wizards and feature models that configure components, visual models that assemble components and generate other artifacts like models, code and configuration files, and source code that completes frameworks extension points, or that creates, modifies, extends or adapts components.
- *Product Deployment* involves creating or reusing default deployment constraints, logical host configurations and executable to logical host mappings, by provisioning facilities, validating host configurations,

reconfiguring hosts by installing and configuring the required resources, and installing and configuring the executables being deployed.

- *Product Testing* involves creating or reusing test assets, including test cases, test harnesses, test data sets, test scripts and applying instrumentation and measurement tools.

Agile Model Driven Development

Agile Modeling (AM) defines a collection of values, principles and practices for modeling software that can be applied on a software development project in an effective and light-weight manner. The Agile Model Driven Development (AMDD) [43] describes an approach for applying AM in conjunction with agile implementation techniques such as Test Driven Development (TDD), code refactoring and database refactoring.

Traditionally, models are thought as diagrams plus any corresponding non-visual documentation, such as use cases or a textual description of business rules. An agile model denotes a model that is *just barely good enough* [44]. By definition, an artifact is just barely good enough means that it is at the most effective point that it could possibly be at. Agile models are just barely good enough when they exhibit the following traits:

- Agile models fulfill their purpose.
- Agile models are understandable.
- Agile models are sufficient accurate.
- Agile models are sufficient detailed.
- Agile models provide positive value.
- Agile models are as simple as possible.

The goal of MDSD is typically to create comprehensive models, and then generate software from these models. This often requires computer aided system engineering (CASE) tools and IT professionals with sophisticated modeling skills. AMDD intends to describe how developers and stakeholders can work together cooperatively to create models which are just barely good enough. It assumes that each individual has some modeling skills, or at least some domain knowledge, that they will apply together in a team in order to get job done. AMDD allows developers to use modeling tools, but does not depend on these tools.

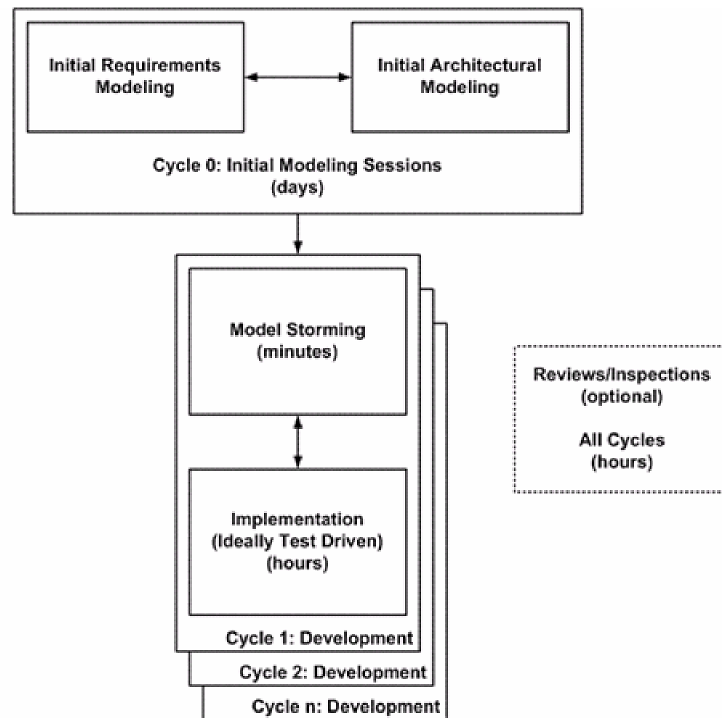

Figure 2-9 The AMDD approach [45]

Figure 2-9 depicts a high-level lifecycle for AMDD for the release of a system [45]. Each box represents a development activity. The initial modeling activity occurs

during cycle/iteration 0 and includes two main sub-activities, initial requirements modeling and initial architecture modeling. The other activities – model storming, reviews, and implementation – potentially occur during any cycle. The time indicated in each box represents the length of an average session.

The *initial modeling* is typically performed during the first week of a project. For short projects (perhaps several weeks in length) it may take a few hours and for long projects (perhaps on the order of twelve or more months) it may take two weeks. During the initial modeling, agile modelers are likely to identify high-level usage requirements models such as a collection of use cases or user stories; identify high-priority technical requirements and constraints; create high-level domain models. In cycle 0, the goal of initial modeling is to get something that is just barely good enough so that the software development team can get coding. In the later cycles both the initial requirements and the initial architecture models will need to evolve as modelers learn more.

The *model storming* session quickly explore in detail a specific issue before it is implemented. Model storming is just in time modeling: identify an issue that needs to be resolved, grab a few team members who can help, the group explores the issue, and then everyone continues on as before. The model storming session typically lasts for five to ten minutes.

The *implementation* practice may include code refactoring, database refactoring and Test-Driven Design. A code refactoring is a simple change to your code that improves its design but does not change its behavioral semantics. Common code refactorings include Rename Process, Remove Control Flag, Change Value to Reference, and Move Process. A database refactoring [46] is a simple change to a database schema

that improves its design while retaining both its behavioral and informational semantics. Database refactorings may focus on data quality, structural changes and performance enhancement. TDD [47] is an approach where tests are identified and written before code is written.

With the AMDD approach, software developers do a little bit of modeling and then a lot of coding, iterating back when need to. The design efforts of developers are now spread out between your modeling and coding activities. AMDD enables software developers to think through larger issues before they move down into the implementation details.

Modeling Languages

As object-oriented programming languages, such as Java and C++, are the key for the object-oriented design, modeling languages are essential for the model-based design. In general, modeling languages fall into the following three categories:

1. Unified (or universal) modeling languages, such as Unified Modeling Language (UML) [48] and Modelica [49], are designed with goals similar to programming languages; they optimized to be broad and intend to offer the advantage for adopters to remain in a single language framework independently from the domain and system category they concerned with. Necessarily, the core language constructs are tailored more toward an underlying technology (e.g. object modeling) rather then to a particular domain - even if extension mechanisms such as UML profiling allow some form of customizability.

2. Interchange languages, such as the Hybrid System Interchange Format (HSIF) [50], are designed for sharing models across analysis tools (hybrid system analysis). Interchange languages are optimized for providing specific quantitative analysis capabilities in design flows via facilitating the integration of a group of tools. Accordingly, they are optimized to cover concepts related to an analysis technology.
3. Domain-specific modeling languages (DSMLs) [13] are tailored to the particular concepts, constraints and assumptions of application domains. They are optimized to be focused: the modeling language should offer the simplest possible formulation that is still sufficient for the modeling tasks. Model-based design frameworks that aggressively use DSMLs, need to support the composition of modeling languages. For example, the MIC infrastructure uses abstract syntax metamodeling and meta-programmable tool suites [11] for the rapid construction of DSMLs with well defined syntax and semantics.

The Unified Modeling Language (UML)

The UML is a visual language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system. The UML offers a standard way to write a system's blueprints, including conceptual things such as business processes and system functions as well as concrete things such as programming language statements, database schemas, and reusable software components [48]. The UML represents a collection of best engineering practices that have proven successful in the modeling of

large and complex systems. Using the UML also helps project teams communicate, explore potential designs, and validate the architectural design of the software.

The UML emerges as the result of combining successful modeling constructs from several object-oriented modeling methods, mainly OMT (Object Modeling Technique) [51], Booch [52] and OOSE (Object-Oriented Software Engineering) [53]. In 1997, the UML is standardized by OMG, as UML 1.1 [54]. Since this version, the Object Constraint Language (OCL) [55] has been adopted as the constraints description language for the UML models in general. Today, the UML is widely-used by the industry in developing object oriented software and the software development process.

Currently, UML 2 [56] [57] is the latest version of UML. It builds on the 1.X versions and is a major update on the UML standards. The new version goes well beyond the Classes and Objects well-modeled by UML 1.X to add the capability to represent not only behavioral models, but also architectural models, business process and rules, and other models used in many different parts of computing and even non-computing disciplines. The main new features in UML 2 include:

- Nested Classifiers: In UML 2, modelers can nest a set of classes inside the component that manages them, or embed a behavior (such as a state machine) inside the class or component that implements it. This capability also allows build up complex behaviors from simpler ones, the capability that defines the Interaction Overview Diagram. Different levels of abstraction can be layered in multiple ways.

- Improved Behavioral Modeling: In UML 2, all different behavioral models derive from a fundamental definition of a behavior (except for the Use Case, which is subtly different but still participates in the new organization).
- Improved relationship between Structural and Behavioral Models: UML 2 allows modelers designate that a behavior represented by (for example) a State Machine or Sequence Diagram is the behavior of a class or a component.

Overall, UML 2 defines 13 basic diagram types, divided into two groups, Structural and Behavioral Modeling Diagrams. Structure Modeling Diagrams define the static architecture of a model. They are used to model the 'things' that make up a model - the classes, objects, interfaces and physical components. In addition, they are used to model the relationships and dependencies between elements. Structure Modeling Diagrams include:

- Package Diagrams are used to divide the model into logical containers or 'packages' and describe the interactions between them at a high level.
- Class or Structural Diagrams define the basic building blocks of a model: the types, classes and general materials that are used to construct a full model.
- Object Diagrams show how instances of structural elements are related and used at run-time.
- Composite Structure Diagrams provide a means of layering an element's structure and focusing on inner detail, construction and relationships.
- Component Diagrams are used to model higher level or more complex structures, usually built up from one or more classes, and providing a well defined interface.

- Deployment Diagrams show the physical disposition of significant artifacts within a real-world setting.

Behavioral Modeling Diagrams capture the varieties of interaction and instantaneous state within a model as it 'executes' over time and include:

- Use Case Diagrams are used to model user/system interactions. They define behavior, requirements and constraints in the form of scripts or scenarios.
- Activity Diagrams have a wide number of uses, from defining basic program flow, to capturing the decision points and actions within any generalized process.
- State Machine Diagrams are essential to understanding the instant to instant condition or "run state" of a model when it executes.
- Communication Diagrams show the network and sequence of messages or communications between objects at run-time during a collaboration instance.
- Sequence Diagrams are closely related to Communication Diagrams and show the sequence of messages passed between objects using a vertical timeline.
- Timing Diagrams fuse Sequence and State Diagrams to provide a view of an object's state over time and messages which modify that state.
- Interaction Overview Diagrams fuse Activity and Sequence Diagrams to provide allow interaction fragments to be easily combined with decision points and flows.

Hybrid System Interchange Format (HSIF)

The Hybrid Systems Interchange Format (HSIF) [58] [59] is a XML-based interchange format for hybrid systems mainly developed researchers at Vanderbilt University and University of Pennsylvania as parts of the DARPA MoBIES project. The goal for HSIF is to share models between hybrid system modeling and analysis tools. Tools could either import HSIF models, export HSIF models or both as shown in Figure 2-10.

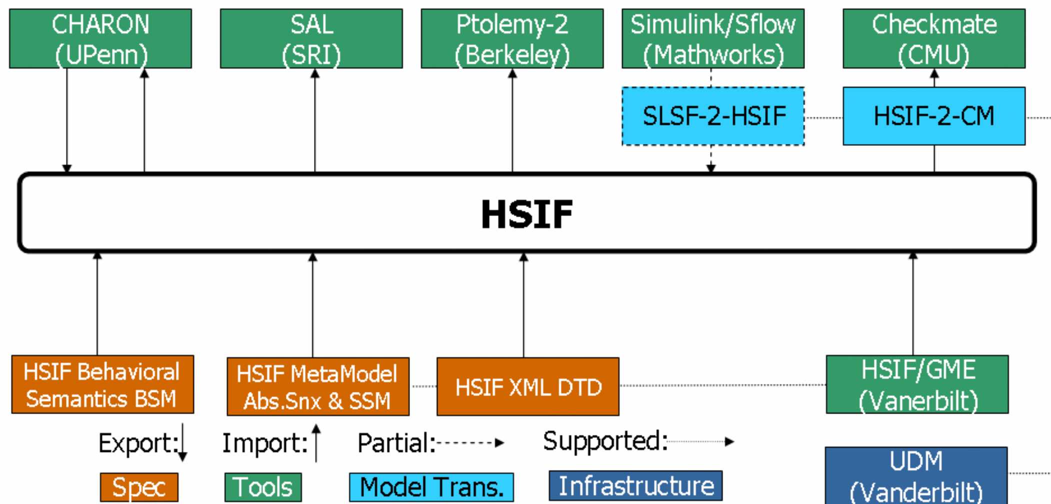

Figure 2-10 Layout of the HSIF tool chain implementation [60]

HSIF models represent a system as a collection of hybrid automata called network. Each hybrid automaton is a finite state machine in which states include constraints on continuous behaviors and transitions describe discrete steps. Automata in a network communicate by means of variables. HSIF supports two kinds of variables: *signals* and *shared variables*. Signals are used to model predictable execution with

synchronous communication between automata. Shared variables are used for asynchronous communication between loosely coupled automata.

HSIF is a powerful interchange format, but it only supports a restrict set of semantics of hybrid systems. For example, it does not support a hierarchical structure of FSMs and prevents "by-construction" zero-time loop among FSMs to eliminate the risk of nondeterministic behavior stemming out of a combination of deterministic subsystem. The limitations to its semantics may deter tool vendors from adopting this interchange format.

Domain-Specific Modeling Language (DSML)

In contrast to a general-purpose modeling language, such as UML, a DSML is a modeling language designed to be useful for a specific task in a fixed problem domain. DSMLs are gaining popularity in the field of software engineering to enhance productivity, maintainability, and reusability of software artifacts, and enable expression and validation of concepts at the level of abstraction of the problem domain.

A well-made DSML captures the modeling concepts, relationships, integrity constraints, and semantics of the application domain and allows modelers to program declaratively through model construction. Domain experts can easily master a DSML, since the domain concepts with which they are already familiar with are incorporated in the modeling language. Unlike UML, both the modeling language and code generators of the DSML are individually tailored to the requirements of each domain. This allows DSML to offer full code generation from design models and produce the efficient code.

DSMLs are essential for MIC applications. They are convenient tools for the design and implementation of embedded software and systems. For example, the Embedded System Modeling Language (ESML) [61] is a DSML developed by the Vanderbilt DARPA MoBIES team for modeling real-time mission computing embedded avionics applications. It is successfully used by Boeing engineers to model component-based avionics applications designed for the Bold-Stroke [62] component deployment and distributed middleware infrastructure. Figure 2-11 presents an example that models Bold Stroke multi-threaded component interaction in ESML. The underlying model interpreters for the ESML will generate XML files that are used during load-time configuration of Bold Stroke, and enable invasively modifying a very large code base from properties specified in an ESML model.

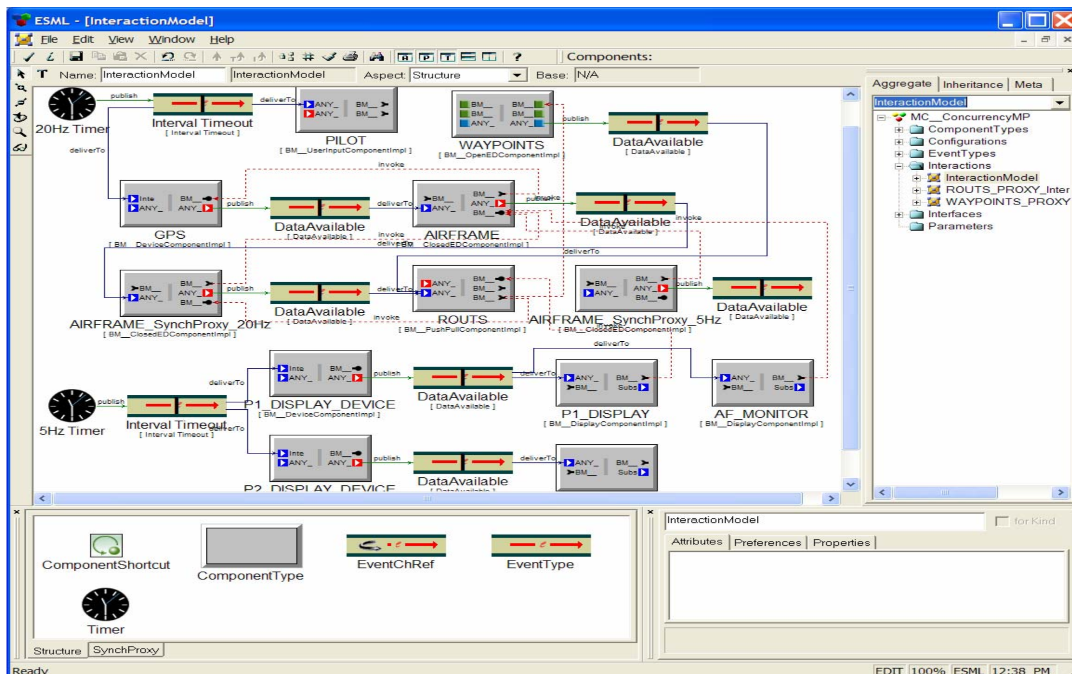

Figure 2-11 Bold Stroke multi-threaded component interaction in ESML

Formal Methods

Formal methods [63] [64] are mathematically based techniques for describing system properties. In most cases, formal methods provide frameworks within which users can specify, develop, and verify systems in a systematic manner. A method is called formal if it has a sound mathematical basis, typically given by a formal specification language. This mathematical basis provides the means of precisely defining notions like consistency, completeness, specification, implementation, and correctness. Formal methods are used to reveal ambiguity, incompleteness, and inconsistency in a system. When used early in the system development process, they can reveal design flaws that otherwise might be discovered only during costly testing and debugging phases. When used later, they can help determine the correctness of a system implementation and the equivalence of different implementations. In recent years, formal methods are widely applied in specifying formal semantics for modeling languages.

This section first introduces widely-used semantics specification approaches. Then, formal specification languages in general and a particular formal specification language, Abstract State Machine, are presented. The last part is a short survey on recent research in formal semantics for modeling languages.

Semantic Specification Methods

Formal semantics [65] is concerned with rigorously specifying the meaning, or behavior, of programs, pieces of hardware etc. The actual semantics specification can be formalized in different ways. Three main formal semantics specification approaches are *operational semantics*, *denotational semantics* and *axiomatic semantics*.

Operational semantics: Specify a set of rules on how the state of an actual or hypothetical computer changes while executing a program. The overall state is typically divided into a number of components, e.g. stack, heap, registers etc. Each rule specifies certain preconditions on the contents of some components and their new contents after the application of the rule. It is similar in spirit to the notion of a Turing machine, in which actions are precisely described in a mathematical way. Typically, the definition of operational semantics of a system has two major steps: the choice or the definition of the virtual machine, the translation of the system into the source code of the machine.

Denotational semantics [66]: Describe the meaning of programs in terms of mathematical functions on programs and program components. Programs are translated into mathematical functions about which properties can be proved using the standard mathematical theory of functions, and especially domain theory. Denotational semantics was developed by Christopher Strachey in mid 1960s. Dana Scott supplied the mathematical foundations in 1969. The idea of denotational semantics is to associate an appropriate mathematical object, such as a number, a tuple, or a function, with each *phrase* of the program. The *phrase* is said to denote the mathematical object, and the object is called the *denotation* of the phrase. Both the denotational semantics and the operational semantics use the same basic idea that defines the semantics via mapping to some mathematical basis. However, while the operational semantics is done at high level of abstraction, usually by means of virtual machines, denotational semantics uses rigorous mathematical objects.

Axiomatic semantics [67] [68]: Specify a set of assertions about properties of a system and how they are affected by program execution. The axiomatic semantics of a

program could include pre- and post-conditions for operations. In particular if you view the program as a state transformer (or collection of state transformers), the axiomatic semantics is a set of invariants on the state which the state transformer satisfies. Based on methods of logical deduction from predicate logic, axiomatic semantics are more abstract than denotational semantics in that there is no concept corresponding to the state of the machine. Rather, the semantic meaning of a program is based on assertions about the relationships that remain the same each time the program executes. The relation between an initial assertion and a final assertion following a piece of code captures the essence of the semantics of the code. Another piece of code that defines the algorithm slightly different yet produces the same final assertion will be semantically equivalent provided any initial assertions are also the same. The proofs that assertions are true do not rely on any particular architecture for the underlying machine; rather they depend on the relationships between the values of the variables. Although individual values of variables change as a program executes, certain relationships among them remain the same. These invariant relationships form the assertions that express the semantics of the program.

Formal Specification Languages

A formal specification language [69] is a language having a well defined syntax and semantics which is suitable for describing or specifying systems of some kind. Unlike most programming languages, which may also have precise syntax and semantics definition and are used to implement a system, specification languages are used during system analysis, requirements analysis and design. Formal specification languages describe the system at a much higher level than a programming language. Indeed, it is

considered as inappropriate if a requirement specification is cluttered with unnecessary implementation detail.

In most cases, specifications must be subject to a process of refinement before they can actually be implemented. The result of such a refinement process is an executable algorithm, which is either formulated in a programming language, or in an executable subset of the specification language at hand. An important use of formal specification languages is enabling the creation of proofs of program correctness. In general, formal specification languages can be categorized as model-oriented specification language, algebraic specification language, and process modeling language.

Model-oriented specification languages [75] specify a system in terms of a state model that is constructed using mathematical objects such as sets and sequences. An *operation* is a function which maps a value of the state together with values of parameters to the operation onto a new state value. A model-oriented specification language is typically used by describing in detail specific mathematical objects which are structurally similar to the corresponding software. It is then permitted to transform these mathematical objects, during the design and implementation of the system, in ways which preserve the essential features of the requirements as initially specified. Popular model-oriented specification languages include Abstract State Machine (ASM) [70], VDM-SL [71], the specification language associated with VDM [72], the Z language [73] and the B language [74].

Algebraic specification languages [75] apply methods derived from abstract algebra or category theory to specify systems. Abstract algebra is the mathematical study of certain kinds or aspects of structure abstracted away from other features of the objects

under study. Algebraic methods are beneficial in permitting key features of systems to be described without prejudicing questions which are intended to be settled later in the development process. While model-oriented specification languages are assumed to be suited better for the description of state based systems (abstract machines), algebraic specification languages are assumed to be better for abstract data type specifications. Popular algebraic specification languages include the OBJ language [76], the Larch Shared Language (LSL) [77], and ACT ONE [78].

Process modeling languages describe concurrent systems. In these languages, expressions denote processes, and are built up from elementary expressions which describe particularly simple processes by operations which combine processes to yield new potentially more complex processes. Popular process modeling languages contain Communicating Sequential Processes (CSP) [79] and Calculus of Communicating Systems (CCS) [80].

Abstract State Machine

Abstract State Machine (ASM), formerly called Evolving Algebras, is a model-oriented specification language. It was first introduced by Yuri Gurevich in [81]. Supporting materials on ASM can be found on ASM community web site [82], in a series of ASM introductory publications [83] [84] [85] [86], or the ASM book [87].

ASM is mathematical machine that models a system or simulates its execution by operating on the global state of the system according to a set of transition rules. Like Turing machine [88], ASM gives operational semantics to algorithms. Turing machine simulation may be very clumsy. In particular, one step of the algorithm may require a

long sequence of steps of the simulating Turing machine. With ASM, the machine makes only a bounded number of steps to simulate one step of the given algorithm. Furthermore, an ASM may be tailored to an arbitrary abstraction level. The ASM thesis [83] claims that every sequential algorithm can be simulated by an ASM on a natural abstraction level.

ASM has been successfully applied to specifying the semantics of many programming languages, including C [89], C++ [90], Java [91] [92] [93] [94], Oberon [95], Smalltalk [96], Prolog [97], Occam [98] and etc. ASM has also been used for describing the semantics of IEEE hardware description language VHDL [99] [100], as well as distributed systems [101] [102], real-time systems [103] [104] and hardware architectures [105] [106]. In particular, the International Telecommunication Union adopted an ASM-based formal semantics definition of SDL as part of SDL language definition [107] [108].

Basic Abstract State Machines

A basic ASM consists of a basic ASM program together with a collection of states and sub-collection of initial states. Basic ASMs are sequential algorithms. Intuitively sequential algorithms are non-distributed algorithm with uniformly bounded parallelism. The latter means that the number of actions performed in parallel are bounded independently of the state or the input. It has been proved that, for every sequential algorithm, there is a basic ASM that simulates the algorithm step by step [109].

States

The notion of ASM state is equivalent to the notion of first-order structure in mathematical logic. A *vocabulary* is a finite collection of *function names*; each name has

a fixed arity. Some names may be marked as *relation names* or *static names*, or both. Relational names may also be called *predicates*. Predicates are special kinds of functions that return Boolean values, and may be used to express all kinds of constraints. Every vocabulary contains the following static names: *true*, *false*, *undef*, the equality sign, and the names of the usual Boolean operations. According to whether or not the values of a function may change, functions are *static* or *dynamic*. Static functions give the basic structure of the system, while the dynamic functions explicitly reflect the dynamics of the system.

A *state A* of a given vocabulary is a nonempty set *X*, together with interpretations of the function names (the basic functions of *A*) and the predicates (the basic relations of *A*). The set *X* is called the base set of *A*. A function name of arity *j* is interpreted as a *j-ary* operation over *X*. A nullary function name is interpreted as an element of *X*.

Basic relations are seen as special basic functions whose only possible values are *true* and *false* and whose default value is *false* rather than *undef*. The value *undef* is the default value for basic functions. Formally a basic function *f* is total but, actually, it is partial in ASM. The intended domain of *f* consists of all tuples *a* with $f(a) \neq undef$.

Many algorithms require additional space as they run. In the abstract setting, this seems to mean that the state acquires new elements. It is more convenient to have a source of new elements inside the state. In ASM, Every state *A* includes an infinite set called the *reserve* of *A*. The *reserve* of *A* contains of all elements of *A* such that

- Every basic relation, with the exception of equality, evaluates to *false* if at least one of its arguments belongs to the reserve.

- Every non-relational basic function evaluates to *undef* if at least one of its arguments belongs to the reserve.
- No basic function outputs an element of the reserve.

The definition of an ASM state is very general. Any kind of static mathematical reality can be described as a first-order structure. In fact, second-order and higher-order structures of logic are special first-order structures. Many-sorted first-order structures (with several base sets called sorts) are special one-sorted structures. The roles of sorts are played by designated unary relations that are called *universe* in ASM.

Updates

A state can be viewed as a kind of memory that maps locations to values. Dynamic functions are those that can change during computation. A *location* of a state A is a pair $l = (f,(x_1,\dots,x_j))$ where f is a *j-ary* dynamic function name in the vocabulary of A and $(x_1,\dots,x_j)$ is a j-tuple of elements of A. The element $y = f(x_1,\dots,x_j)$ is the *content* of that location.

An *update* of A is a pair (l, y'), where l is a location $(f,(x_1,\dots,x_j))$ of A and y' is an element of A. y' must be *true* or *false* if f is a predicate. To fire the update (l, y'), replace the old value $y = f(x_1,\dots,x_j)$ at location l with the new value y' so that $f(x_1,\dots,x_j) = y'$ in the new state.

An *update set* over a state A is simply a set of updates of A. A update set $S = \{(l_1, y_1'),\cdots,(l_n, y_n')\}$ is *consistent* if the location are distinct. In other words, S is *inconsistent* if there are i, j such that $l_i = l_j$ but y_i' is distinct from y_j'. To fire a consistent

update set, fire all the updates simultaneously; to fire an inconsistent update set, so nothing.

Rules and Programs

Expressions are defined recursively, as in first-order logic:

- A variable is an expression.
- If f is an *j-ary* function name and $e_1, \ldots, e_j$ are terms, then $f(e_1, \ldots, e_j)$ is an expression.

Ground expressions are expressions without variables. *Atomic Boolean expressions* are expressions of the form $f(\bar{e})$, where f is a relation name. *Boolean expressions* are built from atomic Boolean terms by means of the Boolean operations.

An *update rule R* has the form:

$$f(e_1, \cdots, e_j) := e_0$$

where f is a *j-ary* dynamic function name and each e_i is an expression. To execute R, fire the update (l, a_0) where $l = (f, (a_1, \ldots, a_j))$ and each a_i is the value of e_i.

The *skip rule* has the form:

skip

A *conditional rule R* has the form:

if e **then** R_1
else R_2
endif

where e is a Boolean expression and R_1, R_2 are rules. To execute R, evaluate the guard e. If e is *true*, then execute R_1, otherwise execute R_2.

A *do-in-parallel* rule R has the form:

do in-parallel

```
    R1
    R2
enddo
```

where R_1, R_2 are rules. To execute R, execute rules R_1, R_2 simultaneously.

An *import rule* R has the form:

```
import x
   R0(x)
endimport
```

where x is a variable and R_0 is a rule. To execute R, fish out any element x of the reserve and execute the rule $R_0(x)$.

A basic ASM program is just a rule. An ASM is give by a program, a collection of legal states and a subcollection of initial states. The program describes one step of the ASM. An ASM is supposed to execute until the state does not change.

Parallel Abstract State Machines

Parallel ASMs are supported by enriching basic rules with first-order expressions and introducing do-in-parallel construct.

The expression $\{t(x) \mid x \in s \textbf{ where } \varphi(x)\}$ denotes the set of all values $t(x)$ where x ranges over those elements of set s that satisfy $\varphi(x)$. This presumes that s is a set expression and $\varphi(x)$ is Boolean. Every state A is required to be closed under tuples and finite sets: if $a_1, \ldots, a_n$ are elements of A then the tuple $(a_1, \ldots, a_n)$ and the set $\{a_1, \ldots, a_n\}$ are elements of A. A also contains standard operations over tuples and sets.

A *do-forall rule* R has the form:

```
do forall x ∈ s
   R0(x)
enddo
```

where x is a variable and $R_0(x)$ is a rule. To execute R, execute all subrules $R_0(x)$ with x in s at once.

Parallel ASMs are parallel algorithm. It is proved in [110] that, for every parallel algorithm, there is a parallel ASM that simulates the given algorithm step by step.

Nondeterministic Abstract State Machines

Basic and parallel ASMs can be made nondeterministic by the use of nondeterministic choose rules.

A *choose rule R* has the form:

choose $x \in s$
    $R_0(x)$
endchoose

where $R_0(x)$ is a rule and x does not occur freely in the set expression s. To execute R, choose any element x of s and execute the subrule $R_0(x)$.

Formal Semantics for Modeling Languages

Many modeling language standards do not contain a formal semantics definition, and only have the syntax definition of the language. This is not because the syntactic notations for modeling languages are more important than their semantics. Instead, a formal semantics is one of the main goals for many modeling languages. For example, the 4th goal of seven design goals for UML is to “Provide a formal basis for understanding the modeling language” [111]. But, normally, the semantics definition is a much harder problem than the syntax definition. Instead of delaying a language standard because of the lack of a formal semantics, natural languages, such as English, are used to informally explain the semantics of a modeling language, and the formal semantics

definition is left to the research community. Researchers will propose different approaches for the formal semantics definition of the language. This process may take several years or even longer until a formal semantics of this modeling language is mature and is accepted by the language's standardization committee. Then, this formal semantics will be adopted as a part of the language standard. This section presents related researches in the formal semantics definition for two widely-used modeling languages: SDL and UML.

Formal Semantics for SDL

SDL (Specification and Description Language) [131] is a modeling language for specifying the behavior of distributed real-time systems in general and telecommunication systems in particular. SDL was first standardized by ITU (Telecommunication Sector of the International Telecommunication Union) in 1976, and after that, upgrades of the SDL standard are officially released every 4 years. Now, SDL has matured from a simple visual language for describing a set of communicating finite state machines to a sophisticated modeling technique with graphical syntax, data type constructs, structuring mechanisms, objected-oriented features, support for reuse, companion notations, tool environments and a formal semantics. It took more than 10 years of language development until the semantics of SDL became defined formally in 1988. This formal semantics was based on a combination of the VDM meta-language Meta-IV and a CSP-like communication mechanism. It has been maintained and extended for subsequent versions of SDL.

In November 1999, a new version of SDL called SDL-2000 was passed by ITU. SDL-2000 incorporates several new features, including object-oriented data types, a

unified agent concepts, hierarchical states, and exception handling. Based on the assessment that the existing Meta-IV program would be too difficult to extend and maintain, it was decided to conceive a new formal semantics for SDL-2000 from scratch. For this purpose, the SDL semantics group [132] was formed in 1998. The main design objectives for SDL semantics include intelligibility, maintainability, expressiveness and executability. After widely discussion and investigation, ASM was finally selected as the underlying formalism for defining SDL semantics.

In November 2000, the formal semantics of SDL-2000 (about 350 pages), referred to as the *ITU-T approach*, was officially approved to become part of the SDL standard [108]. In ITU-T approach, the static semantics covers well-formedness rules and transformations from non-basic language constructs to the core SDL concepts. The dynamic semantics is given to syntactically correct SDL specifications satisfying the well-formed rules, after all transformations have been applied. For a given SDL specification, it defines the corresponding set of computations. The dynamic semantics builds on a so-called SDL Abstract Machine, which is defined using ASM. Next, the transitions of the SDL specification are compiled into code executable on this machine. Finally, a distributed operating system, which initializes and executes the agents of the SDL system, is defined. It is now official policy that if there is an inconsistency between the main body of SDL standard and the ASM-based semantics, then neither the main body of SDL standard nor ASM-based semantics takes precedence when this is corrected.

In the same period, there have been a variety of approaches on formalizing the SDL semantics using various methods. For example, Bozga et al. [133] defined the SDL intermediate representation format IF as the basis for a systematic integration of the

ObjectGeode toolset with different validation tools supporting formal verification and automatic test case generation. In [134], Fischer and Dimitrov proposed a SDL Time Nets model, which is an extended Petri Nets, as a formal basis for verifying SDL protocol specifications. In [135], Bergstra and Middleburg defined a process algebra semantics for a restricted version of SDL. Broy [136] modeled various subsets of basic SDL using stream processing functions of FOCUS [137]. Lau and Prinz [138] proposed an Object-Z based model to define a universal core for SDL as a conceptual framework for dealing with the main building blocks of the SDL language. In [139] [140], Glässer and Karges defined the dynamic semantics of Basic SDL-92/96 in ASM. This work provides a conceptual framework which has further been developed and extended by combing it with the compilation-based view of [141] as well as fundamental concepts from [138] resulting in the final formal semantics of SDL-2000.

Formal Semantics for UML

Until now the goal to provide a formal semantics of UML is reached in a very limited way. The UML standard does not contain a formal semantics. English is used by the UML standard to informally describe the semantics. The UML standard includes a family of structural and behavioral modeling diagrams as described in section 3.1. Many researchers have proposed formal semantics for individual diagrams of the UML – e.g. [112] [113] on UML class diagram, [114] [115] on state machines, [116] [117] on collaboration diagrams, [118] [119] on use cases, [120] [121] on activity diagrams. In general, these approaches to formalize UML semantics can be divided into two main groups: set-valued semantics and translational approach.

In [122], M. Richters and M. Gogolla proposed to define the semantics of UML class diagrams and OCL based on set theory. In this approach, the semantics of a class diagram is described as a set of hypergraphs. Each hypergraph corresponds to a concrete configuration of object instances. The nodes correspond to objects and the edges to association links. The nodes are labeled with the attribute and value mappings, while edges are labeled with names for association ends. This approach directly provides set-valued semantics for UML models and fits well for the definition of OCL constraints within UML models.

The translational approach defines translation rules from UML diagrams to traditional formal specification languages. There exist, among others, translations to Z [123], Object-Z [124], B [125], Larch [126], CASL [127], π-calculus [120], X-machines [118], PVS [128], CSP [129], Petri Nets [130] and ASM [114]. The advantage of this approach is that tools that have already existed for the formal specification languages can be used for reasoning UML models after translation.

CHAPTER III

A SEMANTIC ANCHORING INFRASTRUCTURE

The proposed semantic anchoring infrastructure includes a formal metamodeling framework and a finite set of semantic units. This chapter presents the details of the semantic anchoring methodology and the tool suite for domain-specific modeling language (DSML) design through semantic anchoring.

Formal DSML Specification

Formally, a DSML is a five-tuple of concrete syntax (C), abstract syntax (A), semantic domain (S) and semantic and syntactic mappings (M_S, and M_C) [146]:

$$L = \{C, A, S, M_S, M_C\}$$

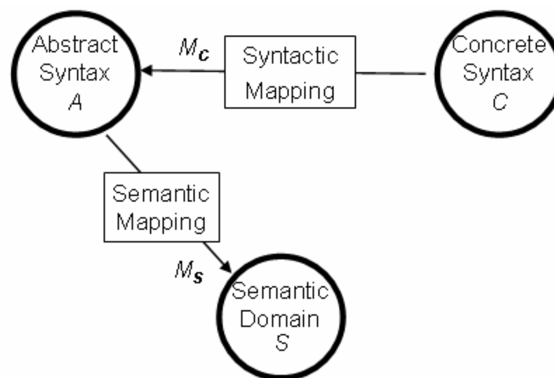

Figure 3-1 Formal DSML specification

As shown in Figure 3-1, the concrete syntax C defines the specific notation used to express models, which may be graphical, textual or mixed. The abstract syntax A

defines the modeling concepts, relationships between these modeling concepts, and well-formedness rules in the language. The semantic domain *S* is usually defined in a formal framework in terms of which the meaning of the models is explained. The syntactic mapping $M_C: C \rightarrow A$ assigns modeling notations (graphical, textual or both) to the modeling concepts defined in the abstract syntax.

The syntax and semantics are the fundamental components of a DSML. The syntax of a DSML provides the modeling constructs that conceptually form an interface to the semantic domain, while the semantics of a DSML gives the meaning behind well-formed domain models. For example, in MIC applications, the semantics of a domain model often prescribes the behavior that simulates a computer-based system.

DSML semantics are defined in two parts: a semantic domain *S* and a semantic mapping $M_S: A \rightarrow S$. The semantic domain S is usually defined in a formal, mathematical framework, in terms of which the meaning of the models is explained. The semantic mapping relates syntactic concepts to those of the semantic domain. In DSML applications, semantics may be either structural or behavioral. The *structural semantics* defines the correct structure of model instances: the abstract syntax defines the set of all correct domain models that satisfy the well-formedness rules. Accordingly, the semantic domain for structural semantics is a set-valued semantics. The *behavioral semantics* may describe the evolution of the state of the modeled artifact along with some time model. Hence, the behavioral semantics is formally captured by a mathematical framework representing the appropriate form of dynamics.

Semantic Anchoring Methodology

Although DSMLs use many different notations, modeling concepts and model structuring principles for accommodating needs of domains and user communities, semantic domains for expressing basic behavior categories are more limited. A broad category of component behaviors can be represented by basic behavioral abstractions, such as Finite State Machine, Timed Automaton, Continuous Dynamics and Hybrid Automaton. This observation led us to the following strategy in defining behavioral semantics for DSMLs:

1. Definition of a set of modeling languages $\{L_i\}$ for capturing semantics of the basic behavioral abstractions and development of the precise specifications for all components of $L_i = < C_i, A_i, S_i, M_{Si}, M_{Ci} >$. We use the term "semantic units" to describe these basic modeling languages.
2. Definition of the behavioral semantics of an arbitrary DSML, $L = < C, A, S, M_S, M_C >$, is accomplished by specifying the M_A: $A \rightarrow A_i$ mapping to a predefined semantic unit L_i. The M_S: $A \rightarrow S$ semantic mapping of L is then defined by the composition $M_S = M_{Si} \circ M_A$, which indicates that the semantics of L is anchored to the S_i semantic domain of the L_i modeling language.

To support the specification of the transformational rules between the abstract syntax of a DSML and the abstract syntax of the semantic unit, a formal metamodeling framework is adopted to support the abstract syntax specification by using MOF-based metamodels [35] and the transformational rule specification by using a formal metamodel transformation language. Semantic units are predefined to capture the behavioral semantics of a finite set of basic Models of Computations (MoCs), such as Finite State

Machine, Timed Automata, Discrete Event System and Synchronous Dataflow. In addition, to integrate semantic units with the formal metamodeling framework, a simple modeling language needs to be developed for each semantic unit. These simple modeling languages act as metamodeling interface of the semantic units.

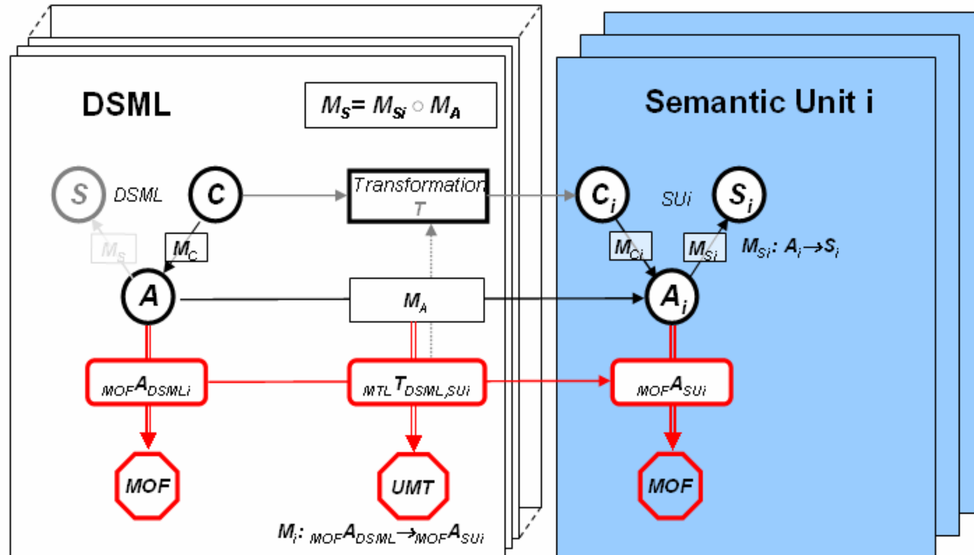

Figure 3-2 The semantic anchoring infrastructure

The semantic anchoring infrastructure, shown in Figure 3-2, includes a finite set of semantic units and a formal MOF-based metamodeling framework. With this semantic anchoring infrastructure, a formal DSML specification can be accomplished with three steps.

1. Specify the DSML syntax $< A, C, M_C >$ by using MOF-based metamodels.
2. Select appropriate semantic units $L_i =< A_i, C_i, M_{Ci}, S_i, M_{Si} >$ that can capture the behavioral aspects of the DSML. Note that a DSML may have multiple behavior aspects. Hence, multiple semantic units may be selected in this step.

3. Specify the semantic anchoring rules $M_A = A \rightarrow A_i$ from the metamodel of the DSML to the metamodel of the selected semantic units by using a formal metamodeling transformation language.

Semantic Anchoring Tool Suite

By integrating a set of existing tools, we develop a semantic anchoring tool suite, as shown in Figure 3-3, to support DSML design using the semantic anchoring methodology. The GME tool suit [11] is used for defining the abstract syntax, A, for a DSML, $L = < C, A, S, M_S, M_C >$, using UML Class Diagrams [56] and OCL [55] as meta-language. The $L_i =< A_i, C_i, M_{Ci}, S_i, M_{Si} >$ semantic unit is specified as an AsmL [147] specification in terms of (a) an AsmL Abstract Data Model (which corresponds to the A_i, abstract syntax specification of the modeling language defining the semantic unit in the AsmL framework), (b) the S_i, semantic domain (which is implicitly defined by the ASM mathematical framework), and (c) the M_{Si}, semantic mapping, defined as a model interpreter written in AsmL.

The M_A: $A \rightarrow A_i$ semantic anchoring of L to L_i is defined as a model transformation using the GReAT tool suite [16]. The abstract syntax A and A_i are expressed as metamodels. Connection between the GME-based metamodeling environment and the AsmL environment is provided by a XML-based syntax conversion. Since the GReAT tool suit generates a model transformation engine from the meta-level specification of the model transformation, any legal domain model defined in the DSML can be directly transformed into a corresponding AsmL data model and can be simulated by using the

AsmL native simulator. In the following, we give explanation of our methodology and the involved tools.

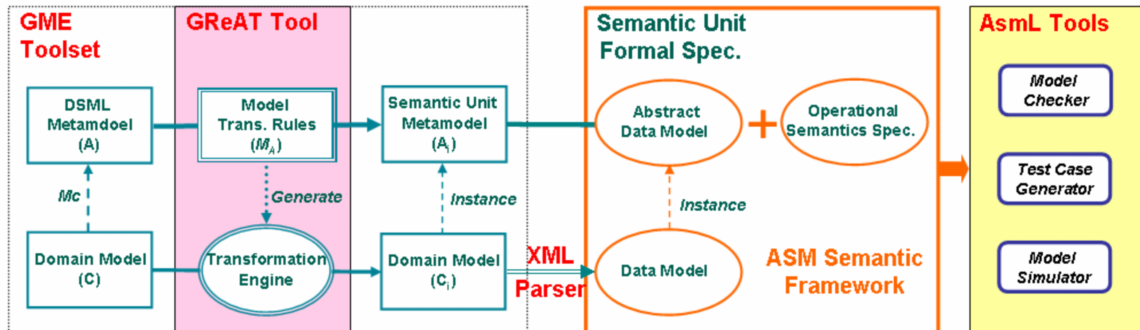

Figure 3-3 The semantic anchoring tool suite

Abstract Syntax Modeling

In the standard OMG four-layer metamodeling architecture [56], a *metamodel* is used to define the abstract syntax of a DSML. A metamodel defines the modeling concepts, their relationships, and any well-formedness rules. Metamodels may be constructed using one of several different metamodeling languages, including the Meta Object Facility (MOF) [35] or UML Class Diagrams [56]. The Object Constraint Language (OCL) [55] is often used to define well-formedness rules.

A DSML's concrete syntax determines how domain models are represented in a domain specific modeling environment. The visualizations are related to the elements of the abstract syntax through the syntactic mapping. The mechanism for accomplishing the syntactic mapping is usually tool-dependent, because the type of representation used depends on the capabilities of the modeling tool which supports the DSML.

The Generic Modeling Environment (GME) is a meta-programmable tool that supports the OMG four-layer metamodeling architecture. GME allows users both to design and to model with domain-specific modeling environments. The GME modeling environments may be created by using GME itself through a process of metamodeling. The GME metamodel is based on UML class diagrams and OCL, and has recently been extended [148] to handle the OMG's MOF standard.

Set-Valued Structural Semantics for Metamodels

The meaning of a DSML metamodel is defined by the set-valued structural semantics: a DSML metamodel defines a set space that contains all well-formed models in the domain. The basic elements in a metamodel are modeling concepts defined by UML classes. Semantically, each modeling concept is a set containing infinite number of instances (modeling objects) of the modeling concept. The relationships among modeling concepts describe the mathematical relationships that modeling objects in a domain model should satisfy. Since the capability for UML class diagrams to express relationships among modeling concepts is limited, OCL constraints are used to specify additional well-formedness rules, which can also be expressed as equivalent mathematical relationships that modeling objects need to satisfy. Hence, a metamodel uses modeling concepts to construct an initial set space. Then, the relationships and well-formedness rules are added to reset the boundary of this set space. Every well-formed domain model is an element in the set space defined in the DSML metamodel.

Figure 3-4 presents a metamodel for a simple Automaton Language, which includes modeling concepts: *State*, *Intial* (*State*) and *Transition* (edge from a source state

to a destination state). This metamodel is used as an example to explain the set-valued structural semantics for metamodels. Semantically, it defines a set space which contains all correct Automata models. This set can be mathematical expressed as

$$AM = \{\mathtt{m} \in State \times Transition \times Initial \mid \mathtt{m[2]} \subseteq \mathtt{m[1]} \times \mathtt{m[1]} \wedge \mathtt{m[3]} \subseteq \mathtt{m[1]}\},$$

where *State*, *Transition* and *Initial* are three sets defined by the UML classes *State*, *Transition* and *Initial* respectively.

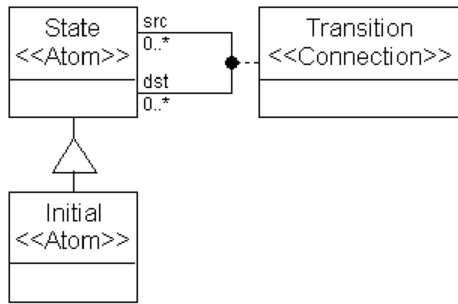

Figure 3-4 Metamodel for a simple Automaton Language

The UML class diagram itself is not able to specify the constraint: if an automaton model has *State* objects, it should have exactly one *Initial* object. Hence, an OCL constraint,

```
Self.parts(State)→size > 0 implies Self.parts(Initial)→size == 1,
```

is added to assert this relationship. This OCL constraint mathematically means adding an additional restriction to set space defined by the UML class diagram. Now, the set *AM* is redefined as

$$AM = \{\mathtt{m} \in State \times Transition \times Initial \mid \mathtt{m[2]} \subseteq \mathtt{m[1]} \times \mathtt{m[1]} \wedge \mathtt{m[3]} \subseteq \mathtt{m[1]} \wedge \mathtt{m[1].size>0} \Rightarrow \mathtt{m[3].size==1}\}.$$

A Formal Framework for Specifying Semantic Units

Semantic anchoring requires the specification of semantic units in a formal framework using a formal language, which is not only precise but also manipulable. The formal framework must be general enough to represent all three components of the M_S: $A \rightarrow S$ specification; the abstract syntax, A, with set-valued structural semantics, the S semantic domain to represent the dynamic behavior and the mapping between them.

Different DSMLs may have varied best-fit formal specification languages to specify their behavioral semantics. However, if a single formal specification language is used to specify behavioral semantics of multiple DSMLs, it becomes easier to support the comparison and composition of multiple DSMLs. Of course, it is also easier for language designers to familiarize themselves with a single formal specification language than with many. Hence, we decided to select a single, easy-to-understand language for DSML behavioral semantics specification. This language must have the following features:

- To avoid ambiguity, it must be mathematically precise.
- It must be sufficiently rich and flexible enough to cover a wide variety of application domains.
- It must support appropriate abstraction mechanisms to increase scalability. Designers should not be forced to over-specify in order to avoid syntactic errors. Specifying the semantics of a DSML should be a process of refinement from higher levels of abstraction to lower, more detailed levels.
- It should use well-known notations and be easy to read and write to avoid cognitive formalization overhead.

- Since we are focused on behavioral semantics, the language should be executable with supporting tools for analysis and verification.

Abstract State Machines (ASM) meets all these requirements and is finally selected as the formal specification language for specifying semantic units. ASM, formerly called Evolving Algebras [84], is a general, flexible and executable modeling structure with well-defined semantics. General forms of behavioral semantics can be encoded as (and simulated by) an abstract state machine [87]. ASM is able to cover a wide variety of domains: sequential, parallel, and distributed systems, abstract-time and real-time systems, and finite- and infinite-state domains. ASM has been successfully used to specify the semantics of numerous languages, such as C [89], C++ [90], Java [91], Prolog [97] and VHDL [99]. In particular, the International Telecommunication Union adopted an ASM-based formal semantics definition of SDL as part of the SDL language standard [108].

The Abstract State Machine Language, AsmL [147], developed by Microsoft Research, provides an industrial-strength language to program ASM. AsmL specifications look like pseudo-code operating on abstract data structures. As such, they are straightforward for programmers to read and understand. A set of tools is also provided to support the compilation, simulation, test case generation and model checking for AsmL specifications. It needs to be mentioned here that the plentiful supporting tools for AsmL specifications was an important reason for us to select AsmL over other formal specification languages, such as Z [73], *tagged signal model* [149] and Reactive Modules [150].

A Formal Framework for Model Transformation

We use model transformation techniques as a formal approach for specifying the M_A: $A \rightarrow A_i$ mapping between the abstract syntax of a DSML and the abstract syntax of a semantic unit. Based on our discussion above, the A abstract syntax of the DSML is defined as a metamodel using UML class diagrams and OCL, and the A_i abstract syntax of a semantic unit is as an Abstract Data Model expressed using AsmL data structure. However, specification of the M_A transformation between the two abstract syntax specifications requires using the same language. In our tool architecture, this common language is the abstract syntax metamodeling language (UML class diagrams and OCL), since the GReAT tool suite is based on this formalism.

This choice requires building a UML/OCL–based metamodeling interface for the Abstract Data Model used in the AsmL specification of a semantic unit. One possible solution is to define a metamodel that captures the abstract syntax of the generic AsmL data structures. The other solution is to construct a metamodel that captures only the exact syntax of the AsmL Abstract Data Model of a particular semantic unit. Each solution has its own advantages and disadvantages. In the first solution, different semantic units can share the same metamodel and the same AsmL translator can be used to generate the data model in the native AsmL syntax. The disadvantage is that the model transformation rules and the AsmL specification translator are more complicated. Figure 3-5 shows a simplified version of the metamodel of generic AsmL data structures as it appears in the GME metamodeling environment. In the second solution, a new metamodel is needed to be constructed for different semantic units, but the transformation rules are simpler and more understandable. Since the metamodel

construction is relatively easier compared with the specification of model transformation rules, we used the second solution to construct metamodels for the AsmL Abstract Data Models of semantic units. We will present metamodel examples using this approach in Chapter 4 (Figure 4-4).

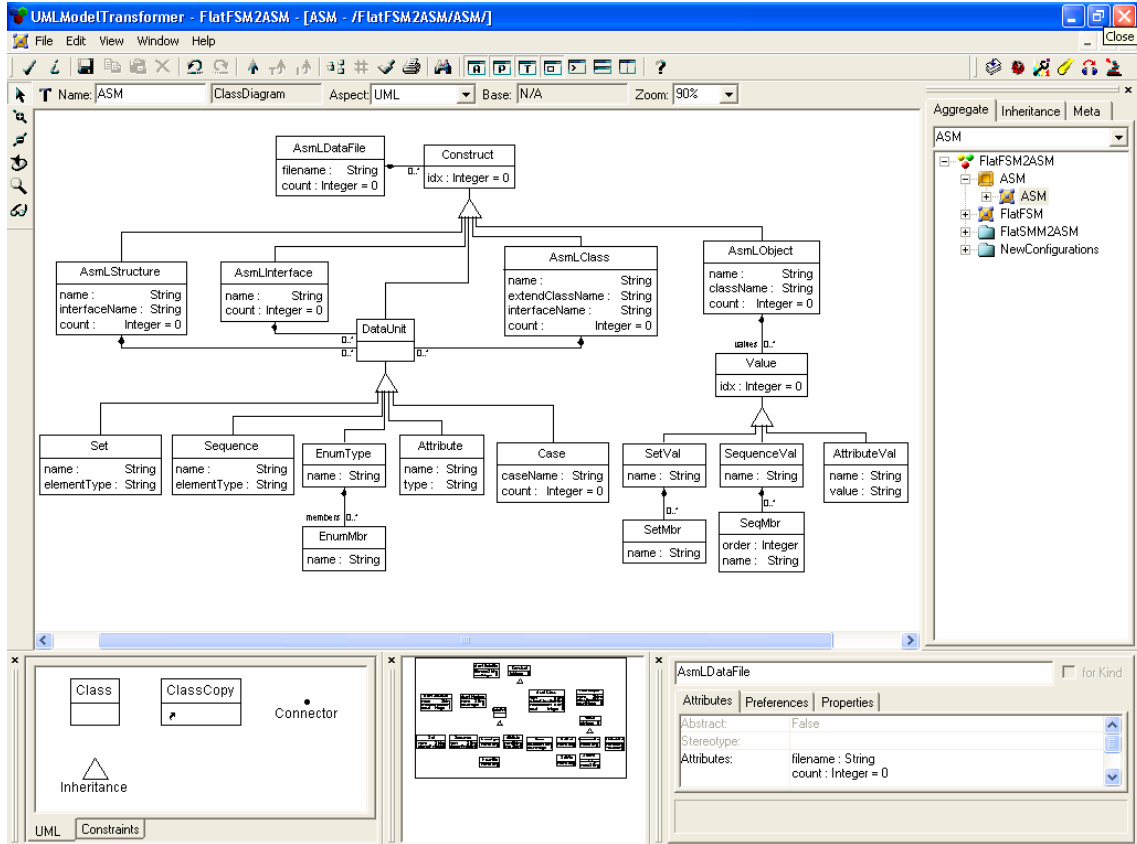

Figure 3-5 Metamodel for a set of AsmL Data Structures

The M_A: $A \rightarrow A_i$ semantic anchoring is specified by using the Unified Model Transformation (UMT) language of the GReAT tool suite. UMT itself is a DSML and the transformation M_A can be specified graphically using the GME tool. The GReAT tool uses GME and allows users to specify model-to-model transformation algorithms as

graph transformation rules between metamodels. The transformation rules between the source and the target metamodels form the semantic anchoring specifications of a DSML. The GReAT engine can execute these transformation rules and transform any allowed domain model to an AsmL model that is then stored in an XML format. Then the AsmL specification translator parses the XML file, performs specification rewriting and generates data model in the native AsmL syntax. Note that UMT provides designers with certain modeling constructs (e.g. "any match") to specify non-deterministic graph transformation algorithms. However, we can always achieve a unique semantic anchoring result by using only the UMT modeling constructs that do not cause the non-determinism.

In summary, semantic anchoring methodology specifies semantics of a DSML by the operational semantics of selected semantic units (defined in AsmL) and by the transformation rules (defined in UMT). The integrated semantic anchoring tool suite ensures that domain models defined in a DSML are simulated according to their "reference semantics" by automatically transforming them into AsmL data models using the transformation rules.

The Abstract State Machine Language

To actually program ASMs in industrial environments, an industrial-strength language is needed. AsmL (ASM Language) is such a language developed in Microsoft Research. A detailed introduction to AsmL is beyond the scope of this thesis. Here we focus on those aspects of AsmL that are most important for the general understanding and that are actually used in this thesis. For an in-depth understanding of AsmL, readers are referred to [147].

Types

Some ASM universes give rise to *types* in AsmL. Other universes are represented as finite sets. An AsmL model may first declare an abstract type *C* and later on concretize that type into a *class*, a *structure*, a finite *enumeration*, or a derived type.

type C
class C

AsmL has an expressive type system that allows one to define new types using finite sets, finite maps, finite sequences, tuples, etc., combined in arbitrary ways. For example, if *A* and *B* are types then the type of all maps from *A* to sets of elements of *B* is this.

Map **of** A **to** set **of** B

Finite sets, sequences, maps are ordinary elements. The common operations on sets, sequences, maps and other built-in data types are available as built-ins. For instance, the binary operation *apply(f, a)* applies a map *f* to an element *a*. The shorthand notation for map application is *f(a)*.

Derived Functions

Derived functions play an important role in application of ASMs. A derived function does not appear in the vocabulary; instead it is computed on the fly at a given state. In AsmL, derived functions are given by methods that return values. A derived function *f* from $A_0 \times A_a \times \cdots A_n$ to *B* can be declared as a global method.

f(x_0 **as** A_0, x_1 **as** A_1, …, x_n **as** A_n) **as** B

The definition of how to compute *f* may be given together with the declaration or introduction later on in the code. Alternatively, if *C* is a class or a structure then *f* can be declared as a method of *C*. Notice that *n* can be *0*.

```
class C
  f(x0 as A0, x1 as A1, …, xn as An) as B
```

A nullary derived function can be introduced as a global method that takes no arguments. For instance

```
f( ) as Boolean return b
```

where *b* is evaluate in a given state.

Constants

A nullary function that does not change during the evolution can be declared as a *constant*.

```
i as Integer = 0
```

A unary static function from a class *C* to *D* can be declared as a constant field of *C* as in the following example.

```
class C
  id as String
```

Variables

There are two kinds of variables, *global variables* and *local variable fields* of classes. Semantically, fields of classes are *unary* functions.

```
var b as Boolean
class C
  var v as Integer
```

Notice that *v* represents a unary dynamic function from *C* to integers.

Dynamic functions of ASMs are represented by variables in AsmL. A dynamic function *v* from $A_0 \times A_a \times \cdots A_n$ to *B* of any positive arity *n* can be represented as a variable map in AsmL.

```
var v as Map of (A1, …, An) to B
```

With the map representation, a normal ASM update $v(a) := b$ corresponds to a partial update of the map variable *v*. A set of consistent ASM updates to *v* corresponds to a set of consistent partial map updates that are combined into a single total update of *v*.

Classes and Dynamic Universe

AsmL classes are special *dynamic universes*. Classes are initially empty. Let *C* and *D* be two dynamic universes such that *C* is a subset of *D* and let *f* be a dynamic function from *C* to integers.

```
class D
class C extends D
  var f as Integer
```

The following AsmL statement adds a new reserve element *c* to *C* and *D*, and initializes *f(c)* to the value *0*.

```
let c = new C(0)
```

Classes are special dynamic universe in that one cannot programmatically remove an element from a class. In general, classes cannot be qualified over like sets but it is possible to check whether a given element is type *C* by using the *is* keyword.

```
if x is C then
```

In order to keep track of elements of a class *C*, one can introduce a variable of type *set of C* that is initially empty.

```
var Cs as Set of C = { }
```

Set-valued variables can be updated partially by inserting and removing individual set members. Several such pair-wise *non-conflicting partial updates* (i.e. do not both insert and remove the same element) are combined into a single *total update* at the end of the step.

```
let c = new C (0)
Cs (c) := true
```

Sets can be quantified over like in the following rule where all the invocations of *R(x)*, one for every element *x* of the set *s*, happen simultaneously in one atomic step.

```
forall x in s
  R (x)
```

CHAPTER IV

SEMANTIC ANCHORING CASE STUDY: FSM DOMAIN IN PTOLEMY

In this chapter, the FSM domain from Ptolemy is used as a case study to explain the semantic anchoring methodology and to illustrate how the semantic anchoring tool suite is applied to design DSMLs. The detailed implementation can be downloaded from [151].

The FSM domain in Ptolemy

Finite State Machine (FSM) has long been used to model the control logic of reactive systems, a class that includes most embedded systems and many software systems. However, conventional FSM models lack hierarchy and thus have a key weakness: the complexity of the model increases dramatically as the number of states increases. In 1987, David Harel proposed the Statecharts model [152], which extends the conventional FSM by supporting the hierarchical composition of states and concurrency. Many Statecharts variations have since been proposed to fit particular syntactic and semantic requirements. In 1999, Edward Lee proposed *charts [153], which allows the composition of hierarchical FSM with a variety of concurrency models and is later implemented as the FSM domain in the Ptolemy tool.

For simplicity, we define a DSML called the FSM Modeling Language (FML) which only supports Ptolemy-style hierarchical FSM. In this section, we give a short

introduction on the basic FSM and hierarchical FSM. For a detailed description of *charts and the FSM domain in Ptolemy, readers may refer to [154].

The Basic FSM

An FSM is a five tuple

$$(S, \Sigma, \Delta, \sigma, s_0)$$

where

- S is a finite set of states.
- Σ is an input alphabet, consisting of a set of input symbols.
- Δ is an output alphabet, consisting of a set of output symbols.
- σ is a transition function, mapping $S \times \Sigma$ to $S \times \Delta$.
- $s_0 \in S$ denotes the initial state.

In one *reaction*, a FSM maps a current state $s \in S$ and an input symbol $\alpha \in \Sigma$ to a next state $t \in S$ and an output symbol $\beta \in \Delta$, where $\sigma(s, \alpha) = (t, \beta)$. Give an initial state and a sequence of input symbols, a trace, which is a sequence of reactions, will produce a sequence of states and a sequence of output symbols.

A *state transition graph* shown in Figure 4-1 is a typical visualization for a FSM. In the figure, each node represents a state and each edge denotes a transition. A transition is labeled by "*guard* / *action* ", where *guard* $\in \Sigma$ and *action* $\in \Delta$. The action without a source state points to the initial state *s*. During one reaction of the FSM, one transition is taken, chosen from all *enabled transitions*. An enabled transition is an outgoing transition from the current state where the guard matches the current input symbol. The execution

of the transition produces the output symbol and set the destination state of the transition as the current state of the FSM.

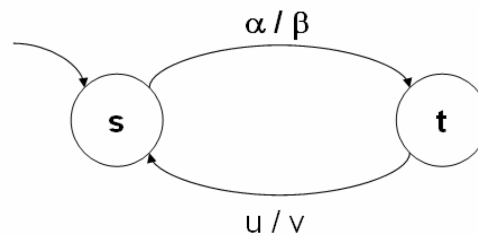

Figure 4-1 A basic FSM

A FSM is *deterministic* if from any state there is at most one enabled transition for each input symbol. A FSM is *reactive* if from any state there is at least one enabled transition for each input symbol. To ensure FSMs to be reactive but not complicate notations, every state is equipped with an *implicit self transition* (a transition whose source state and destination state are the same) for each input symbol that is not a guard of an explicit out transition. Every self transition has a default output symbol, denoted by ε, which has to be an element of Δ.

The Hierarchical FSM

The basic FSM is flat and sequential and has a major weakness: most practical systems have a very large number of transitions and states. Hierarchy is one solution to alleviate this problem. In a hierarchical FSM, a state is allowed to be refined into another FSM. Figure 4-2 shows a simple hierarchical FSM in which state *t* is refined. The inside FSM is called the *slave* and the outside FSM is called the *master*. A state with a slave FSM, such as state *t*, is called a *hierarchical state*. A state without a slave FSM, such as

state *p*, is called an *atomic state*. The input events for the slave FSM are a subset of the input events for the master FSM. The output events from the slave FSM are a subset of the output events from the master FSM.

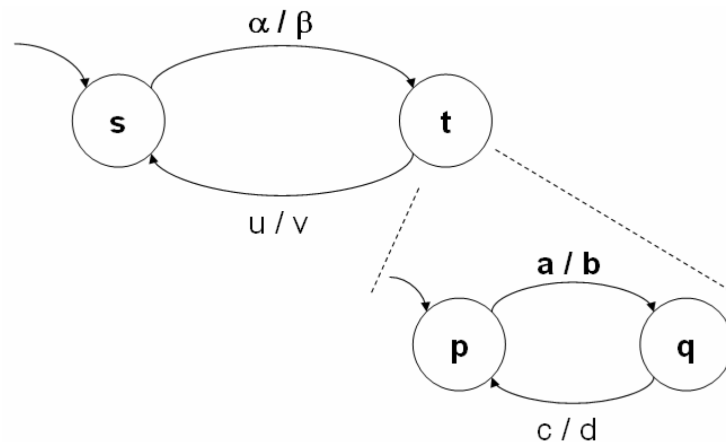

Figure 4-2 A hierarchical FSM

The hierarchy semantics defines how the slave FSM reacts to the reaction of its master FSM. This hierarchy semantics is defined by different tools. The *chart define one reaction of the hierarchical FSM as follow: If the current state is an atomic state, the hierarchical FSM behaves just like an FSM without hierarchy. If the current state is a hierarchical state, first the corresponding slave FSM reacts and then the master FSM reacts. In this way, two transitions are taken for one reaction.

In fact, hierarchy does not reduce the number of states in a FSM. However, it can significantly reduce the number of transition and make a FSM more intuitive and easy to understand.

The Syntax Definition for FML

The syntax of a DSML consists of an abstract syntax *A*, a concrete syntax *C* and a syntactic mapping $M_A: C \rightarrow A$. In our design methodology, a GME metamodel, which consists of UML class diagrams, OCL constraints, and visualization information, is able to define the syntax of a DSML. UML class diagrams define the modeling concepts and their relationships. The OCL constraints specify well-formedness rules for domain models using the concepts defined in the UML class diagrams. The syntactic mapping can be implemented in GME by specifying the visualization information for each class in the UML class diagrams.

Figure 4-3 A UML class diagram for the FML metamodel

Figure 4-3 presents a UML class diagram for the FML metamodel as represented in GME. The classes in the UML class diagram define the domain modeling concepts.

For example, the *State* class denotes the FSM domain concept of state. Instances (*State objects*) of the *State* class can be created in a domain model to represent the states of a specific FSM. Note that the *State* class is hierarchical – each *State* object can contain an entire child FSM. The *LocalEvent* class and the *ModelEvent* class represent respectively the Ptolemy FSM concepts: local event and model event. Local events are only visible within a single FSM model, whereas model events are globally visible.

A set of OCL constraints is added to the UML class diagram to specify well-formedness rules that the UML class diagram itself may not capable of expressing. For example, the constrain

```
Self.parts(State)→size>0 implies
                Self.parts(State)→select(s:State|s.initial)→size=1,
```

is attached to the State class. It specifies that if a *State* object has child states, exactly one child state should be the initial state.

The Semantic Unit Specification for FML

An appropriate semantic unit for FML should be generic enough to express the behavior of FML models. Since our purpose in this chapter is restricted to demonstrate the key steps in the semantic anchoring methodology, we do not investigate the problem of identifying a generic semantic unit for the hierarchical FSM. We simply define a semantic unit, which is rich enough for FML, instead of defining a common semantic model for hierarchical FSM. In the next chapter, we will have detailed discussions on how to specify a semantic unit for a Model of Computation.

The semantic unit specification includes two parts: an Abstract Data Model and a Model Interpreter defined as Operation and Transition Rules on the data structures

defined in the Abstract Data Model. The AsmL Abstract Data Model captures the abstract syntax of the semantic unit data models, and the Operation and Transition Rules specify the behavioral semantics of the semantic unit. Whenever we have a domain model in AsmL (which is a specific instance of the Abstract Data Model), this domain model and the semantic unit specification compose an abstract state machine, which gives the model semantics. The AsmL tools can simulate its behavior, perform the test case generation or perform model checking. Since the size of the full semantic unit specification is substantial, we only show the main part of the specification together with some short explanations.

AsmL Abstract Data Model for FML

In this step, we specify an Abstract Data Model using AsmL data structures, which will correspond to the semantically meaningful modeling constructs in FML. As we mentioned above, the Abstract Data Model does not need to capture every details of the FML modeling constructs, since some of them are only semantically-redundant syntactic sugar. The semantic anchoring (i.e. the mapping between the FML metamodel and the Abstract Data Model) will map the FML abstract syntax onto the AsmL data structures that we specify below.

```
interface Event

structure ModelEvent implements Event

structure LocalEvent implements Event

class FSM
  var outputEvents as Seq of ModelEvent
  var localEvents  as Set of LocalEvent
  var initialState as State
  var children     as Set of State
```

Event is defined as an AsmL abstract data type interface. *ModelEvent* and *LocalEvent* are AsmL structures. They implement the *Event* interface and may consist of one or more fields via the AsmL case construct. These fields are model-dependent specializations of the semantic unit, which give meaning to different types of events. In AsmL, classes may contain instance variables and are the only way to share memory. Structures may contain fields and do not share memory. The AsmL class *FSM* captures the top-level tuple structure of the hierarchical state machine. The field *outputEvents* is an AsmL *sequence* recording the chronologically-ordered model events generated by the FSM. If a generated event is a local event, its generation order does not need to be recorded. So, it will be recorded in the field *localEvents* which is an unordered AsmL *set*. The field *initialState* records the start state of a machine. The *children* field is an AsmL set that records all state objects which are the top-level children of the machine.

```
class State
  var active         as Boolean = false
  var initial        as Boolean
  var initialState   as State?
  var master         as State?
  var slaves         as Set of State
  var outTransitions as Set of Transition
```

State is defined as the first-class type. Note that the variable field *initalState* of the *State* class records the start state of any child machine contained within a given state

object. Initially, a state object is inactive. When the *active* filed is set to true, the state object becomes the current state in its parent machine. The *initial* filed denotes whether the state object is an initial state. The *initalState* filed will be undefined whenever a state has no child states. This possibility forces us to add the ? modifier to express that the value of the field may be either a State instance or the AsmL *null* value. For the similar reason, we add the ? modifier after the *master* field that refers to the parent state. The *slaves* filed is a set that recodes all the child states within the state objects. If this state is an atomic state, this set will be empty. The *outTransitions* filed records all transitions out from the state object.

```
class Transition
  var guard          as Boolean
  var preemptive     as Boolean
  var triggerEvent   as Event?
  var outputEvent    as Event?
  var src            as State
  var dst            as State
```

The AsmL class *Transition* captures the structure of the corresponding modeling concept Transition in FML. The *guard* field records the Boolean value of the guard of the transition object. The *preemptive* field indicated whether the transition object is a preemptive or non-preemptive transition. The *triggerEvent* and *outputEvent* fields record the input event and output event for the transition object respectively. The *src* and *dst* fields respectively record the source state object and destination state object of the transition object.

Behavioral Semantics for FML

We are now ready to specify the behavioral semantics for FML as Operation and Transition rules, which manipulate the AsmL data structures defined above. The specifications start from the top-level machine, and proceeds toward the lower levels.

Top-level FSM Operations

A FSM instance waits for input events. Whenever an allowed input event arrives, the FSM instance reacts in a well-defined manner by updating its data fields and activating enabled transitions. To avoid non-determinism, the Ptolemy FSM domain defined its own priority policy for transitions, which supports both the hierarchical priority concept and preemptive interrupts. The operational rule *fsmReact* specifies this reaction step-by-step. Note that the AsmL keyword *step* introduces the next atomic step of the abstract state machine in sequence. The operations specified within a given step all occur simultaneously.

```
fsmReact (fsm as FSM, e as ModelEvent) =
  step
    let s as State = getCurrentState (fsm, e)
  step
    let t as Transition? = getPreemptiveTrasition (fsm, s, e)
  step
    if t <> null then
      doTransition (fsm, s, t)
    else
      step
        if isHierarchicalState (s) then
          invokeSlaves (fsm, s, e)
      step
        let npt as Transition? = getNonpreemptiveTranstion (fsm, s, e)
      step
        if npt <> null then
          doTransition (fsm, s, npt)
```

First, the rule determines the current state, which might be an initial state. Next, it checks for enabled preemptive transitions from the current state. If one exists, then the

machine will take this transition and end the reaction. If a preemptive transition is taken, the slaves of the current state will not be invoked. If there is no enabled preemptive transition, the rule will first determine if the current state has any child states. If it does, the rule will invoke the slaves (child states) of the current state. Next, it checks for enabled non-preemptive transitions from the current state. If one exists, then the rule will take this transition and end this reaction. Otherwise, it will do nothing and end this reaction.

Invoke Slaves

The operational rule *invokeSlaves* describes the operations required to invoke the child machine in a hierarchical state (the master state). The AsmL construct *require* is used here to assert that this state should be a hierarchical state, and it should have a start state in its child machine. The rule first determines the current slave in the child machine. If there is an active slave, this active slave is the current slave. Otherwise, the start slave is the current slave. The rest of this operational rule is the same as the *fsmReact* rule. The similarity between the reactions of the top-level state machine and any child machine facilitates the Ptolemy style composition of different Models of Computations.

```
function invokeSlaves (fsm as FSM, s as State, e as Event) =
  require isHierarchicalState (s) and s.initialState <> null
  step
    let cs as State = getCurrentSlave (fsm, s, e)
  step
    let t as Transition? = getPreemptiveTrasition (fsm, cs, e)
  step
    if t <> null then
      doTransition (fsm, ids, pt)
    else
      step
        if isHierarchicalState (ids) then
          invokeSlaves (fsm, ids, e)
      step
        let npt as Transition? = getNonpreemptiveTranstion (fsm,ids,e)
      step
        if npt <> null then
          doTransition (fsm, ids, npt)
```

Do Transition

The operational rule *doTransition* specifies the steps through which a machine takes an enabled transition. The assertion for this rule is that the source state of the transition must be the current active state. First, exit the source state of the transition. Next, if the current transition mandates an output event, perform an emit event operation. Finally, make the destination state of the transition an active state.

```
doTransition (fsm as FSM, s as State, t as Transition) =
  require s.active
  step
    exitState (s)
  step
    if t.outputEvent <> null then
      emitEvent (fsm, t.outputEvent)
  step
    activateState (fsm, t.dst)
```

Activate State

The operational rule *activateState* describes the operations required to activate a state. The rule first sets the active field of the state. Then, it determines whether the state is a hierarchical state. If it is not, then the rule attempts to find an enabled instantaneous

transition out of the current state. In Ptolemy, an instantaneous transition is defined as any transition that is outgoing from an atomic state and lacks a trigger event. An instantaneous transition must be taken immediately after entering its source state. If such a transition exists, the rule forces this transition and returns.

```
activateState (fsm as FSM, s as State) =
  step
    s.active := true
  step
    if isAtomicState (s) then
      step
        let t as Transition? = getInstantaneousTransition (s)
      step
        if t <> null then
          doTransition (fsm, s, t)
```

Get Instantaneous Transition

The operational rule *getInstantaneousTransition* finds all the enabled instantaneous transitions from an atomic state. The AsmL construct require is used here to assert that this state should be an atomic state. Since the Ptolemy FSM domain does not support non-determinism, the rule will report a nondeterministic error when more than one transition is enabled. If exactly one is enabled, return the enabled instantaneous transition. Otherwise, return null.

```
getInstantaneousTransition (s as State) as Transition? =
  require isAtomicState (s)
  step
    let ts = {t | t in s.outTransitions where t.triggerEvent = null
      and t.guard }
  step
    if Size (ts) > 1 then
      error "non-deterministic error"
  step
    choose t in ts
      return t
    ifnone
      return null
```

Semantic Anchoring Specification

Having the abstract syntax of FML and an appropriate semantic unit specified, we are now ready to describe the semantic anchoring specifications for FML. We use UMT, a language supported by the GReAT tool, to specify the model transformation rules between the metamodel of FML (Figure 3-3) and the metamodel for the semantic unit shown in Figure 4-4. As we have discussed in Chapter 3, there are two approaches in defining metamodels for semantic units. One solution is to define a UML/OCL-based metamodel that captures the abstract syntax of the generic AsmL data structures. The other solution is to construct a metamodel that captures only the exact syntax of the AsmL Abstract Data Model of a particular semantic unit. We have discussed the advantages and disadvantages of both approaches. The second approach is applied in this example. Figure 4-4 presents a metamodel capturing the exact syntax of the FML Abstract Data Model.

The semantic anchoring specification in UMT consists of a sequence of model transformation rules. Each rule is finally expressed using *pattern graphs*. A pattern graph is defined using associated instances of the modeling constructs defined in the source and destination metamodels. Objects in a pattern graph can play three different roles as follows:

1. *bind*: Match object(s) in the graph.
2. delete: Match objects(s) in the graph, then, remove the matched object(s) from the graph.
3. *new*: Create new objects(s) provided all of the objects marked *Bind* or *Delete* in the pattern graph match successfully.

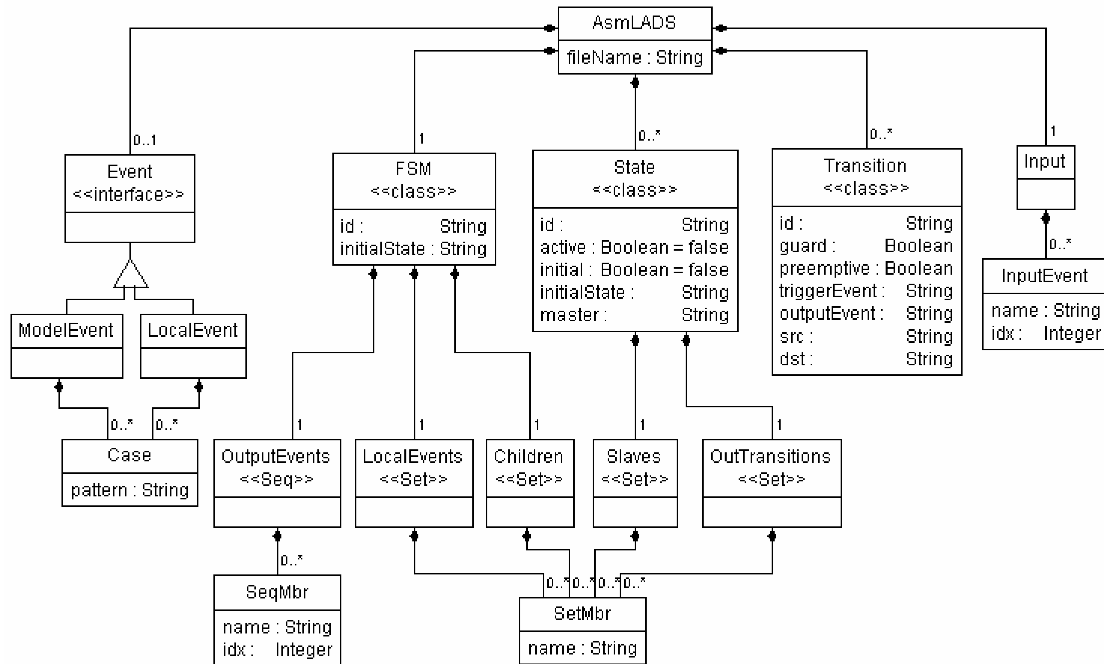

Figure 4-4 Metamodel capturing AsmL Abstract Data Structures for FML

The execution of a model transformation rule involves matching each of its constituent pattern objects having the roles *bind* or *delete* with objects in the input and output domain model. If the pattern matching is successful, each combination of matching objects from the domain models that correspond to the pattern objects marked *delete* are deleted and each new domain objects that corresponds to the pattern objects marked *new* are created.

We give an overview of the model transformation algorithm with a short explanation for selected role-blocks below. The transformation rule-set consists of the following steps:

1. Start by locating the top-level state machine in the input FML model; create an AsmL *FSM* object and set its attribute values appropriately.

2. *Handle Events*: Match the model event and local event definitions in the input model and create the corresponding variants through the *Case* construct in *Event*.
3. *Handle States*: Navigate through the FML *FSM* object; map its child *State* objects into instances of AsmL *State* class, and set their attribute values appropriately. Since the *State* in FML has a hierarchical structure, the transformation algorithm needs to include a loop to navigate the hierarchy of *State* objects.
4. *Handle Transition*: Navigate the hierarchy of the input model; create an AsmL *Transition* object for each matched FML *Transition* object and set its attribute values appropriately.

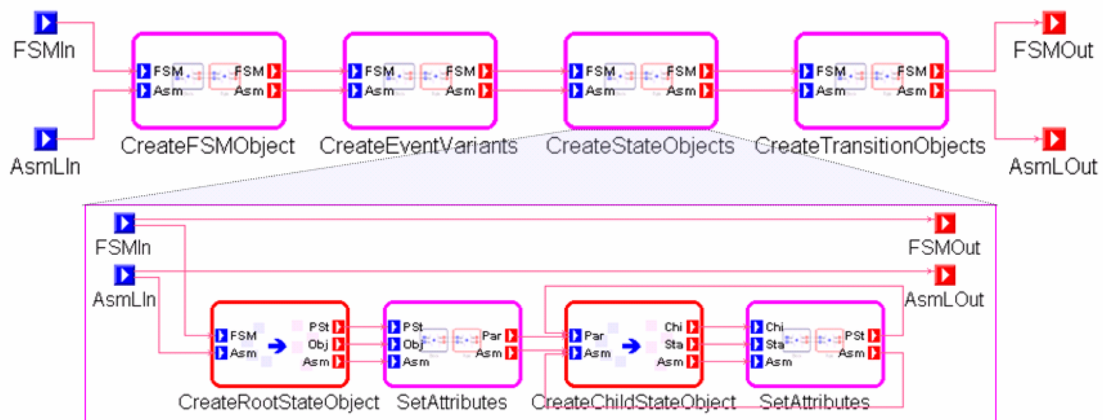

Figure 4-5 Top-level model transformation rule for the FML semantic anchoring specifications

Figure 4-5 shows the top-level transformation rule that consists of a sequence of sub-rules. These sub-rules are linked together through ports within the rule boxes. The connections represent the sequential flow of domain objects to and from rules. The ports *FSMIn*, and *AsmLIn* are input ports, while ports *FSMOut* and *AsmLOut* are output ports. In the top-level rule, *FSMIn* and *AsmLIn* are bound to the top-level state machine in the

FSM model that is to be transformed, and the root object (a singleton instance of *AsmLADS*) in the semantic data model that is to be generated, respectively. The four key steps in the transformation algorithm, as described above, are corresponding to the four sub-rules contained in the top level rule.

The figure also shows a hierarchy, i.e., a sub-rule may be further decomposed into a sequence of sub-rules. The *CreateStateObjects* rule outlines a graphical algorithm which navigates the hierarchical structure of a state machine. It starts from the root state, does the bread-first navigation to visit all child state objects and creates corresponding AsmL *State* objects.

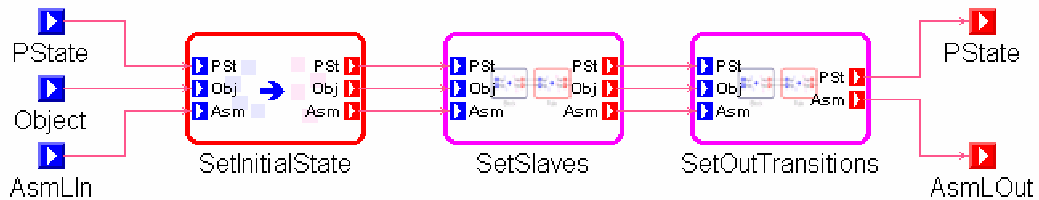

Figure 4-6 Model Transformation Rule: *SetAttributes*

Figure 4-6 presents the *SetAttributes* rule. This rule sets the attribute values for the newly created AsmL *State* object. First, the sub-rule *SetInitialState* checks whether the current FML *State* object is a hierarchical state and has a start state. If it has a start state, set the value of the attribute *initialState* to this start state. Otherwise set the value to null. Then, the sub-rule *SetSlaves* searches for all hierarchically-contained child states in the current state and adds them as members into the attribute *Slave* whose type is a set. Finally, the transitions out from the current state are added as members to the attribute *OutTransitions* by the sub-rule *SetOutTransitions*.

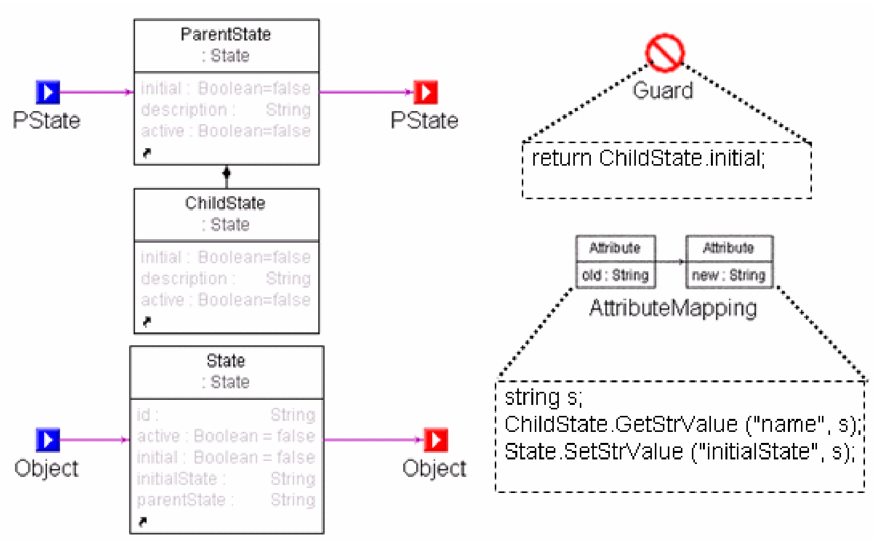

Figure 4-7 Model Transition Rule: *SetInitialState*

The final contents of model transformation rules are pattern graphs that are specified in UML class diagrams. Figure 4-7 shows a part of the *SetInitialState* rule, which is a pattern graph. This rule features a GReAT *Guard* code block and a GReAT *AttributeMapping* code block. This rule is executed only if the graph elements match and the *Guard* condition evaluates to true. The *AttributeMapping* block includes code for reading and writing object attributes. In Figure 4-7, the *Guard* condition claims that the *ChildState* object should be an initial state.

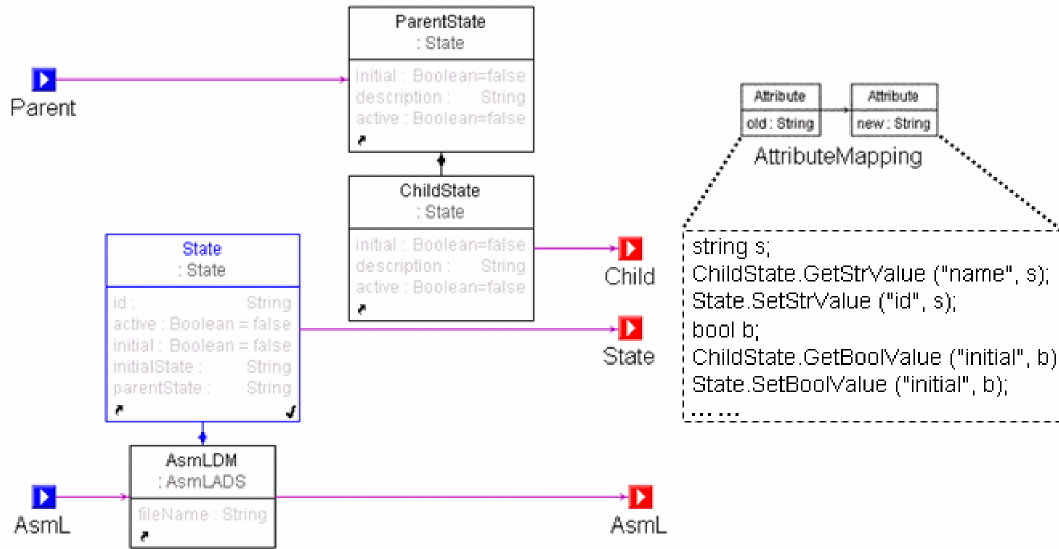

Figure 4-8 Model Transition Rule: *CreateChildStateObject*

The *CreateChildStateObejct* rule, shown in Figure 4-8, creates a new AsmL *State* object when a FML child *State* object is matched. It also enables the hierarchy navigation. Through a loop specified in the *CreateStateObjects* rule (Figure 4-5), the child *State* object output by the *Child* port will come back as an input object to the *Parent* port.

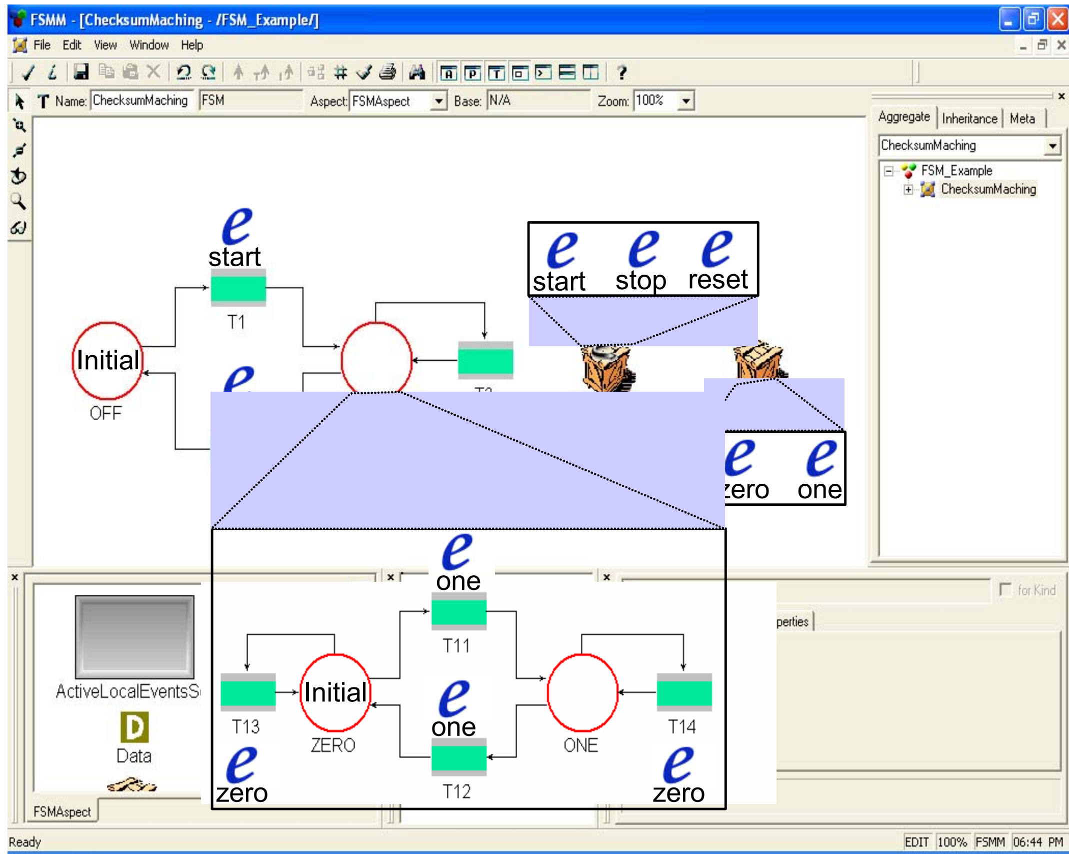

Figure 4-9 A hierarchical FSM model for ComputerStatus

In the semantic anchoring process, the GReAT engine takes a legal FML domain model, executes the model transformation rules and generates an AsmL data model. As an example, we design a simple hierarchical FSM model in the GME modeling environment (Figure 4-9), which simulates the status of a computer. An XML file storing the AsmL data model is generated through the semantic anchoring process. We developed an AsmL specification translator, which can parse this XML file and generate the data model in native AsmL syntax as shown in Figure 4-10. The newly created AsmL data model plus the previously-defined AsmL semantic domain specifications compose an abstract state machine that gives the semantics to the FSM ComputerStatus model.

With these specifications, the AsmL tools can simulate the behavior, do the test case generation and model checks. For more information about the AsmL supported analysis, readers are referred to [147].

```
interface  Event
structure ModelEvent implements Event
  case start
  case stop
  case reset
structure LocalEvent implements Event
  case zero
  case one
var ChecksumMaching = new FSM ([], {}, OFF, {ON, OFF})
var OFF = new State (false, true, null, null, {}, {T1})
var ON  = new State (false, false, ZERO, null, {ONE, ZERO}, {T3, T2})
var ZERO= new State (false, true, null, ON, {}, {T13, T11})
var ONE = new State (false, false, null, ON, {}, {T14, T12})
var T1  = new Transition (true, false, ModelEvent.start, null, OFF, ON)
var T2  = new Transition (true, false, ModelEvent.stop, null, ON, OFF)
var T3  = new Transition (true, false, ModelEvent.reset, null, ON, ON)
var T11 = new Transition (true, false, LocalEvent.one, null, ZERO, ONE)
var T12 = new Transition (true, false, LocalEvent.one, null, ONE, ZERO)
var T13 = new Transition (true, false,LocalEvent.zero, null,ZERO, ZERO)
var T14 = new Transition (true, false, LocalEvent.zero, null, ONE, ONE)
```

Figure 4-10: Part of the AsmL data model generated from the ComputerStatus model

CHAPTER V

A SEMANTIC UNIT FOR TIMED AUTOMATA BASED MODELING LANGUAGES

A key issue in the semantic anchoring methodology is the specification of semantic units. This chapter proposes a semantic unit, which captures the Timed Automata behavior, and uses this as an example to illustrate the whole process of the semantic unit specification. The precise semantics of a wide range of Timed Automata based modeling languages (TAMLs) can then be defined by specifying semantic anchoring rules between a domain-specific TAML and the Timed Automata Semantic Unit.

Semantic Units

Semantic units are designed to capture the operational semantics of a finite set of basic Models of Computations (MoCs), such as Finite State Machine, Timed Automata, Discrete Event System and Synchronous Dataflow. They sever as reusable semantic models in the semantic anchoring infrastructure so that the semantics of an application DSML can be defined by specifying semantic anchoring rules to appropriated semantic units.

By using the semantic anchoring tool suite, a semantic unit is specified in AsmL specification including two parts: an Abstract Data Model and a Model Interpreter defined as Operation and Transition Rules on the abstract data structures. In order to integrate the semantic unit specification with the formal metamodeling framework, which

supports DSML syntax design, a metamodel is designed to capture the Abstract Data Model of the semantic unit. This metamodel can be thought as a metamodeling interface for the semantic unit to support the specification of semantic anchoring rules.

To successfully define a semantic unit for a MoC, the most important task is to understand what the semantics of the MoC is. A MoC may have an evolution history and many modeling language variants. Typically, a MoC is firstly proposed with a mathematical definition to address a certain behavioral pattern. Then, it is implemented by many tool-defined modeling languages which may extend the basic semantics with some tool-defined semantic concepts. As the later evolution of these modeling languages, the semantics may be further changed or new semantic concepts may be added. Modeling language variants enrich the application and also contribute the further development of a MoC. However, they make the precise semantics definition of a MoC difficult.

If a semantic unit only covers the basic semantics of a MoC, many tool-defined semantic concepts that are very useful and have been widely accepted by the related communities may not be captured by the semantic unit. On the contrary, if the semantic unit contains all tool-defined semantic concepts, the semantic unit may be too complicated and some properties (such as analyzability and verifiability) of a MoC may be impaired. It is necessary to do a deep analysis on different modeling language variants of a MoC and make clear the essential semantic concepts needed to be captured in the semantic unit.

Timed Automata

Timed Automata were firstly proposed by Rajeev Alur and David L. Dill [155] for modeling and verification of real-time systems. A timed automaton is a finite-state automaton extended with real-value variables. Such an automaton may be considered as an abstract model of a timed system. The variables model the logical clocks in the system, and are initialized with zero when the system is started, and then increase synchronously in the same rate. Clock constraints (i.e. guards on transitions) are used to restrict the behavior of the automaton. A transition can be taken when the evaluation of clock variables satisfy the guard. Clock variables can be reset to zero when a transition is taken.

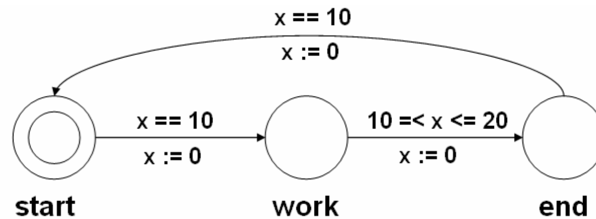

Figure 5-1 A timed automaton example

A timed automaton is typically represented as a graph containing a finite set of nodes or locations and a finite set of labeled edges. Figure 5-1 shows a simple timed automaton example. The timing behavior of the automaton is controlled by the clock variable *x*. The automaton starts from the *start* location where *x* is zero initially. It may leave *start* when *x* equals 10. If it goes from *start* to *work*, *x* will be reset to zero. Then, it needs stay at *work* unless *x* reach 10. When *x* is between 10 and 20, the automaton may go from *work* to *end* and reset the clock *x*, etc.

Timed BÜchi Automata

A guard on an edge of an automaton is only an enabling condition of the transition represented by the edge. However, it can not force the transition to be taken. In other words, an automaton can stay forever in any location idly. In the initial work by Alur and Dill [155], this problem is solved by introducing BÜchi-acceptance condition. A subset of the locations in a automaton are marked as accepting, and only those execution passing through an accepting location infinitely often are considered valid behaviors of an automaton. As the example is Figure 5-1, if the *end* location is marked as accepting, all legal execution of the automaton must visit *end* infinitely many times. This imposes implicit conditions on *start* and *work*. The *start* location must be left when *x* equals 10, otherwise, the automaton will get stuck in *start* and never be able to enter the location *end*. Likewise, the automaton must leave the *work* location when *x* is at most 20 to be able to enter the *end* location.

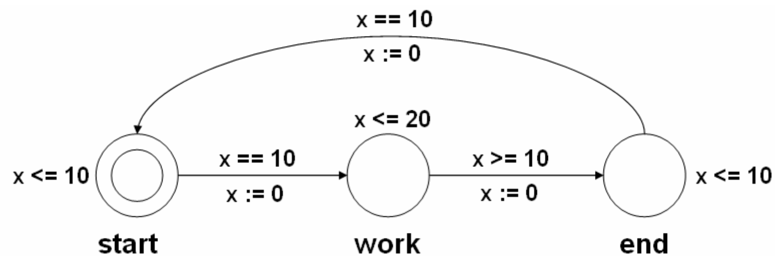

Figure 5-2 A timed safety automaton with location invariants

Timed Safety Automata

In order to force a transition to be taken, timed safety automata [159] introduce a more intuitive notion of progress. Instead of accepting conditions, in timed safety

automata, location may be put local timing constraints called *location variants.* An automaton may remain in a location as long as the clocks values satisfy the invariant condition of the location. For instance, Figure 5-2 presents a timed safety automaton which is equivalent to the timed automaton in Figure 5-1 with the *end* location marked as an accepting location. The location invariants claim that the automaton must leave the *start* and *end* location when x is at most 10, and must leave the *work* location when x is at most 20.

Timed Automata Semantic Unit

Since Timed Automata were proposed, several analysis tools for real time systems, such as UPPAAL [156], IF toolset [157] and Kronos [158], were developed based on this modeling approach. They use TAMLs that have tool dependent differences in their approaches to express communication among concurrent components and action (transition) priorities. The similarities and differences in the syntax and semantics of varied TAMLs may confuse designers and lead to mistakes. There are plenty of examples for language constructs that may appear similar while they express essentially different semantics and language constructs that appear different but have essentially the same semantics.

In this section, we propose a Timed Automata Semantic Unit (TASU) with a support for action priorities as a common semantic model for TAMLs. Semantics of varied TAMLs can be defined by specifying the transformation rules between them and the TASU. The explicit representation of transformation rules, the formal operational semantics specification of TASU and the behavioral simulation support allow designers

to understand and compare languages with different timed automata semantics and help the integration of different analysis tools in design flows. Also, the TASU can work as a semantic reference model for DSMLs with timed automata related behavioral aspects.

Overview of TASU

The initial timed automata model assumes a strong synchrony assumption is adopted for time progress, which means that all clock variables progress at the same rate. Transitions are executed instantly and time progresses only when an automaton is in a location. Constraints on clock variables can be used as conditions for enabling transitions. Transitions can be associated with actions that reset clock variables.

In order to facilitate modeling concurrent real-time systems, many TAMLs (e.g. UPPAAL and IF) extend the original timed automata with parallel composition to specify networks of automata. The supported communication mechanisms vary across tools. In general, such communication can be achieved through shared variables, synchronous signaling, or asynchronous message passing. The asynchronous form of communication can be reduced to a synchronous communication plus a buffering mechanism. To keep balance between the expressiveness and complexity, our proposed semantic unit supports communication only through shared variables and synchronization. If needed, asynchronous communication semantics can be specified via mapping to a composition of synchronous communication and buffering using model transformation.

Transition priority is a very useful concept for reducing non-determinism in models and for modeling interrupts or preemption in real-time systems. Also, dynamic priorities match well with practical implementations of real-time systems. Priority

information is implicitly expressed in certain language constructs of a TAML. For instance, an *urgent location* in UPPAAL indicates that transitions out from this location have a higher priority than that of the time progress transition. The priority of a transition is, in general, time dependent. For example, a *delayable transition* in IF semantically implies that the priority of this transition jumps to a higher value than that of time progress when the enabling condition of this transition is about to be violated by time progress.

Although priority hierarchies of TAMLs are tool-dependent, they have many common features. In order to compare and integrate models from varied TAMLs, we need to establish a generic priority scheme that is capable of capturing all these priority hierarchies. A fundamental common feature is that all these tool-defined priority schemes are built with respect to the *time progress priority*, which enables modeling urgency of actions in real-time systems. An urgent action in real-time systems is modeled as a transition having higher priority than that of time progress. In some TAMLs, urgent transitions are additionally divided into two groups: normal urgent transitions and critically urgent transitions. A critically urgent transition prohibits the execution of any other transitions including time progress and is modeled by an atomic action. An atomic action is composed of a sequence of sub-actions and no interrupts is allowed during the execution of these sub-actions.

In TASU, the priority hierarchy has three layers: the bottom priority (the time progress priority), the top priority, and the urgent priority, which has a series of urgency degrees. No transition can have priority lower than the time progress priority, since such a transition will always be blocked by time progress and have no chance to be executed.

The top priority enables modeling of atomic actions. All transitions in urgent priority block time progress, and their relative priorities are determined by their urgency degrees. We will show that this priority hierarchy is capable of expressing varied priority hierarchies defined by TAMLs.

In the proposed TASU, a real-time system contains a set of concurrent components. Each component is modeled by a timed automaton. Components communicate among each other through shared variables and synchronization. The priority of an action can be dynamically updated with respect to time progress. Enabled actions with higher priorities will block actions with lower priorities. Non-determinism is supported by allowing multiple enabled actions with the same priority. Based on the timed automata model defined in [150], we present an abstract mathematical definition for a timed automaton in the semantic unit.

Given a finite set of variables *V*, a *valuation* for the variables is a function $\nu \in \Re^V$ that assigns a value for each variable from the domain of real numbers. If $|V| = n$ the valuation can be represented as the vector $\bar{v} \in \Re^n$. We denote the valuation for an element $i \in N$ in *V* as v_i. A *linear expression* $\phi(\bar{v})$ over *V* can be expressed as $\sum a_i v_i$ where $a_i \in Z$ (Z denotes a set containing all integers) and $v_i \in V$. A *linear constraint* γ is of the form $\phi(\bar{v})$ **op** c where $\phi(\bar{v})$ is a linear expression over *V*, **op** $\in \{=, <, \leq, >, \geq\}$ and $c \in Z$. We denote the set of linear constraints over the set of variables *V* as **LC**(*V*). A *linear assignment* over *V* is defined as $A\bar{v} + \bar{c}$, where **A** is an $n \times n$ matrix with coefficients from Z and $\bar{c}$ is a vector of Z^n. We denotes a set of linear assignment over the set of

variable V as **LA**(V). The set of *simple assignment* **SA**(V) corresponding to the case when all entries of **A** are 0 and $\bar{c} \geq 0$.

A timed automaton is defined over a set C of resetable clocks and a set V of integer variables. A timed automaton in the semantic unit is a 7-tuple $\langle C, V, \Sigma, L, l_0, E, \Pr i \rangle$ where:

- C is a finite set of n clock variables,
- V is a finite set of integer-valued variables,
- $\sum$ is a finite set of symbols defining the system events,
- L is a nonempty set of locations,
- $l_0 \in L$ is the initial location,
- $E \subseteq L \times \sum \times \mathrm{LC}(C \cup V) \times \mathrm{SA}(C) \cup \mathrm{LA}(V) \times L$ is a set of edges. An edge $\langle l, \alpha, \varphi, \beta, \gamma, l' \rangle$ represents a transition from location l to location l' on symbol α. φ is a guard over clock and integer variables. β represents simple assignment for clock variables and γ is linear assignment over integer variables,
- $\Pr i : E \times R^n \rightarrow N_+$ is a map that assigns to each edge its priority, which is a non-negative integer value, with respect to a give clock evaluation $v \in R^n$, so that $\Pr i(e, v)$ is the priority of edge e at clock value v.

A state of the timed automaton is defined as (l, c, v), where $l \in L$, denoting the current location of the automaton, and *c, v* are valuation of the clock C and integer V variables, respectively. The set of all state is denoted S. The $\xrightarrow{\alpha}$ step relation denotes a *jump transition* which is a discrete and instantaneous transition that changes the location of the automaton as well as the assignment of the integer variables and clocks.

The $\xrightarrow{t}$ step relation denotes a *time transition* that advances all clock variables at the same value. A time transition may affect the priority of edges through the function *Pri*. The priority of a time transition is assumed a constant value, *zero*.

The semantics of a timed automaton model $M = \langle C, V, \Sigma, L, l_0, E, \Pr i \rangle$ in TASU is given as a transition system $T_M = (S, s_0, \rightarrow)$ where S is the set of states, s_0 is the initial state where $c_0 = 0$, and the step relation $\rightarrow$ is the union of the jump transition:

- $(l, c, v) \xrightarrow{\alpha} (l', c', v')$ iff $\exists \;\; e = \langle l, \alpha, \varphi, \beta, \gamma, l' \rangle \in E$ such that
  - $\varphi(c, v) = true \wedge c' = \beta(c) \wedge v' = \gamma(v)$, and
  - $\forall \;\; e' = \langle l, \alpha, \varphi', \beta', \gamma', l'' \rangle \in E$, $\varphi'(c, v) = true \Rightarrow \Pr i(e, c) \geq \Pr i(e', c)$.

and time transition:

- $(l, c, v) \xrightarrow{t} (l, c + t, v)$ iff

  $\forall \;\; e = \langle l, \alpha, \varphi, \beta, \gamma, l' \rangle \in E$ and $\forall \;\; 0 \leq t' \leq t$, $\varphi(c + t', v) = true \Rightarrow \Pr i(e, c + t') = 0$.

A *run* of the timed automata is a finite or infinite sequence of alternating jump and time transition of T_M: $\rho = s_0 \xrightarrow{\alpha 1} s_1 \xrightarrow{t1} s_2 \xrightarrow{t2} s_3 \xrightarrow{\alpha 2} s_4 \cdots$.

The operational semantics for TASU is specified as a Control State ASM. The specification includes two parts: an Abstract Data Model, Operations and Transition Rules. In the ASM formulation, the Abstract Data Model captures the abstract syntax of TASU. The Operations and Transition Rules form a model interpreter that specifies the operational semantics on the data structure defined in the Abstract Data Model. An instance of the Abstract Data Model (we will refer to it as Date Model) and the operational semantics specification form an ASM that specifies the model semantics.

Here, we present only a part of the specification together with short explanations. The full TASU specification can be downloaded from [151].

Abstract Data Model

We choose to define the Abstract Data Model for TASU by using AsmL classes. The *Clock* defines a type for clock variables. The variable field *time* represents the logical time of a clock variable. All clock variables progress at the same rate. For the purpose of model simulation, we introduce an AsmL constant *CLOCKUNIT* to set the granularity of time progress. A system might define a set of global clocks and each component might define its own local clocks. The *globalClocks* is an AsmL *set* containing all global clocks. Note that the set *globalClocks* is empty in the specification of the Abstract Data Model. Elements of this set are model-dependent and will be specified in instances of the Abstract Data Model (in the Data Models). When the Boolean variable *TimeBlocked* is set to *true*, the system explicitly blocks time progress.

```
class Clock
  var time         as Double = 0

const CLOCKUNIT    as Double

var globalClocks   as Set of Clock = {}

var TimeBlocked    as Boolean =  false
```

Location is defined as a first-class type. The Boolean field *initial* indicates whether this location is the initial location of an automaton. The field *outTransitions* is a set that contains all transition objects that are out from this location.

```
class Location
  const id              as String
  const initial         as Boolean
  const outTransitions  as Set of Transition
```

Transition is also defined as a first-class type. The constant fields *blockTime* and *setPrior* in *Transition* indicate whether this transition is a special transition to block time progress or to set the priority of the owner automaton to the top priority. The *blockTime* transition is used by an automaton to explicitly block time progress, while the *setPrior* transition is employed by an automaton to explicitly claim that it is in the process of executing an atomic action, and enabled transitions in all other automata as well as time progress should be prohibited. Note: these two special transition types are adopted for software implementation convenience. They are also represented as edges whose source and destination locations are the same. (See further explanation in the operational semantics specification.) A transition can participate in a synchronous communication in one of two roles: *SEND* and *RECEIVE*, which are defined by the enumeration type *SYNROLE*.

```
class Transition
  const id              as String
  const blockTime       as Boolean
  const setPrior        as Boolean
  const dstLocationID   as String

enum SYNROLE
  SEND
  RECEIVE
```

The class *SignalChannel* captures the synchronous communication among automata. The variable field *senders* and *receivers* record a set of signal senders and receivers, respectively. The Boolean field *broadcast* indicates whether the signal channel is a broadcasting channel. In a broadcasting channel, a sender publishes events without

waiting for receivers and the event will be broadcasted to all receivers that are waiting for it. The event will be lost, if there are no receivers that are waiting for it. A non-broadcasting channel is enabled to fire when there are at least one sender and one receiver that are waiting for synchronization. Only one sender and one receiver can take part in a synchronization communication and a synchronization pair is chosen non-deterministically when several combinations are enabled.

```
class SignalChannel
  const id        as String
  const broadcast as Boolean
  var senders     as Set of (TimedAutomaton, Transition) = {}
  var receivers   as Set of (TimedAutomaton, Transition) = {}
```

The abstract class *TimedAutomaton* defines the base structure for a timed automaton. The variable field *currentLocation* refers to the current location of an automaton. Initially, it refers to the AsmL *null* value. The AsmL construct *Location?* indicates that the value of this field may refer to either a *Location* instance or to the *null* value. When an automaton is in the top priority layer, its *prior* field is set to *true*.

The *TimedAutomaton* class also holds a set of read-only abstract properties and abstract methods. These abstract properties define an abstract data structure that captures the tuple structure of an automaton. The abstract property *locations* and *transitions* are sets that contain all instances of *Location* and *Transition* in an automaton respectively. The abstract property *localClocks* is set of all local clock variables defined in an automaton. The abstract property *syns* is *a map* whose domain consists of transitions that require synchronization. If *t* is a transition in this domain, then *syns(t)* returns a 2-tuple whose first element refers to the corresponding signal channel and whose second element indicates whether *t* acts as a sender or a receiver. The abstract method *TimeGuard(t)* and

DataGuard(t) return a Boolean-valued expression of time conditions and data conditions that are attached to the transition *t*, respectively. The abstract method *DoAction(t)* executes actions attached to a transition *t*. The abstract method *Priority(t)* returns the priority of a transition. Since the priority value of a transition might be dynamically updated, *Priority(t)* returns either an integer value or an integer-valued expression. All these abstract members of *TimedAutomaton* are model-dependent specifications for the semantic unit and will be further specified by a concrete automaton template.

```
abstract class TimedAutomaton
  const id              as String
  var currentLocation as Location? = null
  var prior             as Boolean   = false

  abstract property locations   as Set of Location
    get

  abstract property transitions as Set of Transition
    get

  abstract property localClocks as Set of Clock
    get

  abstract property syns as Map of <Transition,(SignalChannel,SYNROLE)>
    get

  abstract TimeGuard (t as Transition) as Boolean

  abstract DataGuard (t as Transition) as Boolean

  abstract DoAction  (t as Transition)

  abstract Priority  (t as Transition) as Integer
```

The AsmL class *RTSystem* captures the top-level structure of a real-time system that is modeled by timed automata. The *components* field holds concurrent components contained in the system. Each component is an instance of a concrete timed automaton template. These components communicate through a set of signal channels, which are recorded in the field *signalChannels*.

```
class RTSystem
  const components     as Set of TimedAutomaton
  const signalChannels as Set of SignalChannel
```

Operational Semantics

We are now ready to specify the operational semantics for TASU as AsmL Operations and Transition Rules, which interpret the Abstract Data Model defined above. The specifications start from the top-level system, and proceed toward the lower levels.

```
class RTSystem
  Run()
    step Initialize()
    step until fixpoint
      step
        RegisterSignalChannels()
      step
        let T as (TimedAutomaton?,Transition?) = GetNextTransition()
      step
        if TimeBlocked then
          if T.Second = null then
            error("The system is blocked.")
          else
            T.First.DoTransition(t.Second)
        else
          if T.Second = null then
            TimeProgress()
          else
            if T.First.GetPriority()=0 and RandomDecisionIsTrue() then
              TimeProgress()
            else
              T.First.DoTransition(t.Second)
```

An active *RTSystem* instance executes enabled transitions or advances time. The operational rule *Run* of *RTSystem* specifies the top-level system operations as a set of updates. (Note that the AsmL keyword *step* introduces a set of operations that updates the ASM states. All operations within a single step occur simultaneously.) The rule *Run* first initializes all components in the system, which makes the field *currentLocation* of each component refer to its initial location. The next step is executed until the operations

inside the step causes no state changes in the ASM (*fixpoint*). The stopping condition for an AsmL *fixpoint* loop is met if no non-trivial updates have been made in the step. Updates that occur to variables declared in ASMs that nested inside this loop are not considered. An update is considered non-trivial if the new value is different from the old value.

Within the loop, the rule first registers all current transitions that are enabled to participate in synchronization to the corresponding signal channels. A transition is called current if it is a transition from the current location of an automaton. Then the signal channels are able to judge whether they are enabled to trigger communication among components. Next, the rule checks all current transitions and selects an enabled one that has the highest priority as a candidate for the next execution. If such enabled transitions exist, the selected candidate is recorded as a 2-tuple whose first element refers a component and whose second element refers an enabled transition in that component. Otherwise, the second element of the 2-tuple must be a *null* value. Afterwards, the rule checks a set of current system properties and makes decisions on whether to execute the candidate transition or to advance time. There are four possible cases:

1. If time progress is explicitly blocked and the system has no enabled transitions, the rule returns an error message that indicates the system will be blocked indefinitely.
2. If time progress is explicitly blocked and the system has a candidate transition, this candidate transition is executed.
3. If time progress is allowed and the system has no enabled transitions, the rule forces time progress.

4. If time progress is allowed and the system has a candidate transition, the rule checks the priority of this candidate transition. This candidate transition is executed when it has a higher priority than that of time progress. Otherwise, the rule randomly determines to advance time or to execute the candidate transition.

```
abstract class TimedAutomaton
  IsTransitionEnabled (t as Transition) as Boolean
    if (TimeBlocked and t.blockTime) or t.setPrior then
      return false
    else
      return t in currentLocation. outTransitions and TimeGuard(t) and
        DataGuard(t) and IsSynEnabled(t)
```

The operational rule *IsTransitionEnabled* of *TimedAutomaton* examines if a transition is enabled or not. The subrule *IsSynEnabled(t)* checks whether the corresponding signal channel is ready to activate a synchronization among transitions. If the transition *t* does not require to synchronize with other transitions, *IsSynEnabled(t)* always returns *true*.

The block time transition and the *setPrior* transition are two special transitions. A block time transition has an implicit guard, which checks if the *TimeBlocked* field is *false*, and has an implicit action that sets the *TimeBlocked* field to *true* so that the system knows time is explicitly blocked. This implicit guard is added to avoid the infinitely execution of a block time transition. A *setPrior* transition has no guard, but has one more implicit action that sets its owner component as a prior component. It must be enforced immediately after a component enters a location that has a *setPrior* transition. A *setPrior* transition is always considered disabled by the rule *IsTransitionEnabled* of *TimedAutomaton*, since if a *setPrior* transition exists in the current location, it must

already be executed when the current location is first entered. After an execution of a normal transition (transitions other than the block time transition and the *setPrior* transition), the priority status of the system will be refreshed - the *prior* field of a component and the *TimeBlocked* field of the system will be reset to *false*.

For a normal transition, its enabling condition should satisfy all the flowing conditions: (1) its source location refers to the current location; (2) its timing guard is true; (3) its data guard is true; (4) it does not need to synchronize with other transitions or the corresponding synchronization channel is ready to fire.

```
abstract class TimedAutomaton
  GetEnabledTransition() as Transition?
    let ET = {t | t in transitions where IsTransitionEnabled(t)}
    if Size(ET) = 0 then
      return null
    else
      choose t in ET where t.blockTime
        return t
      ifnone
        return (any t | t in ET where not
          (exists t2 in ET where Priority(t2) > Priority(t)))
```

The operational rule *GetEnabledTransition* chooses a transition with the highest priority from all enabled transitions in a component. The rule returns a *null* value if the component has no enabled transition. In short, a block time transition has higher priority than normal transitions. For a normal transition, the larger its priority value, the higher its priority. If there are multiple enabled transitions with the same highest priority value, one of them is randomly selected. This non-determinism is specified by using the AsmL construct **any**. The AsmL specification

```
(any t| t in ET where not
                    (exists t2 in ET where Priority(t2) > Priority(t)))
```

means that a transition is selected from the set *ET* , which contains all enabled transitions, so that no other transitions in *ET* can have higher priority value than the priority of the selected transition.

```
abstract class TimedAutomaton
  DoTransition (t as Transition)
    require IsTransitionEnabled(t)
    if t.blockTime then
      TimeBlocked := true
    else
      if t in syns then
        step
          let CHAN as SignalChannel = syns(t).First
          let MODE as SYNROLE = syns(t).Second
        step
          CHAN.Synchronize(me, t, MODE)
      else
        step
          FinishTransition(t)
```

The operational rule *DoTransition* of *TimedAutomaton* specifies the steps through which a system executes an enabled transition. We use the AsmL *require* construct to assert that this transition must be an enabled one. The semantic unit has two special transitions, the block time transition and the *setPrior* transition. Both of these two transitions are system priority related transitions, and neither of them changes the current location of a component. A block time transition has an implicit action to set the *TimeBlocked true* and has no other actions to do. A *setPrior* transition has one more implicit action that sets its component as a prior component. If a transition needs to synchronize with transitions in other components, the corresponding signal channel organizes synchronization and finishes remaining operations related to the execution of this transition. Otherwise, the operational rule *FinishTransition* is applied to finish the *DoTransition* operation.

```
abstract class TimedAutomaton
  FinishTransition (t as Transition)
    step DoAction (t)
    step
      choose l in locations where location.id = t.dstLocationID
        currentLocation := l
      ifnone
        error(t.id + " has no effective destination location.")
    step
      if exists t1 in outTransitions(currentLocation) where t.setPrior
        then
        me.prior := true
        TimeBlocked := true
      else
        me.prior := false
        TimeBlocked := false
```

The *FinishTransition* rule first executes actions attached to this transition. Next, makes the field *currentLocation* refer to the destination location of the transition. A *setPrior* transition must be enforced immediately after entering its source location to set the priority of its component to the top priority. This enables the owner component to claim that it is in the process of execution an atomic action and no other components and time progress can interrupt this process.

```
class RTSystem
  GetNextTransition() as (TimedAutomaton?, Transition?)
    choose c in components where c.prior
      return (c, c.GetEnabledTransition())
    ifnone
      let EC = {c | c in components where c.HasEnabledTransition()}
      choose c in EC where c.GetEnabledTransition().blockTime
        return (c, c.GetEnabledTransition())
      ifnone
        choose c in EC where not
          (exists c2 in EC where c2.Priority() > c.Priority())
          return (c, c.GetEnabledTransition())
        ifnone
          return (null, null)
```

The rule *GetNextTransition* of *RTSystem* describes the algorithm for the system to select a candidate transition for the next execution. It first looks for components that are in the top priority. If one exists, the subrule *GetEnabledTransition* of *TimedAutomaton* is

then applied to select an enabled transition from this component. If this component has no enabled transitions, *GetEnabledTransition* returns a *null* value. The system will be blocked indefinitely, since a component in the top priority blocks enabled transitions in other components as well as time progress. If no component is in the top priority, the rule chooses a transition with the highest priority from enabled transitions in all components. If no component has enabled transitions, the rule *GetNextTransition* returns a tuple whose two elements are both *null* value.

```
class SignalChannel
  Synchronize (ta as TimedAutomaton, t as Transition, m as SYNROLE)
  require IsEnabled()
    if (broadcast) then
      step forall e in senders + receivers
          (e.First).FinishTransition(e.Second)
          (r.First).FinishTransition(r.Second)
    else
      match m
        SEND:
          ta.FinishTransition(t)
          choose r in receivers
            (r.First).FinishTransition(r.Second)
        RECEIVE:
          ta.FinishTransition(t)
          choose s in senders
            (s.First).FinishTransition(s.Second)
```

The operational rule *Synchronize* specifies operations for a signal channel to organize synchronization among transitions. The subrule *IsEnabled* asserts that this signal channel must be enabled to fire. If the signal channel is a broadcasting channel, the rule synchronizes all transitions waiting for sending and receiving the signal. Otherwise, the signal channel randomly chooses a synchronization pair from its *senders* or *receivers* set. Here, the non-determinism is specified by using AsmL construct **choose**. The operational rule *FinishTransition* is utilized to do actions that are attached with the corresponding transition, and reset the current location of the component. Note that there

is no order for the execution of transitions that participate in synchronization. All of these transitions are executed simultaneously in a single abstract state machine update step.

TASU Metamodel Specification

The semantic anchoring tool suite as shown in Figure 3-3 assumes that model transformation between a TAML and TASU is defined in terms of their abstract syntax metamodels using the graph transformation language UMT and the GReAT tool. Consequently, we need to create an "interface" toward the semantic anchoring tool suite by defining a metamodel for TASU. Since the metamodeling language in MIC is UML/OCL (or MOF), the metamodel is simply the UML/OCL based representation of the TASU Abstract Data Model.

There are two approaches to specify a metamodel capturing the TASU Abstract Data Model. One approach is to define a metamodel that captures the abstract syntax of the generic AsmL data structures (Figure 3-5). The other approach is to construct a metamodel that captures only the exact syntax of the TASU Abstract Data Model. As it is discussed in Chapter 3, each approach has its advantages and disadvantages. Here, we adopt the second approach. The metamodel for TASU Abstract Data Model includes three paradigms as shown in Figure 5-3 and Figure 5-4. In GME, a paradigm sheet is a place where users can construct metamodels or models. A GME model typically contains parts, such as atoms and other models. A GME atom is an atomic part and can not have internal structure. As in Figure 5-3, the *System* is a root model, which contains four kinds of parts: *ComponentInstance*, *Chan*, *Declaration* and *TimedAutomaton*. The atom *ComponentInstance* models a component which is an instance of a timed automaton. Its

attribute *template* refers to a concrete *TimedAutomaton* model that defines the internal structure of this component. The atom *Chan* corresponds to the *SignalChannel* class in the TASU Abstract Data Model. The model *Declaration* and *TimedAutomaton* are GME references whose actual internal structures are defined in separate paradigm sheets. The *Declaration* model contains data and clock types. The *TimedAutomaton* captures the tuple structure of the corresponding class *TimedAutomaton* in TASU Abstract Data Model. It is easy to see the close correspondence between the two.

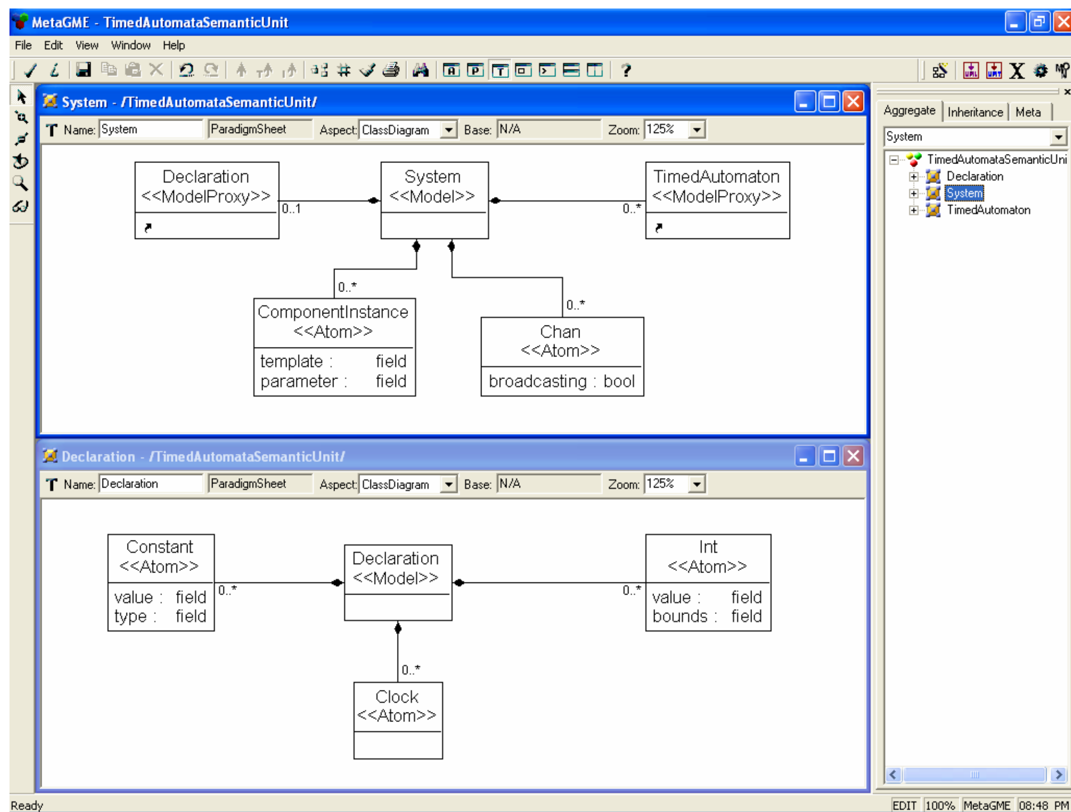

Figure 5-3 The *System* and *Declaration* paradigms of the TASU metamodel

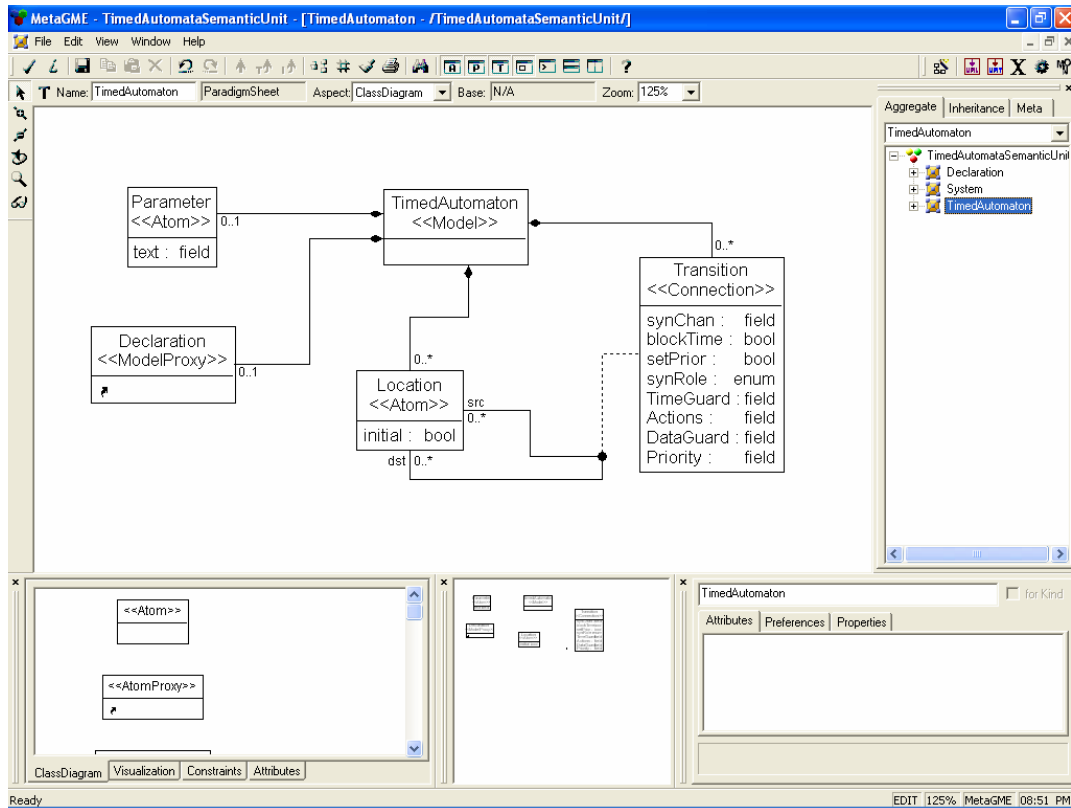

Figure 5-4 The *TimedAutomaton* paradigm of the TASU metamodel

In order to represent a TASU data model (an instance of the TASU Abstract Data Model) more intuitively, we also adds concrete syntax information in the metamodel so that a TASU model can have a graphical representation in the GME modeling environment. Note that a graphical representation for a semantic unit data model is only for the representation convenience in the rest of this chapter and is not necessary for the semantic anchoring methodology, since the graphical representation and the underlying XML representation are only different expressions for the same information.

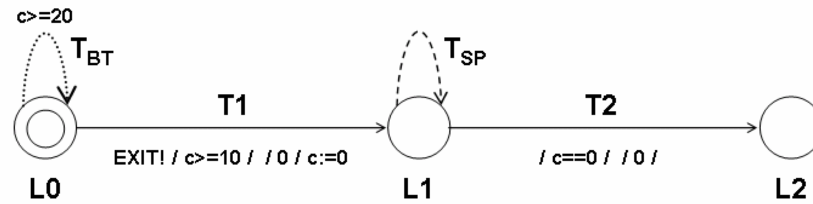

Figure 5-5 A TASU timed automaton (*ComponentKindA*)

```
class ComponentKindA extends TimedAutomaton
  L0  as Location   = new Location ("L0",true, {T1,TBT})
  L1  as Location   = new Location ("L1",false,{T2,TSP})
  L2  as Location   = new Location ("L2", false, {})
  T1  as Transition = new Transition ("T1",false,false,"L1")
  T2  as Transition = new Transition ("T2",false,false,"L2")
  TBT as Transition = new Transition ("TBT",true,false,"L0")
  TSP as Transition = new Transition ("TSP",false,true,"L1")
  c   as Clock      = new Clock ()
  override property locations as Set of Location
    get
      return {L0, L1, L2}
  override property transitions as Set of Transition
    get
      return {T1, T2, TBT, TSP}
  override property localClocks as Set of Clock
    get
      return {c}
  override property syns as Map of <Transition,(SignalChannel,SYNROLE)>
    get
      return { T1 -> (EXIT, SEND)}
  override TimeGuard (t as Transition) as Boolean
    match t.id
      "T1" : return c.time >= 10
      "T2" : return c.time == 0
      "TBT": return c.time >= 20
      _    : return true
  override DataGuard (t as Transition) as Boolean
    match t.id
      _   : return true
  override DoAction (t as Transition)
    match t.id
      "T1": c.time := 0
      _   : skip
  override Priority (t as Transition) as Integer
    match t.id
      _   : return 0
```

The semantic anchoring tool suite provides a translator, which translates TASU models built in the GME modeling environment into data models in native AsmL syntax. To illustrate this process, we show the visual representation of a simple TASU timed automaton as an example in Figure 5-3. After the translation, the equivalent AsmL specification is presented below the figure. The AsmL class *ComponentKindA* is a concrete timed automaton that overrides the abstract members of the abstract *TimedAutomaton* class defined the TASU Abstract Data Model.

In the TASU domain modeling environment, a normal transition is represented as a continuous direct-line. Information attached to a normal transition includes five segments ordered in sequence: a signal channel, a time guard, a data guard, priority and actions. The corresponding segment is left empty if a transition does not have that information. A block time transition (e.g. T_{BT} in Figure 6-5) is represented by a dot direct-line. Only a time guard can be attached to a block time transition. A dash direct-line is utilized to represent a *setPrior* transition, e.g. T_{SP} in Figure 5-5. No additional information might be put on a *setPrior* transition, since the information attached to a *setPrior* transition is predefined by the semantic unit and can not be modified by a component.

Semantic Anchoring to TASU

Semantic anchoring of a TAML means defining the transformation rules to TASU. In MIC, the transformation rules are specified in terms of the TAML and TASU metamodels. As it is shown in Figure 3-3, the GReAT tool can understand the transformational specification (which is the transformational semantics of the selected

TAML) and generate a model transformer, which can translate any legal TAML models to TASU models. The TASU models (as we showed it above) can then be translated to AsmL data models. It is AsmL data model plus the operational semantics specification that defines the semantics of the corresponding TAML model. In this section, we illustrate the semantic anchoring process using the two popular TAMLs, UPPAAL and IF languages. It must be noted that we do not need to verify the semantic equivalence between our specification and those used internally by the tools, since the semantics of TAMLs is defined by the TASU specification and the semantic anchoring rules. Some semantic anchoring rules are explained briefly in the rest of this section. The full specifications by using the semantic anchoring tool suite can be downloaded from [151].

Semantic Anchoring for the UPPAAL Language

We give an overview of the model transformation algorithm with a short explanation for selected role-blocks below. Figure 5-6 shows the specification of the top-level transformation rule in the GReAT tool. The transformation rule-set consists of the following steps:

1. Start by locating the system in the input UPPAAL model; create a System object in the TASU model and set its attributes values appropriately.
2. *Create Declaration*: Match the global clock variables, integer variables and constants in the input UPPAAL model; create the corresponding objects in the TASU model.
3. *Create Timed Automaton*: Match the Timed Automaton templates in the input UPPAAL model; create the corresponding TASU *TimedAutomaton* objects.

4. *Create Signal Channnel*: Match the signal channels in the input UPPAAL model; create the corresponding *Chan* objects in the TASU model and set the attributes values appropriately.
5. *Create Components*: Match the components in the input UPPAAL model; create the corresponding *ComponentInstance* objects in the TASU model and set the attributes values appropriately.

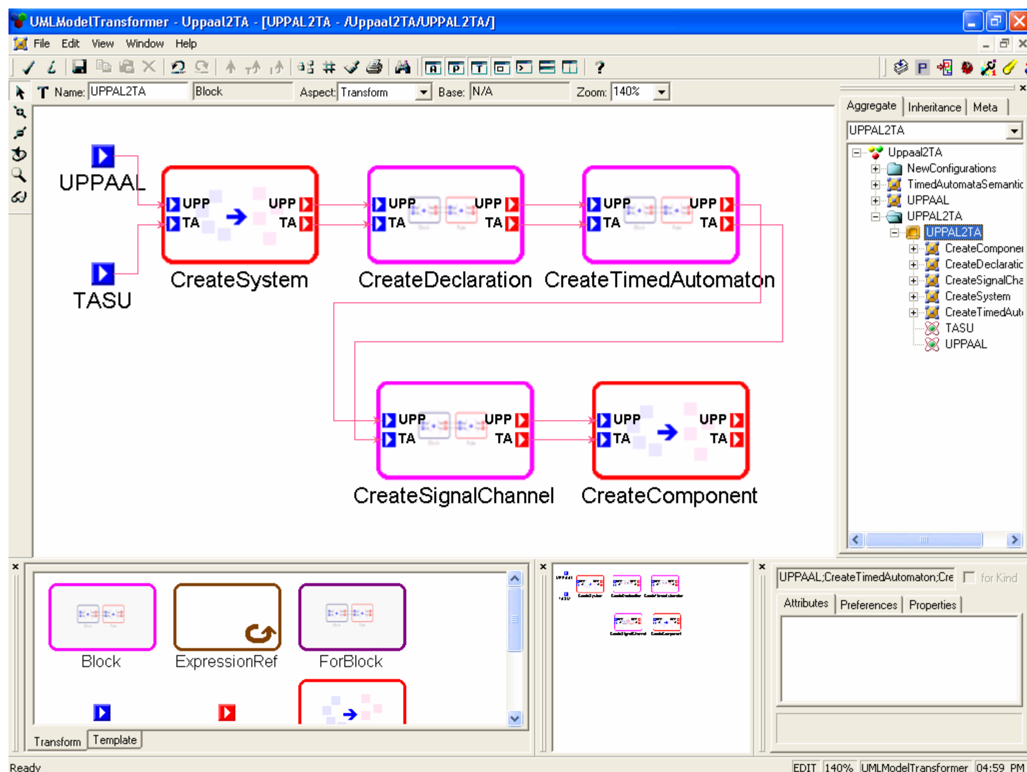

Figure 5-6 Top-level model transformation rule for the UPPAAL semantic anchoring specification

The third step is the key step in the transformation from a UPPAAL model to a TASU model. We give some explanations for sub-rules contained in this step. A trivial

one-to-one mapping can realize the mapping for those modeling constructs that are the same both in the UPPAAL automaton and in the TASU automaton.

Location invariants

The notion of location invariants was first introduced by the Timed Safety Automata [159], which is then adopted by the UPPAAL language. Time constraints put on locations are called *location invariants.* An automaton may remain in a location as long as the clock variables satisfy the invariant condition of that location. When the invariant condition is about to be violated by time progress, the automaton must be forced to leave this location.

A location with an invariant condition in UPPAAL is semantically equivalent to a location in TASU with a blocked time transition whose time guard is the critical condition of the invariant condition. The pseudo code for this transformation rule is

foreach location **in** a UPPAAL automaton
  create a new location **in** the corresponding TASU automaton
  set the attributes **in** the new location
  if the UPPAAL location has location invariants **then**
    create a block time transition **on** the new created TASU location
    set the time guard **in** the block time transition
  endif
  ……
end foreach.

Figure 5-7 shows a pattern graph, which is a part of the GReAT specification of the transformational rule for UPPAAL locations with location invariants. This pattern graph actually implements the pseudo code between **if** and **endif**. The *Guard* implements the **if** condition. If the *Guard* is true, a new block time transition is created (the blue-color *Transition* in the pattern graph). The *AttributeMapping* includes code that set the time guard in the new created transition object.

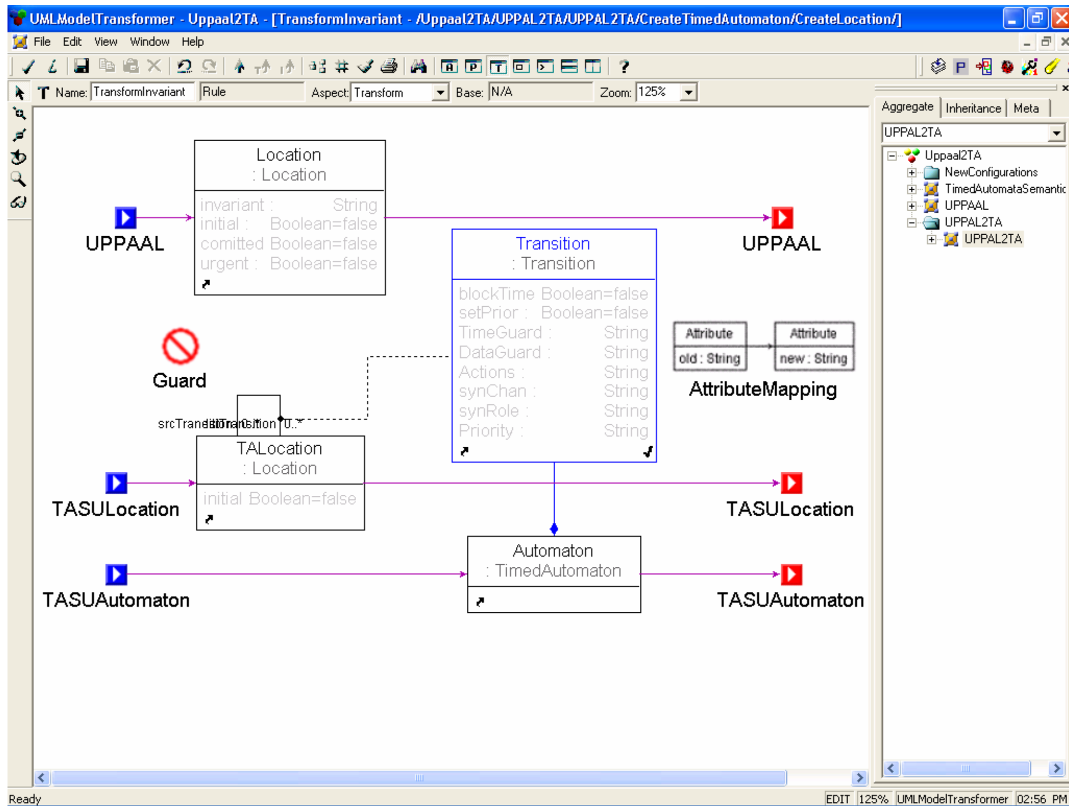

Figure 5-7 A pattern graph in the GReAT specification of
the transformational rule for UPPAAL locations with Location Invariants

An example shown in Figure 5-8 is presented to further illustrate this transformation rule. In the figure, a simple UPPAAL time automaton (Figure 5-8 (a)) whose *start* location has an invariant condition on the clock variable c can be mapped to an equivalent timed automaton in TASU (Figure 5-8 (b)). We briefly explain the behavior of the TASU automaton. Before the critical condition, c equals 5, is satisfied, the block time transition T_{BT} is not enabled and the automaton may stay in or leave the location *start*. If the automaton is still in the *start* location when the critical condition is reached, the block time transition T_{BT} is taken immediately. Now time is not allowed to progress until the automaton leaves the *start* location.

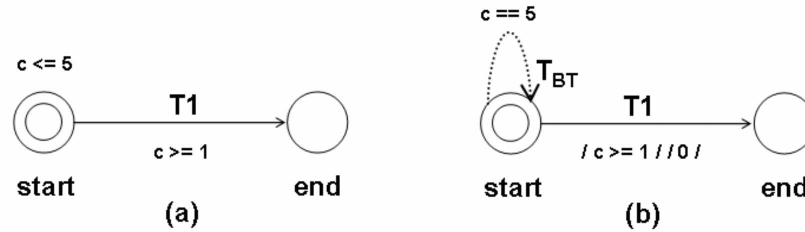

Figure 5-8 Semantic anchoring for a UPPAAL automaton with location invariants

For the rest of transformational rules, we will not present the pseudo code and the corresponding GReAT specification. Instead, only examples are used to explain the transformational rules. With examples and the corresponding explanations, it is easy for readers to derive the pseudo code. The completed GReAT specification can be downloaded from [151].

Urgent/Committed locations

There are three kinds of locations in UPPAAL that are normal locations with or without invariants, urgent locations and committed locations. Time may not pass in urgent or committed locations. However, urgent locations allow instantaneous interleaving with other components, while committed location does not. In UPPAAL, a location marked ∪ denotes an urgent location and the one marked C is committed.

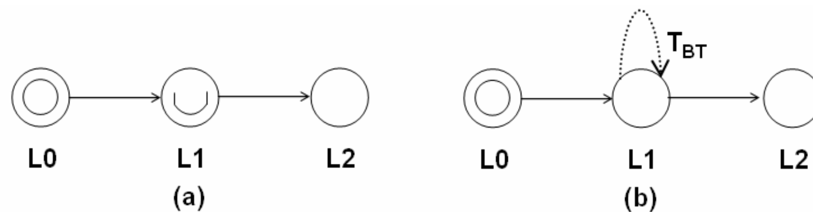

Figure 5-9 Semantic anchoring for a UPPAAL automaton with urgent locations

In the anchoring, an urgent location is mapped to a normal location plus a block time transition with no time guard. Figure 5-9 shows a simple example. In the figure, (a) is a UPPAAL automaton, and (b) is the anchoring automaton in TASU. In anchoring automaton, the time is blocked as long as the automaton stays in the location *L1*, but the enabled transitions in other automata (components) can be execute in this period. This is semantically equivalent to the UPPAAL automaton in Figure 5-9 (a).

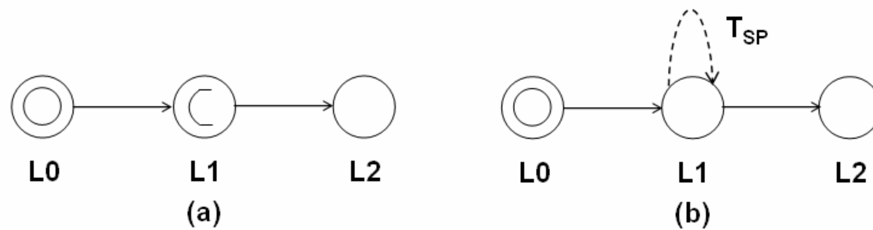

Figure 5-10 Semantic anchoring for a UPPAAL automaton with committed locations

An automaton in a committed location blocks both time progress as well as enabled transitions in all other automata. This functionality is equivalent to a *setPtior* transition in TASU, which sets the priority of the owner automaton to the top priority. With the execution of a *setPtior* transition, an automaton claims that it is in the process of executing an atomic action, and enabled transitions in all other automata as well as time progress should be prohibited until the atomic action is finished. As shown in Figure 5-10, (a) is a UPPAAL automaton and (b) is the semantic equivalent automaton in TASU after transformation.

Figure 5-11 Semantic anchoring for a UPPAAL automaton with urgent synchronization

Urgent synchronizations

A signal channel in UPPAAL may be declared as urgent. If a transition with an urgent channel is enabled, time delay must not occur before this transition is taken. TASU does not have the urgent channel concept, but the same effects can be achieved through setting the priority of a transition to be an integer greater than *zero* (depending on the relative priority with respect to other actions). Since the transition has higher priority than that of time progress, there will be no time delay before the execution of this transition if it is an enabled transition. Figure 5-11 shows an example to illustrate the transformation for urgent synchronization. Note that the signal channel *EXIT* in (a) is declared elsewhere as an urgent channel, while the one in (b) is only a normal channel in TASU.

Semantic Anchoring for the IF Language

The IF language is another well-known TAML to model asynchronous communicating real-time systems. In IF, a real-time system contains a set of processes, which are running in parallel and interacting through asynchronous signal routes. Each process is modeled by an extended timed automaton. This section is focused on the semantic anchoring for the IF specialized language constructs including: transitions with

three types of deadline, unstable states and asynchronous signal routes with different policies.

Lazy/Delayable/Eager transitions

The IF language does not support *location invariants* explicitly, but the same behavior can be achieved through utilizing transitions with different deadlines. An IF transition may have one of three types of deadlines (lazy, delayable and eager), which indicates the priority of a transition with respect to time progress. A lazy transition is never urgent and always allows time progress. An eager transition is urgent and prohibits time progress as soon as it is enabled. A delayable transition becomes urgent when it is about to be disabled by time progress and allows time progress otherwise. A lazy transition is equivalent to a normal transition in TASU with a priority value *zero* (the priority of time progress). The same behavior for an eager transition can be achieved by setting the priority of the corresponding transition in TASU to be an integer greater than *zero* (depending on the relative priority with respect to other actions).

A delayable transition implies that the priority of this transition jumps to a higher value than that of time progress when the enabling condition of this transition is about to be violated by time progress. An example for the transformation is shown in Figure 5-12. Note that the transition *T1* in (a) is an IF delayable transition while the one in (b) is a normal transition in TASU. We give a briefly explanation for the behavior of the IF automaton in (a). During $10 <= c < 20$ (where c is a clock variable), the transition *T1* is enabled but the system can randomly make choices on whether it takes this transition or advances time. When c reaches 20, the transition *T1* will be disabled by any further time progress. At this moment, the system should execute *T1* before advancing time. The

automaton in the Figure 5-12 (b) employs the expression *if c == 20 return1 else return 0* to specify the dynamic priority of this transition. The priority of the *T1* transition jumps to a higher value than the time progress priority (*zero*), which ensures *T1* to be executed as soon as *c* reaches *20*, if the automaton is still in the s*tart* location.

Figure 5-12 Semantic anchoring for an IF automaton with delayable transitions

Unstable locations

Unstable locations in IF have similar meaning to committed locations in UPPAAL. A process entering an unstable location must continue immediately by firing some transitions at that location and soon on, until a stable location will be reached. Unstable locations is an IF way to define an atomic action as a sequence of transitions from one stable location to another stable location. So the anchoring approach for an IF unstable location is also the same as that for a UPPAAL committed location. Like the approach shown in Figure 5-10, each unstable location in IF is mapped to a normal location with a *setPrior* transition in the semantic unit.

Asynchronous signal routes

The IF language imports a language construct, the signal route, to facilitate modeling the asynchronous communications among processes. Signal routes can be thought as specialized processes for the delivery of signals between normal processes. The behavior of signal routes is implicitly defined by a set of policies:

- *queueing policy* denotes how the message in transit are tackled by the signal route. Two options are available respectively, #fifo, order-preserving, using a queue-based storage, and #multiset, no order-preserving, using a multiset storage;
- *delivering policy* denotes how the messages are delivered. Three options are available here, respectively, #peer, which means delivery to the instance indicated in the output action using the **to** construct, #unicast, means delivery to one of the running instances and #multicast, means delivery to all running instances existing at the signal route endpoint;
- *reliability policy* has two options, #reliable, which means that message are not lost, or #lossy, which means that messages could be lost when transiting through the signal route;
- *delaying policy* denotes the delay associated with the signal route and can be respectively, #urgent, means immediate (0-time) delivery, #delay[a, b], means that any message entering the signal route will eventually leave it after *a* and not later than *b* units of time, and #rate[a, b], means that it takes between *a* and *b* units of time per message to be delivered by the signal route.

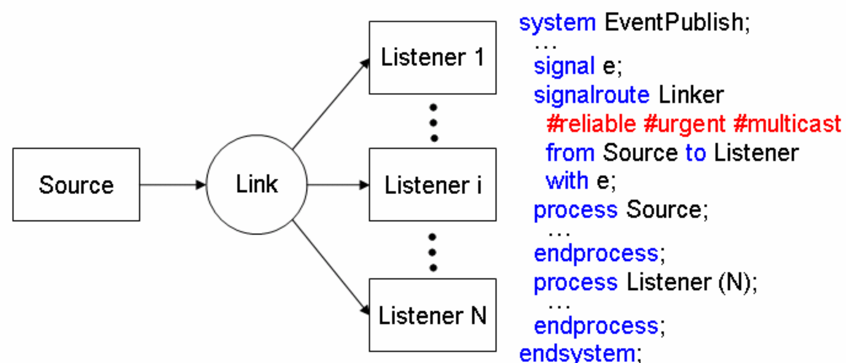

Figure 5-13 An IF asynchronous model with policies #reliable, #urgent and #multicast

To ensure analyzability, the TASU can only handle simple events. In particular, we assume that communication events are only signals and are not attached with data. In this situation, two queueing policy options, #fifo and #multiset, are equivalent with respect to our semantic unit. An IF asynchronous model, with other three different policies, will result in different anchoring models in TASU. However, the general ideas that guide the anchoring approaches for an IF asynchronous model with different policies are very close. We use an example to illustrate our anchoring approach for different policies. The left part of Figure 5-13 depicts an Event Publish system, in which a sender process multicasts events to N receiver processes through an asynchronous signal route. The right part of the Figure is the IF specification for this system. The sender process is defined by the process template *Source* and the receiver process is defined by the process template *Listener*. The sender and receivers are connected through a signal route *Link* whose transmission policies are set to #reliable, #urgent and #multicast. The detailed structures of the *Source* and *Listener* processes are not shown, since they do not affect the anchoring approach for the signal route.

During an asynchronous communication, the *Source* process first publishes an event and continues its following tasks without waiting. The signal route *Link* receives and buffers this event immediately. The buffered events will be delivered to target processes according to the pre-specified policies of the signal route. The option #reliable indicates that all events will be transmitted successfully without loss. The option #urgent denotes that the time required for a complete event transmission is *zero*. The option #multicast means that the signal route multicast events to all receiver processes.

Figure 5-14 presents the structure of the anchoring model in TASU. The signal route in IF is a black box that realizes the asynchronous communication among processes. In a TASU model, it is explicitly modeled as an automaton representing a Transmitter to transmit events plus a set of automata representing Buffers to buffer events for receivers. Figure 5-14 (a) displays the overall structure of the system. The Transmitter receives events from the *Source* process through the non-broadcasting signal channel *e*, and publishes events to N Buffers through the broadcasting signal channel *eR*. Each Buffer saves events and delivers them to the corresponding *Listener* process through the corresponding non-broadcasting signal channel eR_i. All signal channels in Figure 5-14 are synchronous channels in TASU.

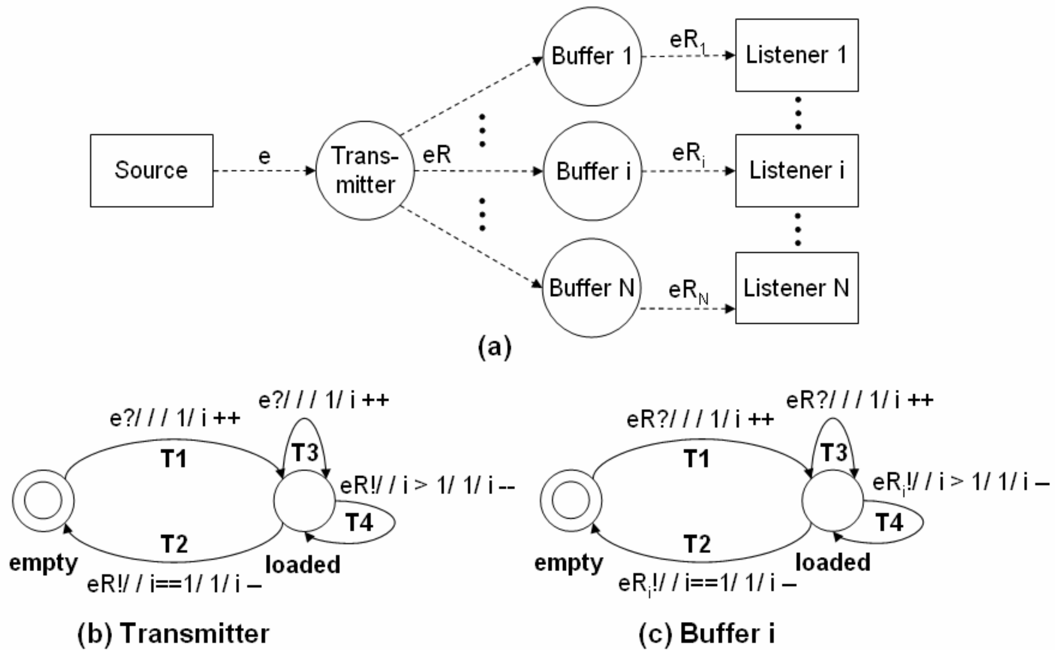

Figure 5-14 The semantic anchoring model for the IF asynchronous model in Figure 5-13

Figure 5-14 (b) presents the structure of the Transmitter automaton. The Transmitter starts from the *empty* location. When it receives an event, it moves to the *loaded* location. The Transmitter will stay in the *loaded* location, and execute the receiving (*T3*) or publishing (*T4*) events transition as long as it has at lest one buffered event after the transition. If it publishes its last buffered event, the Transmitter takes the *T2* transition and moves back to the *empty* location. The integer variable *i* is employed to record the number of buffered events. Figure 5-14 (c) shows the structure of the automaton representing the ith Buffer, which has the similar structure as the Transmitter automaton. The priority values of all transitions in both automata are set to *1*, which indicates that these transitions are urgent. So, there is no time delay during the event delivery.

In this report, we omit the verification for the semantic equivalence between the IF asynchronous model in Figure 5-13 and the TASU synchronous model in Figure 5-14. If the policies of an asynchronous model are changed, the synchronous model also needs to be modified to capture the changed behavior. In fact, there may have multiple solutions to capture the changed behavior. However, we give a simple solution as a clue for other possible approaches.

The behavior of the reliability policy #lossy can be achieved through adding transitions in the *loaded* location of the Transmitter, which randomly drop buffered events. If the signal channel *eR* is set as a non-broadcasting channel, the model in Figure 5-14 has the same behavior as the model in Figure 5-13 with the delivering policy set to #unicast, which means that an event is delivered to a randomly selected receiver process. If the delivery policy is set to #peer, the source process must specify a specific target

process to send events. In this case, the synchronous model has only one *Listener* process and one *Buffer* process. Therefore, it runs in the same manner as the corresponding asynchronous model applying the delivery policy #peer.

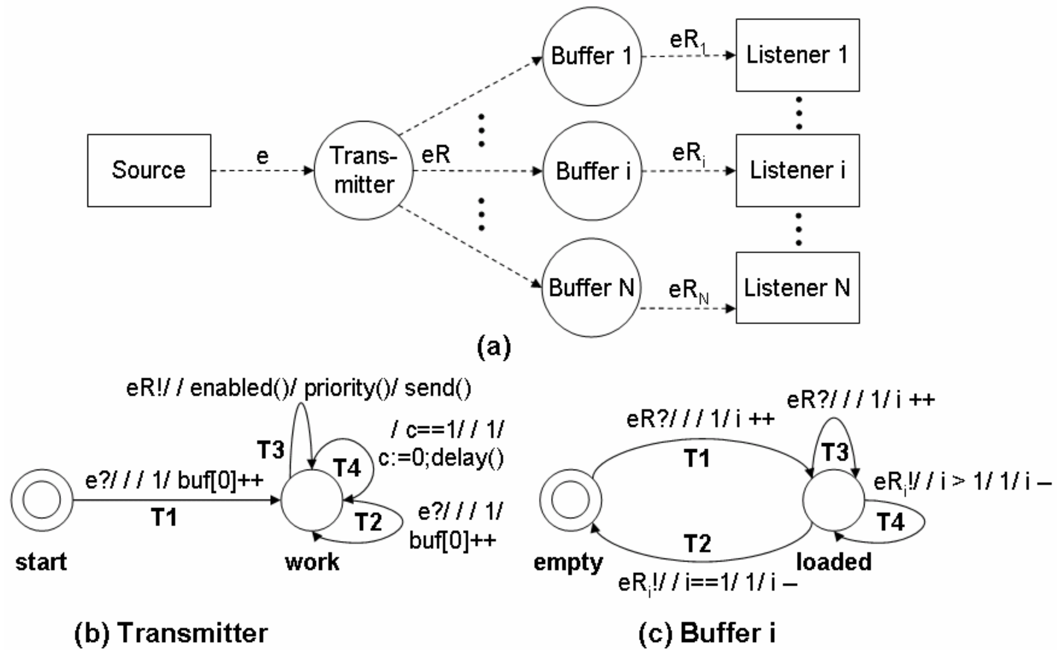

Figure 5-15 The semantic anchoring model for the IF asynchronous model in Figure 5-13 with the delaying policies changed from #urgent to #delay[a, b]

The delaying policy #delay[a, b] can be achieved by adding local clock variables to measure the buffered time for each received event, and adding time conditions and dynamic priorities on the publishing events transitions of Transmitter to control the delay for the event delivering. Figure 5-15 shows the structure of the anchoring model for the IF asynchronous model in Figure 5-13 with the delaying policies changed from #urgent to #delay[a, b]. Comparing Figure 5-15 with Figure 5-14, only the Transmitter automaton is changed to reflect the modification of the delivering policy. The Transmitter automaton

defines a clock variable *c* and an integer array *buf* whose size is *b*. The *buf[i]* (*i* is between zero and *b-1*) records the number of events that have been delayed for *i* time units.

Initially, the Transmitter automaton starts from the location *start* and the clock variable *c* and all elements in the *buf* array are zero. When an event comes, the transition *T1* is taken immediately and *buf[0]* is set to *1*. The Transmitter automaton will always stay in the *work* location after the execution of the *T1* transition. If more events come, the transition *T2* will be taken immediately which increases the value of *buf[0]*. Whenever the time progress for one unit, the transition *T4* will be enabled and should be taken immediately, which reset the clock variable *c* and do function *delay()*. The function *delay()* increases the delay time for all buffered events by one time unit, which sequentially sets the value of *buf[i]* to the value of *buf[i-1]*, where *i* moves from *b-1* to *1*, and the set *buf[0]* to zero. The transition *T3* is a crucial one that controls the event delivery. The function *enabled()* (**if exist** *i*, **where** $a-1 \leq i \leq b-1$, *buf[i]* > *0* **then return** *true* **else return** *false*) checks whether there exist events that have been buffered for more than *a* time units. The priority of transition *T4* is a dynamic value set by the function *priority()* (**if** *buf[b-1]* > 0 **then return** *1* **else return** *0*). This means that time progress should be blocked if there are events that have been buffered for *b* time units. The *send()* function randomly chooses an *i*, where $a-1 \leq i \leq b-1$ and *buf[i]* > *0*, and reduce the value of *buf[i]* by one.

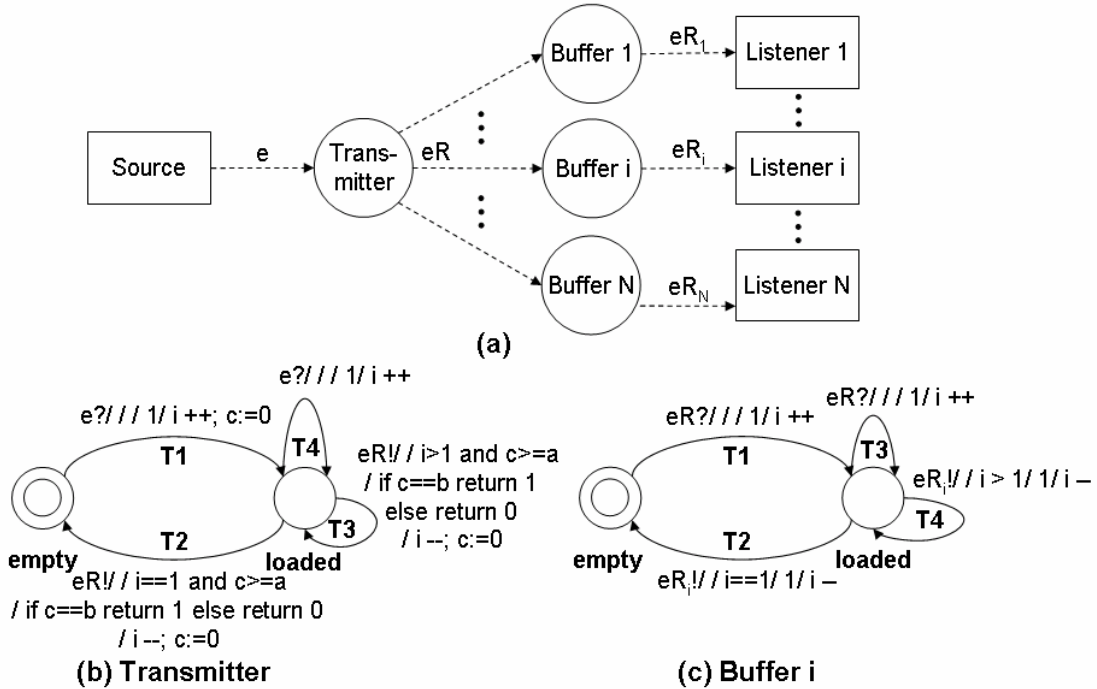

Figure 5-16 The semantic anchoring model for the IF asynchronous model
in Figure 5-13 with the delaying policies changed from #urgent to #rate[a, b]

Likewise, the delaying policy # rate[a, b] can be captured by adding a local clock variable to measure the time passing for the Transmitter automaton, and setting time guards and dynamic priorities on the publishing events transitions to control the rate of the event delivering. Figure 5-16 shows the semantic anchoring model for the IF asynchronous model in Figure 5-13 with the delaying policies changed from #urgent to #rate[a, b]. The Transmitter automaton defines a clock variable *c* to control the rate of the event delivery.

CHAPTER VI

SEMANTIC UNIT COMPOSITION

In Chapter 3, we state that a DSML may have multiple behavioral aspects that are associated with different behavioral categories. In this case, the semantics of a DSML can be specified as the composition of multiple semantic units. Semantic unit composition reduces the required effort from DSML designers and improves the quality of the specification. In this chapter, we discuss the general rules that can guide semantic unit composition. An industrial-strength modeling language, EFSM (Extended Finite State Machine Language), is used as a case study to illustrate the compositional semantics specification.

Compositional Specification of Behavioral Semantics

In the semantic anchoring infrastructure, we define a finite set of *semantic units*, which capture the semantics of basic behavioral and interaction categories. If the semantics of a DSML can be directly mapped onto one of these basic categories, its semantics can be defined by simply specifying the model transformation rules between the DSML and the Abstract Data Model of the semantic unit. However, in heterogeneous systems, the semantics is not always fully captured by a predefined semantic unit. If the semantics is specified from scratch (which is the typical solution if it is done at all) it is not only expensive but we loose the advantages of anchoring the semantics to (a set of) common and well-established semantic units. This is not only loosing reusability of

previous efforts, but has negative consequences on our ability to relate semantics of DSMLs to each other and to guide language designers to use well understood and safe behavioral and interaction semantic “building blocks” as well.

Our proposed solution is to define semantics for heterogeneous DSMLs compositionally. If the composed semantics specifies a behavior which is frequently used in system design, (for example composition of SDF interaction semantics with FSM behavioral semantics defines semantics for modeling signal processing systems [162]) the resulting semantics can be considered a *derived semantic unit*, which is built on *primary semantic units*, and could be offered up as one of the set of semantic units for future anchoring efforts. The composition method we describe in the rest of the paper is strongly influenced by Gossler and Sifakis framework for composition [163] and has commonalities with composition approaches used in Ptolemy [162] and Metropolis [164] by clearly separating behavior and interaction. In the following we provide a brief overview of the composition approach that will be followed by a detailed case study.

Mathematically, a composed semantics is represented as a tuple $CS = <A, R>$. Similarly to [163], we model semantic unit composition as *structural* and *behavioral compositions* (see Figure 6-1). In the Figure, we represented ASM instances that include an m data model, the R rule set and the S dynamic state variables updated during runs. The structural composition defines relationships among selected elements of Abstract Data Models using partial maps. In Figure 6-1, we demonstrate semantic composition with two semantic units, SU1 and SU2. The structural composition yields the composed Abstract Data Model $A = <A_C, A_{SU1}, A_{SU2}, g_1, g_2>$, where g_1, g_2 are the partial maps between concepts in A_C, A_{SU1}, and A_{SU2}.

Behavioral composition is completed by the R_C set of rules that together with R_{SU1} and R_{SU2} form the R rule set for the composed semantics. The role of the R_C set of rules is to receive the possible sets of actions that can be offered by the embedded semantic units using the `Get(…)` calls, to restrict these sets according to the interactions created by the structural composition and to send back selected subset of actions through the `Run(…)` call to complete their next step. The executable actions are represented as partial orders above the set of actions. (This will be shown in detail in the following Sections.)

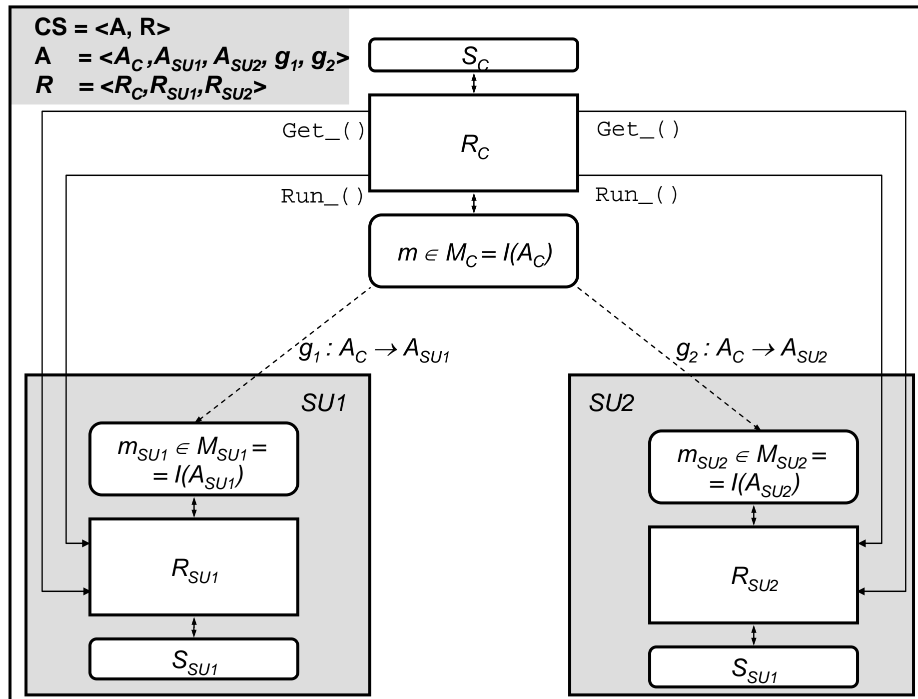

Figure 6-1 A graphical representation for semantic unit composition

Remark: The behavioral composition specifies a controller, which restricts the executions of actions. Since the behavior of the embedded semantic units can be

described as partial orders on the sets of actions they can perform, the behavioral composition can be modeled mathematically as a composition of the partial orders.

In the rest of this paper, we first describe a simplified version of EFSM, called SEFSM, which only includes the modeling constructs that determine the core behavioral semantics of EFSM. Then, we apply structural and behavioral composition for two primary semantic units, Finite State Machine (FSM-SU) and Synchronous Dataflow (SDF-SU), to define the semantics of SEFSM. As the first step in the composition sequence, we define semantics for Action Automata (AA). We use the semantics of AA as a new derived semantic unit (AA-SU) that we further compose with SDF-SU to SEFSM as the composition of individual components whose semantics are defined by obtaining the semantics for EFSM. Due to space limitations, we need to omit many details of the specification. The full semantics specifications can be downloaded from [151].

SEFSM Overview

EFSM has been developed by General Motors Research to specify vehicle motion control (VMC) software [165]. In order to satisfy the requirements of the VMC domain, EFSM provides a narrow and precisely-defined set of modeling constructs that can represent concurrent FSMs, mathematical functions, data types, physical units, value ranges, and a hierarchical signal and event type structure. Because the semantic anchoring methodology focuses on the behavioral semantics of a DSML, many modeling constructs in EFSM, such as those related to type structures, physical units and value ranges, have little influence on the behavioral semantics specification. Hence, we

introduce a simplified version of EFSM, called SEFSM, which only includes those modeling constructs in EFSM that determine the core behavioral semantics of EFSM.

A SEFSM model is a synchronous reactive system including a set of components communicating through event channels and data channels. The connections do not form event and data propagation loops. Global states are considered as delay variables that may be read and updated during reactions. In each computation cycle, a SEFSM system is first activated by an incoming event; this event is then propagated through event channels and activates internal components; the reaction of internal components may produce additional events; new generated events will continue the propagation and activation cycle until conclusion. According to the synchrony assumption, a computation cycle will be finished before the next incoming event triggers a new reaction.

A SEFSM model integrates a set of stateless *computational functions* $t = f(t_1, \ldots, t_n)$ that consume input data and produce output data. SEFSM separates events from data as they are for different purposes. Events determine which components are to be activated and the order of activations. An incoming event, while activating a component, also affects the decision on which functions within that component are to be executed. All input data required by the functions to be executed should be available already when the owner component is activated.

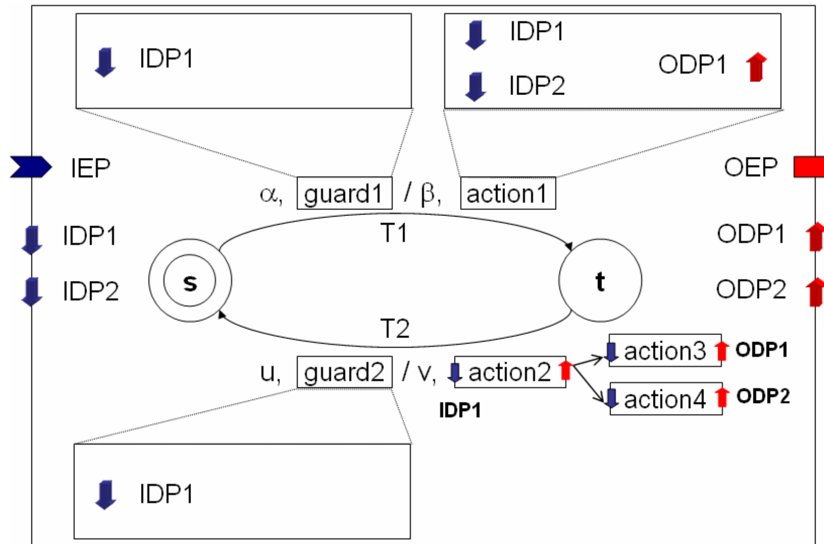

Figure 6-2 A simple SEFSM component model

A SEFSM component is an FSM-based model. We use a simple component model shown in Figure 6-2 as an example to explain the structure and the behavior of SEFSM components. The component communicates with other components through *ports*, including a single *input event port* (IEP), an *output event port* (OEP), two *input data ports* (IDP1 and IDP2) and two *output data ports* (ODP1 and ODP2). As shown in the figure, the component includes an FSM, where *transitions* are labeled with a *trigger event*, a *guard*, an *output event* and set of *actions*. Guards and actions are *computational functions* within the component and receive their input data through *input data ports.* The execution of an action (a function) may produce new data, while the execution of a guard only returns a Boolean value for the true or false evaluation. Therefore, an action has a set of *output data ports* while a guard does not.

Whenever a component receives an event, it consumes the event and evaluates which *transition* is enabled. A transition is enabled if its *source state* is the *current state*,

its *trigger event* matches the *incoming event* and the evaluation of its guard function returns true. For safety reasons, EFSM intentionally prohibits non-determinism. If the enabled transition is labeled with an *output event* and *actions*, the component generates the output event and executes the actions, which may produce output data. The new created event and data are stored in the corresponding event and data ports. Note that the output event and output data of a component should be delivered to destination components simultaneously and the delivery process takes logical (per the synchrony assumption) zero time.

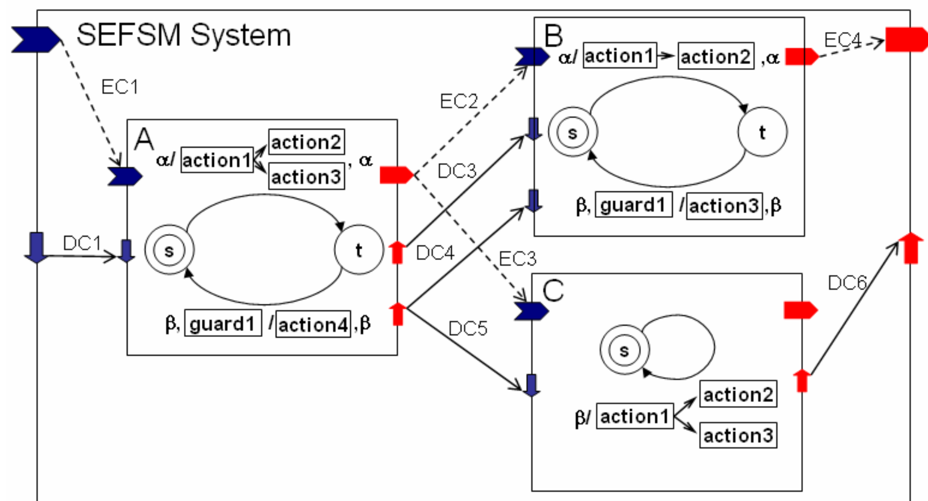

Figure 6-3 A simple SEFSM system model

A SEFSM system consists of a set of components, event channels, data channels, an input and an output event port, and a set of input and output data ports. To illustrate this, Figure 6-3 presents a simple SEFSM system model, including three components A, B and C. Event channels are represented as dashed lines and data channels are shown as concrete lines. Multiple event channels can be connected to the same output event port,

but only one event channel can be connected to an input event port. This restriction eliminates the possibility that a component may receive multiple events during one reaction. A data port is also not allowed to receive multiple data since this will cause a non-deterministic decision on which data is to be used in the computation. However, data channels are allowed to merge, if at most one of the merged data channels actually delivers data in one computation cycle. When a component is activated by an event, some of its input data ports may be empty. However, those data ports that provide data for evaluating guards and for executing actions must contain data.

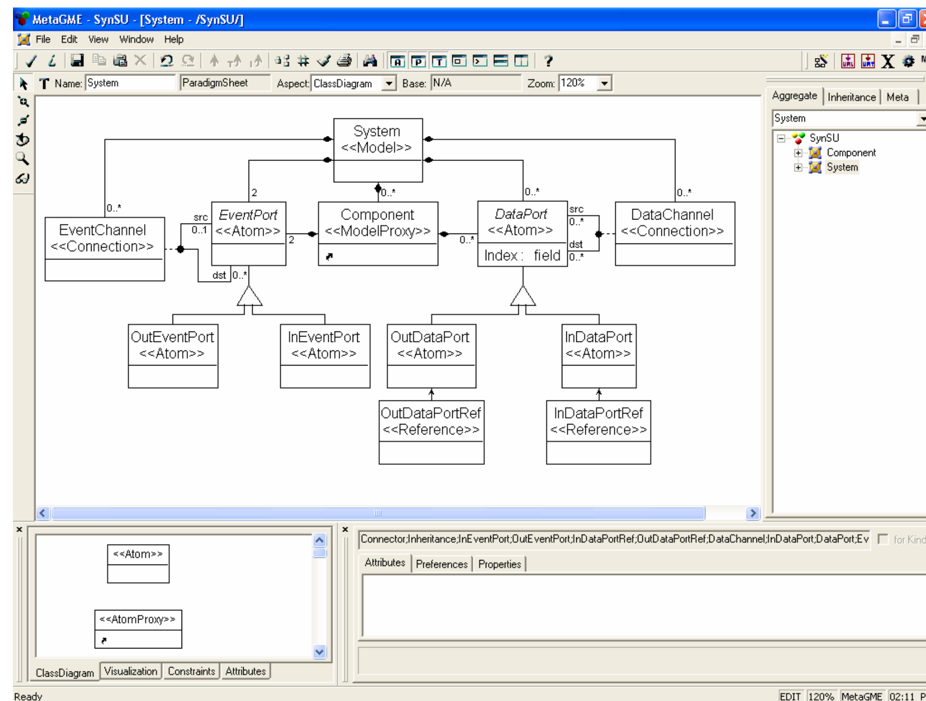

Figure 6-4 A paradigm in the SEFSM metamodel defining the system structure

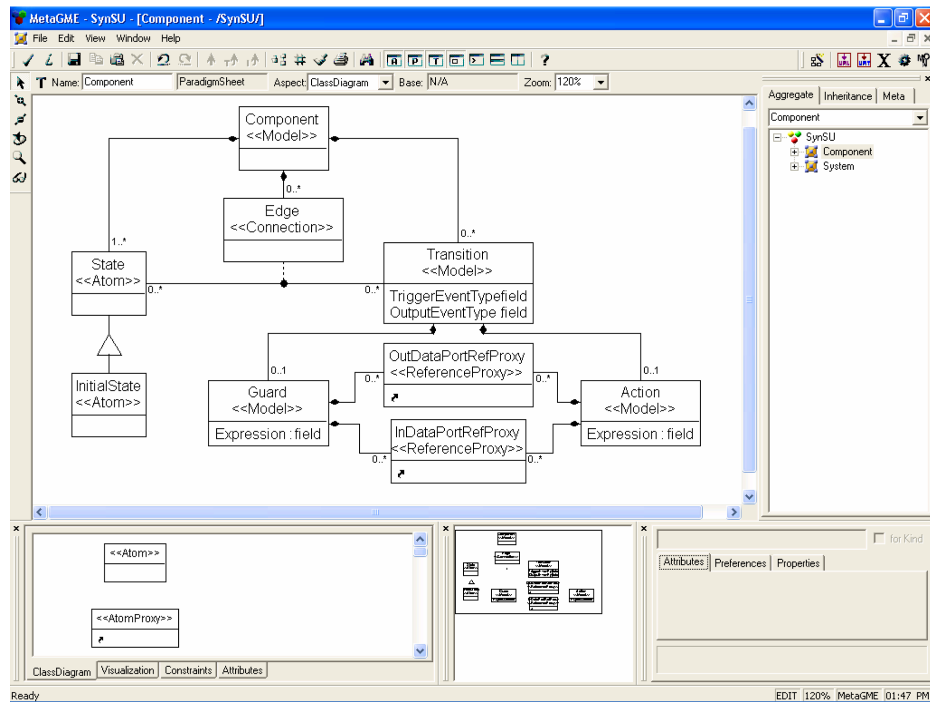

Figure 6-5 A paradigm in the SEFSM metamodel defining the component structure

The abstract syntax of SEFSM is defined by a UML/OCL-based metamodel, including two paradigms as shown in Figure 6-4 and 6-5. The paradigm in Figure 6-4 defines the system-level structure and the one in Figure 6-5 defines the component-level structure. Note that the *InDataPortRefProxy* and *OutDataPortRefProxy* classes in Figure 6-5 refer to the *InDataPort* and *OutDataPort* classes in Figure 6-4, respectively. In this way, data ports in a Guard or an Action are enforced to refer to the corresponding data ports defined in the component.

Figures 5 and 6 show the abstract syntax metamodel of SEFSM in MetaGME [9] (a UML/OCL-based metamodeling language). The metamodel in Figure 5 and 6 define the sub-language for representing the system-level structure and for the component-level structure, respectively. A set of OCL (Object Constraint Language) constraints [55] are

added to the SEFSM metamodel to specify well-formedness rules for the models. For example, the OCL constrain,

```
self.connectedAs("dst")->size()=1 and
                              self.connectedAs("src")->size()=1,
```

is attached to the *Transition* class in Figure 6, which claims a transition object should have a single source state and a single destination state.

It is easy to see that the abstract syntax metamodels and the textual description of the behavior are insufficient for the precise understanding of the semantics of SEFSM. For example, the metamodel specification does not reveal the complex interdependency between the event flow and the data flow structure of the components that both define partial orders for the evaluation of guards and execution of actions.

Primary Semantic Units Used

In the following section we briefly elaborate the primary semantic units FSM-SU and SDF-SU that we use to compose the semantics of, first SEFSM Components, and then SEFSM Systems. We describe these semantics unit here directly with their AsmL specifications as they are readable and executable.

Finite State Machine Semantic Unit

The Finite State Machine Semantic Unit (FSM-SU) defines the behavioral semantics of the basic non-deterministic FSM, which can be mathematically defined as a 5-tuple

$$\langle S, \Sigma, \Delta, \sigma, s_0 \rangle$$

where

- S is a finite set of states;
- Σ is an input alphabet, consisting of a set of input symbols;
- Δ is an output alphabet, consisting of a set of output symbols;
- $\sigma \subseteq S \times \sum \times \Delta \times S$ is a set of transitions;
- $s_0 \in S$ denotes the initial state.

The specification contains two parts as we mentioned earlier: an Abstract Data Model $A_{FSM\text{-}SU}$ and Operations and Transformation Rules $R_{FSM\text{-}SU}$ on the data structures defined in A. The AsmL abstract class *FSM* prescribes the top-level structure of a FSM, including a set of states, transitions, relationships between states and transitions, and relationships between transitions and event types. All the abstract members of *FSM* are further specified by a concrete FSM, which is an instance of the Abstract State Model.

```
structure Event
  eventType  as String
class State
  id         as String
  initial    as Boolean
  var active as Boolean = false
class Transition
  id as String
abstract class FSM
  id as String
  abstract property states         as Set of State
    get
  abstract property transitions    as Set of Transition
    get
  abstract property outTransitions as Map of <State, Set of Transition>
    get
  abstract property dstState         as Map of <Transition, State>
    get
  abstract property triggerEventType as Map of <Transition, String>
    get
  abstract property outputEventType  as Map of <Transition, String>
    get
```

The operational semantics of FSM-SU is specified as a set of AsmL rules. Two rules that are important in behavioral composition are briefly explained here. The rule

Run specifies the top-level system reaction of a FSM when it receives an event. Note that the '?' modifier after *Event* means the return from the *Run* rule may be either an event or an AsmL *null* value.

```
abstract class FSM
  React (e as Event) as Event?
    step
      let CS as State = GetCurrentState ()
    step
      let enabledTs as Set of Transition = {t | t in outTransitions
(CS) where e.eventType = triggerEventType(t)}
    step
      if Size (enabledTs) = 1 then
        choose t in enabledTs
          step
            CS.active := false
          step
            dstState(t).active := true
          step
            if t in me.outputEventType then
              return Event(outputEventType(t))
            else
              return null
      else
        if Size(enabledTs) > 1 then
          error ("NON-DETERMINISM ERROR!")
        else
          return null
```

The operational rule *GetCurrentState* returns the current state of a FSM. A state is considered as the current state if it is active. If a FSM has multiple active states, the rule reports an error. If it has no active state, the initial state is considered as the current state.

```
abstract class FSM
  GetCurrentState () as State
    step
      let currents = {s | s in me.states where s.active}
    step
      if Size (currents) > 1 then
        error ("FSM has multiple active states")
      else
        if Size (currents) = 0 then
          return GetInitialState ()
        else
          choose s in currents
            return s
```

Synchronous Dataflow Semantic Unit

The Synchronous Dataflow Semantic Unit (SDF-SU) defines the behavioral semantics of the Synchronous Dataflow (SDF) that can be mathematically expressed as a 5-tuple

$$\langle N, P, C, f_{ip}, f_{op} \rangle$$

where:

- N is a finite set of nodes;
- P is a finite set of ports;
- $C \subseteq P \times P$ is a finite set of channels;
- $f_{ip} : N \to 2^P$ is a map that assigns each node to its input ports;
- $f_{op} : N \to 2^P$ is a map that assigns each node to its output ports.

The AsmL specification of the Abstract Data Model $A_{SDF\text{-}SU}$ is shown below. *Token* is defined as an AsmL structure to package data. (We included only three types of data (integer, double and Boolean) in the specification using the AsmL construct *case*.) *Port* and *Channel* are defined as first-class types. The Boolean attribute *exist* of a port indicates whether the port has a valid data token. When all the input ports of a node have

valid data tokens, the node is enabled to fire. In the semantics specification, *Fire* is an abstract function. A concrete node will override the abstract function *Fire* with a computational function. The AsmL abstract class *SDF* captures the top-level structure of a model. The abstract property *inputPorts* contains a sequence of the SDF model's input ports that does not belong to any internal nodes. The abstract property *outputPorts* expresses the similar meaning.

```
structure Value
  case IntValue
    v as Integer
  case DoubleValue
    v as Double
  case BoolValue
    v as Boolean

//Data Token, it may contain a value or a null data
structure Token
  value as Value?

//Data Port, when exist is true, the port has an effective data token
class Port
  id        as String
  var token as Token   = Token (null)
  var exist as Boolean = false
class Channel
  id as String
  srcPort as Port
  dstPort as Port

abstract class Node
  id as String
  abstract property inputPorts  as Seq of Port
    get
  abstract property outputPorts as Seq of Port
    get
  abstract Fire ()
```

```
abstract class SDF
  id as String
  abstract property nodes       as Set of Node
    get
  abstract property channels    as Set of Channel
    get
  abstract property inputPorts  as Seq of Port
    get
  abstract property outputPorts as Seq of Port
    get
```

Two key operational rules in the $R_{SDF\text{-}SU}$ specification are explained here. The operational rule *GetEnabledNode* returns a set of nodes in a SDF model that are ready to fire.

```
abstract class SDF
  GetEnabledNodes () as Set of Node
    return {n | n in me.nodes where forall p in n.inputPorts where
p.exist}
```

The operational rule *Fire* specifies the behaviors that a SDF model takes to fire a node. The AsmL construct *require* asserts that the node to fire should be an enabled one. The *Fire* function of *Node* is defined by the node itself. However, it should consume the data tokens in all input ports of the node and produce data tokens to all output ports of the node. Otherwise, the rule will report an error. If an output port of the node is connected with multiple channels, the data token in it will be duplicated and propagated along all these channels. If the destination port of a channel already has an effective data token, the rule will report a non-deterministic error since it does not know which data token should be placed in the port.

```
abstract class SDF
  Fire (n as Node)
    require n in me.GetEnabledNodes ()
    step
      n.Fire ()
    step
      if exists p in n.inputPorts where p.exist then
        error ("After the firing of a node, all input tokens should be
consumed by the node.")
    step
      if exists p in n.outputPorts where not p.exist then
        error ("After the firing of a node, each of its output port
should have one effective token.")
    step
      forall c in me.channels where c.srcPort in n.outputPorts
        if c.dstPort.exist then
          error ("Non-deterministic error.")
        else
          c.dstPort.token := c.srcPort.token
          c.dstPort.exist := true
        c.srcPort.exist := false
```

The operational rule *Run* specifies the steps it takes to execute a set of nodes. This rule can be considered as a composition interface for SDF-SU. In the beginning, some of the nodes in the set may not be enabled, but they are supposed to be enabled by the execution of already enabled ones. The rule non-deterministically chooses an enabled node from the set and fires it. The execution of a node consumes the data tokens in all input ports of the node and produce them to all output ports as well. The operational rule *Fire* executes the node and propagates data tokens produced by the execution of the node through all the connected channels. The rule *Run* reports error if there are no enabled nodes in the set while the set is not empty.

```
abstract class SDF
  Run (ns as Set of Node)
    step while Size(ns) <> 0
      choose n in ns where n in GetEnabledNodes ()
        remove n from ns
        Fire (n)
      ifnone
        error ("Some Nodes are not enabled to fire.")
```

Compositional Semantics Specification for SEFSM Components

As we described before, the behavior of individual SEFSM components can be divided into two different behavioral aspects: the FSM-based behavior expressing reactions to events and the SDF-based behavior controlling the execution of computational functions (actions and guards). In this section, we formally specify the behavioral semantics of SEFSM components as composition of two primary semantic units: FSM-SU and SDF-SU. The compositional semantics specification consists of two parts: (1) an Abstract Data Model defining the structural composition $<A_C, A_{FSM\text{-}SU}, A_{SDF\text{-}SU}, g_1, g_2>$, where $g_1: A_C \rightarrow A_{FSM\text{-}SU}$, and $g_2: A_C \rightarrow A_{SDF\text{-}SU}$ are structural relation maps; and (2) Operations and Transformation Rules specifying the behavioral composition $<R_C, R_{FSM\text{-}SU}, R_{SDF\text{-}SU}>$.

Structural Composition

The structural composition defines mapping from elements in the Abstract Data Model of the composed semantic unit to elements in the FSM-SU model and those in the SDF-SU model. Figure 6-6 shows the role of the FSM-SU and SDF-SU in the SEFSM component model by restructuring the example in Figure 6-2. In the modified structure, the FSM model controls the event-related behaviors, while the SDF model takes charge

of the data-related computations. Comparing Figure 6-2 and 6-6, we can find that the overall structure of the FSM model closely matches that of the original SEFSM component, except for events, guards and actions. The trigger events and the output events in the FSM model are renamed. The guards and actions are represented as nodes in the SDF model. The relationships between the FSM model and the SDF model are specified by two maps: *GuardMap* and *ActionMap*. In this section, we only briefly explain how these two maps help to relate the FSM model with the SDF model. More details will be introduced in the following behavioral composition section.

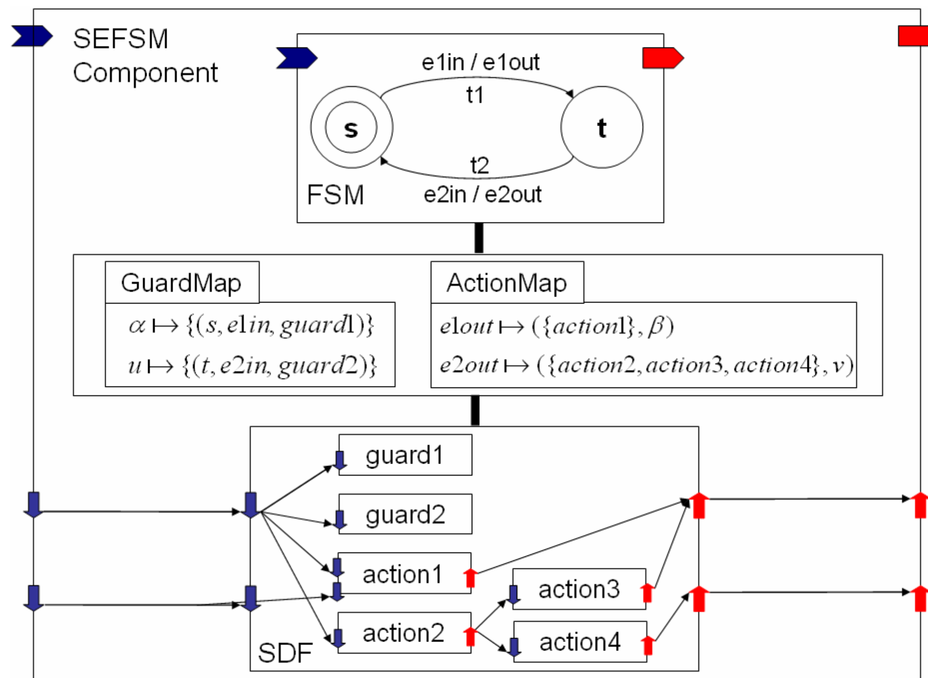

Figure 6-6 The compositional structure of the SEFSM component originally shown in Figure 6-2

The new compositional structure is built in a way that each transition in the original component is decomposed into three parts: a transition in the FSM model, a node

representing the guard and a node representing the action in the SDF model. In the original component, a transition can be unambiguously located by the combination of the source state, the trigger event, and the guard. In the compositional structure, the information can be expressed by a 3-tuple *(s, e, n)*, where *s* refers a state in the FSM model; *e* is a local trigger event in the FSM model; and *n* represents a node in the SDF model. When a component receives an event, this event is a global event and will not be directly forwarded to the FSM model. The *GuardMap* maps this global event to a set of 3-tuples, each tuple referring to a transition in the original component whose trigger event matches this global event. Using the example in Figure 6-2 again, the event α is the trigger event only for the transition *T1*. In the compositional structure as shown in Figure 6-6, the *T1* transition is decomposed into the *t1* transition in the FSM model, whose source state is *s* and trigger event is *e1in*, and the *guard1* and *action1* node in the SDF model. As a result, *GuardMap* assigns the event α to the set *{(s, e1in, guard1)}*.

```
class EventPort
  id         as String
  var evnt   as Event  = Event ("")
  var exist  as Boolean = false
abstract class Component
  id      as String
  abstract property inPort as EventPort
    get
  abstract property outPort as EventPort
    get
  abstract property GuardMap as Map of <String, Set of(String, String, Node?)>
    get
  abstract property ActionMap as Map of <String, (Set of Node, String?)>
    get
  abstract property fsm as FSM
    get
  abstract property sdf as SDF
    get
```

Behavioral Composition

In essence, the behavioral composition specifies the rules R_C, which is akin to a component-level controller (or scheduler) that orchestrates the executions and interactions of the FSM model and the SDF model.

The execution of a transition in the original (SEFSM) component can be decomposed into a three-step process: (1) the evaluation of the guard functions for all outgoing transitions from the current state as nodes in the SDF model; (2) selection of an enabled transition in the FSM model; and (3) the execution of actions of the transition as nodes in the SDF model. The three steps are related to each other by the maps *GuardMap* and *ActionMap*. The output event produced by the execution of a transition in the FSM model is a local event. *ActionMap* maps it to a 2-tuple *({n}, e)*, where *{n}* refers to a set of nodes (actions) in the SDF model and *e* refers to a global output event that will be propagated out of the component. For instance, the execution of the *t2* transition of the FSM model in Figure 6-6 generates a local event *e2out*. As the *t2* transition corresponds to the *T2* transition in the original component (Figure 6-2), which is attached with actions: *action2*, *action3* and *action4*, and an output event *v*, the *ActionMap* maps the local event *e1out* to a 2-tuple *({action2, action3, action4}, v)* accordingly.

The rules verbalized above are specified in AsmL as Operation and Transition Rules. The operational rule *Run* of *Component* specifies the top-level component operations as a sequence of updates. The AsmL construct *require* asserts that the component's input event port must have a valid event. The rule first consumes the event in the port and checks whether this event triggers further updates in the component. If the event does, the rule *MapToLocalInputEvent* returns the corresponding local event used to

trigger the FSM model; if not, a *null* value is returned and the reaction is completed. If a valid local event is returned, it activates the FSM model. The reaction of the FSM model returns a local output event. If the SEFSM component produces an output event in this reaction, the rule *MapToGlobalOutputEvent* maps the local event to the global output even, which is then stored in the output port of the component.

```
abstract class Component
  React ()
    require inPort.exist
    step
      inPort.exist := false
      let localEvent as Event? = MapToLocalInputEvent (inPort.evnt)
    step
      if localEvent <> null then
        step
          let e as Event? = fsm.React (localEvent)
        step
          let globalEvent as Event? = MapToGlobalOutputEvent (e)
        step
          if globalEvent <> null then
           outPort.evnt := globalEvent
           outPort.exist := true
```

The operational rule *MapToLocalInputEvent* maps the global event received by the component to a local event that activates the FSM model, and evaluates guards placed as nodes in the SDF model. First, *GuardMap* maps the received event to a set of 3-tuples *{(s, e, n)}*, each of which can locate a transition in the component whose trigger event matches this event. A transition is an enabled one if it satisfies all the three conditions: (1) its trigger event matches the received event; (2) its source state is the current state; (3) the evaluation of its guard is true.

GuardMap returns the set of all tuples that satisfy the first condition. Then, the rule enquires the current state of the FSM model using the operational rule *GetCurrentState* and removes those tuples whose first element does not refer to the current state. The third element in the tuple refers to a node in the SDF which is actually

a guard in the component. If this element is a *null* value, it indicates the corresponding guard is default true. The rule evaluates the guards and removes those tuples whose guard evaluation returns false. All the remaining tuples in the set then refer to the current enabled transitions in the component. If the size of this set is larger than *1*, the rule reports a non-deterministic error; if the set is empty, the rule returns a *null* value due to no enabled transition; otherwise, the rule creates and returns a local input event to activate the FSM model.

```
abstract class Component
  MapToLocalInputEvent (e as Event) as Event?
    step
      if e.eventType in GuardMap then
        step
          var enabledTransitions as Set of (String, String, Node?) =
GuardMap (e.eventType)
        step
          let s as State = fsm.GetCurrentState ()
        step
          forall g in enabledTransitions
            if g.First <> s.id then
              remove g from enabledTransitions
        step
          forall g in enabledTransitions where g.Third <> null
            if not EvaluateGuard (g.Third) then
              remove g from enabledTransitions
        step
          if Size(enabledSet) > 1 then
            error ("NON-DETERMINISM ERROR")
          else
            if Size (enabledSet) = 1 then
              choose g in enabledSet
                return Event (g.Second)
            else
              return null
      else
        return null
```

The operational rule *MapToGlobalOutputEvent* maps a local event produced by the FSM model to a global output event, and executes actions placed as nodes in the SDF model. First, *ActionMap* maps a local output event to a 2-tuple *({n}, e)*. If the component

needs to execute actions in this reaction, *{n}* refers to a set of nodes in the SDF model which encapsulates those actions. If the component produces an output event in this reaction, *e* is the type of that output event; otherwise, *e* is a *null* value. If *e* is not a *null* value, the rule creates and returns the corresponding global output event; otherwise, the rule returns a *null* value.

```
abstract class Component
  MapToGlobalOutputEvent (e as Event) as Event?
    step
      if e.eventType in ActionMap then
        step
          let actionTuple as (Set of Node, String?) =
ActionMap(e.eventType)
        step
          sdf.Run (actionTuple.First)
        step
          if actionTuple.Second <> null then
            return Event (actionTuple.Second)
          else
            return null
```

The semantics of SEFSM components is defined as the composition of the two semantic units: FSM-SU and SDF-SU. We observe that this behavioral semantics specification is not limited to the SEFSM components. It actually specifies the semantics of a common behavioral category that captures the reactive computation behaviors. Therefore, we can consider the compositional semantics specification of SEFSM components as a new derived semantic unit, called Action Automaton Semantic Unit (AA-SU). We leverage this AA-SU in the following section to compositionally specify the semantics of SEFSM Systems.

Compositional Semantics Specification for SEFSM Systems

A SEFSM system is composed of a set of components, which communicate with each other through event channels and data channels. The semantics of SEFSM systems is defined as the composition of AA-SU and SDF-SU. The compositional semantics specification for SEFSM includes: (1) an Abstract Data Model defining the structural composition $<A_C, A_{AA\text{-}SU}, A_{SDF\text{-}SU}, g_1, g_2>$, where $g_1: A_C \rightarrow A_{AA\text{-}SU}$, and $g_2: A_C \rightarrow A_{SDF\text{-}SU}$ are structural relation maps; and (2) Operations and Transformation Rules specifying the behavioral composition $<R_C, R_{AA\text{-}SU}, R_{SDF\text{-}SU}>$.

Structural Composition

The structural composition defines the communication relationships among components, in terms of an event flow and a data flow. The event flow is constructed using event channels connecting the input/output event ports. The data flow is created by connecting the input/output data ports with data channels. As it is shown in Figure 6-7, we reuse again the SDF-SU semantic unit to model the interaction semantics for the data flow. It is important to note that although the SDF sections of the individual components together with the SDF interaction among the components are integrated into a single SDF model, this is still a model system. Due to the integration with the FSM sections, always only a subset of the SDF nodes is involved in a reaction of the SEFSM system. We chose not to declare the event flow interaction model as a semantic unit. Figure 6-7 presents the role of the AA-SU, SDF-SU and the event flow interactions in the SEFSM system model by restructuring the example in Figure 6-3. This new structure gives a much clearer

expression for the control dependency among components and the data dependency among computational functions (actions and guards).

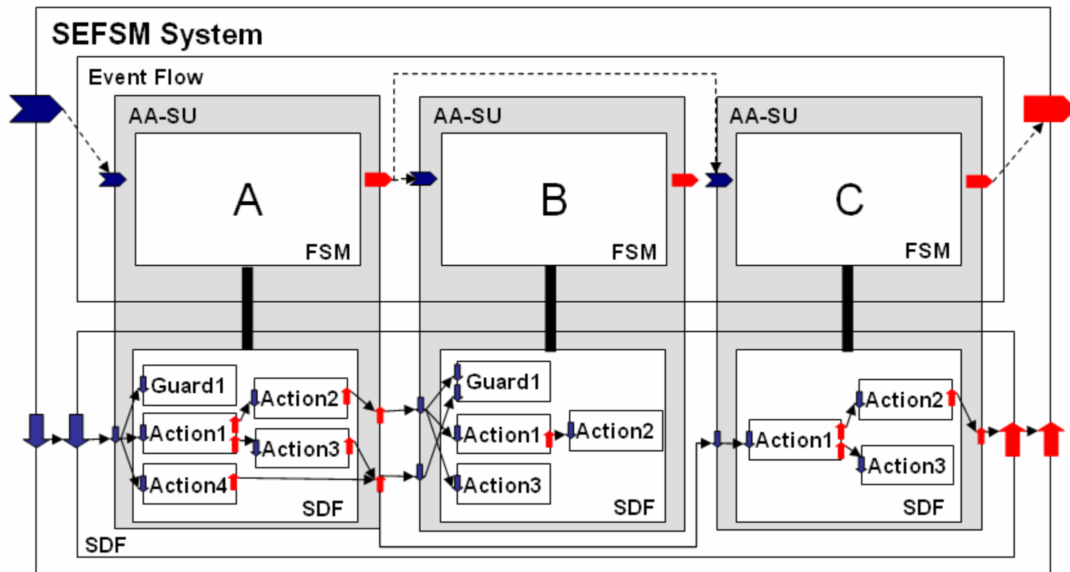

Figure 6-7: The compositional structure of the SEFSM system originally shown in Figure 6-3

The AsmL abstract class *System* captures the top-level structure of a SEFSM system. The abstract property *components* is a set that holds all components in a system. The control dependency among components is expressed by a set of event channels contained in the abstract property *channels*. The data dependency among computational functions is described by a SDF model. Each component has a reference to this SDF model. The relationship between a component and the SDF model is defined by the AA-SU (e.g. the abstract property *GuardMap* and *ActionMap* in the class *Component*).

```
class EventChannel
  id      as String
  srcPort as EventPort
  dstPort as EventPort

abstract class System
  abstract property inPort     as EventPort
    get
  abstract property outPort    as EventPort
    get
  abstract property components as Set of Component
    get
  abstract property channels   as Set of EventChannel
    get
  abstract property sdf        as SDF
    get
```

Behavioral Composition

The behavioral composition of the SEFSM system defines a system-level controller (or scheduler) that controls the executions and the order of the executions of components, event channels and the SDF model. The operational rule *Runt* of *System* specifies the top-level system operations as a sequence of updates. The AsmL construct *require* asserts that the system should have a valid input event. Firstly, the rule propagates the event in the input event port of the system along all the connected event channels to the destination ports that refer to the input event ports of components. In the meantime, the operational rule *Initialize*, defined in the SDF-SU, propagates the data tokens in the input ports of the SDF model along the connected data channels to the destination ports that refer to the input ports of nodes. The next step is to keep running until the operations inside the step cause no further state updates in the ASM (*fixpoint*). The stopping condition for an AsmL *fixpoint* loop is met if no non-trivial updates have been made in the step. Updates that occur to variables declared in ASMs that nested

inside this loop are not considered. An update is considered non-trivial if the new value is different from the old value.

Within the loop, the rule first activates all the components who receive an event. The reactions of these components then produce new events. If new events are produced, the rule propagates them to the destination components and continues the loop; otherwise, the loop is stopped. Finally, the rule *ClearPorts* defined in SDF-SU is utilized to clear all the input data ports in the SDF model because the SEFSM system does not store the data generated in the last computation cycle.

```
abstract class System
  Run ()
    require inPort.exist
    step
      forall c in me.channels where c.srcPort.exist
        c.dstPort.evnt := c.srcPort.evnt
        c.srcPort.exist := false
        c.dstPort.exist := true
      ddf.Initialize ()
    step until fixpoint
      step
        forall comp in me.components where comp.inPort.exist
          comp.React ()
      step
        forall c in me.channels where c.srcPort.exist
          c.dstPort.evnt := c.srcPort.evnt
          c.dstPort.exist := true
          c.srcPort.exist := false
    step
      ddf.ClearPorts ()
```

This behavioral semantics is actually not unique to SEFSM. Rather, it captures the common behavior of event-driven synchronous reactive systems. Therefore, we can consider the compositional semantics specification of SEFSM as a new derived semantic unit for event-driven synchronous reactive systems. Details of the specification clearly demonstrates the similarities between the semantics of SEFSM and well known event-

driven synchronous reactive systems and opens up the possibility of utilizing a rich variety of analytical techniques that have been developed in that domain.

CHAPTER VII

RESULTS, CONCLUTIONS AND FUTURE WORK

Results

This section evaluates the semantic anchoring methodology from four aspects: the precision of the semantics specification, the validation support, the satisfaction of DSML designers, and the efficiency of the compositional semantic specification approach (CSSA). The evaluation is based on the empirical data from applying the semantic anchoring methodology to the definition of the EFSM semantics and the Ptolemy FSM domain semantics. In order to have a more thorough and objective evaluation on this methodology, more empirical data from real applications may be needed, but that is beyond the scope of this thesis.

Precision of the Semantics Specification

Originally, the EFSM semantics was informally described by the designers in English. Because of the ambiguity of natural languages, it is difficult for the designers to find the hidden errors and ambiguities in EFSM. After explicitly specifying the EFSM semantics in AsmL, the precision property of the AsmL specification expose some hidden errors and ambiguities in the original EFSM documents, some of which are listed here:

- In the original EFSM definition there is no difference between an event port and a data port in a FSM. We find that the event port should be differentiated from the

data port. A FSM should have a single input event port and a single output event port, and may have multiple input and output data ports.

- The original EFSM does not explicitly specify the composition semantics of FSMs. This is an error since two possible composition approaches, the parallel composition and the sequential composition, will generate totally different system behaviors.
- The original EFSM does not explicitly specify the communication semantics among FSMs. This is also an error since two possible communication mechanisms, the synchronous communication and the asynchronous communication, will significantly change the system behaviors.
- In the original EFSM, the input arguments in both a FSM and a computational function are required to have valid data when the FSM or the function is to be executed. However, this requirement is not true for a FSM. When a FSM is activated by an input event, its input arguments are not required to have valid data.
- The original EFSM does not explicitly specify whether a transition allows a single action or multiple actions. If multiple actions are allowed, it should have means for users to specify the order of these actions.

It is difficult for language designers to find errors and ambiguities when only a natural language is used to describe the semantics of a DSML. When the semantics of a DSML is precisely specified in AsmL, many of these errors and ambiguities are discovered. It is a meaningful improvement to find semantic faults in the early stages of the DSML lifecycle, since these faults will be very costly in the downstream stages.

Validation Support

Another way to determine the correctness of a specification is to execute the specification directly. The AsmL specification is executable and allows DSML designers to test for errors in the specification. With the behavioral semantics specification of EFSM, the AsmL tools support the simulation and the test case generation of EFSM models, which allows designers to have a better understanding of the semantics and find possible errors by testing.

Satisfaction of the EFSM Designers

We have closely cooperated with the EFSM designers when specifying the EFSM semantics. Whenever we found any possible ambiguities or errors in the original EFSM documents, we discussed them with the designers to confirm that they are semantic faults. If they are, we also gave suggestions on how to correct them. Overall, the EFSM designers are satisfied with the AsmL semantics specification and are willing to accept our specification as the formal semantics definition for EFSM.

Efficiency of the Semantic Anchoring Methodology

We measure the efficiency of the semantic anchoring methodology from two aspects: the efficiency of the compositional semantics specification approach (CSSA) and the time effort of specifying the semantic anchoring rules.

Both the CSSA and the transitional semantics specification approach (specifying semantics from scratch without reusing semantic units) are applied to define the semantics of EFSM. Table 7-1 shows the comparison between these two approaches.

Though the CSSA produces a larger specification, it requires less new lines of specification (LOS) and takes fewer man-hours because existing semantic units are reused. This experiment confirms the efficiency of CSSA compared with the traditional approach when a DSML has multiple behavioral aspects associated with different behavioral categories.

Table 7-1: Comparison between the CSSA and the traditional approach

Specification Approaches	Total LOS	New LOS	Time (man-hours)
CSSA	~ 400	~ 150	~ 20
Traditional Approach	~ 300	~ 300	~ 30

We measure the time effort for specifying the semantic anchoring rules by using a case study on the semantics definition of the Ptolemy FSM domain (Chapter IV). The time to specify the semantic anchoring rules, which map any Ptolemy FSM models to the AsmL data models, is about 24 man-hours, while the time to specify the AsmL specification is about 80 man-hours. Therefore, it is safe for us to estimate that specifying the semantic anchoring rules takes less time efforts than specifying the semantics directly in AsmL. When the semantic units are predefined, DSML designers only need to specify the semantic anchoring rules. Hence, in this situation, the semantic anchoring methodology reduces the overall semantics definition time. When DSML designers need to specify the semantics from scratch (in the worst case), the effort to specify the semantic anchoring rules is still worthwhile since the semantic anchoring rules enable the direct execution of domain models, which helps designers to test for errors in the specification.

Conclusion

A survey on Model-Driven Software Engineering reveals a trend towards the adoption of DSMLs in model-based design. Among the many approaches, Model-Integrated Computing (MIC) is a pioneer in advocating DSMLs. Though DSMLs have many successful applications, the lack of formal semantics definition is the main concern that slows down the adoption of DSMLs.

While abstract syntax metamodeling alleviates part of this problem, explicit and formal semantics specification has been an unsolved problem whose significance has not even recognized. For instance, the UML SPT profile (UML Profile for Schedulability, Performance and Time) does not have precisely defined semantics [142], which creates possibility for semantic mismatch between design models and modeling languages of analysis tools. This is particularly problematic in the safety critical real-time and embedded systems domain, where semantic ambiguities may produce conflicting results across different tools. The research community has put forth much effort to define semantics of modeling languages by means of informal mathematical text [58] or formal mathematical notations [160]. In either case, precise semantics specification for DSMLs remains a challenge.

This thesis proposes a semantic anchoring infrastructure for MIC, which includes a finite set of well-made semantic units and a formal MOF-based metamodeling framework. A semantic unit is a formal semantics specification defining the semantics of a behavioral category that captures the behavioral pattern of a class of systems. In the embedded software and systems domain, there exists a finite set of basic behavioral

categories such as Finite State Machine, Timed Automata, Discrete Event System and Synchronous Dataflow. A semantic unit is called a primary semantic unit if it specifies the semantics of a basic behavioral category. A finite set of primary semantic units are predefined in a formal specification language AsmL.

We have also developed a semantic anchoring tool suite that enables the transformational specification of DSML semantics to semantic units. If the semantics of a DSML falls into the finite set of basic behavioral categories, its semantics can be defined by specifying the semantic anchoring rules to a primary semantic unit. When a DSML has multiple behavioral aspects associated with different behavioral categories, a single primary semantic unit can not capture the semantics of this DSML. Alternatively, we proposed a compositional semantics specification approach (CSSA), which defines the semantics of a heterogeneous DSML as the composition of primary semantic units. This approach can reduce the required effort from DSML designers and improve the quality of the specification.

If the composed semantics specifies a behavior which is frequently used in system design, (for example composition of SDF interaction semantics with FSM behavioral semantics defines semantics for modeling signal processing systems [162]) the resulting semantics can be considered a derived semantic unit, which is built on primary semantic units, and could be offered up as one of the set of semantic units for future anchoring efforts.

This thesis also includes three case studies for different purposes. The FSM domain in Ptolemy is used as a case study to explain the semantic anchoring methodology and to illustrate how the semantic anchoring tool suite is applied to design

DSMLs. The Timed Automata Semantic Unit (TASU) is defined as an example to illustrate how to specify semantic units. An industrial-strength modeling language, EFSM, is employed as a case study to explain the compositional semantics specification approach.

Future Work

Semantic anchoring describes an approach towards formal DSMLs and a solid semantic foundation for the MIC approach, but it is still in its early stages. Substantial further research is required to mature and concretize this approach. Some of the research directions are listed as following:

- Identify the best underlying formal semantic framework, which is general enough to cover a broad range of behavioral categories and also has a strong tool support. The current semantic anchoring tool suite adopts ASM as the formal semantic framework, and, correspondingly, AsmL as the formal specification language. ASM has many advantages, but it can not specify continuous behaviors. Some other promising formal methods, such as PVS [161] and tagged signal model [149], may be considered.
- Identify and specify an appropriate set of semantic units. In this thesis, we defined a semantic unit that captures a common semantics for Timed Automata based modeling languages. The semantic anchoring infrastructure needs to be filled with a set of well-made semantic units. In the embedded software and system domain, the semantic unit candidates may include semantic units for hybrid automata, synchronous languages, discrete event systems and synchronous dataflow.

- Integrate semantic units with the corresponding analysis and verification tools. A set of commercial or academic tools may support the simulation, analysis, modeling checking and verification of models whose behaviors satisfy certain behavioral categories. If the semantic units are integrated with the corresponding analysis tools, the semantic anchoring will not only define the semantics of DSMLs but also provide the tool-supported analysis of DSMLs.
- Identify the composition patterns among semantic units and build a framework to automate the semantic unit composition. This thesis proposes a compositional semantics specification approach that defines the semantics of a heterogeneous DSML as a composition of semantic units. This approach moderates some specification efforts from DSML designers, but it still requires substantial specification work. Further research is needed to identify the composition patterns among semantic units and develop a framework to automate semantic unit composition.

REFERENCES

[1] Gary Cernosek, Eric Naiburg, "The Value of Modeling", IBM White Papers, June 10, 2005.

[2] Object Management Group, "Overview and Guide to OMG's Architecture", OMG Document omg/03-06-01, June 21, 2003.

[3] Object Management Group, "Model Driven Architecture (MDA)", OMG Document ormsc/2001-07-01, July 9, 2001.

[4] Object Management Group, "Common Object Request Broker Architecture", http://www.omg.org/technology/documents/formal.corba_iiop.htm.

[5] Object Management Group, "CORBA Component Model", http://www.omg.org/technology/documents/formal/components.htm.

[6] Sun Microsystems, "Java 2 Enterprise Edition (J2EE)", http//java.sun.com/j2ee/.

[7] Sztipanovits J., Karsai G., "Model-Integrated Computing", IEEE Computer, pp. 110-112, April, 1997.

[8] Karsai G., Sztipanovits J., Ledeczi A., Bapty T., "Model-Integrated Development of Embedded Software", Proceedings of the IEEE, Vol. 91, Number 1, pp. 145-164, January, 2003.

[9] Model-Integrated Computing, the web site http://www.isis.vanderbilt.edu/research/mic.html.

[10] Ledeczi A., Bakay A., Maroti M., "Model-Integrated Embedded Systems", in Robertson, Shrobe, Laddaga (eds) Self Adaptive Software, Springer-Verlag LNCS, Vol. 1936, February, 2001.

[11] The Generic Modeling Environment, http://www.isis.vanderbilt.edu/Projects/gme/

[12] Ledeczi A., Maroti M., Bakay A., Karsai G., Garrett J., Thomason IV C., Nordstrom G., Sprinkle J., Volgyesi P., "The Generic Modeling Environment", Workshop on Intelligent Signal Processing (WISP 2001), Budapest, Hungary, May, 2001.

[13] Ledeczi A., Bakay A., Maroti M., Volgyesi P., Nordstrom G., Sprinkle J., Karsai G., "Composing Domain-Specific Design Environments", Computer, Vol. 34, No. 11, pp. 44-51, November, 2001.

[14] Karsai G., Maroti M., Ledeczi A., Gray J., Sztipanovits J., "Composition and Cloning in Modeling and Meta-Modeling", IEEE Transactions on Control System Technology, Vol. 12, No. 2, March 2004.

[15] Ledeczi A., Nordstrom G., Karsai G., Volgyesi P., Maroti M., "On Metamodel Composition", IEEE CCA 2001, CD-Rom, Mexico City, Mexico, September 5, 2001.

[16] The Graph Rewriting and Transformation, http://www.isis.vanderbilt.edu/Projects/mobies/downloads.asp#GREAT.

[17] Agrawal A., Karsai G., Shi F., "Graph Transformations on Domain-Specific Models", ISIS Technical Report, ISIS-03-403, November, 2003.

[18] Szemethy T., Karsai G., "Platform Modeling and Model Transformations for Analysis", Journal of Universal Computing Science, 10, 10, pp. 1383-1408, November 23, 2004.

[19] Sprinkle J., Karsai G., "A Domain-Specific Visual Language for Domain Model Evolution", Journal of Visual Languages and Computing, vol. 15, no. 2, April, 2004.

[20] Agrawal A., "A Formal Graph-Transformation Based Language for Model-to-Model Transformations", PhD Dissertation, Vanderbilt University, Dept of EECS, August, 2004.

[21] Neema S., Sztipanovits J., Karsai G., .Ken Butts, "Constraint-Based Design-Space Exploration and Model Synthesis", Proceedings of the Fifth ACM International Conference on Embedded Software (EMSOFT 2003), LNCS 2855, Philadelphia, PA, October, 2003.

[22] Mohanty S., Prasanna V., Neema S., Davis J., "Rapid Design Space Exploration of Heterogeneous Embedded Systems using Symbolic Search and Multi-Granular Simulation", Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES), Berlin, Germany, June, 2002.

[23] Bryant R., "Symbolic Manipulation with Ordered Binary Decision Diagrams", School of Computer Science, Carnegie Mellon University, Technical Report CMU-CS-92-160, July 1992.

[24] The Mozart Project, http://www.mozart-oz.org/.

[25] Magyari E., Bakay A., Lang A., Paka T., Vizhanyo A., Agrawal A., Karsai G., "UDM: An Infrastructure for Implementing Domain-Specific Modeling Languages", The 3rd OOPSLA Workshop on Domain-Specific Modeling, OOPSLA 2003, Anahiem, California, October 26, 2003.

[26] WOTIF, http://repo.isis.vanderbilt.edu/tools/get_tool?WOTIF.

[27] Zonghua Gu, Shige Wang, Sharath Kodase, and Kang G. Shin, "An End-to-End Tool Chain for Multi-View Modeling and Analysis of Avionics Mission Computing Software", 24th IEEE International Real-Time Systems Symposium (RTSS 2003), Cancun, Mexico.

[28] Model-Driven Software Development, http://www.mdsd.info/

[29] Bran Selic, "The Pragmatics of Model-Driven Development", IEEE Software, Vol. 20, issue 5, 2003.

[30] Eclipse Modeling Framework, http://www.eclipse.org/emf/.

[31] Software Factories, http://msdn.microsoft.com/vstudio/teamsystem/workshop/sf/default.aspx.

[32] Agile Modeling, http://www.agilemodeling.com/.

[33] Frank Budinsky, David Steinberg, Ed Merks, Ray Ellersick, Timothy Grose, "Eclipse Modeling Framework", Published by Addison Wesley Professional, August, 2003.

[34] Frank Budinsky, "The Eclipse Modeling Framework – Moving into Model-Driven Development", Dr. Dobb's Journal August, 2005.

[35] Object Management Group, "Meta Object Facility (MOF) 2.0 Core Specification", ptc/04-10-15, October, 2003.

[36] Gary Cernosek, "Next-Generation Model-Driven Development", IBM White Paper, December 2004.

[37] Graphical Editing Framework, http://eclipse.org/gef/.

[38] S. Kelly, "Comparison of Eclipse EMF/GEF and MetaEdit+ for DSM", http://www.softmetaware.com/oopsla2004/kelly.pdf, access November 20, 2004.

[39] Daniel Lee, "Display a UML Diagram using Draw2D", Eclipse Corner Article, http://www.eclipse.org/articles/Article-GEF-Draw2d/GEF-Draw2d.html.

[40] J. Greenfield and K. Short, "Moving to Software Factories", Software Development Magazine, July 2004.

[41] J. Greenfield and K. Short, S. Cook and S. Kent, "Software Factories: Assembling Applications with Patterns, Models, Frameworks and Tools", Wiley. 2004.

[42] Mauro Regio and Jack Greenfield, "A Software Factory Approach To HL7 Version 3 Solutions", Microsoft White Paper, June 2005.

[43] Agile Modeling Documents, "Agile Model Driven Development", http://www.agilemodeling.com/essays/amdd.htm.

[44] Scott W. Ambler and Ron Jeffries, "Agile Modeling: Effective Practices for Extreme Programming and the Unified Process", John Wiley & Sons, March 2002.

[45] Scott W. Ambler, "The Object Primer: Agile Model-Driven Development with UML 2.0", Cambridge University Press, 3 edition, March 2004.

[46] Scott W. Ambler, "Agile Database Techniques: Effective Strategies for the Agile Software Developer", John Wiley & Sons, 2003.

[47] Astels. D, "Test Driven Development: A Practical Guide", Upper Saddle River, NJ: Prentice Hall, 2003.

[48] Object Management Group, UML Project, http://www.uml.org/.

[49] Peter Fritzson and Peter Bunus Modelica, "A General Object-Oriented Language for Continuous and Discrete-Event System Modeling and Simulation", Proceedings of the 35th Annual Simulation Symposium, Apr., 2002.

[50] The Hybrid System Interchange Format, http://www.isis.vanderbilt.edu/Projects/Mobies/downloads.asp#HSIF.

[51] James R Rumbaugh, Michael R. Blaha, William Lorensen, Frederick Eddy and William Premerlani, “Object-Oriented Modeling and Design”, Prentice Hall, Englewood Cliffs, 1991.

[52] Grady Booch, “Object Solutions: Managing the Object-Oriented Project (Addison-Wesley Object Technology Series)”, Addisson Wesley, 1995.

[53] I. Jacobson, “Object-Oriented Software Engineering: A Use Case Driven Approach (Addison-Wesley Object Technology Series)”, Addisson Wesley, 1994.

[54] Object Management Group, “Unified Modeling Language Specification, v1.1”, January 1997.

[55] Object Management Group, “UML 2.0 OCL final adopted specification”, ptc/03-1014, 2003.

[56] Object Management Group, “UML 2.0 Infrastructure”, ptc/04-10-14, November 2004.

[57] Object Management Group, “UML Superstructure Specification, v2.0”, formal/05-07-04, August 2005.

[58] MoBIES Group, “HSIF semantics”, The University of Pennsylvania, 2002.

[59] Agrawal A., Simon G., Karsai G., “Semantic Translation of Simulink/Stateflow models to Hybrid Automata using Graph Transformations”, International Workshop on Graph Transformation and Visual Modeling Techniques, Electronic Notes in Theoretical Computer Science, Vol. 109, December 2004.

[60] Jonathan Sprinkle, “Generative Components for Hybrid Systems Tools”, Journal of Object Technology, Vol. 4, No. 3, 2005.

[61] The Embedded System Modeling Language, http://www.isis.vanderbilt.edu/Projects/mobies/downloads.asp?newlogin=2#ESML.

[62] David C. Sharp, “Component-Based Product Line Development of Avionics Software”, First Software Product Lines Conference (SPLC), Denver, Colorado, August 2000.

[63] Wing, J.M., “A Specifier's Introduction to Formal Methods”, Computer, Vol. 23, Issue 9, September 1990.

[64] Hall, A., “Seven myths of formal methods”, IEEE Software, Vol. 7, Issue 5, September 1990.

[65] Hanne Riis Nielson, Flemming Nielson, “Semantics with Applications: A Formal Introduction”, Wiley Professional Computing, Wiley, Revised Edition 1999.

[66] Joseph E. Stoy, “Denotational semantics: The Scott-Strachey Approach to Programming Language Theory”, MIT Press, 1977.

[67] C. A. R. Hoare, “An axiomatic basis for computer programming”, Communications of the ACM, Vol. 12, Issue 10, October 1969.

[68] Kenneth Slonneger and Barry L. Kurtz, “Syntax and Semantics of Programming Languages, A Laboratory Based Approach”, Addison-Wesley, 1995.

[69] Formal Specification Languages, http://www.rbjones.com/rbjpub/cs/csfm02.htm.

[70] E. Boerger and R. Staerk, “Abstract State Machines: A Method for HighLevel System Design and Analysis”, Springer-Verlag, 2003.

[71] Dawes, John, “VDM-SL Reference Guide”, Pitman 1991.

[72] C. B. Jones, “Systematic Software Development Using VDM”, Prentice-Hall 1989.

[73] Antoni Diller, Z An Introduction to Formal Methods, John Wiley & Sons Inc., 1994.

[74] Kevin Lano, “The B Language and Method: A Guide to Practical Formal Development”, Springer-Verlag, FACIT series, 1996.

[75] J.L. TURNER & T.L. McCLUSKEY, “The Construction of Formla Specifications: An. Introduction to the Model-based and Algebraic Approaches”, McGraw Hill, 1994.

[76] Joseph Goguen and Grant Malcolm, “Algebraic Semantics of Imperative Programs”, MIT Press, 1997.

[77] J. V. Guttag and J. J. Horning, “A Larch shared language handbook”, Science of Computer Programming, Vol. 6, Issue 2, March 1986.

[78] Jan de Meer, Rudolf Roth and Son Vuong, “Introduction to algebraic specifications based on the language ACT ONE”, Computer Networks and ISDN Systems, Vol. 23, Issue 5, February 1992.

[79] C.A.R. Hoare, “Communicating Sequential Processes”, Communications of the ACM Vol. 21, Issue 8, August 1978.

[80] R. Milner, “A Calculus of Communicating Systems”, Springer-Verlag, 1982.

[81] Yuri Gurevich, “Logic and the Challenge of Computer Science”, Chapter in Current Trends in Theoretical Computer Science, ed. E. Boerger, Computer Science Press, 1988.

[82] Abstract State Machines - A Formal Method for Specification and Verification, http://www.eecs.umich.edu/gasm/.

[83] Yuri Gurevich, "Evolving Algebras: An Attempt to Discover Semantics", Current Trends in Theoretical Computer Science, eds. G. Rozenberg and A. Salomaa, World Scientific, 1993.

[84] Yuri Gurevich, "Evolving Algebras 1993: Lipari Guide", Specification and Validation Methods, ed. E. Börger, Oxford University Press, 1995.

[85] Yuri Gurevich, "May 1997 Draft of the ASM Guide", University of Michigan EECS Department Technical Report CSE-TR-336-97.

[86] Yuri Gurevich, "Sequential Abstract State Machines Capture Sequential Algorithms", ACM Transactions on Computational Logic, Vol. 1, NO. 1, July 2000.

[87] Egon Börger and Robert Stärk, “Abstract State Machines: A Method for High-Level System Design and Analysis”, Springer-Verlag, 2003.

[88] Alan M. Turing, “On Computable Numbers with an Application to the Entscheidungsproblem”, Proc. London Math. Soc. (2), 42, 1937.

[89] Yuri Gurevich and James K. Huggins, "The Semantics of the C Programming Language". Computer Science Logic (CSL'92), Springer-Verlag LNCS 702, 1993.

[90] Charles Wallace, "The Semantics of the C++ Programming Language", Specification and Validation Methods, ed. E. Börger, Oxford University Press, 1995.

[91] Egon Börger and Wolfram Schulte, "A Programmer Friendly Modular Definition of the Semantics of Java". In J. Alves-Foss, ed., "Formal Syntax and Semantics of Java", Springer-Verlag LNCS 1523, 1998.

[92] Egon Börger and Wolfram Schulte, "A Practical Method for Specification and Analysis of Exception Handling: A Java/JVM Case Study", IEEE Transactions on Software Engineering, Vol. 26, NO. 10, October 2000.

[93] Yuri Gurevich, Wolfram Schulte, and Charles Wallace, "Investigating Java Concurrency using Abstract State Machines", In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, eds., Abstract State Machines: Theory and Applications, Springer-Verlag LNCS 1912, 2000.

[94] Robert Stärk, Joachim Schmid, and Egon Börger, Java and the Java Virtual Machine: Definition, Verification, Validation, Springer-Verlag, 2001.

[95] Philipp W. Kutter and Alfonso Pierantonio, "The Formal Specification of Oberon", Journal of Universal Computer Science, Vol. 3, NO. 5 (1997).

[96] Marcin Mlotkowski, Specification and Optimization of the Smalltalk programs, Ph.D. Thesis, University of Wroclaw, 2001.

[97] Egon Börger and Dean Rosenzweig, "A mathematical definition of full Prolog", In Science of Computer Programming, 1994.

[98] Yuri Gurevich and Lawrence S. Moss, "Algebraic Operational Semantics and Occam", 3rd Workshop on Computer Science Logic (CSL'89), Springer-Verlag LNCS 440, 1990.

[99] Egon Börger, Uwe Glässer, and Wolfgang Muller, "Formal Definition of an Abstract VHDL'93 Simulator By EA-Machines", In C. Delgado Kloos and P.T. Breuer, eds., Formal Semantics for VHDL, Kluwer Academic Publishers, 1995.

[100] Hisashi Sasaki, Kazunori Mizushima, and Takeshi Sasaki, "Semantic Validation of VHDL-AMS by an Abstract State Machine", In Proceedings of IEEE/VIUF

International Workshop on Behavioral Modeling and Simulation (BMAS'97), Arlington, VA, October 20-21, 1997.

[101] Zsolt Németh and Vaidy Sunderam, "A Formal Framework for Defining Grid Systems". In Proceedings of the Second IEEE/ACM International Symposium on Cluster Computing and the Grid, CCGRID2002, Berlin, May 2002.

[102] Paula Glavan and Dean Rosenzweig, "Communicating evolving algebras", In E. Börger, H. Kleine Büning, G. Jäger, S. Martini, and M. M. Richter, editors, "Selected papers from CSL'92 (Computer Science Logic)", Springer-Verlag LNCS 702, 1993.

[103] Yuri Gurevich and James K. Huggins, "The Railroad Crossing Problem: An Experiment with Instantaneous Actions and Immediate Reactions", In Computer Science Logic, Selected papers from CSL'95, ed. H.K. Büning, Springer-Verlag LNCS 1092, 1996.

[104] Danièle Beauquier and Anatol Slissenko, "Verification of Timed Algorithms: Gurevich Abstract State Machines versus First Order Timed Logic", In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, eds., Abstract State Machines -- ASM 2000, International Workshop on Abstract State Machines, Monte Verita, Switzerland, Local Proceedings, TIK-Report 87, Swiss Federal Institute of Technology (ETH) Zurich, March 2000.

[105] Egon Börger and Giuseppe Del Castillo, "A formal method for provably correct composition of a real-life processor out of basic components (The APE100 Reverse Engineering Study)", In Y. Gurevich and E. Boerger, "Evolving Algebras Mini-Course", Technical Report BRICS- NS-95- 4, BRICS, University of Aarhus, July 1995.

[106] Egon Börger and S. Mazzanti, "A Practical Method for Rigourously Controllable Hardware Design", in J.P. Bowen, M.G. Hinchey, D. Till, eds., ZUM'97: The Z Formal Specification Notation, Springer-Verlag LNCS 1212, 1997.

[107] Uwe Glässer and René Karges, "Abstract State Machine Semantics of SDL", Journal of Universal Computer Science, Vol. 3, NO. 12, 1997.

[108] ITU-T recommendation Z.100 annex F: SDL formal semantics definition. International Telecommunication Union, Geneva, 2000.

[109] Yuri Gurevich, "Sequential Abstract State Machines Capture Sequential Algorithms", ASM Transaction on Computational Logic, Vol. 1, NO. 1, July 2000.

[110] Andreas Blass and Yuri Gurevich, "Abstract State Machines Capture Parallel Algorithms", ACM Transactions on Computational Logic, Vol. 4, Issue 4, October 2003.

[111] Object Management Group, "What is OMG-UML and Why Is It Important?", 1997, http://www.omg.org/news/pr97/umlprimer.html.

[112] William E. McUmber and B.H.C. Cheng, “A General Framework for Formalizing UML with Formal Languages”, Proceedings of the 23rd International Conference on Software Engineering, 2001.

[113] Heinrich Hussmann, “Loose Semantics for UML/OCL”, In H. Ehrig, B.J.Krämer, A.Ertas (eds), Proceedings of the 6th World Conference on Integrated Design and Process Technology (IDPT 2002), Pasadena, California, June 23-28, 2002.

[114] Egon Börger, Alessandra Cavarra and Elvinia Riccobene, “Modeling the Dynamics of UML State Machines”, Proceedings of the International Workshop on Abstract State Machines, Theory and Applications, Springer-Verlag, LNCS Vol. 1912, 2000.

[115] Gihwon Kwon, “Rewrite rules and operational semantics for model checking UML statecharts”, In Andy Evans, Stuart Kent, and Bran Selic, editors, Proceedings of the Third International Conference on the Unified Modeling Language (UML 2000), Springer-Verlag, LNCS, Vol. 1939, 2000.

[116] Gregor Engels, Jan Hendrik Hausmann, Reiko Heckel and Stefan Sauer, “Dynamic Meta Modeling: A Graphical Approach to the Operational Semantics of Behavioral Diagrams in UML”, Proceedings of 3rd International Conference on the Unified Modeling Language (UML2000), October, 2000.

[117] Reiko Heckel and Stefan Sauer, “Strengthening UML Collaboration Diagrams by State Transformations”, 4th International Conference on Fundamental Approaches to Software Engineering (FASE 2001), Springer-Verlag, LNCS, Vol. 2029, 2001.

[118] D.Dranidis, K.Tigka and P.Kefalas, “Formal Modelling of Use Cases with X-machines”, Proceedings of the 1st South-East European Workshop on Formal Methods (SEEFM’03), November 2003, Thessaloniki, Greece.

[119] Gunnar Övergaard and Karin Palmkvist, “A Formal Approach to Use Cases and Their Relationships”, Selected papers from the First International Workshop on The Unified Modeling Language «UML»'98: Beyond the Notation, Springer-Verlag, LNCS Vol. 1618, 1998.

[120] Yang Dong and Zhang ShenSheng, “Using π-calculus to Formalize UML Activity Diagram”, 10th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS'03), 2003.

[121] Börger, E., Cavarra, A. and Riccobene, E., “An ASM Semantics for UML Activity Diagrams”, Proceedings of the 8th international Conference on Algebraic Methodology and Software Technology, Springer-Verlag, LNCS Vol. 1816, 2000.

[122] Richters, M. and Gogolla, M, “On Formalizing the UML Object Constraint Language OCL”, Proceedings of the 17th international Conference on Conceptual Modeling, T. W. Ling, S. Ram, and M. Lee, Eds. Springer-Verlag, LNCS, Vol. 1507, November 1998.

[123] J.-M. Bruel and R. B. France, “Transforming UML Models to Formal Specifications”, UML’98: The Unified Modeling Language, Mulhouse, France, Springer-Verlag, LNCS Vol. 1618, 1998.

[124] S.-K. Kim and D. Carrington, “A Formal Mapping Between UML models and Object-Z Specifications”, ZB 2000: Formal Specification and Development in Z and B, York, UL, Springer-Verlag, LNCS Vol. 1878, 2000.

[125] Meyer, E. and Souquières, J., “A Systematic Approach to Transform OMT Diagrams to a B Specification”, Proceedings of the Wold Congress on Formal Methods in the Development of Computing Systems-Volume I - Volume I, J. M. Wing, J. Woodcock, and J. Davies, Eds. Springer-Verlag, LNCS, Vol. 1708, 1999.

[126] R. Bourdeau and B. Cheng, “A Formal Semantics for Object Model Diagrams”, IEEE Transactions on Software Engineering 21(10), 1995.

[127] Reggio, G., Cerioli, M. and Astesiano, E., “Towards a Rigorous Semantics of UML Supporting Its Multiview Approach”, Proceedings of the 4th international Conference on Fundamental Approaches To Software Engineering, LNCS, Vol. 2029, Springer-Verlag, 2001.

[128] M. Kyas, H. Fecher, F. S. de Boer, J. Jacob, J. Hooman, M. van der Zwaag, T. Arons and H. Kugler, ” Formalizing UML Models and OCL Constraints in PVS”, International Workshop on Semantic Foundations of Engineering Design Languages (SFEDL'04).

[129] Muan Yong Ng and Michael Butler, "Towards Formalizing UML State Diagrams in CSP," First International Conference on Software Engineering and Formal Methods (SEFM'03), 2003.

[130] Luciano Baresi and Mauro Pezzè, “Petri Nets as Semantic Domain for Diagram Notations”, Electr. Notes Theor. Comput. Sci. 127(2): 29-44 (2005).

[131] ITU-T Recommendation Z.100(11/99), Languages for telecommunications applications – Specification and Description Language (SDL), International Telecommunication Union, Geneva, 2000.

[132] SDL Formal Semantics Project, ITU-T Study Group 10: SDL Semantics Group, http://rn.informatik.uni-kl.de/projects/sdl/.

[133] Marius Bozga, Jean-Claude Fernandez, Lucian Ghirvu, Susanne Graf, Jean-Pierre Krimm, Laurent Mounier and Joseph Sifakis, “IF: An intermediate representation for SDL and its applications”, SDL '99 The Next Millennium, 9th International SDL Forum, Montréal, Québec, Canada, June 1999.

[134] J. Fischer and E. Dimitrov, “Verification of SDL Protocol Specifications Using Extended Petri Nets”, Proceedings of the Workshop on Petri Nets and Protocols of the 16th International Conference on Applications and Theory of Petri Nets, Torino, Italy, 1995.

[135] J.A. Bergstra and C.A. Middleburg, “Process Algebra Semantics of ϕ SDL”, Technical Report, UNU/IIST Report No. 68, UNU/IIST, The United Nations University, April 1996.

[136] M. Broy, “Toward A Formal Foundation of the Specification and Description Language SDL”, Formal Aspects of Computing 3 (3), 1991.

[137] Manfred Broy, Frank Dederich, Claus Dendorfer, Max Fuchs, Thomas Gritzner and Rainer Weber, “The Design of Distributed Systems - An Introduction to FOCUS”, Technical Report, TUM-I9202, Technische Universität München, January 1992.

[138] S. Lau and A. Prinz, “BSDL: The Language--Version 0.2”, Technical Report, Department of Computer Science, Humboldt University, Berlin, August 1995.

[139] U. Glässer and R. Karges, “Abstract state machine semantics of SDL”, Journal of Universal Computer Science 3 (12), 1997.

[140] U. Glässer, “ASM Semantics of SDL: Concepts, Methods, Tools”, Proceedings of 1st Workshop of the SDL Forum Society on SDL and MSC (SAM 98), Berlin, Germany, June 29- July 1, 1998.

[141] R. Gotzhein, B. Geppert, F. Rößler and P. Schaible, “Towards a New Formal SDL Semantics”, Proceedings of 1st Workshop of the SDL Forum Society on SDL and MSC (SAM 98), Berlin, Germany, June 29- July 1, 1998.

[142] Susan Graph, Ileana Ober: “How Useful is the UML profile SPT Without Semantics?”, Workshop on the usage of the UML profile for Scheduling, Performance and Time (SIVOES '04), Toronto Canada, 2004.

[143] Kai Chen, Janos Sztipanovits, Sandeep Neema, Matthew Emerson and Sherif Abdelwahed, “Toward a Semantic Anchoring Infrastructure for Domain-Specific Modeling Languages,” Proceedings of the Fifth ACM International Conference on Embedded Software (EMSOFT’05), September 19-22, 2005, Jersey City, New Jersey, USA.

[144] Kai Chen, Janos Sztipanovits, Sherif Abdelwahed and Ethan Jackson, “Semantic Anchoring with Model Transformations”, European Conference on Model Driven Architecture -Foundations and Applications (ECMDA-FA), November 7-10, Nuremberg, Germany, LNCS 3748.

[145] Kai Chen, Janos Sztipanovits and Sherif Abdelwahed, "A Semantic Unit for Timed Automata Based Modeling Languages", Accepted by 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2006), April 4 - 7, 2006.

[146] T. Clark, A. Evans, S. Kent and P. Sammut, “The MMF Approach to Engineering Object-Oriented Design Languages”, Workshop on Language Descriptions, Tools and Applications, April, 2001.

[147] The abstract state machine language. http://www.research.microsoft.com/fse/asml.

[148] Emerson M., Sztipanovits J., Bapty T., “A MOF-Based Metamodeling Environment”, Journal of Universal Computer Science, 10, October 2004, pp. 1357-1382.

[149] E. Lee and A. Sangiovanni-Vincentelli, “A denotational framework for comparing models of computation”, IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, 17(12), 1998.

[150] R. Alur and T. A. Henzinger, “Reactive modules”, Form. Methods Syst. Des., 15(1), 1999, pp. 7–48.

[151] The semantic anchoring tool suite. http://www.isis.vanderbilt.edu/SAT.

[152] D. Harel, “Statecharts: A visual formalism for complex systems”, Science of Computer Programming, 8(3):231–274, 1987.

[153] A. Girault, B. Lee, and E. A. Lee, “Hierarchical finite state machines with multiple concurrency models”, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 18(6):742–760, June 1999.

[154] B. Lee, Specification and Design of Reactive Systems. PhD thesis, University of California, Berkeley, 2000.

[155] Rajeev Alur and David L. Dill, “A Theory of Timed Automata”, Journal of Theoretical Computer Science, 126 (2), 1994, pp. 183-235.

[156] Kim G. Larsen, Paul Pettersson and Wang Yi, “UPPAAL in a Nutshell”, Springer International Journal of Software Tools for Technology Transfer 1(1+2), 1997.

[157] Marius Bozga, Susanne Graf, Ileana Ober, Iulian Ober and Joseph Sifakis, “Tools and Applications II: The IF Toolset”, In Flavio Corradinni and Marco Bernanrdo,editors, Proceedings of SFM'04 (Bertinoro, Italy), LNCS vol. 3185, Springer-Verlag, 2004.

[158] Sergio Yovine, “Kronos: a Verification Tool for Real-time Systems”, Journal on Software Tools for Technology Transfer, 1, October 1997.

[159] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine, “Symbolic Model Checking for Real-time Systems”, Journal of Information and Computation, 111(2), 1994, pp. 193-244.

[160] G. Hamon and J. Rushby. “An Operational Semantics for Stateflow”, In Fundamental Approaches to Software Engineering: 7th International Conference, Springer-Verlag, 2004, pp. 229–243.

[161] Judy Crow, Sam Owre, John Rushby, N. Shankar, and Dave Stringer-Calvert, “Evaluating, Testing, and Animating PVS Specifications”, CSL Technical Report, March 30, 2001.

[162] Johan Eker, Jörn W. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Stephen Neuendorffer, Sonia Sachs, Yuhong Xiong, “Taming Heterogeneity The Ptolemy Approach”, *Proceedings of the IEEE*, volume 91, pages 127–144, 2003.

[163] Gossler, G., Sifakis, J., "Composition for Component-Based Modeling", *Science of Computer Programming,* vol. 55, 2005.

[164] Balarin, F., Lavagno, L., Passerone, C., Sangiovanni-Vincentelli, A. L., Sgroi, M., and Watanabe, Y., "Modeling and Designing Heterogeneous Systems", In *Concurrency and Hardware Design, Advances in Petri Nets* Lecture Notes In Computer Science, vol. 2549, pages 228-273, Springer-Verlag, London, 2002.

[165] S. Birla, S. Wang, S. Neema, and T. Saxena, "Addressing cross-tool semantic ambiguities in behavior modeling for vehicle motion control", In *Automotive Software Workshop 2006*, San Diego, CA, April 2006.