

THE STRUCTURAL SEMANTICS OF MODEL-BASED DESIGN:  
THEORY AND APPLICATIONS

By

Ethan K. Jackson

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, 2007

Nashville, Tennessee

Approved:

Professor Janos Sztipanovits

Professor Gabor Karsai

Professor Gautam Biswas

Professor Sherif Abdelwahed

Professor Constantine Tsinakidis

# TABLE OF CONTENTS

	Page
<b>LIST OF FIGURES</b> . . . . .	iv
<b>LIST OF TABLES</b> . . . . .	vi
<b>Chapter</b>	
<b>I. INTRODUCTION</b> . . . . .	1
<b>II. BACKGROUND</b> . . . . .	3
Introduction . . . . .	3
The Benefits of the Language View . . . . .	4
Models of Computation . . . . .	5
Representing Time . . . . .	6
Representing Concurrency . . . . .	9
Simulating MoCs . . . . .	12
Simulating Transition Systems . . . . .	13
Simulating Process Networks . . . . .	17
Domain-Specific Compilers . . . . .	23
Describing Syntax . . . . .	24
Semantic Analysis . . . . .	29
Discussion and Conclusion . . . . .	39
<b>III. FORMALIZING STRUCTURAL SEMANTICS</b> . . . . .	44
Introduction . . . . .	44
The Formal Semantics of Domains and Domain Construction . . . . .	45
Existing Candidate Formalisms . . . . .	48
Regular and Context-free Languages . . . . .	48
Extensions of Graph Structures . . . . .	49
Instances of ADTs . . . . .	51
Algebraic Approach . . . . .	52
Transformational Semantics and Model Transformations . . . . .	56
Metamodels and Metamodeling . . . . .	58
Applications of Structural Semantics . . . . .	61
Formalizing Model-Integrated Computing . . . . .	61
MiniMeta: A Formalized MIC Tool Suite . . . . .	63
Defining the eMOF Domain . . . . .	65
The Horn Domain $D_{Horn}$ . . . . .	68
eMOF Transformation onto $D_{Horn}$ . . . . .	72
Implementing MiniMeta with GME/GrEAT . . . . .	73
Discussion and Conclusion . . . . .	81
Acknowledgments . . . . .	82
<b>IV. AUTOMATED MODEL CONSTRUCTION AND ANALYSIS</b> . . . . .	83
Preliminaries - Metamodels, Domains, and Logic . . . . .	83
Analysis of Nonrecursive Horn Domains . . . . .	87
Extensions, Tools, and Future Directions . . . . .	95
<b>V. STRUCTURAL INTERFACES FOR ADAPTIVE SYSTEMS</b> . . . . .	98
Introduction . . . . .	98

Structural Semantics . . . . .	101
Time-Model Dynamics . . . . .	105
Scenario-based Adaptation . . . . .	107
Calculating Scenario-Regular Interfaces . . . . .	110
Interfaces from Horn Logic with Negation . . . . .	112
Conclusions and Future Work . . . . .	116
<b>Appendix</b>	
<b>A. TRANSFORMATION FROM FSA TO C . . . . .</b>	<b>117</b>
Transformation rules . . . . .	117
C Code generated from FSA . . . . .	124
<b>B. SPECIFICATION OF META-LEVEL COMPONENTS . . . . .</b>	<b>125</b>
Partial eMOF Transformation . . . . .	125
Example of Generated Domain in Prolog . . . . .	126
eMOF Domain . . . . .	127
<b>BIBLIOGRAPHY . . . . .</b>	<b>129</b>

## LIST OF FIGURES

1.	An abstract machine with a precise notion of time. . . . .	8
2.	Example of concurrency in hardware notations. . . . .	9
3.	Example of a process network and its associated constraint system . . . . .	11
4.	Timed-automaton represented by Specification 5. . . . .	17
5.	Simulation of Specification 5 with ASML . . . . .	19
6.	The Elevator problem in Ptolemy II . . . . .	21
7.	Simulating the elevator problem in Ptolemy II. . . . .	23
8.	Example of a model represented as a directed graph . . . . .	25
9.	Several digraphs; graph I is not in the language, but II and III are in $\mathcal{L}_{DAG}$ . . . . .	25
10.	Example metamodel for hierarchical finite state machines. . . . .	28
11.	Example models in the metaprogrammable modeling environment GME: I. FSA model II. Assembly code model III. Access control model IV. Synchronous dataflow model . . . . .	30
12.	Metamodel of the input language; a finite state acceptor (FSA) language. . . . .	33
13.	Metamodel of the output language; a structured subset of C. . . . .	35
14.	Graph rewriting rule that creates the STATE enumeration elements. . . . .	36
15.	Rule creates the declarations, loop, and main switch objects. . . . .	37
16.	Sequencing and encapsulation of the graph rewriting rules as a block. . . . .	38
17.	Simple example of a communication protocol. . . . .	40
18.	One possible representation of the protocol using HFSMs. . . . .	41
19.	Interpretation of the specification assuming synchronous product. . . . .	43
20.	Model-based view of the meta-level design choices. . . . .	43
21.	(a) Example of two domains. (b) A model from a digital signal processing domain. . . . .	46
22.	A graph containing only 2-paths . . . . .	49
23.	Example metamodel with many relational concepts . . . . .	50
24.	Some classes as ADTs . . . . .	52
25.	Two positive domains of same signature are related by the <i>wellform</i> symbol . . . . .	55
26.	MetaGME metamodel for HFSM. . . . .	59
27.	Abstract view of the metamodeling process . . . . .	60
28.	The architecture for the MiniMeta Tool Flow. . . . .	63
29.	Detailed view of MiniMeta metamodeling facility implementation using MetaGME, GME, and and embedded Prolog Engine. . . . .	74
30.	A MetaGME metamodel of the eMOF Domain. . . . .	75
31.	(a) An eMOF metamodel in GME of DSP domain. (b) Translation of metamodel to definite clauses and verification that metamodel is well-formed, using a Prolog engine. . . . .	75
32.	Results of check after inheritance cycle is added to DSP metamodel. . . . .	76
33.	Implementation of the domain generation facility using MetaGME, GME, GReAT, and an embedded Prolog engine. . . . .	77
34.	Invocation of the domain generator component. . . . .	79
35.	An example DSP model created using the DSML generated in Figure 31.a . . . . .	79
36.	Results of checking the DSP model against the formal domain definition . . . . .	80
37.	Annotation of the DSP metamodel with an additional constraint . . . . .	80
38.	DIGRAPH: A simple metamodel for labeled directed graphs . . . . .	83
39.	UNSAT: A complex metamodel with no finite non-trivial models . . . . .	83
40.	Model that (partially) adheres to the UNSAT metamodel . . . . .	84
41.	(Left) CYCLE: a positive NHD in FORMULA syntax. (Right) Backwards chaining graph generated from goal <i>wellform(X)</i> . . . . .	90
42.	Solution tree generated from backwards chaining graph of Figure 41 . . . . .	91
43.	Constraint system shown as a forest of union-find trees. . . . .	93
44.	(Left) A malformed input model (top), a well-formed embedding, and a minimal embedding (bottom). (Middle) Initial constraint system showing only sink trees and disequality constraints. (Right) Minimized constraint system. . . . .	94

45.	Interfaces and subcomponents of an adaptive component. . . . .	100
46.	An example of model structure. . . . .	101
47.	<i>DF</i> - A dataflow metamodel with constraint annotations . . . . .	104
48.	A general $\Delta$ -interface acting on an input model . . . . .	106
49.	Example of scenario-based adaptation . . . . .	108
50.	Lattice generated by the adaptation scenarios of <i>R</i> . . . . .	111
51.	Finds the initial state and creates an assignment that initializes the <i>currentState</i> variable . . . . .	117
52.	Creates a <i>case</i> and empty <i>switch</i> block for each <i>State</i> instance. . . . .	118
53.	Creates non-loop transitions . . . . .	120
54.	Creates loop transitions . . . . .	121
55.	Builds feedback messages . . . . .	121
56.	Sequencing of transformation rules . . . . .	122
57.	Example FSA acceptor . . . . .	122
58.	Partial C abstract syntax tree generated from example FSA. . . . .	123
59.	Simulation of generated FSA model . . . . .	123
60.	C code generated from AST model of an FSA . . . . .	124

## LIST OF TABLES

1.	Extending the universe $U$ for timed-automation . . . . .	14
2.	Extending the signature $\Upsilon$ for timed-automata . . . . .	15
3.	Kernel of simulation engine . . . . .	16
4.	Main simulation loop . . . . .	17
5.	The data model for a simple timed-automaton . . . . .	18
6.	Pseudocode for executing a finite state acceptor. . . . .	34
7.	Set of modeling concepts for DSP domain. . . . .	53
8.	A partial encoding of Figure 21.b with ground terms. . . . .	53
9.	Table of Vertex Primitives for MiniUML Metamodels . . . . .	65
10.	Table of Edge Primitives for MiniUML Metamodels . . . . .	66
11.	Encoding of concepts in the Horn Domain. . . . .	71
12.	Encoding of eMOF vertex primitives . . . . .	127
13.	Encoding of eMOF edge primitives . . . . .	128

## CHAPTER I

### INTRODUCTION

This thesis presents a systemic study of the structural semantics of model-based design. Structural semantics have a long history in computer science, and were studied early on under the moniker *language syntax*. This early work gave us regular and context-free languages, as well as tools for generating parsers from simple descriptions of language grammars. These efforts were a key step in the formal specification of programming languages [1]. In practice, these advances made it possible to design programming languages with sophisticated syntax, without spending significant design cycles on parser implementation. However, the applications of syntax in traditional programming language design seems to have stopped here. Most modern programming languages are implemented so that parsing is the first tier in a multi-tiered analysis process. The second tier is typically a *type check*, which is based on the formal descriptions of *type systems*. Unlike syntax, research on type systems continues to evolve in many different directions. This seems to makes sense in programming languages, because language syntax is primarily a user-interface issue. Parsing simply renders the program in a form suitable for further analysis.

Jumping ahead to the mid-1980's, model-based design began to evolve as a means to specify and implement embedded, distributed, and heterogeneous software systems. Model-based design collects together a number of principles addressing the design of such systems. We shall discuss model-based design in much more detail in the next chapter. At this juncture let us briefly mention that domain-specific modeling languages (DSMLs) play an important role in model-based design. DSMLs are programming languages tailored to some problem niche. At first glance this distinction may appear to make little difference. A programming language has the same basic parts regardless of its intended scope. However, deeper inspection shows that there are essential differences that must be taken into account. This thesis explores the impact of domain-specificity on the structural semantics of DSMLs.

There are two significant differences between the structural semantics of DSMLs in model-based design, and the language syntax of traditional general-purpose languages (GPL). First, DSML structural semantics encode essential properties of the problem domain, while GPL syntax does not. For example, imagine that one is to design an embedded system, and does so using the C language to create a program  $P$ . It is quite clear that the syntactical correctness of the program  $P$  reveals little information about the correctness of the implementation. Contrarily, if one uses a dataflow-like DSML to create a program  $P'$ , then the syntactical correctness of  $P'$  might formally witness freedom from deadlock. Thus, aligning the programming language with the problem domain yields languages where syntactic correctness is some reflection of behavioral correctness. In this sense, the structural semantics encode high-level invariants relevant to all possible solutions in the problem domain.

If the intent of language syntax is purely to provide a user-interface, then strong restrictions on expressiveness of syntax are essential. For example, regular and context-free languages impose strong restrictions on syntax, but result in programs that are easily parsed for later stages of program analysis. However, in model-based design syntax is intended to prune away bad solutions. Often times domain-specific invariants can not be expressed with simple regular or context-free languages. Thus, expressiveness is the second differentiator of DSML and GPL structural semantics. The structural semantics of DSMLs must be sufficiently expressive to capture meaningful invariants of the problem domain.

These two observations show that the structural semantics of DSMLs represents an important open problem. In this thesis we provide an encompassing formalization of structural semantics that also addresses *metamodeling* and *model transformations*, which are an integral part of the model-based approach. Fortunately for us, model-based design provides many opportunities for utilizing structural semantics. Thus, there are a number of interesting applications: (1) Proving the correctness of model transformations presupposes a formal definition of model structure. Our results provide an important step toward this goal. (2) Satisfaction of structural invariants may ensure properties like schedulability or deadlock-freedom, in which case well-formedness amounts to a proof of these properties. From this perspective, it is reasonable to develop algorithms that automatically construct well-formed models. An engineer might begin with a malformed description of a system, and then automatically convert this malformed description to a well-formed model. This conversion procedure can be viewed as repairing the errors in the model. (3) Adaptive systems and dynamic architectures are systems that evolve over many possible models. Using structural semantics, it is possible to ensure that a such systems always evolve through well-formed models.

This thesis proceeds as follows: The second chapter reviews model-based design and the roles that DSMLs play in this process. The third chapter develops the formal foundations for the structural semantics of DSMLs. The fourth chapter applies this formalism for automated theorem proving, and the fifth chapter applies the formalism for the use in dynamic architectures.



## CHAPTER II

### BACKGROUND

#### Introduction

Today's software systems pose unique challenges to traditional software engineering methodologies. First, the demands placed on software systems continue to evolve in both the functional and non-functional realms, and along many interacting axes: architectural, temporal, and, physical. Second, the sheer scale of software continues to grow, both on a per node basis, and in the number of distributed nodes that compose a system. Third, non-engineering disciplines, such as the legal field, are impacting design choices in poorly understood ways. For example, the recently enacted HIPAA law will impact how medical records can be digitally stored and accessed[2]. On one hand, this trifecta validates exactly what software engineers have always argued: Software must be designed methodically; off-the-cuff implementations will almost surely fail. On the other hand, engineering approaches that focus on sequential systems isolated in a comfortable computational environment are not sufficient for methodically designing today's large-scale and heterogeneous software systems.

The term *model-based design* encompasses a spectrum of engineering approaches, all of which address the complexity of modern system design. Most model-based approaches share a central dogma: *The application context must be defined before architecting a solution.* By *application context* we mean a description of the world in which the solution will operate. Typically the application context includes the temporal properties of computation, the concurrency and synchronization properties of communication, and the conditions under which deadlock or other malevolent behaviors arise. These attributes are specific to the context, and affect any solution placed in the application context. Particular model-based tools metaphorize the application context differently. The application context may be viewed as a *platform*, *actor class*, *model of computation*, or *domain-specific modeling language*. The authors of [3] argue that all of these perspectives are essentially the same. Nonetheless, it is useful to think in terms of one (or more) of these metaphors. In this chapter we review model-based design from the perspective that an application context can be represented with a domain-specific modeling language (DSML).

Embedded and heterogeneous systems were the genesis for model-based design, but the approach has wider applicability to software engineering as a whole. A secondary purpose of this review is to present model-based design to the software engineer from the perspective of domain-specific modeling languages. We chose this perspective because the field of programming languages is already familiar to many software engineers. DSML design can be viewed as an extension of traditional language design. These extensions permit the application context to be described as a sort of programming language; programs that adhere to

the language correspond to systems that are well-behaved when immersed in the application context. The engineer’s job is to select or construct a DSML that captures the essential characteristics of the application context. In the next section we discuss the benefits of the language view in more detail. Section II.3 presents the formal foundations of DSML semantics. Section II.4 describes how DSML programs can be executed (simulated) on traditional machines. Section II.5 examines syntax and compiler construction. We conclude in Section II.6. Through out this chapter we emphasize concrete code examples, providing the reader with tangible snapshots of a number of model-based tools.

### The Benefits of the Language View

In the simplest sense, a software system is a list of instructions and data executed on a machine. Of course, a list of instructions is just a carefully crafted list of data that adheres to the syntax of the programming language in which it was written. Thus, reiterating the observation that many have made before, a program is just a list of data. A program alone is meaningless without a machine to execute it, but when coupled with such a machine, a complex dynamical system emerges. Programming languages allows us to *represent* complex dynamical systems, in a compact form, as syntactically correct data [4].

The traditional data/machine view is a useful one, but in its unaltered form, it does not work well for distributed, embedded, and heterogeneous systems. Traditional programming languages are based on Turing machines, and this has significant drawbacks: First, Turing-like machines do not match the actual dynamics that distributed and embedded systems exhibit [5]. For example, the Turing machine must be extended to model the communication delays or unreliable channels experienced by a distributed system. Additional extensions are needed to capture the continuous dynamics experienced by embedded systems that sense and manipulate a physical environment [6]. Second, Turing-like machines are so expressive that it may be impossible to know if certain software requirements have been met. For example, the Halting Problem is undecidable for Turing machines. Deadlock-freedom is closely related to the halting problem, and is often undecidable. Thus, if a software system must be deadlock-free, then it may be unsafe to design such a system with the expressiveness of a Turing machine, for which the problem is undecidable (or intractable).

Model-based design addresses these issues by supporting the data/machine paradigm for many distinct types of machines. It also provides tools for defining new machine types and programming languages for those machines. In the model-based community the machine types are called *models of computation* (MoCs), and the programs are called *models*. Thus, a model is a structural (syntactic) artifact that defines a dynamical system when coupled with a particular MoC. The programming languages for particular MoCs are called *domain-specific modeling languages* (DSMLs) because they target only some machine types. This approach offers software engineers “methods and syntaxes that are closer to their application domain” [7]. This encourages the engineer to use the MoC that best reflects the reality of the environment in which design

must take place.

From the perspective of traditional software engineering, many of the techniques for DSML construction are similar to traditional programming language construction. DSMLs are created according to the following recipe: First, a mathematical description of an abstract machine (MoC) is developed. Second, an implementation of the abstract machine on a traditional Von Neumann architecture is constructed. This is similar to implementations of the Java Virtual Machine (JVM) on various platforms [8]. Third, a modeling language with a well-defined syntax is defined using tools similar in spirit to BNF-based (Backus-Naur Form) parser generators. Fourth, techniques, e.g. syntax-directed translation[9], and patterns, e.g. the visitor pattern [10], are used to translate a model into a set of instructions for the abstract machine. Since the machine has an implementation on existing architectures, the model can be simulated for the purpose of analysis or converted into a final native-code implementation.

Despite these similarities, there are some deep theoretical differences between traditional language design and today's model-based DSML design approaches. First, as we have already discussed, DSMLs support many different notions of computation. Second, DSMLs extend the expressiveness of syntax. Historically, syntaxes have been chosen for their ease of use and ease of parsing. Resounding figures, like Dijkstra and Hoare, argued for both these properties, and history bares their mark [1]. The model-based community uses syntaxes to filter out behaviorally incorrect models, or as a first-pass before verification [11]. The trade-off is often made that syntaxes with high parsing complexity are tolerated in exchange for the ability to detect badly designed systems early. We now describe these issues in more detail, beginning with extensions to formal notations of computation.

### Models of Computation

The traditional Turing machine, which contains a finite state controller with an infinite tape, must be rethought for today's engineering landscape. This is not because software systems run on drastically different hardware architectures (there are some exceptions) where the Turing model is invalid; rather software systems run in drastically new environments with drastically new requirements. For example, the *Object Management Group* (OMG), which maintains standards for the widely-used Universal Modeling Language (UML), defines a standard for specifying distributed data-centric applications. The following list enumerates some of the two-dozen possible requirements that can be placed on distributed applications (See the data distribution service (DDS) specification [12]):

1. *Deadlines* - A reader in the network requires a new piece of data within every  $T$  units of time.
2. *Reliability* - A reader demands how much of the known data must be delivered to that reader; impacts overall resource usage in the network.

3. *Lifespan* - A writer places an expiration date on data; after this time, the data is no longer valid.

Software with these requirements must be understood as both temporal and concurrent. However, providing a suitable formal definition of time and concurrency is not easy. For the remainder of this section we will explore various formal notions of time and concurrency as extensions to the traditional untimed Turing machine.

### Representing Time

The concept of *time* is an integral component of modern system requirements. In order to know if the requirements have been met, we must first get an idea of how a software system evolves across time. Traditionally, the temporal properties of software are measured with profilers like ATOM [13]. However, profilers cannot decide if timing requirements have been met without performing an unbounded number of analyses. A more conservative approach is to estimate the worst-case execution time (WCET), but WCET is highly correlated with implementation choices [14]. For example, the precise cache replacement policy affects WCET. We could fix all of these implementation details at the beginning of the engineering process, but this is contradictory to almost all modern engineering approaches wherein a design evolves from a high-level specification to a low-level implementation.

Methodologically, software should be *designed* with certain timing characteristics, instead of just measuring those characteristics *a posteriori*. However, as we have already discussed, traditional programming languages do not support the programmatic specification of timing properties, because the underlying machine model does not include a notion of time. This has been addressed by changing the underlying machine model to include a precise notion of time. Programs, which are just data interpreted by the machine, define dynamical systems with precise temporal properties. One such widely-adopted extension is *timed-automata*, but before we discuss this, let us recall some basic definitions. Consider that all physical machines have finite state; ignoring time, we can describe a machine as a *finite state automaton* (FSA)  $A_F = \langle Q, Q^0, Q^F, \rightarrow, \Pi \rangle$  over an input alphabet  $\Sigma_i$  and output alphabet  $\Sigma_o$ :

1.  $Q$  is a finite set of states
2.  $Q^0 \subseteq Q$  is a set of initial states
3.  $Q^F \subseteq Q$  is a set of final states
4.  $\rightarrow \subseteq Q \times \Sigma_i \times Q$  is a transition relation where  $s \xrightarrow{\alpha} s'$  indicates that the system transitions to  $s'$  when it is in state  $s$  and observes  $\alpha$ .
5.  $\Pi : Q \rightarrow \Sigma_o$  is a mapping from states to observations.

In this case, we can imagine that the input alphabet  $\Sigma_i$  contains the basic instructions and data recognized by the machine. A program is fed, instruction-by-instruction and datum-by-datum, to the automaton  $A_F$ .

In response, the machine transitions through a sequence of states  $s_0 \rightarrow s_1 \rightarrow \dots \rightarrow s_n$  and we observe a dynamical system that looks like the sequence  $\Pi(s_0), \Pi(s_1), \dots, \Pi(s_n)$ .

The timed-automata formalism extends this model, allowing states to modulate *clocks*, which count the passage of time [15]. Clocks may also be reset, so that they forget the elapsed amount of time. To be more precise, a set of clocks is a set of variables  $X$  that can be evaluated by a *clock valuation*  $v$ , assigning a positive real value to each clock. A transition may be taken if a certain input letter has been observed and the current clock valuation satisfies the guard of the transition, where a guard is conjunction of terms of the form  $(c < q)$ ,  $(c \leq q)$ ,  $(q < c)$ , and  $(q \leq c)$  for  $c \in X, q \in \mathbf{Q}_+$ . The guard of a transition is satisfied for a valuation  $v$  if each term  $(v(c) \text{ op } q)$  is valid in  $\mathcal{R}$ , where  $\text{op} \in \{<, \leq, >, \geq\}$ . Let  $\Phi(X)$  be the set of all such guard terms. A *timed-automaton*  $A_T = \langle V, V^0, V^F, X, E, \Pi \rangle$  over an input alphabet  $\Sigma_i$  and output alphabet  $\Sigma_o$  is given by:

1.  $V$  is a finite set of *locations*
2.  $V^0 \subseteq V$  is a set of initial locations
3.  $V^F \subseteq V$  is a set of final locations
4.  $X$  is a finite set of clock variables
5.  $E \subseteq V \times (\Sigma_i \cup \epsilon) \times \Phi(X) \times \mathcal{P}(X) \times V$  is a set of *switches*  $\langle s, a, g, \lambda, s' \rangle$  where the system may transition from  $s$  to  $s'$  if it observes the input letter  $a$  (or no input letter if  $a = \epsilon$ ) and the clock valuation  $v$  satisfies  $g$ . If the transition occurs, then the clocks  $\lambda \subseteq X$  are reset to the value 0.
6.  $\Pi : V \rightarrow \Sigma_o$  maps locations to observations.

Without delving too far into the theory of timed-automata, we can build an intuition for how this extension allows us to develop software in new ways. Let us imagine that we have a machine that supports several instructions:

1. **add**  $R_i, R_j, R_k$  causes  $R_k \leftarrow R_i + R_j$
2. **mul**  $R_i, R_j, R_k$  causes  $R_k \leftarrow R_i \times R_j$
3. **load**  $R_i, C$  causes  $R_i \leftarrow C$ , where  $C$  is a data value

Figure 1 shows a partial abstract machine, modeled as a timed-automaton, that reads the above assembly-language and modifies its state accordingly. The machine initially begins in a state where all registers have value 0 ( $R_i = 0$ ). In the first round of fetching (FETCH1) the machine can accept the data **LOAD**, but this will take between  $q_F^{\min}$  and  $q_F^{\max}$  units of time, as measured by the clock  $x_1$ . The range  $[q_F^{\min}, q_F^{\max}]$  captures the time it takes to fetch an instruction; this can be viewed as temporal non-determinism. After the load instruction is accepted, the machine expects a pair of data  $(R_i, C_j)$ , indicating which register should receive what data. Though a different state and transition must exist for every possible pair, the figure shows such a state and transition for the pair  $(R_1, C_1)$ . This transition is guarded by a range for the latch time  $q_L$ . After this time, the system goes to the state  $R_1 = C_1, R_{i \neq 1} = 0$ .

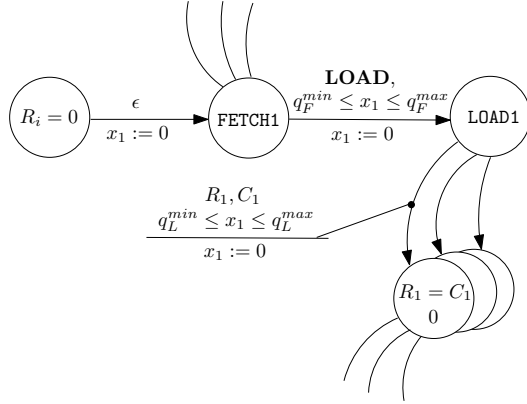


Figure 1. An abstract machine with a precise notion of time.

Given an abstract model such as this, a program consists of sequence of *timed events* of the form  $(d_i, t_i)$  where  $d_i \in \Sigma_i$  and  $t_i \in \mathcal{R}_+$ . A pair  $(d_i, t_i)$  denotes that the  $i^{\text{th}}$  instruction and/or data is fed to the machine at time  $t_i$ . If the machine accepts this sequence of timed events, then the untimed program can be executed with the specified timing properties. The set of all programs the machine  $M$  can accept is the *language*  $\mathcal{L}(M)$ . Typically the programmer does not specify timing information for every instruction. Instead, the programmer may define a function  $f$  using an *untimed* sequence of instructions/data ( $f \equiv d_0, d_1, \dots, d_n$ ), and then augment the basic **CALL** instruction with a requested timing range: **CALL**  $f [t^{\min}, t^{\max}]$ . This augmentation means that the call to function  $f$  is valid if there exists an accepted sequence of timed events that have the same instructions/data  $d_i$ , but execute within the time interval  $[t^{\min}, t^{\max}]$ . In another words, **CALL** succeeds if  $\exists (d_i, t_i)_{i \in I} \in \mathcal{L}(M)$ ,  $t_n - t_0 \in [t^{\min}, t^{\max}]$ . Since the abstract machine model is precise, it is possible to algorithmically decided if such a timing property is satisfied. No performance evaluation is necessary.

Though we have carried this example through with timed-automata, the same process can be repeated for other abstract machines. For example, this approach was applied to *time-triggered architectures* by defining a virtual machine, called the *E Machine*, that executes an extended assembly language [16]. The E machine includes assembly instructions that start periodic tasks (**schedule**  $j$ , for a task  $j$ ) and suspend tasks for a specified amount of time (**future**  $n, a_j$ , for  $n$  a unit of time,  $a_j$  an address in  $j$ ). The authors of this work also developed a high-level language called *Giotto* that is compiled into timed assembly code for the E machine. Once in this form, schedulability of the programs can be checked [17].

## Representing Concurrency

The previous examples extended computing to incorporate time, but not necessarily concurrency. Notice that a timed-automaton can be completely sequential, while still associating timing information with the sequential steps. Mathematically, we can explain how concurrently running automata interact by defining a *product operator* that converts a set of concurrent automata into a single monolithic automaton. This single automaton contains enough states and transitions to capture all the possible ways that each concurrent automaton could evolve with respect to the others. This is also a problem: The *product automaton* generally contains a combinatorial number of states and transitions, which makes it difficult to analyze and difficult for engineers to understand.

Finding the ideal means to express concurrency has been a research goal for decades. One approach is to build software from data transformers that consume and emit data through wire-like connections [18]. This approach is motivated by highly current hardware systems, which process data this way. For example, Figure 2 shows a simple one-bit adder (without a carry-in). The sum of the two bits ( $i_1, i_2$ ) is just the exclusive-OR and the carry-out is the logical AND of the bits. We imagine that bits arrive on the inputs

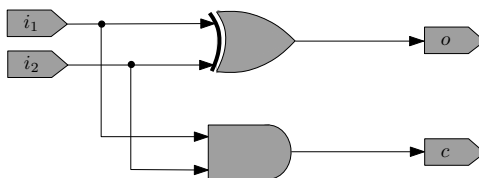


Figure 2. Example of concurrency in hardware notations.

and then flow through the wires to the XOR and AND gates. These gates read the data, process it, and then pass data onto the output wires. Notice that data can move simultaneously on different wires, so that the XOR and AND gates can produce outputs simultaneously. (The fan-out on the wires duplicates data.) Software systems using this approach are called *dataflow graphs*, *dataflow process networks*, or *process networks*, depending on the exact details of the computation. The computational objects are often referred to as processes, dataflow operators, actors, or nodes. The communication wires between nodes are similarly termed connections, channels, links, or edges. The process network view is attractive for several reasons:

1. States in an automaton are, by default, mutually exclusive and hence sequential. Processes in a network, by default, run in parallel and are thus concurrent.
2. The communication mechanism uses private point-to-point connections that cannot be modified by other processes. This eliminates the strange interactions that occur with shared variables.
3. Only data passes between processes; not control. Each process encapsulates its own control loop.

The behavioral properties of process networks depend heavily upon the properties of the processes and channels. For example, if we decide that processes are connected by infinite FIFOs, block on reads, and do not block on writes, then the system will always calculate the same results regardless of when individual processes read and write data. The proof of this relies on some technical assumptions about processes, and is due to G. Kahn [19]. Consequently, such dataflow systems are called Kahn Process Networks (KPNs). Amazingly, KPNs are immune to most of the problems that plague concurrent programming.

Unfortunately, KPNs cannot be implemented because they require infinite memory. However, there are many classes of process networks that can be implemented. Most of these are obtained by starting with the KPN model, and then bounding the FIFOs while requiring all processes to consume and produce data in some predictable fashion. For example, processes might always consume  $n$  units of data to produce  $m$  units of data, regardless of the particular data. In general, once the communication mechanism is bounded, the system becomes less immune to concurrency, unless the process behaviors are restricted in a corresponding way.

It is possible to define the semantics of classes of process networks using (concurrent) automata theory, but this is not the most intuitive formalism. It is more natural to imagine that processes map sequences of data “tokens” to sequences of data “tokens”. We make this more precise following the notation presented in [20]. Let  $\Sigma$  be an alphabet containing the possible data values that appear on connections. The set  $\Sigma^*$  contains all finite sequences of data (*Kleene closure* of  $\Sigma$ ), and the set  $\Sigma^{\mathbb{Z}^+} = \{f | f : \mathbb{Z}^+ \rightarrow \Sigma\}$  contains all infinite sequences of data. Let  $S = \Sigma^* \cup \Sigma^{\mathbb{Z}^+}$  be the set of all finite and infinite sequences of data tokens. A process  $P : S \rightarrow S$  maps sequences to sequences.

The internal state of a process can be completely abstracted away by defining the mapping appropriately. Consider the classic example of a system that remembers if it has seen an even or odd number of a particular input  $a$ . An automaton would do this using at least two states. A process has access to the entire input history, so it is not necessary to model this state. For example, take  $\Sigma = \{a, b\}$  and  $P_{eo}$  such that the  $i^{\text{th}}$  element in the sequence  $P_{eo}(S)$  is  $a$  if there are an even number of  $a$  tokens in the input subsequence  $[s_0, s_1, \dots, s_i]$ . Otherwise, the  $i^{\text{th}}$  element is  $b$ . We must also consider the *empty sequence*  $\perp$  that contains no data. Define  $P_{eo}(\perp) = \perp$ . The process has access to the entire to sequence, so we do not need to describe how  $P_{eo}$  remembers the number of  $a$  tokens seen.

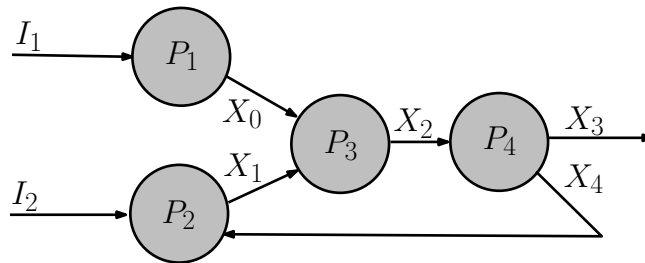
$$\begin{aligned} P_{eo}(\perp) &= \perp \\ P_{eo}([a]) &= [b] \\ P_{eo}([a, b, b, a]) &= [b, b, b, a] \end{aligned}$$

The properties of process networks depend heavily on the properties of individual processes. The most important properties of processes relate similar input sequences to similar output sequences. A sequence  $S$



is a prefix of a sequence  $S'$ , written  $S \sqsubseteq S'$ , if  $s_i = s'_i, 0 \leq i < \text{len}(S)$ .<sup>1</sup> The empty sequence  $\perp$  is a prefix of every sequence. A process  $P$  is *monotonic* if whenever  $X \sqsubseteq Y$ , then  $P(X) \sqsubseteq P(Y)$ . The example process  $P_{eo}$  is such a process. Without a property like monotonicity, it may be impossible to know the output of a process without feeding it an arbitrarily large amount of data. It is often the case that processes exhibit a stronger property called *continuity*. A process  $P$  is continuous if for every *ascending chain* of sequences  $C = \{X_0 \sqsubseteq X_1 \sqsubseteq \dots\}$  then  $P(\bigvee C) = \bigvee P(C)$ , where  $\bigvee \mathbf{Y}$  denotes the least upper bound of a set of sequences  $\mathbf{Y}$  with respect to prefixes.

Processes, such as the AND gate, read from more than one input channel. We handle this by extending processes to map from an  $n$ -tuple of sequences to an  $m$ -tuple of sequences, i.e.  $P : S^n \rightarrow S^m$ . It is also useful to define a projection operator (or projection process)  $\pi_{i,n} : S^n \rightarrow S$  that extracts the  $i^{\text{th}}$  sequence from an  $n$ -tuple of sequences, i.e.  $\pi_{i,n}((S_0, S_1, \dots, S_{n-1})) \mapsto S_i$ . Using projections, an entire process network can be represented as a set of equations that constrain the behaviors of each process in the network. The solutions to these equations yield the legal global behaviors of the network. To calculate the response of the network to a set of input sequences, we view the inputs as fixed constraints and solve for the internal/output sequences  $\{X_0, X_1, \dots, X_{n-1}\}$  that satisfy these constraints. Figure 3 shows an example of a process network



$$\begin{aligned}
 X_0 &= P_1(I_1) \\
 X_1 &= P_2(I_2, X_4) \\
 X_2 &= P_3(X_0, X_1) \\
 X_3 &= \pi_{0,2} \circ P_4(X_2) \\
 X_4 &= \pi_{1,2} \circ P_4(X_2)
 \end{aligned}$$

Figure 3. Example of a process network and its associated constraint system

and its associated constraint system. Solving these constraints can be tricky. For example, by substitution  $X_2 = P_3(X_0, P_2(I_2, \pi_{1,2} \circ P_4(X_2)))$  is a function of itself. A solution to this constraint must be a *fixed point* of the form  $X_2 = f(X_2; I_1, I_2)$ , where  $f$  is parameterized by  $I_1, I_2$ . In general, we can view an entire network as a solution to a fixed point equation of the form  $\mathbf{X} = F(\mathbf{X}, \mathbf{I})$ , where  $\mathbf{I}$  is a fixed set of input sequences

<sup>1</sup>By this definition, if  $S$  and  $S'$  are both infinite, then  $s_i = s'_i, i \geq 0$  therefore  $S = S'$ .

and  $\mathbf{X} = \{X_0, X_1, \dots, X_{n-1}\}$ .

This mathematical model lends itself to concurrency for several reasons. First, if the network contains continuous processes, then the response to a set of input sequences can be calculated iteratively by first feeding the set  $\mathbf{I}$  of external input sequences into the network with all the internal sequences initialized to the empty sequence  $X_i = \perp$ . The processes calculate a new set of sequences  $X_i^1$  using the initial value  $\perp$  for all of the internal inputs. This procedure is iteratively repeated; in the next iteration, each process uses the results from the previous iteration as inputs, i.e.  $\mathbf{X}^{j+1} = F(\mathbf{X}^j, \mathbf{I})$ . The procedure terminates when two consecutive iterations produce the same sequences, i.e.  $X_i^k = X_i^{k+1}$ ,  $0 \leq i < n$ . In this case,  $\mathbf{X}^k$  is the fixed point of the equation  $\mathbf{X} = F(\mathbf{X}, \mathbf{I})$ . Amazingly, this constructive process can be implemented by concurrently running processes that send data across the channels until the entire network stabilizes [21]. Thus, we can actually view a network of processes much like a circuit that stabilizes after some transient period of communication. (Analogously, some networks will not stabilize in finite time.) Verifying properties of process networks works in a similar manner. For example deadlock (also called *causality*) can be detected by an iterative procedure that analyzes how individual process consume and produce data tokens. An elegant exposition of causality analysis can be found in [22].

Comparing process networks with automata shows that there are some advantages of expressing concurrency with processes. The behaviors expressed by concurrent automata include every possible interleaving modulo a particular synchronization mechanism. The process network model allows us to move away from this, by viewing the computational objects (processes) as inherently concurrent instead of inherently mutually exclusive. This view provides benefits at both the implementation and verification levels. Automata typically communicate via *synchronous broadcast*: When a state emits an event this event is instantaneously observed by all other automata in the system. Implementing synchronous broadcast requires sophisticated distributed algorithms [23]. Verification of deadlock in concurrent automata may require analysis over all the product states, while many classes of process networks admit a simple analysis of token consumption and production rates. This is not an argument against automata. There also exist classes of process networks where many properties are undecidable. Additionally, automata are an intuitive imperative style of specification that continues to prove useful. Nevertheless, there are certainly situations where the process network viewpoint is appropriate.

### Simulating MoCs

A purely mathematical description of an MoC is necessary, but not sufficient for model-based design. In particular, engineers need something more tangible, e.g. derived algorithms that check model properties. At the very least, we expect to be able to simulate models on conventional machines. This is typically done by constructing a program that manipulates MoC-specific quantities, as represented in a traditional machine.

For example, a timed-automaton can be simulated like a traditional FSM, except that the simulator must manage the clocks and evaluate guards. It is important to remember that the length of time it takes to simulate a model may bear little or no resemblance to the predicted temporal properties of that model within the MoC. This is not surprising, considering that a simulator does not implement a MoC, but approximates it. With this caveat in mind, there are several approaches to MoC simulation, each of which leverages traditional software engineering principles.

### Simulating Transition Systems

The best approach to simulation depends on the particular MoC. Automata-based MoCs can be readily simulated, because they are already defined in terms of execution steps (evaluate guards/change state); i.e. they are *operational* definitions [24]. In fact, advances in automata-based specification languages have made it possible to simultaneously specify the operational semantics of an MoC and simulate that specification. Two key insights make this possible: First, most automata-like structures can be reformulated into a very simple structure called a *transition system* (TS). A transition system is a structure  $T = \langle \Gamma, \rightarrow \rangle$ , where

1.  $\Gamma$  is a set of configurations (or locations or states)
2.  $\rightarrow \subseteq \Gamma \times \Gamma$  is a binary relation on configurations.
3. If  $(q, q') \in \rightarrow$ , then the system can transition from state  $q$  to state  $q'$ .

Plotkin’s influential notes on *structural operational semantics*(SOS) [24] show how various operational semantics can be rendered as transition systems. Interestingly, reformulating an arbitrary structure into a transition system requires generalizing the notation of state. For example, Plotkin points out that an FSA can be made into a transition system if  $\Gamma = Q \times \Sigma_i^*$ , where  $\Sigma_i^*$  is the set of all finite strings over  $\Sigma_i$ . Squeezing an FSA into a TS yields a TS with an infinite number of configurations ( $|\Gamma| = |Q| + |\aleph_0|$ ), even though  $Q$  is a finite set. Thus, correctly defining the notion of configuration is essential to applying the TS formalism. However, this simple example shows that producing the correct reformulation is both practically and theoretically non-trivial. Gurevich, in his work on *abstract state machines* (ASM) [25], generalized the notion of configuration so that it could easily encompass many different structures. Specifically, he proposed that configurations should be *algebras* over a fixed signature  $\Upsilon$ , and a system transitions from one algebra to another.

An algebra  $A$  is a structure  $A = \langle U, \Upsilon \rangle$ , where  $U$  is called the *universe* of the algebra, and  $\Upsilon$  is called the *signature* of the algebra. A signature names a set of operations (function symbols)  $f_1, f_2, \dots, f_n$ , and defines the number of arguments (arity) required by each operation. The expression  $arity(f_i)$  denotes the arity of function symbol  $f_i$ ; clearly  $arity(f_i) \geq 0$  must hold. An operation of the algebra is a mapping from an  $arity(f_i)$ -tuple of  $U$  to  $U$ ;  $f_i : U^{arity(f_i)} \rightarrow U$ . Let  $\mathcal{C}(U, \Upsilon)$  be the class of all algebras defined over universe

$U$  with signature  $\Upsilon$ . An abstract state machine  $A$  over  $(U, \Upsilon)$  is a transition system with  $\Gamma \subseteq \mathcal{C}(U, \Upsilon)$ . The particular operations of the algebra form the state, so two states  $s, s'$  differ if there exists an operation  $f_i$  and a tuple  $t \in U^{\text{arity}(f_i)}$  such that  $f_i^s(t) \neq f_i^{s'}(t)$ . The notation  $f_i^s(t)$  indicates the operation  $f_i$  applied to  $t$  in algebra  $s$ .

Given this generalization, it is possible to implicitly define complex ASMs in a programmatic style. The language *ASML*[26] allows ASMs to be characterized by a set of statements of the form:

1. “**if** *conditional* **then** *update*”, where
2. *conditional* is a term  $f_i(x_1, x_2, \dots, x_{\text{arity}(f_i)})$  that yields a boolean value when evaluated against the current state  $s$
3. *update* is a pair  $(f_j(y_1, y_2, \dots, y_{\text{arity}(f_j)}), u)$  such that  $u \in U$ .

If the current state is  $s$ , and  $f_i^s(x_1, x_2, \dots, x_{\text{arity}(f_i)})$  evaluates to **true**, then the system may transition to a new state  $s'$ . In  $s'$  all the operations are the same as in  $s$ , except for operation  $f_j$  that maps the tuple  $(y_1, y_2, \dots, y_{\text{arity}(f_j)})$  to  $u$ . A single update changes exactly one operation at exactly one tuple, which is the smallest possible change that makes two states different. Let  $l = (y_1, y_2, \dots, y_{\text{arity}(f_j)})$ , then:

$$(f_i^s(x_1, x_2, \dots, x_{\text{arity}(f_i)}) = \mathbf{true}) \Rightarrow (s, s') \in \rightarrow,$$

$$\text{where } s' = \begin{cases} f_{k \neq j}^{s'} = f_k^s \\ f_j^{s'}(l' \neq l) = f_j^s(l') \quad , \text{ and } s' \in \Gamma \\ f_j^{s'}(l) = u \end{cases} \quad (\text{II.1})$$

To illustrate this, we will specify the execution rules of a timed-automaton as an implicitly defined ASM using ASML. We begin by describing the members of the universe  $U$ . By default, ASML adds many members to  $U$  including the real numbers (`Double`<sup>2</sup>), the integers (`Integer`), and  $\{\text{true}, \text{false}\}$  (`Boolean`). Specification 1 lists the necessary ASML code that extends the universe  $U$ . Lines 1-3 declare that  $U$  contains

**Spec 1.** Extending the universe  $U$  for timed-automation

---

```

1: enum LocationName
2: enum InputLetter
3: enum Clocks
4:
5: class Transition
6:   l as LocationName
7:   lp as LocationName
8:   i as InputLetter
9:   r as Set of Clocks
10:  var g as Map of (Map of Clocks to Double)
11:                        to Boolean

```

---

<sup>2</sup>Actually, the type `Double` is 64-bit floating point.

three new subuniverses, each of which contains a finite number of (enumerated) elements. We do not need to actually enumerate the distinguished elements at this point. The *LocationName* subuniverse is a reservoir of names for discrete states. The *InputLetter* subuniverse is a reservoir of letters for input alphabets  $\Sigma_i$ . Finally, the *Clocks* subuniverse contains names for clock variables. We call these *reservoirs*, because a single automaton does not need to use every element in each subuniverse, just as it does not need to use every integer in  $\mathbf{Z}$ . However, we can rely on  $U$  to contain the needed elements. Our usage of the term reservoir is similar in spirit to the usage of the term *reserve* in [25]. A reserve contains names for objects that may be dynamically introduced into a running ASM, though this example does not dynamically introduce new elements into the universe.

Transitions are more complex structures, but we can easily handle them with ASML. Line 5 declares a new subuniverse called *Transition*. (Note that the keyword **class** implies some additional technicalities.) Each member of this subuniverse is a 5-tuple of the form  $(l, i, g, r, l')$ , with the obvious relationship to transitions in timed-automata. One important detail is the representation of the guard  $g$ . Recall that a guard is evaluated against a clock valuation  $v : X \rightarrow \mathcal{R}_+$ , which maps clocks to nonnegative reals. Mathematically, this means that a guard maps clock valuations to booleans. For example, consider a guard  $x \leq 12$ , where  $x$  is a clock. This guard is really a mapping  $g : (X \rightarrow \mathcal{R}_+) \rightarrow \mathcal{B}$ , such that  $\forall v, g(v) \mapsto (v(x) \leq 12)$ . In ASML a (partial) function from set  $X$  to  $Y$  is identified with the notation **Map of  $X$  to  $Y$** . Thus, lines 10-11 identify  $g$  as map from clock evaluations to booleans. The reader may ignore the keyword **var**. The purpose of this keyword is to allow us to make  $g$  a partial function over the relevant valuation, which changes as the automaton executes.

The actual state of the system is captured by operations of the signature  $\Upsilon$ . However, not every operation contributes to state; ASML automatically provides many non-state operations, e.g. addition over integers. The keyword **var** identifies operations that do contribute to state. Contrarily, we can use the keyword **const** to denote an operation that does not effect state. Specification 2 lists the key members of  $\Upsilon$ . Line 12

**Spec 2.** Extending the signature  $\Upsilon$  for timed-automata

---

```

12: var  $v = \{ \text{clki} \rightarrow 0.0 \mid \text{clki in clocks} \}$ 
13: var  $crnt = \mathbf{any} \text{ qi} \mid \text{qi in } q0$ 
14: const  $time = \mathbf{new} \text{ Transition}(\text{empty}, e, \{\rightarrow\}, \{\}, \text{empty})$ 

```

defines the unary clock valuation operation, which initially maps every clock to zero. The notation  $m = \{ x \rightarrow y \}$  specifies that  $m$  maps value  $x$  to value  $y$ . Similarly,  $crnt$  is a nullary function that identifies the current discrete state of the system. Line 13 initializes  $crnt$  to some discrete state from the set of initial states  $q0$ . The ASML keyword **any** implements a non-deterministic choice, and picks some  $qi$  from the set of initial states. Finally,  $time$  is a special transition in every timed-automaton. At every choice point, the

system may take a “regular” enabled transition, or it may take the *time* transition. If the system takes the *time* transition, then all clocks are incremented by a fixed amount  $\epsilon$ . This discretization is an artifact of simulating a continuous system on a discrete machine. The *time* transition is a permanent part of every timed-automaton, so it is marked as constant.

The kernel of the simulator examines the enabled transitions available from the current state, and then takes one. Taking a transition may cause a change in state, which means that the operations  $v$  and  $crnt$  are updated. The key is to specify the rules for changing this state. Specification 3 shows the rules for finding the enabled transitions and taking one of those transitions. Lines 16-19 collect up the enabled transitions

### Spec 3. Kernel of simulation engine

---

```

15: TakeATransition()
16:   let takeTrans = any tj | tj in ( { tr | tr in transitions
17:     where tr.l = crnt and ( exists (ai,ti)
18:     in input where ( ai = tr.i and ti =  $v(t)$  ) )
19:     and (tr.g( $v$ ) = true) } union { time } )
20:
21:   if (takeTrans.l = empty) then
22:      $v := \{ \text{clk}i \rightarrow v(\text{clk}i) + \text{epsilon} \mid \text{clk}i \text{ in } \text{clocks} \}$ 
23:   else crnt := takeTrans.lp
24:      $v := \{ \text{clk}i \rightarrow 0.0 \mid \text{clk}i \text{ in } \text{takeTrans.r} \}$  union
25:     {  $\text{clk}i \rightarrow v(\text{clk}i) \mid \text{clk}i \text{ in } \text{clocks} - \text{takeTrans.r} \}$ 

```

---

and then non-deterministically choose one. An enabled transition  $tr$  is one that starts at the current state ( $tr.l = crnt$ ), satisfies the time guard ( $tr.g(v) = true$ ), and for which there exists an input pair  $(\alpha, \tau)$  that satisfies the trigger of a transition. The set *input* contains all the input pairs used during the simulation. A transition  $tr$  is triggered by a pair  $(\alpha, \tau)$  if  $\alpha = tr.i$  and  $\tau = v(t)$ , where  $t$  is mapped to the current time. ( $t$  is a clock that is never reset.) The *time* transition is unioned with the enabled transitions, and then one is non-deterministically chosen and placed in *takeTrans*. Lines 21-25 update the state. If *takeTrans* is the *time* transition, then the current valuation  $v$  is updated to a new map  $v'(x) \mapsto v(x) + \epsilon$ , effectively incrementing every clock by  $\epsilon$  units of time (Line 22). The ASML notation  $:=$  indicates a state update. If *takeTrans* is not the *time* transition, then the current discrete state is updated to *takeTrans.lp* (Line 23), and the valuation  $v$  is updated by setting  $v(x) \mapsto 0$  for each clock  $x$  in the reset set *takeTrans.r*.

The final piece of the simulator indefinitely takes transitions. A timed-automaton can always take the *time* transition, so the simulator does not terminate (except by the user’s request). Specification 4 shows the main simulation loop. In ASML we can just “call” the procedure *TakeATransition*; really this procedure call represents the composition of ASMs. The keyword **step** causes all the of the updates of the form  $expr_1 := expr_2$  to be applied simultaneously. ASML does not sequentialize updates, but performs many updates of the form of Equation II.1 at once. Thus, the specification does not require discrete state

#### Spec 4. Main simulation loop

```
26: Main()
27:   step while true
28:     step TakeATransition()
29:     step UpdateGuardMaps()
30:     WriteLine(v + ": " + crnt )
```

to change (Line 23) before clock resets occur (Line 24). The last point that deserves explanation is the *UpdateGuardMaps* of Line 29. This procedure redefines the guard maps after each transition, so that they are defined for the current valuation  $v$ . This is necessary because maps must be explicitly enumerated in ASML, and we cannot enumerate a complete guard map, as it has an infinite domain. Instead, we continually redefine each guard map  $g$  with respect to the current valuation  $v$ .

Though some effort is required, the basic timed-automata semantics can be described with a 30 line specification. In order to actually simulate a specific timed-automaton, we must add the necessary data to the specification. ASML allows a specification to be split across multiple lexical units, so we can keep the simulator as a pure abstract unit, and then add the model-specific data in a different file. Following the terminology of [27], this file is called the *abstract data model*. Specification 5 lists the data model for the simple automaton of Figure 4. Lines 10-12 define the *UpdateGuardMaps* for this specific data model.

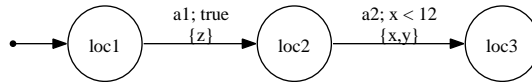


Figure 4. Timed-automaton represented by Specification 5.

Otherwise, the data model<sup>3</sup> is quite close to the original mathematical definition of a timed-automaton in Section II.2. After a data model and simulator have been combined, the model can be immediately simulated. Figure 5 shows a single partial simulation trace.

#### Simulating Process Networks

Specifying the simulation semantics of process networks can be a challenging task. (This can also be true for automata-based MoCs, e.g. hybrid automata.) There are two challenges to simulator development: First, the simulator must be able to determine how processes will respond to a partial sequence of data tokens.

<sup>3</sup>To save space, we have left out the elements of the enumerations in Lines 1-3 of Spec 1. These are also included in the data model.

### Spec 5. The data model for a simple timed-automaton

---

```
1: const epsilon = 0.1
2: q = { loc1, loc2, loc3 }
3: q0 = { loc1 }
4: input = { (a1, 0.3), (a1, 0.4), (a1, 0.5), (a2, 5.0), (a2, 12.1) }
5: clocks = { x, y, z, t }
6: transitions = [
7:   new Transition( loc1, a1, { v → true }, { z }, loc2),
8:   new Transition( loc2, a2, { v → (v(x) < 12.0) }, { x, y }, loc3) ]
9:
10: UpdateGuardMaps()
11:   transitions(0).g := { v → true }
12:   transitions(1).g := { v → v(x) < 12.0 }
```

---

Second, the simulator must contain a constructive procedure that correctly calculates the fixed point of a given process network. A typical simulation engine does not provide a control loop for every process, but uses a single thread of control that processes may borrow for short intervals. This permits the simulator to micromanage the evolution of each process, which is often necessary to efficiently direct the network towards a correct fixed point. The order and duration that processes gain control is called a *schedule*. A correct schedule effectively sequentializes a process network and is also the iterative procedure that leads a particular network to its fixed point. Thus, a simulator is an algorithm that generates the appropriate iterative procedure (schedule) for the arbitrary network it simulates.

Process networks are categorized by the difficulty of producing a schedule. Some classes can be *statically scheduled*, meaning a correct schedule can be calculated using only the topology of the network and the rules governing how processes consume and produce data [28]. Statically schedulable networks correspond to systems where the actual data values do not significantly impact how much data the processes consume and produce. Contrarily, *dynamically schedulable* networks may adjust the number of tokens they consume and produce based on the exact data values carried by the tokens. These networks cannot be scheduled without knowing the exact values of the data sequences. As a result, a simulator must continually adjust the schedule as external stimulus arrives [29].

Besides calculating the schedule for the entire network, the simulator must also calculate the individual (partial) responses of each process to a (partial) input stream. This too can be challenging because process behaviors can be idiosyncratic. The authors of [20] give an example of a monotonic process that produces different outputs depending on whether it is presented with a finite or infinite sequence. Mathematically, this process is easy to specify and has nice properties, but programmatically it is highly anomalous. The heterogeneous modeling and simulation framework *Ptolemy II* addresses these issues by providing an *abstract*



```

{ t→0.0, z→0.0, y→0.0, x→0.0}: loc1
{ t→0.1, z→0.1, y→0.1, x→0.1}: loc1
{ t→0.2, z→0.2, y→0.2, x→0.2}: loc1
{ t→0.3, z→0.3, y→0.3, x→0.3}: loc1
{ t→0.4, z→0.4, y→0.4, x→0.4}: loc1
{ t→0.4, z→0.0, y→0.4, x→0.4}: loc2
{ t→0.5, z→0.1, y→0.5, x→0.5}: loc2

```

Figure 5. Simulation of Specification 5 with ASML

*semantics* for process network simulation [3]<sup>4</sup>. An abstract semantics is a structured set of rules governing how process networks are described to the simulation framework. These rules allow the simulator to generate schedules for process networks with minimal additional work from the software engineer. Additionally, the framework restricts process behaviors to those that can be described programmatically.

Ptolemy II’s abstract semantics addresses this by requiring processes (called *actors* in Ptolemy II) to be specified by a set of firing rules. An actor *fires* by consuming input data and/or producing output data. A firing rule characterizes the conditions necessary for an actor to fire. A firing rule  $\mathbf{R}$  is an  $m$ -tuple of sequences,  $(r_0, r_1, \dots, r_m)$  where  $m$  is the number of inputs exposed by an actor. A rule is satisfied by an  $m$ -tuple of input sequences  $(I_0, I_1, \dots, I_m)$  if each sequence  $r_i$  is a prefix of the corresponding input sequence,  $r_i \sqsubseteq I_i, 0 \leq i < m$ . The sequences  $r_i$  are usually expressed as *patterns* where the pattern  $[*]$  is a prefix of any sequence with one token. For example, an actor with three inputs may have a firing rule  $\mathbf{R} = ([*, *], \perp, [1])$ . Such an actor would fire only if the first input had at least two tokens, the second input had zero or more tokens, and the third input had the data value 1 as its first token. An actor may have a set of firing rules  $\{\mathbf{R}_1, \mathbf{R}_2, \dots, \mathbf{R}_k\}$  and fires if at least one rule is satisfied. The number of tokens produced by an actor can be similarly described. Process networks with these types of firing rules are called *dataflow process networks*.

Ptolemy II is implemented in Java, and basic actors are implemented by subclassing the *AtomicActor* class [30]. This class introduces a number of important methods that a simulator can use to gather information about the actor. There is an *initialize* method that is called once per simulation, and initializes the actor’s internal state. It can also provide the simulator with an outline of the firing rules. We should mention that actor interfaces are more complex than simple channel readers/writers; they have typed and named *ports*. Consider a *SimpleActor* with two inputs  $a, b$  and one output  $c$ . We can specify that *SimpleActor* initially requires two tokens on  $a$ , one token on  $b$ , and produces three tokens on  $c$  by adding the following code to the initialize method:

```

a_tokenConsumptionRate.setToken(new IntToken(2));
b_tokenConsumptionRate.setToken(new IntToken(1));
c_tokenProductionRate.setToken(new IntToken(3));

```

<sup>4</sup>Ptolemy II also supports the simulation of discrete and hybrid automata.

The *Token* object encapsulates basic data values, hence *IntToken(3)* contains the integer value 3. By setting the appropriate token consumption and production members, the simulator can estimate the firing rule. This is an estimation because the consumption parameters do not indicate whether specific data values are required. To provide this functionality, each actor has a *prefire* member that returns *true* if the actor can fire, and *false* otherwise. The *prefire* member can test the values of the data tokens.

```
public boolean prefire() throws ... {
    return b.hasToken(0) && ( ((IntToken)b.get(0)).intValue() == 1 );
    ... }
```

This code in the *prefire* method requires the port *b* to have the integer value 1. As a simplification, the reader may ignore the argument 0 passed to the *get* and *hasToken* functions.

The Ptolemy II abstract semantics separates actor functionality between three methods: *prefire*, *fire*, *postfire*. The *prefire* method determines if the actor can fire. The *fire* method gets and sends tokens, but should *not* modify the internal state of the actor. Finally, the *postfire* method modifies internal state and may present a new firing rule to the simulator. As the simulator proceeds it will call the *prefire* method exactly once, the *fire* method zero or more times, and the *postfire* method exactly once. There is good reason for this: Sometimes finding a fixed point requires the simulator to test how an actor responds to different input values, necessitating many calls to the *fire* method per simulation step. If the *fire* method changes the internal state, then the actor is irrevocably advanced many times. By removing state changes from the *fire* method, the simulator can test how the actor responds to different data values without the actor remembering these tests. Unfortunately, not all actors can be implemented this way. Actors that do not follow this rule cannot be used in classes of process networks that require this rule.

Each dataflow class may put restrictions on the firing rules, the data that passes between actors, the channel properties, and the separation of state. Ptolemy II encapsulates these rules within a *director*. This permits a “plug-and-play” approach to simulation: The user models a network independently of the class, and then plugs in a director that simulates the network with respect to some class. Of course, some actors may break the rules of a class, and cannot be simulated by the corresponding director. Actors that can be simulated under many classes are called *behaviorally-polymorphic actors*. It was shown in [31] that dataflow classes can be modeled as a type system (lattice), such that if an actor can be correctly simulated in one type (class) of dataflow, then it can be correctly simulated in all subtypes of that dataflow class. These software engineering techniques allow simulators and actors to be reused correctly and with minimal effort from the engineer.

Figure 6 shows how all of these tools have been put together to effectively simulate a classic problem in software engineering, the *Elevator Problem* [32]. Even simple versions of the elevator problem are wrought with details concerning when and how buttons, indicators, and elevators respond. In this simplified version of the problem we focus on how process networks can effectively model the concurrency in the system. Our

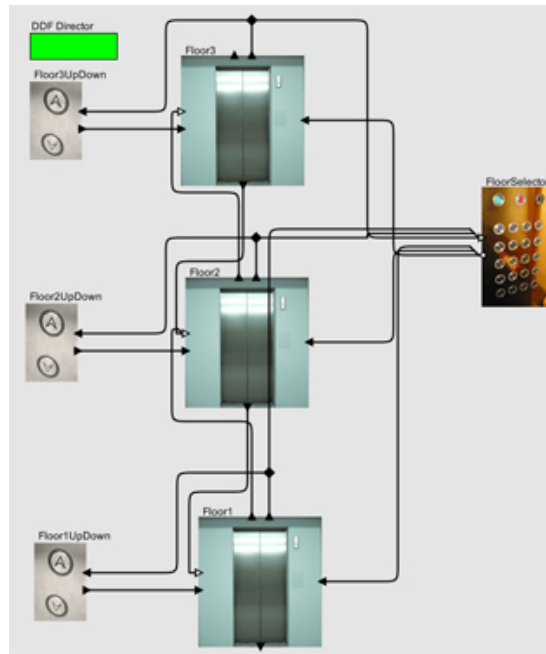


Figure 6. The Elevator problem in Ptolemy II

simplified view of the elevator problem is as follows:

1. A building has  $n$  floors with one elevator,
2. The elevator repeats the procedure:
  - a. If the elevator's direction is *up*, then it moves up until it reaches the top floor, at which point it moves down.
  - b. If the elevator's direction is *down*, then it moves down until it reaches the bottom floor, at which point it moves up.
3. The elevator starts at the bottom floor and so it has the direction *up*.
4. Each floor has an up/down panel. The elevator stops at floor  $i$  if the  $i^{th}$  up/down panel has been pressed in the direction the elevator is going and the elevator is at the  $i^{th}$  floor.
5. The elevator has a request panel with buttons  $\{floor_{min}, \dots, floor_{max}\}$ . The elevator stops at the  $i^{th}$  floor if it is at the  $i^{th}$  floor and the  $i^{th}$  request button has been pressed.

Concurrency appears in a number of places. Each floor has an up/down button that can be pressed independently of the elevator's request panel. Meanwhile, the elevator moves between floors. With process networks we can naturally describe the movement of the elevator as a data token that moves between floor actors. The center column of actors in Figure 6 shows three *Floor* actors, each connected to the other. The

floors pass the elevator around, which is a list of the form  $[dir, f_1, f_2, \dots, f_m]$ . The first element in the list specifies the direction of the elevator, and the remaining elements in the list are the unsatisfied floor requests.

Each floor actor must remember if the elevator is stopped at that floor, and in which direction the elevator was going. Thus, the *Floor* class has the following private members:

```
private boolean hasElevator = false;
private boolean sawGoingUp = false;
```

The  $i^{th}$  floor actor has two elevator input ports  $inFromAbove_i$ ,  $inFromBelow_i$  and two elevator output ports  $outToAbove_i$ ,  $outToBelow_i$ . Each  $inFromAbove_i$  can receive the elevator from  $outToBelow_{i+1}$ , and each  $inFromBelow_i$  can receive the elevator from  $outToAbove_{i-1}$ . (This holds, except for the top and bottom floors, which have some inputs unconnected.) Given the specification of the elevator, we can write the firing rules for the floors. If the elevator was not seen going up, then it must come from below. The firing rule for this state is  $(\perp, [*])$ . If the elevator was seen going up, then the next time it will come from above, so the firing rule is  $([*], \perp)$ . If the floor has the elevator, then it will send it out with no inputs, i.e.  $(\perp, \perp)$ . The elevator starts at the bottom floor, so initially the first firing rule will always apply. We subclass the *initialize* member to contain:

```
inFromAbove_tokenConsumptionRate.setToken(new IntToken(0));
inFromBelow_tokenConsumptionRate.setToken(new IntToken(1));
```

Similarly, the *postfire* method presents the correct firing rule to the simulator.

```
if (hasElevator) {
    inFromAbove_tokenConsumptionRate.setToken(new IntToken(0));
    inFromBelow_tokenConsumptionRate.setToken(new IntToken(0));
}
else if (sawGoingUp) {
    inFromAbove_tokenConsumptionRate.setToken(new IntToken(1));
    inFromBelow_tokenConsumptionRate.setToken(new IntToken(0));
}
else {
    inFromAbove_tokenConsumptionRate.setToken(new IntToken(0));
    inFromBelow_tokenConsumptionRate.setToken(new IntToken(1));
}
```

Each floor has its own up/down panel, as shown by the left-hand column of *UpDown* actors. The panels can be pressed independently from each other. The  $i^{th}$  *UpDown* actor has one output port  $upDownControl_i$ , which sends out the state of the panel. The panel state can be “no buttons pressed”, “only down”, “only up”, or “both buttons”. The  $i^{th}$  panel has one input port  $request_i$ , however we do not require this port to have any tokens for the panel to fire. When the port does have tokens, the panel sends its state through the  $upDownControl_i$  port. In this way, the panel is always active, but it only sends its state when explicitly requested. Thus, the *fire* method contains the code:

```
if (request.hasTokens(0)) {
    request.get(0);
    upDownControl.send(0, new IntToken(panelState));
}
```

The button panel inside the elevator is represented by the *FloorSelector* actor, which works in essentially the same way as the up/down panels. We connect a single floor selector actor to every floor, because it is shared across all the floors. When the elevator first arrives at a floor it sends a request for the status of the corresponding up/down panel and the floor selector. It then waits for the requests to arrive. The floor selector sends a (possibly empty) list of all the current floors selected. The request phase requires an additional state variable in the floor actor that records if the fire method should send the elevator out or send requests to the panels. Before the  $i^{th}$  floor sends the elevator, it appends any new requests to the elevator token and deletes any requests that were satisfied by arriving at the  $i^{th}$  floor.

Using the process network approach, we model concurrency by creating actors for each entity in the system and channels between communicating entities. We set the firing rules to capture when entities are active with respect to the state of the system. By using the Ptolemy II abstract semantics, we can easily extend the framework and correctly simulate the system. The rectangle in the upper-left hand corner represents the particular director<sup>5</sup> that we have chosen for simulation. Figure 7 shows some simulation results. This approach to the elevator problem has been studied in detail using Petri Nets [33], which can be considered as a class of process networks with a particular firing rule.

<pre>Initially at floor 1 going up Floors requested: {1,2} Reached floor 1 going up Floors requested: {1,2} Picked up passengers going up Dropped off passengers at this floor Outstanding requests at floor 2 Reached floor 2 going up Floors requested: {3} Skipped passengers going down Dropped off passengers at this floor Outstanding requests at floor 3</pre>	<pre>Reached floor 3 going down Floors requested: {} Picked up passengers going down Dropped off passengers at this floor No outstanding requests Reached floor 2 going down Floors requested: {1,2,3} Skipped passengers going up Dropped off passengers at this floor Outstanding requests at floors 1, and 3</pre>
--	---

Figure 7. Simulating the elevator problem in Ptolemy II

### Domain-Specific Compilers

We began this discussion by noting that programs are just syntactic constructs that can be executed by machines. We then extended the fundamental notion of a computing machine to include time and concurrency, and we showed how these machine classes can be simulated on traditional machines. We complete the circle of ideas by describing how programming languages are designed for extended computational classes. The key ingredients of a programming language for arbitrary MoCs are the same as those of traditional languages [34].

<sup>5</sup>This is the dynamic data flow (DDF) director.

1. a syntax describing well-formed programs
2. an editor for constructing programs
3. a compiler that translates programs into simulator instructions

As we have seen before, these ingredients will be extended in various ways to suit the increased complexity imposed by today's design problems. As a matter of terminology, the model-based community uses the term *model* for the object that is traditionally called a *program*. This terminology emphasizes that models may execute on totally different machines from a traditional program, even though we may be able to (approximately) simulate models on traditional machines. In another words, *models* are intended to *model* phenomena beyond the scope of Von Nuemann-like architectures, while *programs* are targeted for this class of architectures.

### Describing Syntax

Traditional programming languages evolved under pressures to move from assembly-based programming towards higher-level and methodologically sound languages. This evolution took two forms: syntactic and semantic. Syntactically, programming languages evolved to provide more complex notational mechanisms beyond lists of assembly instructions. Semantically, the language primitives evolved to represent many possible sets of assembly instructions, instead of a single instruction.

Pioneers in language design emphasized two properties of language syntax: (1) Syntax should be specified precisely. (2) Algorithms should exist that easily parse the syntax [1]. Foundational work on regular expressions and grammars showed that syntax can be defined precisely, and parsers can be automatically generated from these definitions. Modern programming languages are usually specified as BNF grammars, and these correspond to context-free languages. Beyond a handful of constructs (e.g. declaration of variables before their use) context-free languages support most of the syntactic flexibility found in mainstream languages. Furthermore, restrictions on the BNF grammars lead to efficient parser implementations (e.g. LALR, shift-reduce parsers) [34]. These technologies have solidified themselves as the de facto approach for syntax specification. Consequently, most language evolution occurs on the semantic side. For example, even Dijkstra's famous argument against `goto` statements is an argument on the semantics of `goto`; not its syntactic representation [35].

Unlike traditional programming languages, model-based design continues to evolve syntax, because many models are naturally represented as graphs and well-formed models correspond to graphs with complex structural constraints. Figure 8 shows a typical embedded system model using a process network-like notation. Assume that a process fires when every input has a token, and a process produces a token on every output when it fires. In this case, the connections in the model also indicate data dependency; a process  $p$  depends on  $q$  (written  $q \rightarrow p$ ) if there is a directed path from  $q$  to  $p$  in the model. Under these assumptions, a

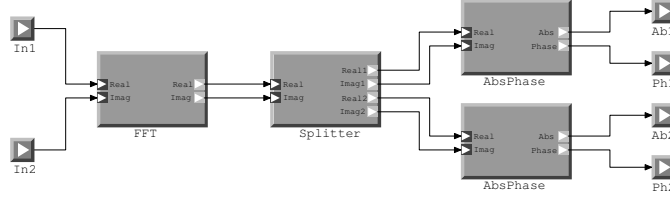


Figure 8. Example of a model represented as a directed graph

network deadlocks if a process depends on itself ( $p \rightarrow p$ ). Thus, models should be constrained so that cycles are disallowed.

Handling these sorts of constraints requires an expressiveness of syntax not found in traditional approaches. In the interest of space, we will show that this constraint does not correspond to a regular language. The reader may continue the analysis for context-free languages. The first step in the analysis is to provide an encoding of a directed acyclic graph as strings from an alphabet. In order to simplify the problem, we will throw out all syntactic adornments, and consider strings that list the edges of a graph in an arbitrary order. Figure 9 shows several example graphs. A digraph is encoded as a string by arbitrarily

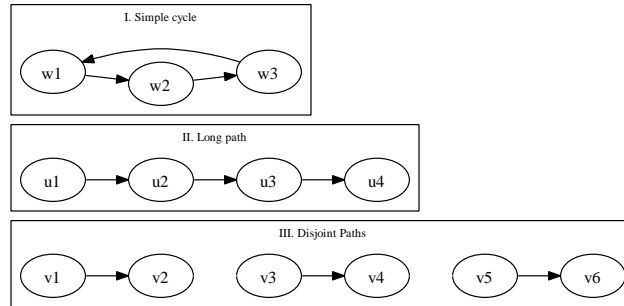


Figure 9. Several digraphs; graph I is not in the language, but II and III are in  $\mathcal{L}_{DAG}$ .

ordering the edge relation, and then listing the vertices incident on each edge. For example, the simple cycle of Figure 9.I could be encoded as  $w_1w_2 w_2w_3 w_3w_1$ ; another possibility is  $w_1w_2 w_3w_1 w_2w_3$ . For simplicity, we will ignore graphs with *orphans*, i.e. vertices with no edges. A permutation of the vertex labels  $v_1v_2 v_3v_4 \dots v_{n-1}v_n$  corresponds to a set of disjoint 2-paths; Figure 9.III is an example of such a graph. Let  $V$  be a set of vertices, and define  $\mathcal{L}_{DAG}(V)$  to be the language of directed acyclic graphs on  $V$  vertices:

1.  $\mathcal{L}_{DAG}(V) \subseteq \Sigma^*$ , where  $\Sigma = V$
2.  $\forall u \in \mathcal{L}_{DAG}(V), 2 \mid |u|$

3.  $\forall u \in \mathcal{L}_{DAG}(V)$ ,  $G(u) = (V(u), E(u))$  is acyclic, where  $V(u) = \bigcup_{i=1}^{|u|} u_i$  and  $E(u) = \bigcup_{i=1}^{\lfloor \frac{|u|}{2} \rfloor} (u_{2i-1}, u_{2i})$ .

Property 1 states that the alphabet of the language is exactly the vertex labels. Property 2 requires each string to have even length, because there are always two vertices per edge. The most important property is 3, which associates a graph with a string  $u$  and requires this associated graph to be acyclic.

The next task is to check if the language is regular. Already, we have some intuition that the language is not regular, so the first plan of attack is to check if it fails the well-known *Pumping Lemma*, which states the following: If  $\mathcal{L}$  is a regular language then  $\exists n > 0$  such that  $\forall u \in \mathcal{L}$  where  $|u| \geq n$ ,  $u$  can be written as the concatenation of substrings  $x, y, z \in \Sigma^*$ , ( $u = xyz$ ) such that:

1.  $\forall i \geq 0, xy^i z \in \mathcal{L}$ , ( $i \in \mathbb{Z}_+$ )
2.  $|y| > 0$
3.  $|xy| \leq n$ .

It is easy to see that this language will *not* fail the Pumping Lemma, or extensions thereof [36]. If a graph is acyclic, then deleting an edge, i.e. setting  $i = 0$ , will not make the graph cyclic. We can always decompose the string representation of an acyclic graph with more than 3 edges ( $n = 6$ ) into three parts:  $x = u_1 u_2$ ,  $y = u_3 u_4$ ,  $z = u_5 \dots u_{|u|}$ . In this case, duplicating the edge  $u_3 u_4$  an arbitrary number of times will not make the graph cyclic. Thus, the Pumping Lemma is not helpful for reasoning about  $\mathcal{L}_{DAG}(V)$ .

In order to show that  $\mathcal{L}_{DAG}(V)$  is not regular, we must make the more difficult argument that there does not exist any deterministic finite state automaton (DFA) that accepts the language. This can be done with the *Myhill-Nerode Theorem* that uses an equivalence relation  $\equiv_{\mathcal{L}}$  over the strings of an arbitrary language  $\mathcal{L}$ :

1.  $\equiv_{\mathcal{L}} \subseteq \mathcal{L}^2$  is reflexive, symmetric, and transitive
2.  $x \equiv_{\mathcal{L}} y$  if  $\forall z \in \Sigma^*$ , ( $xz \in \mathcal{L} \Leftrightarrow yz \in \mathcal{L}$ )

The theorem states that there exists a DFA that recognizes  $\mathcal{L}$  iff  $\equiv_{\mathcal{L}}$  contains a finite number of equivalence classes; a review and some extensions of this theorem can be found in [37]. In order to capture the language of all finite directed acyclic graphs, we choose the vertex set to be a countably infinite set  $\Delta$  ( $|\Delta| = |\aleph_0|$ ). Let  $G$  be any acyclic digraph without orphans, and let  $s(G)$  be any string  $u \in \mathcal{L}_{DAG}(\Delta)$  such that  $G(u) = G$ . Consider any two acyclic graphs  $G$  and  $H$  where  $G$  contains two vertices  $v_1, v_2$  such that:

1.  $v_1 \neq v_2 \in V_G$
2.  $v_1, v_2 \notin V_H$
3.  $(v_1, v_2) \in E_G$

We notice that  $s(G) \not\equiv_{\mathcal{L}} s(H)$  because  $s(G)v_2v_1 \notin \mathcal{L}_{DAG}(\Delta)$  but  $s(H)v_2v_1 \in \mathcal{L}_{DAG}(\Delta)$ . We can always find an infinite number of distinct finite labeled digraphs  $G$  and  $H$  that satisfy (1)-(3), therefore  $\equiv_{\mathcal{L}}$  has an infinite number of equivalence classes and  $\mathcal{L}_{DAG}(\Delta)$  is not a regular language. Another approach the proof



is to analyze a particular equivalence class; the set of all strings  $u$  such that  $G(u)$  is isomorphic to a graph of disjoint 2-paths. These strings are just permutations of  $|u|$  vertices, and it can be shown that no DFA with  $|u|$  states can correctly distinguish the language of disjoint paths from graphs with cycles. Thus, for any DFA with  $n$  states, it will not correctly identify finite DAGs with at least  $n$  vertices. In the interest of space, we do not present this alternative proof here.

It is possible to extend syntax to capture complex structural constraints encountered in model-based design. We accomplish this by noting that traditional syntax is defined over a particular algebraic structure called the *free monoid*. The free monoid  $M_F$  over  $\Sigma$  is an algebra with universe  $U$  whose elements are generated by  $\Sigma$ , and has a binary operator  $\circ$  that concatenates elements of the alphabet into strings. In another words the “string”  $\sigma_1\sigma_2\sigma_3$  can be viewed as repeated applications of the concatenation operator, e.g.  $(\sigma_1 \circ \sigma_2) \circ \sigma_3$ . Additionally,  $M_F$  has a distinguished element  $\epsilon$  called the *identity element*, and satisfies the following axioms:

1. Associativity:  $\forall \sigma_{1,2,3} \in U, (\sigma_1 \circ \sigma_2) \circ \sigma_3 = \sigma_1 \circ (\sigma_2 \circ \sigma_3)$
2. Identity:  $\forall \sigma \in U, \epsilon \circ \sigma = \sigma = \sigma \circ \epsilon$

In this case,  $\epsilon$  is the empty string. A natural extension is to construct syntax from a general algebra, and not just this particular class of algebras. We have explored this by generalizing syntax to sets of terms over the *term algebra*[38] of an arbitrary signature  $\Upsilon$ . Though the next chapter fully describes these results, we briefly summarize this now. We associate a set of operators with the concepts of the language [39]. For example, directed graphs utilize vertices and edges. Associate a unary function symbol  $v$  for the concept of vertex, and a binary function symbol  $e$  for the concept of edge. The term algebra  $T_\Sigma(\{v, e\})$  contains all *terms*, i.e. all possible ways to apply  $v, e$  to each other and members of  $\Sigma$ . A model is just a subset of these terms. A language of models  $\mathcal{M}$  is a subset of the powerset of terms:  $\mathcal{M} \subseteq \mathcal{P}(T_\Sigma(\Upsilon))$ . Thus, we might describe the graph in Figure 9.I as the set of terms  $\{v(w_1), v(w_2), v(w_3), e(w_1, w_2), e(w_2, w_3), e(w_3, w_1)\}$  from  $T_\Sigma(\{v, e\})$ . Notice that this encoding removes the artifact that edges had to be ordered in a string.

Just as with regular and context free languages, we need algorithms that decide if models (sets of terms) are well-formed. Deductive logic provides a natural framework for reasoning about sets of terms, because it allows us to *derive* new terms from old ones. A particular model  $m \in \mathcal{M}$  is well-formed, if well-formedness can be derived using some predetermined *consequence operator*  $\vdash$  (inference procedure) with axioms that characterize well-formed models. This replaces the DFA or pushdown automata (PDA) of regular and context free languages with a tunable inference procedure and axioms that characterize the well-formed structures of the language. We can adjust the expressiveness of language syntax by selecting the appropriate consequence operator. We use the term *structural semantics* instead of syntax, because the formal foundations may be arbitrarily expressive.

These extensions provide a formal underpinning for the syntax of models, but they do *not* suggest a

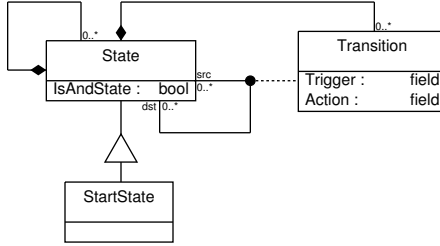


Figure 10. Example metamodel for hierarchical finite state machines.

particular syntactic notation for describing syntax (e.g. BNF grammars). The model-based community has employed a notation for defining syntax based on a subset of the Unified Modeling Language (UML) called *class diagrams* [40]. A class diagram that defines the syntax of a language is called a *metamodel*. UML-based metamodeling can be formalized using the extensions just described, though it has long been used without a formal characterization of the associated structural semantics. With this in mind, we informally summarize metamodeling with UML class diagrams. As the name suggests, a class diagram enumerates a set of classes. Each class encapsulates named members that are also typed. For example, Figure 10 shows a metamodel that describes the syntax of a hierarchical state machine language using the particular notation of *Meta-GME* [41]. The boxes in the model are class definitions, and class members are listed under the class names. For example, the **Transition** class has **Trigger** and **Action** members, both of type **field** (or string). The metamodel also encodes a family of graphs by associating some classes with vertices and other classes with edges. The **State** and **StartState** classes correspond to vertices; instances of the **Transition** class are edges. The diagram also declares which vertex types can be connected together, and gives the edge types that can make these connections. The solid lines passing through the connector symbol ( $\bullet$ ) indicate that edges can be created between vertices, and the dashed line from the connector to the **Transition** class indicates that these edges are instances of type **Transition**. The diagram encodes yet more rules: Lines that end with a solid diamond ( $\diamond$ ) indicate hierarchical containment, e.g. **State** instances can contain other states and transitions. Lines that pass through a triangle ( $\triangle$ ) identify inheritance relationships, e.g. a **StartState** inherits the properties of **State**.

Metamodels may also include more complicated constraints. For example, *multiplicity constraints* specify that vertices of type  $t_v$  must have between  $n_{min}$  and  $n_{max}$  incident edges of type  $t_e$ . In Figure 10 all multiplicity constraints contain the entire interval  $[0, \infty)$ , denoted  $0..*$ . More complicated constraints, e.g. graphs must be acyclic, can be denoted via a constraint language. The Object Constraint Language (OCL) is commonly paired with UML class diagrams to denote complex constraints. OCL is a strongly-typed first-

order calculus without side effects [42]. For example, the following side-effect free helper method can be used to check for cycles:

```

Descendants( children : ocl::Bag ) : ocl::Bag
  if(children.count( self ) < 2) then
    Bag{self} + self.connectedFCOs("dst") ->
      iterate( c ; accu = Bag{} | accu +
        c.Descendants(children + Bag{self}) )
  else( children ) endif

```

The *Descendants* method can be called on any vertex of any type in the model. The special identifier *self* refers to the object on which the method was invoked. Cycles are collected by passing a multiset (called a *Bag* in OCL) of previously seen vertices through recursive invocations of *Descendants*. Initially, a vertex *v* is passed an empty bag: *v.Descendants(Bag{ })*. If *v* has been seen only zero or one times, then all of its immediate children are iterated over using the expression *self.connectedFCOs("dst")→iterate*; the placeholder *c* is the iterator “variable”. Each immediate child is passed the current bag of visited vertices unioned with the current vertex. In this case, the method returns a multiset union of all vertices reachable from the current vertex and its immediate children. If the current vertex is already in the bag two or more times, then it may be in a cycle, so the passed in bag is immediately returned ending the recursion. Using the helper method, we require every vertex in the graph to satisfy the following invariant:

```

self.Descendants(Bag{ }).count( self ) < 2

```

This invariant only checks if the initiating vertex is contained twice in its descendants, but the invariant is checked for every vertex. This correctly detects cycles even in multigraphs.

Traditional language design employs parser generators or compiler compilers to automatically generate software that parses a particular syntax. These tools have been generalized by the model-based community to support metamodels and complex constraints on metamodels. The adjective *metaprogrammable* is used to describe tools that can conform themselves to a particular metamodel. For example, the metaprogrammable model editor called *GME* (Generic Modeling Environment) can reconfigure itself to construct models that adhere to a particular metamodel [43]. Figure 11.I shows the result of reconfiguring GME with the hierarchical automata metamodel of Figure 10. Several models from other DSMLs are also shown. Double-clicking on a state (blue circle) causes GME to open a window that contains the internal states and transitions of a state. The full GME metamodeling language supports many more features including *ports*, which are the structural representation of interfaces, and multiple aspects, which partition a modeling language into multiple dependent views.

## Semantic Analysis

The final component of an MoC-specific compiler is the code generator. In traditional language design the parsing phase of the compiler produces an abstract syntax tree, and the *semantic analysis* phase of the

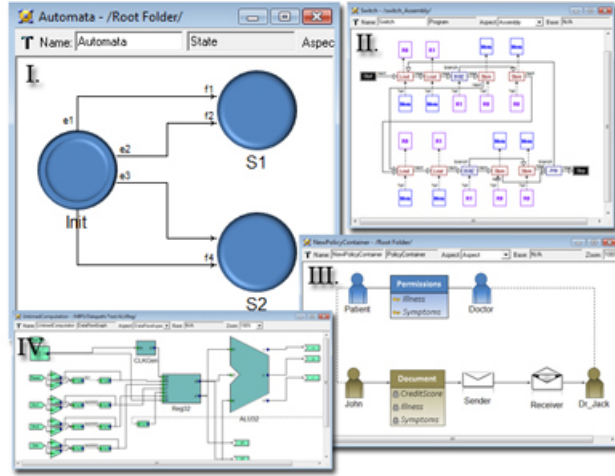


Figure 11. Example models in the metaprogrammable modeling environment GME: I. FSA model II. Assembly code model III. Access control model IV. Synchronous dataflow model

compiler walks this tree and emits code. (Some consider code emitting as a separate phase.) Most of the software engineering effort occurs in this later phase, because the many intricacies of the language prevent automatic generation of this component. The same problem occurs for MoC-specific compilers/interpreters. Additionally, the semantic analysis phase walks a generalized graph, not just a tree. The earliest methodological approaches to semantic analysis emphasized generic well-structured APIs (application programming interfaces) that simplified traversal of arbitrary model structure. For example, GME provides such a C++ API called *BON* (Builder Object Network) that represents model elements as instances of three classes: *Atom*, *Model*, and *Connection*. (BON also provides classes for additional structural features, but these are beyond the scope of this review.) Instances of *Atom* correspond to vertex-like elements that do not have any further substructure, while instances of *Model* correspond to elements with substructure. Instances of *Connection* correspond to edge-like modeling elements. The API provides a suite of methods to traverse the BON representation of a model. For example, we can get all of the outgoing edges from an *Atom* *a*.

```
std::set<BON::Connection> connections = a->getOutConnLinks();
```

Similarly, we may collect all the elements in the substructure of a *Model* *m*.

```
std::set<BON::FCO> subelements = m->getChildFCOs();
```

Note that the class *BON::FCO* is an abstract superclass of the basic model elements. Using this API, we can easily traverse the containment hierarchy of an arbitrary model.

Common traversal methods, like the one above, can be generalized into well-known software engineering patterns, e.g. the *Visitor Pattern*. The BON API supports a number of these patterns, thereby improving

```

void GetHierarchy (BON::Model m, std::set<BON::FCO>& substructs) {
    //Add the current model
    substructs.insert(m);
    //Iterate over each child
    std::set<BON::FCO> subs = m->getChildFCOs();
    for (std::set<BON::FCO>::iterator i = subs.begin(); i != subs.end(); ++i) {
        substructs.insert(*i);
        if (BON::Model(*i)) GetHierarchy (BON::Model(*i), substructs);
    }
}

```

the reusability and maintainability of code, while decreasing the time to produce a working compiler. A significant evolution of the API approach occurred through the development of the *Unified Data Model* (UDM) [44]. UDM generates a custom API from a metamodel by converting elements of the class diagram into C++ classes. Attributes become typed members in the generated classes, and methods for accessing/mutating attributes and traversing model connections/hierarchy are automatically generated. This allows the software engineer to leverage the C++ type system when developing a domain-specific compiler.

The API approaches are effective for implementation, but less useful for high-level specification of the semantic analysis phase. Ideally we would like to specify the compiler backend without appealing to the implementation details of the underlying API. This goal has been pursued for traditional compiler construction, where it can be assumed that the parser produces an abstract syntax tree (AST). The authors of [45] view the semantic analysis phase as a set of patterns that are matched against an input AST. They provide a language for abstractly describing subtree patterns along with actions that should be executed in response to those patterns. In this way, the compiler backend can be generated from the pattern/action descriptions. This not only reduces coding effort, but also provides a high-level specification of the compiler backend.

Clear specification of the compiler backend is essential for domain-specific languages. In model-based design the compiler output is often used as input to formal verification tools. If the compiler produces incorrect output, then the results of downstream verification tools may be inaccurate. For example, behavioral properties of timed-automata can be checked with verification tools such as IF [46] or Uppaal [47]. The verification results faithfully capture the properties of an input model only if the compiler produced an accurate timed-automaton representation of that model. Thus, it is critical to provide some notion of compiler correctness. Admittedly, this correctness problem is still open; in fact Hoare identifies it as a grand challenge for computing [48]. Nevertheless, approaches based on high-level compiler specification seem the most feasible.

The model-based community views compiler specification as a *model transformation* problem. Let  $\mathcal{M}$  and  $\mathcal{M}'$  be two sets of models. A model transformation is a mapping  $\tau : \mathcal{M} \rightarrow \mathcal{M}'$ . Notice that a compiler  $S$  can be viewed as a model transformation  $\tau_S$ . If the compiler  $S$  generates C code, then the codomain of  $\tau_S$  is just  $\mathcal{M}_C$ , the set of all syntactically well-formed C programs. Model transformations are typically specified using

*graph rewriting rules*, which are a generalizations of the subtree matching patterns used in [45]. Abstractly, a graph rewriting rule or *production* is a pair of graphs  $(L, R)$ . A rule can be applied if the input graph (*host graph*)  $G$  contains a subgraph  $S(G)$  that is isomorphic to  $L$ . In this case,  $S(G)$  is removed from  $G$  and a subgraph  $S'$  isomorphic to  $R$  is put in its place. By “replacing”  $S(G)$  with  $S'$ , we implicitly mean that  $S'$  is reconnected into  $G$  in some manner. This mechanism is called the *embedding*, and the flexibility of the embedding mechanism affects the expressiveness of the graph rewriting system [49]. Despite this, most practical graph rewriting systems opt for simpler and more intuitive embedding mechanisms. A comparison of existing graph rewriting tools can be found in [50].

A model transformation may be viewed as a set of graph rewriting rules. A mapping  $\tau$  converts an input model to an output model by repeatedly applying rewriting rules until no more rules can be applied. This procedure encounters problems similar to those of process networks. Is the transformation determinate, i.e. does it produce the same result regardless of the order in which the rules are tested? Does it have a finite fixed-point, i.e. does the transformation terminate? These questions are difficult to answer because, in general, rewriting rules are neither commutative nor associative so the order of application cannot be ignored. Some approaches to analysis of graph rewriting systems can be found in [51] [52].

Model transformations have been successfully applied to a number of DSMLs. A particularly relevant example was presented in [53] where the authors developed a domain-specific compiler from the well-established Stateflow/Simulink DSML to C using model transformations. We will present a scaled-down version of this work that generates C code from a finite state acceptor (FSA) language, also using model transformations to implement the compiler. In particular, we will use the *Graph Rewriting and Transformation* (GReAT) language [54], which was used by the authors of [53]. GReAT is integrated with the GME tool-suite, and permits simple descriptions of rewriting rules in terms of input and output metamodels.

Figure 12 shows the “input” metamodel of the transformation. (We can expect that all models fed to the compiler will conform to this metamodel.) Input models consist of finite state acceptors. An instance of FSA contains instances of the **State** and **Transition** classes. In order to simplify the rewriting rules, we have separated the initial states and acceptor states into two subclasses: **Initial** and **Acceptor**. This simplification does not permit acceptor states that are also initial. Instances of the **Event** class enumerate members of the input alphabet  $\Sigma$ . The attribute **guard** of a transition is a textual field that names one of the **Event** instances.

The output metamodel encodes a structured subset of C needed to implement FSAs. We will present this metamodel by working backwards from the generated code. The generated C code is based on a well-known and efficient technique for implementing automata in C [55]. Specification 6 outlines the approach in pseudocode. Two enumerations encode the states and events of the automaton (Lines 1,2). A variable called *currentState* stores the current state of the automaton, and it is initialized to the initial state *Sk* (Line

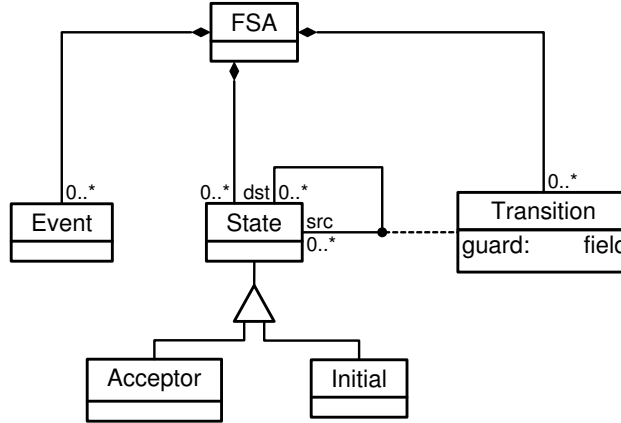


Figure 12. Metamodel of the input language; a finite state acceptor (FSA) language.

4). The program must have a mechanism to read events from the environment; we assume that a class `EventStream` exists to accomplish this task (Line 3). Most of the work is done in the **while** loop (Line 7) that repeatedly reads an event from the environment and stores it in the variable `currentEvent`. The loop contains nested **switch** statements; the outer **switch** chooses a case using the `currentState` (Line 8), and contains a **case** statement for every state of the automaton (e.g., Line 9). Each outer **case** contains an inner **switch** that chooses a case using the `currentEvent`. The inner switches contain **case** statements for each possible transition that the automaton can take from the corresponding state. The labels of the inner cases are the `EVENT` enumeration elements that guard the transitions. For example, if an automaton had the transition  $S_1 \xrightarrow{E_i} S_i$ , then the code would have the inner **case** shown in Line 11. This case correctly updates the `currentState` to the new state  $S_i$  (Line 12). This encoding scheme effectively implements a transition table by using the efficient **switch** statement. Though not shown, each inner **switch** contains a **default** case that breaks the simulation loop. This halts the machine if an improper sequence of events is presented.

Using the structure of the C code as a guide, we obtain the metamodel shown in Figure 13. This metamodel almost exclusively relates classes by containment; a reflection of the fact that it encodes a C abstract syntax tree. The root of the AST is an instance of `FSAProgram`, which contains exactly one child node of type `Declarations` and one of type `SimLoop`. The `Declarations` node contains one or more children of type `Variable` and one or more children of type `Enumeration`. The `Variable` class has an attribute `type` for specifying the type of the variable. Each instance of `Enumeration` contains 1 or more instances of `EnumElement`. Thus, the `Declarations` subtree contains all the parts for defining the necessary variables and enumerations that implement a FSA. The `SimLoop` subtree is necessarily more complicated. In particular, the `SimLoop` must know which variable corresponds to the `EventStream` and which corresponds to the `currentState`. This is

**Spec 6.** Pseudocode for executing a finite state acceptor.

---

```
1: enum STATES { S1 = 0, S2, ..., Sn };
2: enum EVENTS { E1 = 0, E2, ..., Em };
3: EventStream evStream;
4: int currentState = Sk;
5: int currentEvent;
6:
7: while (evStream.read(currentEvent) ) {
8:     switch (currentState) {
9:         case S1:
10:            switch (currentEvent) {
11:                case Ei:
12:                    currentState = Si; break;
13:                case Ej ... } break;
14:            case S2 ... }
15: }
```

---

handled by a feature of GME called a *reference association*. A reference association is much like a reference in traditional languages; it “points” to another object in the model (program). The *SimLoop* node contains exactly one instance of *EventStreamVar*, which is a reference to a variable. Presumably, the *EventStreamVar* instance will refer to the actual variable that should be used to read events from the environment. Similarly, the *SimLoop* instance contains a *CurrentEventVar*, which refers to the *currentEvent* variable. (In GME the reference association is indicated with a directional association with rolename *refers*.) The rest of the metamodel describes how **switch**, **case**, and variable assignments can be nested. In order to provide user feedback a case statement may contain an instance of the *Message* class, which is a placeholder for a **printf** statement.

The output metamodel closely resembles the set of C code ASTs corresponding to FSA implementations. It is a simple exercise to generate actual code from such an AST model, and we can be (fairly) sure that such a generation procedure is correct. The more complicated task is the transformation of a FSA model into a C AST model. Figure 14 shows the graph rewriting rule that fills the *STATE* enumeration with elements. In order to understand this rule, we must describe GReAT in more detail. Graph transformation tools locate all subgraphs of the input that are isomorphic to rule patterns. Unfortunately, subgraph isomorphism is computationally difficult (NP-complete), so it must be implemented carefully. GReAT’s approach is to provide rules with *context vertices*; rules only match subgraphs that include the context vertices. Context vertices are passed into a rule via “input ports”. The rule in Figure 14 is passed two context vertices, as indicated by the two input ports labeled *FSAIn* and *StatesIn*. The context vertices are actually typed instances of metamodel classes; they can be cast to particular types by making connections from the ports to class instances. For example, the connection from *FSAIn* to an instance called *iFSA* of type *FSA* casts the



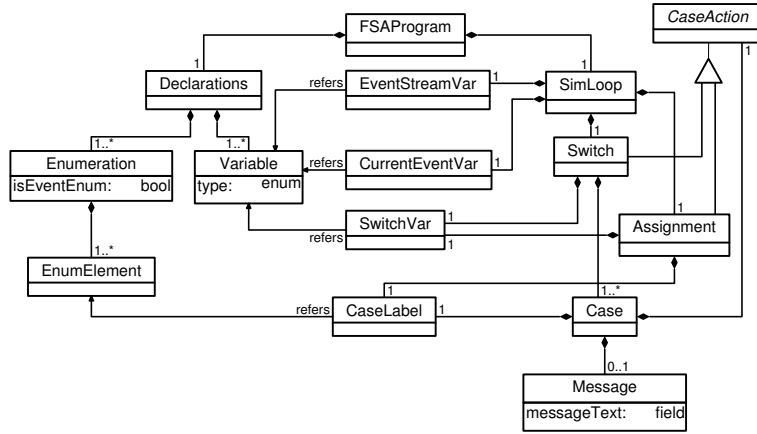


Figure 13. Metamodel of the output language; a structured subset of C.

context vertex *FSAIn* to an *FSA* instance. The instance name *iFSA* is used to refer to the context vertex locally within the rule, and is not the actual name of the instance. As the names imply, this rule is provided with the *FSA* instance of the input model and *STATES* enumeration instance of the output model.

A GREAT rule finds all subgraphs that contain the context vertices and are isomorphic to the submodels drawn in solid lines. In particular, this rule generates a match for each instance of *State* contained in *iFSA*; the matching *State* instance is locally named *iState*. The instances submodels in dotted lines represent objects that are added to the input/output graphs. In this case, for each state in the FSA a corresponding *EnumElement* is instantiated and put inside of the *STATES* enumeration. GREAT provides a useful feature called *crosslinking* that allows temporary marking of vertices in the input/output graphs. The dotted line from *iState* to *iNewElem* creates a temporary edge between the matched state and the newly created enumeration element. This crosslink allows the transformation engine to “remember” each state/element pair. Once the new enumeration element is created, it has a default name. This name will be used for code generation, so it should be set to something more appropriate. GREAT provides *attribute mappings* for modifying the values of instance attributes. Figure 14 contains an attribute mapping called *SetNames*, which sets the name of each enumeration element to *STATE\_sname*, where *sname* is the name of the corresponding *State* instance. This simple rule performs a number of actions that would otherwise have to be coded. The declarative backbone of the graph transformation approach allows compact rules to accomplish many tasks.

Figure 15 shows the rule that creates the basic structures of the C code. A *MainProgram* is created, which contains a *Declarations* section and a *SimLoop* section. The essential variables and enumerations are declared, and the outer *Switch* and *SwitchVar* are created. Recall that the simulation loop must know which variable is the *currentEvent* and which is the *eventStream*. The rule creates an instance of *CurrentEventVar*, a reference

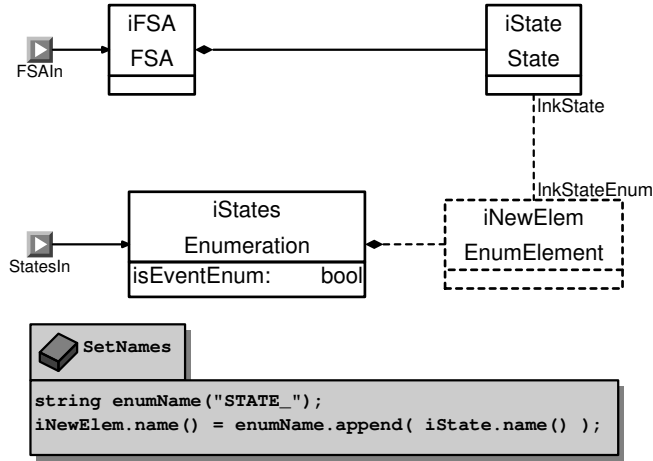


Figure 14. Graph rewriting rule that creates the STATE enumeration elements.

to a variable, and creates a *refers* association from the *CurrentEventVar* instance to the *currentEvent* variable in the declarations section. This way, the simulation loop has a reference to the correct variable. A similar mechanism specifies the appropriate *eventStream* variable to the simulation loop. The *SetTypes* attribute mapping sets the typenames of the C variables. This rule can be executed if a single instance of *FSA* is found in the input graph. Since there is exactly one instance of *FSA*, the rule fires exactly once. Finally, notice that this rule contains “output” ports. These ports pass matched/created vertices out of the rule so they can be used as context vertices for other rules. The *iFSA* instance, *States* and *Events* enumerations are passed out. These vertices will be used as context vertices for the rule in Figure 14.

Passing context vertices between rules provides a natural way to sequence rules together. Figure 16 shows how rules can be explicitly sequenced in a dataflow-like fashion. The oblong labeled *BuildMainObjects* encapsulates the rule of Figure 15. It exposes three outputs ports that pass out the *FSA* instance, and the enumerations. The oblong labeled *BuildStateEnum* encapsulates the rule in Figure 14; it is passed the *FSA* instance and the *STATES* enumeration provided by the *BuildMainObjects* rule. The *BuildEventEnum* rule is almost identical to *BuildStateEnum*, but fills in the *EVENTS* enumeration and requires the *EVENTS* enumeration for context. Rules cannot execute until they have their context, and rules without data dependencies can be executed in parallel or sequenced in an arbitrary order. For example, one can imagine that *BuildStateEnum* and *BuildEventEnum* are applied concurrently. There is one important caveat. Unlike true dataflow systems, rules do have non-local effects because they all operate on the same global graphs. Therefore, the order of execution may affect the outcome, even if rules do not have explicit data dependencies. Finally, GReAT allows rules to be hierarchically grouped into *blocks*. Thus, we can group these rules into

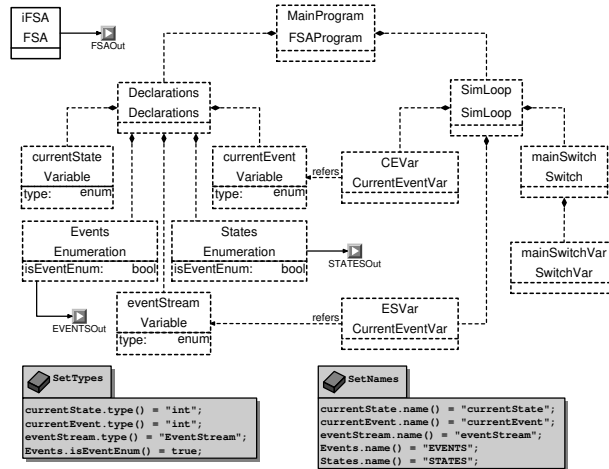


Figure 15. Rule creates the declarations, loop, and main switch objects.

one large block called *BuildDeclarations*. Blocks also have interfaces for passing context vertices. By default, all the rules inside the block execute before any vertices are passed out of the block. In the interest of space, the remaining transformation rules are included in Appendix A.

For the sake of completeness, we will briefly describe how C code is generated from an AST model. After an FSA model is transformed into an AST model, an API-based traverser walks the AST model and emits C code to a file. The AST model is already a tree structure (with the exception of references), so a simple depth-first walk of the model suffices to generate code. The fragment below shows part of the code generation procedure written with the *BON* API in C++.

```

bool Component::WriteCode (BON::FCO n, NODETYPES nt, NODETYPES pt) {
    switch (nt) {
        ...
    case AST_CASE:
        if (!ProcessCase(n,pt)) return false;
        break;
    case AST_ENUM:
        writeFile << "enum " << n->getName() << " { ";
        if (!ProcessElements(n)) return false;
        writeFile << " }; \n"; break;
    case AST_ELEM:
        writeFile << n->getName(); break;
        ...
    }
    return true;
}

```

The *WriteCode* method is passed a node from the AST model ( $n$ ) and an enumeration value that describes the type of the node ( $nt$ ). The type of the parent of  $n$  is also provided ( $pt$ ). The method contains one **switch** statement with a **case** for each node type. For example, if  $n$  is an enumeration node (identified by the constant *AST\_ENUM*), then the method outputs the C fragment: `enum NAME {`, and recursively calls

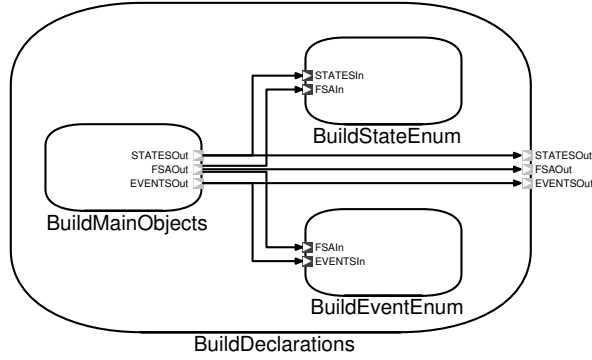


Figure 16. Sequencing and encapsulation of the graph rewriting rules as a block.

*WriteCode* on the enumeration elements via the *ProcessElements* method.

```

bool Component::ProcessElements (BON::FCO enumeration) {
    std::set<BON::FCO> elems = BON::Model(enumeration)->getChildFCOs("EnumElement");
    for (iterator fit = elems.begin(); fit != elems.end(); ++fit) {
        if (!WriteCode(*fit,AST_ELEM,AST_ENUM)) return false;
        ...
        ... writeFile << ", ";
    } writeFile << enumeration->getName() << "_Count"; return true;
}

```

This method collects all the enumeration elements in the set *elems* and then iteratively calls *WriteCode* on each element. *WriteCode* simply prints the name of the enumeration element and returns. Each element is separated by a comma and a final “count” element is appended to the enumeration. Thus, the actual C code produced looks like: `enum NAME { V1 = 0, V2, ..., Vn, NAME_Count };`. The remaining AST node types are handled in a similar fashion. Appendix A shows example output of the code generator.

Even though this example code generator is quite simple, it is not entirely trivial. Needless to say, without graph transformations this code generator would have been even more complicated. The transformation approach allows us to minimize the amount of code necessary to implement the semantic analysis phase. Additionally, formal verification techniques may make it possible to verify that the transformation is correctly implemented.

The transformation approach also supports reuse of model transformations through function composition. For example, given modeling languages  $\mathcal{M}_A, \mathcal{M}_B, \mathcal{M}_C$  and transformations  $\tau_{A,B} : \mathcal{M}_A \rightarrow \mathcal{M}_B$ ,  $\tau_{B,C} : \mathcal{M}_B \rightarrow \mathcal{M}_C$ , we may construct a new map  $\tau_{A,C} : \mathcal{M}_A \rightarrow \mathcal{M}_C$  defined by  $\tau_{A,C} = \tau_{B,C} \circ \tau_{A,B}$ . The new map  $\tau_{A,C}$  transforms models from language  $\mathcal{M}_A$  to  $\mathcal{M}_C$  via  $\mathcal{M}_B$ . Function composition allows reuse of the maps  $\tau_{A,B}$  and  $\tau_{B,C}$ . At first glance, it may appear that this style of reuse is purely academic. However, model-based design employs this style of reuse extensively, precisely because it simultaneously supports many different languages and abstraction layers. We show a concrete example of this in the discussion.

## Discussion and Conclusion

The genesis of model-based design was the heterogeneity and resource constraints of embedded and distributed systems. As a result, the literature surrounding model-based design may seem foreign to traditional software engineers. However, as we have shown, most concepts in model-based design are systematic extensions of well-established design techniques. Additionally, these extensions will have broader impact as the “traditional” software realm evolves to become more like embedded and heterogeneous systems. This is already happening on two fronts. First, traditional software applications (e.g. word processing, spreadsheets) are being recast into service-oriented and data-centric architectures where it is natural to impose complex non-functional requirements related to time and network usage. (This was discussed in the introduction with DDS.) Second, next-generation processor architectures, like the CELL [56], are condensed heterogeneous systems. Exploiting the power of these processors will require dramatic changes to traditional software design techniques so that the inherent heterogeneity is utilized. These issues have been extensively explored in [57].

As we have shown, the DSML perspective provides a convenient metaphor for extending existing techniques in software engineering. However, there are other metaphors that also have significant utility; *Platform-based design*[58] is one such alternative. Platform-based design describes the application context with a set of *components*. Components are simultaneously structural and behavioral: They have interfaces and are connected together through these interfaces. A platform includes structural rules restricting how components can be connected. Components also encapsulate behaviors, and interact with each other through their interconnections. This viewpoint deemphasizes the separation between the structural and behavioral semantics. What we call models are referred to as *platform instances*, and are instantiations of components and component interconnections. Platform-based design differs from other views by emphasizing semi-automatic/automatic system synthesis from high-level specifications [59]. The *Metropolis*[60] project aims to develop tools for automatic synthesis between generic platforms.

We now conclude by viewing a classic design problem through the eyes of model-based design. Figure 17 shows the classic sketch of simple communication protocol between a sender and a receiver. As usual, the flowchart is intended to be a clear high-level specification of the following communication protocol: The sender broadcasts an *Ack* while the receiver waits for the *Ack*. The waiting receiver times out every 10ms, though it returns to the waiting state after this timeout. If the receiver observes the *Ack*, then it broadcasts a *Nack*. Similarly, after the sender announces an *Ack* it waits 10ms for a *Nack*. If *Nack* does not arrive in this interval, then the sender broadcasts the *Ack* again. At the very least, we would like to know if the sender and receiver can reach the *Done* boxes in the flowchart, assuming they both start at the same time. Traditionally, there are two approaches to this design problem. The first approach is to immediately code an approximation of this flowchart, and then test it. We need not mention that this first technique is doomed to

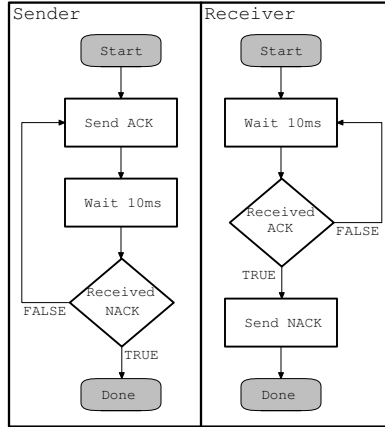


Figure 17. Simple example of a communication protocol.

failure. The second approach is to be more precise about the meaning of the diagram. For example, we might decide that hierarchical concurrent finite state machines (HFSTs) provide a more precise characterization of the concurrency in the model. This might lead us to diagram in Figure 18.

This HFST representation contains two implicit “clock” automata that emit time tick events. This representation discretizes time, because there are no time events that occur within a hypothetical interval. The fact that we used two clocks instead of one clock suggests that we assume the clocks are not synchronized. However, in order to make this precise we must define the synchronization mechanism between the automata. There are many possible choices, none of which are inherently right or wrong. However, these choices drastically affect analysis of the high-level models. For example, if we choose the synchronous product of finite state transducers (FSTs) as the composition mechanism, then the HFST is equivalent to the flattened transducer in Figure 19.

This transducer has the property that from the state start  $(S_1, A, T_1, B)$  the acceptor state  $(S_3, A, T_3, B)$  is always reachable from all future states. However, had we chosen a different composition mechanism, such as an asynchronous product (shuffle product), then the analysis would have yielded a different result. In the asynchronous case, it is always possible for the sender and receiver to miss each others messages, so there is path from the start state for which the acceptor state is not reachable. Which prediction accurately reflects reality depends on how the final system will be implemented. However, this presents a paradox: Choosing the right high-level specification depends on knowledge of the implementation, but the high-level specification is supposed to precede implementation.

Software engineers argue that design choices must be carefully contemplated, usually within the framework of a design methodology. Traditionally, a *design choice* refers to a decision about the architecture of

a point design. However, the above example shows that there are other design choices that affect the entire class of possible designs. These choices are equally important, because they influence whether analysis and verification at the pre-implementation phases reflect the properties of the future implementation. Using the enhancements of model-based design we can capture “meta-level” design choices, as shown in Figure 20. Each oval represents the structural semantics of a DSML. The top oval is the flowchart language, and the row beneath lists various hierarchical automata structures. On the left there is a hierarchical FSM language, in the center is a hierarchical timed-automata language (HTA), and on the right is a hierarchical hybrid automata language (HHA). The meta-level design choices define how a flowchart model is projected onto these other languages. We can explicitly characterize these choices by writing modeling transformations between the languages. For example, the transformation that assigns a unique automaton to each clock of the flowchart is shown by the arrow labeled *locally asynchronous clocks*. Similarly, the transformation that creates one unique global clock is shown by the arrow labeled *external global time events*. If we prefer to consider time as dense and globally synchronized, then we would consider possible projections onto the HTA language. If time is dense but clocks tick at different rates, then we would consider projections onto the HHA language. For each hierarchical language, the semantics of concurrency is defined in terms of product operators that transform a hierarchical model into a flat automaton structure. Figure 20 shows the flattened automata languages below their hierarchical counterparts, and shows model transformations from the hierarchical version to the flat version. These transformations capture the ways that concurrent automata interact. As the diagram shows, there is not one unique way to view concurrency, but a spectrum of possibilities. Finally, code and analysis models can be easily generated from the simple flattened languages, and then tested or verified using tools based on the corresponding formalism. (See Section II.3.)

The framework of Figure 20 permits the systematic exploration of meta-level design choices. The domain-specific language is the key ingredient that allows this framework to emerge. Given a flowchart model  $m$ , we

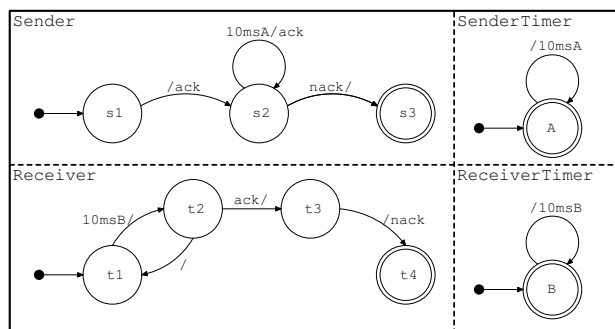


Figure 18. One possible representation of the protocol using HFSMs.

can explore the impact of meta-level choices by choosing a sequence of transformations  $\tau_1, \tau_2, \dots, \tau_n$  where the domain of  $\tau_1$  is the flowchart language and the codomain of  $\tau_n$  is the syntax of simulation instructions for a simulator. Then, by composition,  $(\tau_n \circ \dots \circ \tau_2 \circ \tau_1)m$  yields a simulation artifact. The sequence of maps captures the meta-level choices, and the simulation artifact captures the final outcome of these choices. Notice that the number of *semantic variants* of a language is equal to the number of unique paths from the language to a leaf. In general, this is combinatorial in the number vertices in the diagram. This fact alone shows why it can be quite difficult for a designer to make the right meta-level choices. At the same time, it means that a particular  $\tau_i$  may appear in an exponential number of semantic variants, yield significant reuse of the DSMLs. In conclusion, the model-based approach uses the DSML to capture the properties of a particular application context at a particular level of abstraction. Model transformations link DSMLs, providing a reusable framework for exploring meta-level design choices.



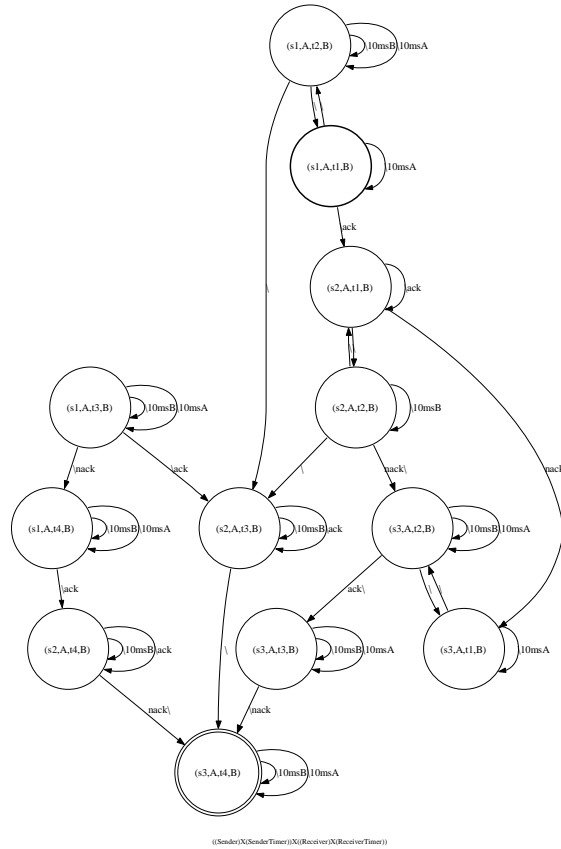


Figure 19. Interpretation of the specification assuming synchronous product.

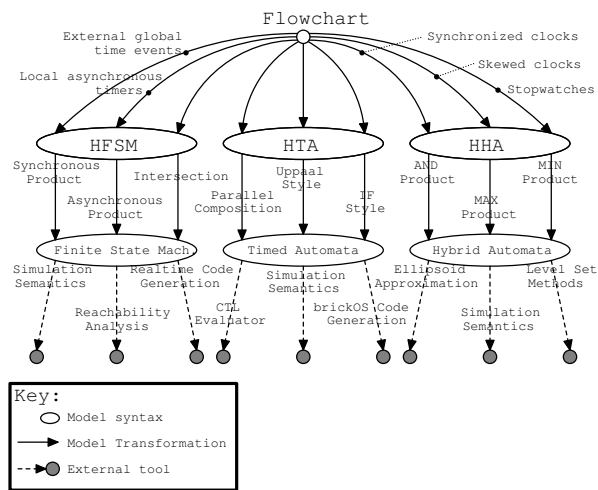


Figure 20. Model-based view of the meta-level design choices.

## CHAPTER III

### FORMALIZING STRUCTURAL SEMANTICS

#### Introduction

Model-based design has emerged as a powerful approach to modern system design. Two core principles lie at the heart of its success: First, accidental complexity can be reduced using domain-specific languages and tools. Second, essential complexity can be incrementally controlled through domain-specific abstractions [61][62][11]. Accidental complexity arises from the semantic mismatch between domain-specific concepts and implementation languages. For example, embedded systems often employ simple hardware-like models of concurrency wherein data is emitted and consumed by software components. Traditional software techniques require these concurrency concepts to be rewritten in terms of threads with mutexes and semaphores, even though thread-based concurrency is a drastically different concurrency model. This mismatch can be a major source of accidental complexity [63]. The model-based approach is to rapidly prototype domain-specific modeling languages (DSMLs) that support the precise concepts (e.g. concurrency concepts) needed for the problem domain. This reduces accidental complexity.

Essential complexity increases as a system approaches implementation. During implementation, details absent from the higher-level system specifications (or *abstract models*) are introduced. For example, an abstract model may utilize an ideal unbounded FIFO. Any implementation augments the ideal FIFO with additional details bounding its size. Implementation details of this sort introduce essential complexity that must be handled with great care. Model-based design employs a “stack” of DSMLs (also called a *platform stack*) to handle this [64]. The language at the top of the stack is an abstract, ideal view of the domain semantics. The lower languages in the stack augment the semantics, adding necessary details. This approach allows the engineer to define an abstract view of the system, and then gradually increase the complexity by migrating the design down the platform stack. At each step, the more abstract model can be compared with the more detailed model to ensure that the same functional and non-functional requirements are satisfied (also called *refinement verification*) [65].

The *domain-specific modeling language* is the key ingredient for managing design complexities. A most remarkable feature of a DSML is the extent to which model structure alone can be used to check model properties and link DSMLs in a platform stack. For example, cycles or knots in a *dataflow graph* (a structural property) may indicate a deadlock (a behavioral property) in the implementation [66]. DSMLs based on concurrent automata can use transformations on structure (e.g. *synchronous product*, *shuffle product*, etc...) to capture interaction and communication. For example, automata-like models can be transported to lower levels in a platform stack by replacing many concurrent automata with a large flattened automaton. All

of this can be done solely using model structure, i.e. without invoking the behavioral semantics of model elements. In fact, the quest for a fundamental theory of model transformations has driven the field in many interesting directions[67].

Even when structure is not enough to reason about models, it still plays an important role: It is the supporting skeleton on which other styles of semantics are built. For example, Plotkin-style structural operational semantics (SOS) make explicit reference to structure [68][69]. Other approaches use model structure as input to a mathematical execution machine [70]. Even some denotational approaches define denotations on top of the model structure [71]. Model-based tools support the rapid definition of rich structures through *metamodeling*, and model transformation tools, such as GReAT[72] and Viatra[73], utilize the metamodel descriptions of structure.

Despite the fundamental role of structure in the model-based approach, it remains largely unformalized (along with other a number of other concepts [74][75]). To address this and other issues, we present a mathematical formulation of structure, giving a precise tool-independent definition that is amendable to formal analysis. This work also provides a formal understanding of model transformations and metamodeling, yielding a complete picture of the structural basis of model-based design. In this sense, we broaden the term *structural semantics* to include the semantics of model transformation and metamodeling. Section III.2 presents our formalization, leaving some key parameters free for later specialization. We begin by formalizing the set of all structurally well-formed models that belong to a DSML. We use these sets as the basis for formalizing model transformations. Finally, we define metamodeling as a transformation from models to sets of well-formed models. Section III.3 illustrates one specialization that leads to decidable algorithms for analyzing domains and model transformations. We also show how our formalization can be immediately applied to existing metaprogramable tools such as GME\GReAT. Section III.4 concludes with a short discussion and plans for future work. This work is a continuation of work originally presented at the ACM Conference on Embedded Software in 2006 [39].

### **The Formal Semantics of Domains and Domain Construction**

The structural semantics characterizes the set of all well-formed mathematical structures of a particular application context. We call this set of structures a *domain*, and a common domain is shown in the foreground of Figure 21.a. The basic building blocks of this domain are the hardware components (ASICs and cards) that can be plugged into the circuit board. Each block on the circuit board encodes a restriction on the actual ASIC that can be placed in a particular location. For example, the block labeled CPU encodes the constraint that a CPU, not a RAM module, must be placed at that point. Constraints can be more complicated than simple placement rules. For example, Figure 21.a also requires that if a CPU of type A is placed on the board, then a RAM module of type B cannot be placed on the board. A *model* is a description

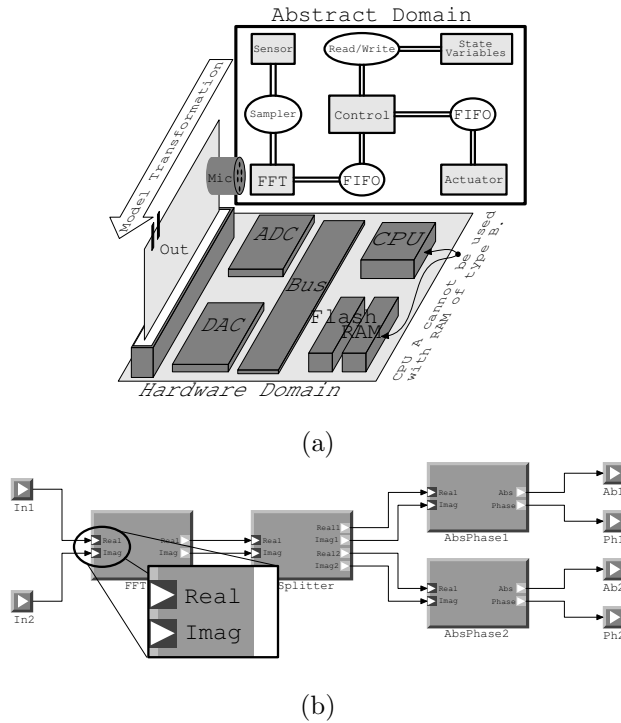


Figure 21. (a) Example of two domains. (b) A model from a digital signal processing domain.

that has no remaining degrees of freedom, e.g., every place on the circuit board has some hardware assigned to it. A *well-formed model* is a model that satisfies all the constraints imposed on its construction. The set of all well-formed models contains all the meaningful structures of a domain. It is important to note that the set of well-formed models can be defined without giving a behavioral meaning to the constructs that participate in the model. For example, we do not need to give any details about what CPUs, RAMs, and buses do in order to check well-formedness of the circuit board models.

This example shows that a domain is a set models, models are built from some structural primitives, and good models can be distinguished from bad models. Formally, a domain is characterized by the following information:

1. a set  $\Upsilon$  of concepts, components, or primitives from which models are built,
2. a set  $R_{\Upsilon}$  of all possible model realizations,
3. a set of constraints  $C$  over  $R_{\Upsilon}$ .

The model realizations in  $R_{\Upsilon}$  are all the ways that models can be built from the available primitives. A domain's set of *well-formed models* is the set of all model realizations that satisfy the constraints. We write this set as

$$D(\Upsilon, C) = \{r \in R_{\Upsilon} \mid r \models C\}. \quad (\text{III.1})$$

The notation  $r \models C$  can be read as “ $r$  satisfies the constraints  $C$ ”. The first essential issue in formalizing a domain is to decide how the set of all model realizations  $R_{\Upsilon}$  relates to the set of concepts  $\Upsilon$ . The relationship between  $\Upsilon$  and  $R_{\Upsilon}$  must capture the types of structures commonly encountered in embedded systems design. For example, the background pane of Figure 21.a shows a diagram outlining the connectivity of components in a data processing application. This typical domain bares similarities to labeled directed graphs (sets of vertices, binary relations), as well as type systems (vertices can be rectangles or ellipses). Additionally, structural concepts like hierarchy and attributes are commonly found in embedded system domains. The primitive set  $\Upsilon$  must be capable of encoding all of these concepts, and the set of all model realizations  $R_{\Upsilon}$  must include all the ways that these concepts can be used together.

Though these general observations direct us towards candidate formalisms, they do not narrow down the candidates to reasonably few possibilities. We propose that a candidate formalism should also satisfy the following “non-functional” requirements:

1. Relevance - The formalism should capture, at least, the relevant structural concepts supported by today’s DSML tools.
2. Scalability - It should be possible to “scale” the formalism to include new structural concepts introduced as DSML methodologies evolve.
3. Expressiveness - The formalism should express a wide range of constraints on structure.

Let us provide some justification for these points. Research on DSML methodologies has been ongoing for over a decade and has yielded tools and standards such as: the Generic Modeling Environment (GME)[43], the Eclipse Modeling Framework (EMF)[76], the Graph Rewriting and Transformation (GReAT) language[72], the Visual Automated Model Transformation (VIATRA) tools[73], the Meta Object Facility (MOF) specification[77], and the Queries/Views/Transformations (QVT) specification[78] (to name a few). A formalism that does not support most of the structural concepts provided by these tools will be moot for most developers. Instead of discarding mature tools due to their informality, we should believe that a formal underpinning can be discovered.

DSML tools and their related specifications continue to evolve at a rapid pace. For example, the Unified Modeling Language (UML) standard has been under development since 1994 and is continually evolving via the Object Management Group (OMG). MOF, which is now its second iteration, spun off from UML and the two standards continue to impact each other. Besides specifications, new tools are also being produced. For example, Microsoft is producing a domain-specific language toolkit for Visual Studio[79]. A number of highly visible academic tools (e.g. Ptolemy II[3] and Metropolis[62]) are expanding platform-based design, directly impacting DSML tools. Thus, a formalism must scale up with the growth of the field, otherwise it risks becoming irrelevant.

Our notions of relevance and scalability are with respect to model structure: Do model elements have

hierarchy, interfaces, or multiple aspects? The use of these structural mechanisms can be restricted by the constraints  $C$ . By *expressiveness*, we mean the capabilities of the underlying constraint system to restrict model structure. The expressiveness of the candidate formalism directly impacts the possible analyses that can be performed on the DSML. For example, OCL (Object Constraint Language)[42] is a first-order calculus used for constraining model structure. As such, OCL has the potential to make model well-formedness semi-decidable. Thus, we may have to adjust the expressiveness of the underlying formalism in order to achieve analyzability. This is a common tactic used in embedded systems design.

### Existing Candidate Formalisms

**Regular and Context-free Languages** Regular and context-free languages have a long history in traditional language design, and are worth considering. If we were to apply this framework, we would choose the modeling primitives to be an (infinite) alphabet  $\Sigma$ , i.e.  $\Upsilon = \Sigma$ . The set of all model realizations would be the set of all finite strings  $R_\Upsilon = \Sigma^*$ . Finally, the constraint system  $C$  would be either a finite state acceptor (regular languages) or a pushdown automata (context-free languages). This framework has two main drawbacks: First, strings are not an optimal encoding for relational structures. Second, the constraint systems are not sufficiently expressive for capturing typical constraints.

Consider the dataflow system of Figure 21.b. Structurally, it resembles a labeled directed graph, where the vertex labels distinguish dataflow components, and the edges denote the flow of data between components. In fact, many domains are relational in the sense that some elements can “stand on their own” (e.g. vertices), while other elements only exist between elements (e.g. edges, or arbitrary  $n$ -ary relations). Thus, as a litmus test, consider representing arbitrary directed graphs (without orphans) as strings of the language  $\mathcal{L}_{DG}$ . Let  $s \in \mathcal{L}_{DG}(\Sigma)$  if  $s \in \Sigma^*$  is a string of even length. An orphan-less directed graph  $G$  with the vertex set  $V \subseteq \Sigma$  is mapped to a string  $s(G)$  as follows: Let  $s(G) = v_{e_1} u_{e_1} v_{e_2} u_{e_2} \dots v_{e_n} u_{e_n}$  for any ordering of the edge relation  $E = \{(v_i, u_i)\} \subseteq V^2$ . Similarly, a string  $s$  can be converted to a graph  $G(s)$  where  $V = \bigcup_{i=1}^{|s|} \{s_i\}$  and  $E = \bigcup_{i=1}^{\frac{|s|}{2}} \{(s_{2i-1}, s_{2i})\}$ . Notice that  $\mathcal{L}_{DG}$  is the “simplest” language for capturing orphan-less digraphs, in the sense that there are no syntactic adornments and no unnecessary restrictions are placed on  $\mathcal{L}_{DG}$ . It is easy to see that the strings of all equally simple representations are in bijective correspondence. Thus,  $\mathcal{L}_{DG}$  is worth studying as an exemplar of the formal language approach. One immediate observation is that  $s(G)$  is not unique; in fact, there are a combinatorial number of possible representations of a digraph  $G$ . This occurs because we force an unnecessary linear ordering on relations that do not need such an ordering. The only way out is to arbitrarily choose some way to order edges, but this is not a particularly attractive solution.

The expressive power of regular and context-free languages is also problematic. As an extreme example, define a well-formed graph to be one that contains no path of length greater than two. By the definition

of  $s(G)$ , well-formed graphs correspond to even strings that do not repeat vertex labels. For example,  $s(G) = v_1v_2v_3v_4v_5v_6$  for the well-formed graph of Figure 22. Given a set of labels  $\Sigma$ , the language of orphan-less 2-paths  $\mathcal{L}_{2p}(\Sigma) \subset \mathcal{L}_{DG}(\Sigma)$  is the set of all permutations of subsets of  $\Sigma$ . The language of *all finite graphs* of orphan-less 2-paths is  $\mathcal{L}_{2p}(\Sigma)$  where  $\Sigma$  is countably infinite. By the Myhill-Nerode theorem[37],  $\mathcal{L}_{2p}$  is not a regular language because it corresponds to an equivalence relation with an infinite number of equivalence classes. Thus, regular languages will have trouble capturing the more complex graph-theoretic constraints encountered in many DSMLs. Similar arguments hold for context-free languages (e.g. recognition of acyclic digraphs). Of course, the limitations of regular and context-free languages are well-known in traditional language design. Traditionally, the more complicated constraints of a programming language are pushed into later phases of the compiler, i.e. into the type system and semantic analysis phase. This solution does not work for our situation, because our goal is to capture all of the structural constraints of a DSML in an explicit and tool-independent fashion.

**Extensions of Graph Structures** Relational structures play an important role in the structural semantics of model-based design. Often there is an underlying graph structure present in DSMLs. Thus, we might directly employ a graph theoretic framework to formalize model structure. Take  $\Upsilon = \Delta$  where  $|\Delta| = |\aleph_0|$  is a countably infinite set of vertex labels. Then, the possible model realizations are given by the set of all finite graphs with vertices from  $\Delta$ :  $\mathcal{G} = \{(V, E) | V \subset \Delta, E \subseteq V^2\}$  and  $V$  is finite. The major drawback of this approach is that it does not easily capture other relational components of DSMLs, of which there are many.

A range of relational concepts can be found in the metamodeling languages currently used to specify DSML structure. Figure 23 shows an example UML-like metamodel that contains many interesting relations. (We will discuss metamodeling in more detail later.) Informally, the boxes represent classes; classes can have attributes, which are typed member fields. In UML notation, the class called *EdgeClass* relates instances of *ClassB*, and also has an attribute. Graph structures called *attributed graphs* have been created to capture these concepts. Notice that edges will have to be distinguished by labels, because two edges between the same vertices may have different values for their attributes. Following the notation of [80], an edge-attributed graph (E-Graph) is a combination of graph nodes  $V_1$ , data nodes  $V_2$ , graph edge labels  $E_1$ , node-to-data edge

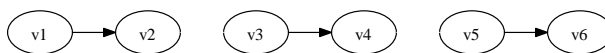


Figure 22. A graph containing only 2-paths

labels  $E_2$ , and edge-to-data labels  $E_3$ . Because edges are distinguishable, a set of functions  $(source_i)_{i \in \{1,2,3\}}$  and  $(target_i)_{i \in \{1,2,3\}}$  designate the source and destinations of the various edge types. Thus, a graph  $G_1$  is a structure:

$$G_1 = \left\langle \begin{array}{c} V_1, V_2, E_1, E_2, E_3, \\ (source_i)_{i \in \{1,2,3\}}, (target_i)_{i \in \{1,2,3\}} \end{array} \right\rangle.$$

This definition does not capture the types of the attributes, necessitating a further extension. Following the notation of [81], an algebra  $\mathbf{D}$  is introduced  $\mathbf{D} = \langle D, (f_j)_{j \in J} \rangle$  where the universe  $D$  contains all possible data values of attributes, and the operations  $f_j$  are useful operations on data values. For example, the universe  $D = \mathbb{Z} \cup \mathbb{B} \cup \Sigma^*$  contains integers, booleans, and strings over an alphabet  $\Sigma$ .  $D$  contains a set  $S$  of distinguished elements called *sorts*, which give names to the relevant subsets of  $D$ . For example, if  $S = \{int, bool, string\}$ , then  $D_{int} = \mathbb{Z}$ ,  $D_{bool} = \mathbb{B}$ , and  $D_{string} = \Sigma^*$ . Given this algebra, the data vertices  $V_2$  are those elements of  $D$  that are members of the relevant sorts, i.e.  $V_2 = \bigcup_{s \in S} D_s$ . A *typed attributed graph* is:

$$G_2 = \left\langle \begin{array}{c} D, S, V_1, E_1, E_2, E_3, \\ (source_i)_{i \in \{1,2,3\}}, (target_i)_{i \in \{1,2,3\}}, \\ (f_j)_{j \in J}, sort \end{array} \right\rangle.$$

where  $sort : S \rightarrow \mathcal{P}(D)$  maps each sort name to the corresponding subset of  $D$ . We dropped  $V_2$  because it is defined by the information  $D, S, sort$ .

These extensions incorporate attributes, but not other common relations. In the interest of space, we shall consider just one more relation, the *containment* relation. Containment allows instances to have internal structure. Referring back to the metamodel of Figure 23, the edge from *ClassA* to *RootClass* (terminating with a diamond symbol) indicates that instances of *RootClass* contain instances of *ClassA*. Containment allows substructures to be encapsulated within model elements; it is a form of information hiding. *Hierarchical graphs* extend graphs so that vertices can contain graphs. Following the notation of [82], let  $\mathcal{H}$  be the set of all hierarchical graphs. A hierarchical graph  $G_H$  contains a set  $F$  of *frame edges* and a mapping  $contents : F \rightarrow \mathcal{H}$  that assigns a hierarchical graph  $contents(f)$  to each frame edge  $f \in F$ . At first

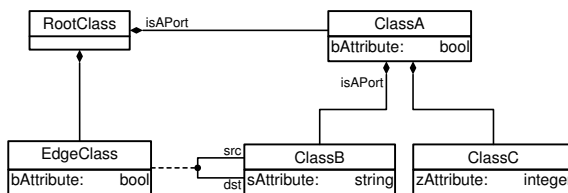


Figure 23. Example metamodel with many relational concepts



glance this definition appears circular, but it can be defined inductively as to avoid any circularity. Combining this extension with typed attributed graphs, let  $\mathcal{H}_{TAG}$  be the set of all hierarchical typed attributed graphs, then:

$$G_3 = \left\langle \begin{array}{c} D, S, V_1, E_1, E_2, E_3, F \\ (source_i)_{i \in \{1,2,3\}}, (target_i)_{i \in \{1,2,3\}}, \\ (f_j)_{j \in J}, sort, contents \end{array} \right\rangle.$$

These extensions still do not handle all structural concepts. For example, *ports*, which are projections of internal structure onto the external structure, are not handled. Figure 21.b shows ports on the dataflow components. Edges that cross hierarchy, like those found in representations of *StateChart* models, are also problematic.

These graph-theoretic extensions can be useful for expanding model transformation techniques that are based on graph rewriting rules. In fact, many of the aforementioned extensions were proposed by researchers in the model/graph transformation community, where the emphasis is on the underlying graph structure. However, our goal is to present a general theory for the structural semantics of model-based design. From this perspective, there is no reason to shape the general theory around just one of the many relational structures found in DSMLs. As we have shown with  $G_1, G_2$ , and  $G_3$ , too much emphasis on graphs contradicts our scalability requirement.

**Instances of ADTs** One final candidate that we examine comes from the long history of abstract datatypes (ADTs) in object oriented programming[83]. Our focus is on the structural aspects of models, so we view classes as very simple ADTs without operations. (If operations were specified in the classes, they would be ignored.) In this case, the modeling primitives are the sets of basic datatypes and descriptions of the ADTs:

$$\Upsilon = \langle S_{basic}, S_{ADT}, (D_s)_{s \in S_{basic}}, nfields, field \rangle.$$

The set  $S_{basic}$  defines the basic datatype names (the set of sorts) from which ADTs can be built, e.g.  $S_{basic} = \{int, bool, string\}$ . The sets  $(D_s)_{s \in S_{basic}}$  (the carriers) contain the values of the basic datatypes, e.g.  $D_{int} = \mathbb{Z}$ ,  $D_{string} = \Sigma^*$ . The set  $S_{ADT}$  contains the names of ADTs, e.g.  $S_{ADT} = \{Thing, Buyer\}$  as shown in Figure 24. The operation  $nfields : S_{ADT} \rightarrow \mathbb{Z}_+$  maps each ADT to the number of fields in the ADT:  $nfields(Thing) \mapsto 3$ ,  $nfields(Buyer) \mapsto 2$ . The mapping  $field : S_{ADT} \times \mathbb{Z}_+ \rightarrow S_{basic}$  describes the basic type of the  $i^{th}$  field of ADT  $t$ . For example  $field(Thing, 1) \mapsto string$ ,  $field(Buyer, 2) \mapsto bool$ . Given this information, we can construct the set  $A_t$  of all possible instances of an ADT  $t$ :

$$A_t = \prod_{i=1}^{nfields(t)} D_{field(t,i)}$$

For example, the set of all *Buyer* instances is the Cartesian product  $A_{Buyer} = D_{string} \times D_{bool}$ . A model realization may contain an arbitrary number of instances of each ADT, so the set of all model realizations

is the Cartesian product of the powersets:

$$R_{\Upsilon} = \prod_{t \in S_{ADT}} \mathcal{P}(A_t)$$

This basic construction may be extended to allow duplicates of the same instance (using multisets with multiplicities), or distinguishable duplicates (using unique IDs). The major drawback of this approach is the difficulty of incorporating any relations. Again, there is no consistent mechanism for incorporating relations of arbitrary arity. However, this approach and the previous approaches all provide some insights that we will utilize in our formalism. In particular, we will convert a metamodel into a signature  $\Upsilon$ , and then construct the *term algebra* generated by an alphabet  $\Sigma$  over  $\Upsilon$ . The generated signature precisely captures the relational structures of a domain, generalizing the binary relations that are emphasized by the graph-theoretic approach. The terms of the term algebra generalize the strings found in the formal language approach. A model realization a set of terms, and the set of all model realizations is the powerset of terms. This incorporates the powerset construction found in the ADT approach.

### Algebraic Approach

Mathematically, we must find a formalization that naturally captures the possible complex relations between model elements. We chose a simple, yet flexible, algebraic approach that can handle the structural concepts typically found in model-based design: Model primitives are represented as *function symbols* and model realizations are subsets of the *term algebra* generated by these function symbols. (This description uses the language of modern algebra, see [84].) We begin with an example to illustrate this approach. Figure 21.b shows a model that belongs to a domain for Digital Signal Processing (DSP) systems. We will work backwards from this single model to the domain of all DSP models. To begin, we must extract the primitive concepts used to build DSP systems. Examining Figure 21.b, we see that this model has inputs and outputs at the far left and right side, as well as a number of DSP primitives (FFT, phase/magnitude extraction, and signal demultiplexing), which can be instantiated multiple times. The zoomed in box shows that the primitives have interfaces, which are sets of uniquely identifiable *ports*. In order to capture these concepts,

Thing	
description:	string
price:	int
refurbished:	bool

Buyer	
fullname:	string
paid:	bool

Figure 24. Some classes as ADTs

we will describe a set of  $n$ -ary function symbols for encoding the modeling concepts. Table 7 lists the basic concepts of the DSP domain written as  $n$ -ary function symbols.

Table 7. Set of modeling concepts for DSP domain.

$$\Upsilon = \left\{ \begin{array}{l} \mathit{insig}(X) : X \text{ is system-wide input signal} \\ \mathit{outsig}(X) : X \text{ is system-wide output signal} \\ \mathit{prim}(X) : X \text{ is a basic DSP operation} \\ \mathit{iport}(X, Y) : X \text{ has an input port called } Y \\ \mathit{oport}(X, Y) : X \text{ has an output port called } Y \\ \mathit{inst}(X, Y) : X \text{ is an instance of the DSP operation } Y \\ \mathit{flow}(X_1, Y_1, X_2, Y_2) : \text{Data goes from } \mathit{oport } Y_1 \text{ on } X_1 \text{ to } \mathit{iport } Y_2 \text{ on } X_2 \end{array} \right.$$

The function symbols clearly encode the important concepts needed to build DSP models. However, in order for the function symbols to encode actual models we need a set of constants that stand for distinguishable model elements. For example, in Figure 21.b there is a DSP block called FFT. We capture this by writing  $\mathit{prim}(\text{FFT})$ , where the name FFT is a constant from some underlying set of constants. Mathematically,  $\mathit{prim}(\text{FFT})$  is called a *ground term* (or just *term*). A model is a set of ground terms where each ground term expresses information about some particular constants, using the function symbols. Table 8 shows a partial encoding of the DSP model as ground terms<sup>1</sup>. Notice that terms can be arbitrarily nested, naturally expressing relations over relations.

Table 8. A partial encoding of Figure 21.b with ground terms.

Primitives	$\mathit{prim}(\text{FFT}), \mathit{prim}(\text{Splitter}), \mathit{prim}(\text{Phase})$
Ports	$\mathit{iport}(\mathit{prim}(\text{FFT}), \text{Real}), \dots, \mathit{oport}(\mathit{prim}(\text{FFT}), \text{Imag})$
Inputs	$\mathit{insig}(\text{In1}), \mathit{insig}(\text{In2})$
Outputs	$\mathit{outsig}(\text{Ab1}), \mathit{outsig}(\text{Ph1}), \dots, \mathit{outsig}(\text{Ph2})$
Instances	$\mathit{inst}(\text{FFT}, \mathit{prim}(\text{FFT})), \dots, \mathit{inst}(\text{AbsPhase1}, \mathit{prim}(\text{Phase}))$
Flows	$\mathit{flow}(\mathit{insig}(\text{In1}), \mathit{insig}(\text{In1}), \mathit{inst}(\text{FFT}, \mathit{prim}(\text{FFT})), \mathit{iport}(\mathit{prim}(\text{FFT}), \text{Real})), \dots$

A single model is a set of ground terms, therefore the set of all model realizations  $R_\Upsilon$  contains all possible sets of ground terms that can be formed from  $\Upsilon$  and a set of constants from some (infinite) alphabet  $\Sigma$ . We will make this more precise using the language of algebra. Assume an underlying vocabulary  $\mathcal{V}$  that provides function names (symbols), then  $\Upsilon$  is a *signature*; a partial function from function names to the non-negative integers,  $\Upsilon : \mathcal{V} \rightarrow \mathbb{Z}_+$ . The set  $\mathbf{dom}\Upsilon$  is the set of function symbols used to encode models, and the integer

<sup>1</sup>In order to simplify the encoding, we assume that every input/output is also a port with the same name as the input/output.

assigned to each symbol is the corresponding arity of the function. An  $\Upsilon$ -algebra  $\mathbf{A} = \langle A, \Upsilon \rangle$  is a structure where  $A$  is a set called the *universe* of the algebra, and  $\Upsilon$  is a signature. Each function symbol  $f$  in the signature denotes a mapping  $f : A^{\Upsilon(f)} \rightarrow A$  from an  $\Upsilon(f)$ -tuple of the universe back to the universe.

Given a signature  $\Upsilon$  and an alphabet  $\Sigma$ , there exists a special algebra  $T_{\Upsilon}(\Sigma)$  called the *term algebra generated by  $\Sigma$*  with the following properties:

**Definition 1.** Let  $\Upsilon$  be a signature and  $\Sigma$  be an alphabet, then the *term algebra*  $T_{\Upsilon}(\Sigma)$  is defined by the following:

1.  $\langle \Sigma \rangle = T_{\Upsilon}(\Sigma)$
2.  $\mathbf{im}f \cap \Sigma = \emptyset$
3.  $f(t_1, \dots, t_{\Upsilon(f)}) = g(t'_1, \dots, t'_{\Upsilon(g)})$  iff  $f = g, t_k = t'_k$  for  $1 \leq k \leq \Upsilon(f)$ .

The notation  $\langle \Sigma \rangle = T_{\Upsilon}(\Sigma)$  can be read “ $\Sigma$  generates  $T_{\Upsilon}(\Sigma)$ ”. A subset  $X$  of the universe  $A$  *generates* an algebra  $\mathbf{A}$  if every member of the universe can be reached by repeated applications of functions to the elements of  $X$ . The term algebra is special, because two functions return the same value iff the functions are the same and their arguments are the same. This means that every element of the universe can be uniquely identified by writing the unique term that produces the element: e.g.  $inst(\mathbf{AbsPhase1}, prim(\mathbf{Phase}))$  is some unique element. This property means that it is unnecessary to explicitly define the universe of a term algebra; instead we can just write terms. The set of all model realizations is the powerset of terms.

**Definition 2.** The set of model realizations  $R_{\Upsilon}$  for a signature  $\Upsilon$  is given by  $R_{\Upsilon} = \mathcal{P} \left( T_{\Upsilon}(\Sigma) \right)$ .

The set of model realizations  $R_{\Upsilon}$  contains many model realizations. In fact, given one function symbol  $f$  of arity one and one constant  $\alpha$ ,  $T_f(\{\alpha\})$  contains a countably infinite number of models. (Consider that all the natural numbers can be generated by 0 and  $succ(\cdot)$ , the successor operation.) As a result,  $R_{\Upsilon}$  may contain many models that combine the functions symbols in ways contrary to our intensions. Consequently, we need a mechanism for deciding whether sets of terms are well-formed or malformed. Preferably, the formalization of well-formedness should support formal analysis. One natural candidate is the representation of formal logics via *consequence operators*. A consequence operator  $\vdash$ , in the sense of Tarski, is a mapping from sets of terms to sets of terms, i.e.  $\vdash : \mathcal{P}(T_{\Upsilon}(\Sigma)) \rightarrow \mathcal{P}(T_{\Upsilon}(\Sigma))$ . Given a set  $T$  of ground terms and a set of *axioms*  $\Theta$ , then  $T \vdash_{\Theta} T'$  yields all the ground terms  $T'$  that can be derived from  $T$  by repeated applications of the axioms  $\Theta$  and the inference rules of the underlying logic. We will also write  $T \vdash t$ , where  $t$  is a single term, indicating that  $t \in T'$ . Classical consequence operators are *extensive*, *isotone*, and *idempotent*, though we will make use of *nonmonotonic* consequence operators that do not have the isotone property.

We propose a simple decision procedure, using consequence operators, to separate well-formed models from malformed models. Add a new function symbol  $wellform(\cdot)$  to the signature  $\Upsilon$ , then a model  $M$  is well-formed if  $\exists x \in T_{\Upsilon}(\Sigma), M \vdash_{\Theta} wellform(x)$ .  $M$  is well-formed if there is some ground term of the form

$wellform(x)$  that can be derived from  $M$ . The axioms  $\Theta$  capture the ways that  $wellform(\cdot)$  terms can be derived, and vary from domain to domain. Sometimes it is easier to characterize the malformed models, in which case we augment  $\Upsilon$  with a  $malform(\cdot)$  symbol. A model is well-formed if  $\forall x \in T_{\Upsilon}(\Sigma), M \not\vdash_{\Theta} malform(x)$ , i.e. if it is impossible to prove any  $malform(\cdot)$  term from  $M$ . In this case, the axioms  $\Theta$  capture the ways that models can be malformed. Notice that consequence operator allows us to adjust the expressiveness of the entire constraint system, and the axioms  $\Theta$  capture the constraints of individual domains.

A *domain* has the following parts: An alphabet  $\Sigma$ , a signature  $\Upsilon$ , called the *domain signature*, a signature  $\Upsilon_C$ , called the *constraint signature*, and a set of constraints  $C$  for deriving model well-formedness.  $\Upsilon_C$ , an extension of  $\Upsilon$ , contains all the necessary symbols for deriving well-formedness. By “extension”, we mean that  $\Upsilon_C$  contains at least the symbols of  $\Upsilon$ , and assigns the same arity to those symbols:  $\mathbf{dom} \Upsilon \subset \mathbf{dom} \Upsilon_C$  and  $\Upsilon = \Upsilon_C|_{\mathbf{dom} \Upsilon}$ . Domains are subdivided into two disjoint classes: *positive* and *negative*. Positive domains must include the unary symbol  $wellform$  in  $\Upsilon_C$ ; a model is well-formed if any ground term of the form  $wellform(\cdot)$  can be deduced. Negative domains must include the unary function symbol  $malform$  in  $\Upsilon_C$ ; a model is well-formed if no ground term of the form  $malform(\cdot)$  can be deduced.

**Definition 3.** A domain  $D$  is a structure of the form  $\langle \Upsilon, \Upsilon_C, \Sigma, C \rangle$ .  $\Upsilon$  and  $\Upsilon_C$  are signatures with functions symbols from  $\mathcal{V}$ , agreeing at their overlap.  $\Sigma$  is an alphabet, and  $C$  is a set of axioms over the terms  $T_{\Upsilon_C}(\Sigma)$ . A domain is *positive* if  $\Upsilon_C(wellform) \mapsto 1$ ; it is *negative* if  $\Upsilon_C(malform) \mapsto 1$ .

This scheme allows domains to be analyzed. For example, Figure 25 illustrates how two domains can be compared with each other. Consider the case of two positive domains  $D_1, D_2$  with identical  $\Upsilon$  signatures

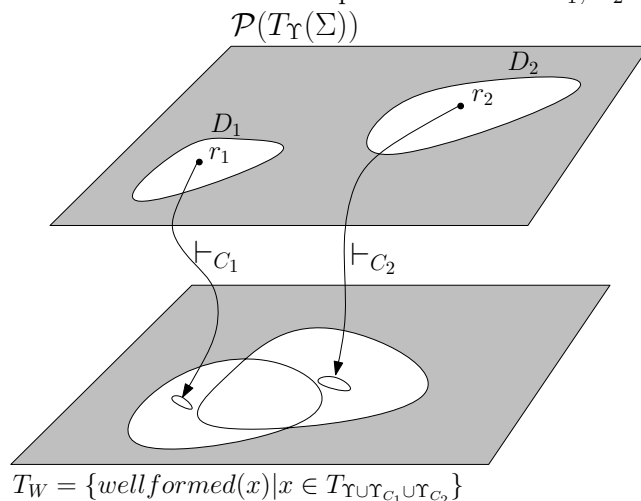


Figure 25. Two positive domains of same signature are related by the  $wellform$  symbol

and agreeing<sup>2</sup> constraint signatures  $\Upsilon_C$ . The upper plane shows the well-formed models of each domain

<sup>2</sup>The signatures assign the same arity to symbols of the same name.

as a subset of the powerset of terms. Each well-formed model in  $D_i$  infers one or more terms of the form  $wellform(x)$  under the consequence relation  $\vdash_{C_i}$ , as shown on the lower plane. Let  $T_W$  be all the terms of the form  $wellform(x)$ . Two domains can be compared by working backwards from  $T_W$  in the following way: Check if there exists a model realization  $r \in R_{\Upsilon}$  such that  $\exists w \in T_W, r \vdash_{C_1} w$ , but  $\neg \exists u \in T_W, r \vdash_{C_2} u$ . If there exists such an  $r$ , then the domains cannot be equal because  $r$  is well-formed in the first domain and not in the second. If there does not exist such an  $r$ , then  $D_1 \subseteq D_2$ . In this case, check the opposite direction for an  $r'$  such that  $\neg \exists w \in T_W, r' \vdash_{C_1} w$  and  $\exists u \in T_W, r' \vdash_{C_2} u$ . Again, if no such  $r'$  can be found, then  $D(\Upsilon_1, C_1) = D(\Upsilon_2, C_2)$ . A similar comparison can be made for negative domains. If an appropriate style of logic is selected (i.e. an appropriate consequence relation) then domain equivalence is decidable.

**Proposition 4.** *Two positive domains  $D_1, D_2$  are equivalent (written  $D_1 \cong D_2$ ) iff  $R_{\Upsilon_1} \cap R_{\Upsilon_2} \neq \emptyset$  and there is no model  $r \in R_{\Upsilon_1} \cap R_{\Upsilon_2}$  such that*

$$\exists w \in T_W, r \vdash_{C_i} w \text{ and } \forall u \in T_W, r \not\vdash_{C_j} u \text{ for } i \neq j \in \{1, 2\}.$$

This is not a particularly deep proposition, but it shows how domain equivalence can be translated into the language of consequence operators. A number of other important properties can be written in the language of logic: Is a domain non-empty, i.e. does it contain at least one well-formed model? Does there exist a well-formed embedding of a model realization  $r$ , i.e. can a malformed model be converted into a well-formed model? In the second half of the chapter we specialize our framework using a consequence operator for which these problems are decidable.

### Transformational Semantics and Model Transformations

The model in Figure 21.b describes a computational apparatus that operates on a continuous stream of data. Though the meaning of this diagram may appear obvious because of the way the model is drawn, we cannot rely on this obviousness as a definition of how a model defines a system. Instead, we must explain precisely how DSP models define computational systems. In practice, this is done by specifying a code generator that produces an implementation from a model. A code generator might map models from the DSP domain to models of a C++ domain. Thus, meaning is affixed to a domain by specifying a mapping from models in one domain to models in another domain. In the model-based community semantics specified with transformations are called *transformational semantics*. We shall choose a more context-neutral term and call any mapping between model realizations an *interpretation*.

**Definition 5.** An *interpretation*  $\llbracket \cdot \rrbracket$  is a mapping from the model realizations of one domain to the model realizations of another domain.

$$\llbracket \cdot \rrbracket : R_{\Upsilon} \mapsto R_{\Upsilon'} \tag{III.2}$$

A single domain may have many different interpretations, and these form a family of mappings  $(\llbracket \cdot \rrbracket_j)_{j \in J}$ . For some model  $r \in R_{\Upsilon}$ , we denote the  $j^{\text{th}}$  interpretation of  $r$  as  $\llbracket r \rrbracket_j$ . The interpretations capture the (dynamic) semantics of a domain. For example, verification tools map a non-trivial class of models onto the boolean domain  $\{\mathbf{true}, \mathbf{false}\}$ . We can think of this verification tool as an interpretation  $\llbracket \cdot \rrbracket_{\text{Verify}}$  that maps models onto a domain containing exactly two models. Similarly, simulators map models onto execution traces. The set of all traces can be collected together into a domain of well-formed traces, and a simulator can be expressed as the mapping  $\llbracket \cdot \rrbracket_{\text{Sim}}$  onto this trace domain. (Trace domains often have interesting constraints that separate the well-formed traces from the malformed ones [85].) Notice that there is no difference between transformational semantics and model transformations. Any framework that supports model transformations also supports specification of transformational semantics. We can now define a DSML:

**Definition 6.** A *domain specific modeling language* (DSML)  $L$  is a pair comprised of its domain and interpretations.

$$L = \left\langle D, (\llbracket \cdot \rrbracket_j)_{j \in J} \right\rangle. \quad (\text{III.3})$$

This definition differs from those presented elsewhere [43] because we expose the components of the structural semantics, while ignoring all together the “concrete syntax”. But, other than this emphasis, there is little conceptual difference between our definition and others.

Every domain has at least one interpretation, which is its structural interpretation. Let  $\Upsilon_B$  contain two nullary function symbols  $\{\mathit{true}, \mathit{false}\}$ , and let the set of well-formed models be  $\left\{ \{\mathit{true}\}, \{\mathit{false}\} \right\}$ . The structural interpretation of a domain  $\llbracket \cdot \rrbracket_{\text{struc}}$  is a mapping onto  $R_{\Upsilon_B}$  according to:

$$\llbracket r \rrbracket = \begin{cases} \{\mathit{true}\}, (r \models C) \\ \{\mathit{false}\}, (r \not\models C) \end{cases} \quad (\text{III.4})$$

The structural interpretation maps a model  $r$  to the *true* model if  $r$  satisfies its structural constraints. Otherwise  $r$  is mapped to *false*.

The framework of formal logic can also be used to specify interpretations. Recall that a model is just a set of ground terms. Given a model  $r$  and some axioms  $\tau$ , we can deduce more terms with  $\vdash_{\tau}$ . If  $\tau$  is correctly defined, then the new ground terms  $G'$ , where  $r \vdash_{\tau} G'$ , yield the transformed model. We will make this more precise by first defining a *transformation*.

**Definition 7.** A *transformation*  $T$  is a three tuple:

$$T = \langle \Upsilon, \Upsilon', \tau \rangle \quad (\text{III.5})$$

where  $\Upsilon, \Upsilon'$  are disjoint signatures, and  $\tau$  is a set of axioms over  $T_{\Upsilon \cup \Upsilon'}(\Sigma)$ .

A model  $r \in R_{\Upsilon}$  is transformed to a model  $r' \in R_{\Upsilon'}$  by first finding the largest set of deductions  $\psi$ , such that  $r \vdash_{\tau} \psi$ . The resulting set of deductions is projected onto the term algebra of  $\Upsilon'$ , producing a model purely in  $R_{\Upsilon'}$ .

**Definition 8.** Given a transformation  $T$ , a *transformational interpretation*  $\llbracket \cdot \rrbracket^T$  is a mapping:

$$\llbracket \cdot \rrbracket^T : R_{\Upsilon} \rightarrow R_{\Upsilon'}, \quad \llbracket r \rrbracket^T \mapsto \left( \psi \cap T_{\Upsilon'} \right), \quad \text{where } r \vdash_{\tau} \psi. \quad (\text{III.6})$$

Interpretations that preserve the structural semantics of domains are particularly important to embedded system design. These *structure preserving maps* possess the weakest property that one would expect a correct transformational semantics to possess. These maps are also important in correct-by-construction design [86][87][88].

**Definition 9.** An interpretation preserves the structural semantics (is structure preserving) if, whenever a model  $r$  is well-formed, the transformed model  $\llbracket r \rrbracket^T$  is also well-formed:

$$\forall r \in R_{\Upsilon}, (r \models C) \Rightarrow (\llbracket r \rrbracket^T \models C') \quad (\text{III.7})$$

It should be mentioned that the verification of weak properties, such as structure preservation, is still a major open problem in the model transformation community. Our approach allows some of these properties to be transcribed into formal logic, and then proved with an existence proof: Find an  $r \in R_{\Upsilon}$  such that  $\exists w \in T_W, r \vdash_C w$ , but  $\forall u \in T'_W, \llbracket r \rrbracket^T \not\vdash_{C'} u$ . If no such  $r$  exists, then the map is structure preserving. Again, this process may be decidable if a decidable logic is chosen for  $\vdash$ , and  $\tau$  is written correctly.

**Proposition 10.** *Given a transformation  $T$  between two positive domains  $D_1, D_2$ , then  $T$  is structure preserving iff  $\neg \exists r \in R_{\Upsilon_1}$  such that*

$$\exists w \in T_{W_1}, r \vdash_{C_1} w \text{ and } \forall u \in T_{W_2}, \llbracket r \rrbracket^T \not\vdash_{C_2} u$$

Domains and transformations between domains are specified with the same underlying mathematical apparatus. This unification allows the introduction of metamodeling in a mathematically consistent fashion, as we shall show next.

## Metamodels and Metamodeling

DSML structures and interpretations provide the most basic foundations for model-based design. In this section we formalize more advanced DSML design principles using our formalization as a foundation. Specifically, we formalize the *metamodeling process* by which new domains are rapidly defined via the construction and interpretation of *metamodels*. A metamodel is a model that belongs to a special DSML called a *metamodeling language*. The metamodeling language provides an interpretation that maps metamodels to domains.



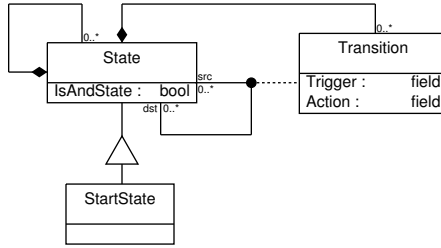


Figure 26. MetaGME metamodel for HFSM.

This process allows users to concisely “model” their domain, and then generate the domain concepts and constraints from the model.

Figure 26 shows a metamodel for hierarchical finite state automata written in a UML notation [40]. The boxes in the model are class definitions, and class members are listed under the class names. For example, the **Transition** class has **Trigger** and **Action** members, both of type **field** (or string). The metamodel also encodes a graph class by associating some classes with vertices and other classes with edges. The **State** and **StartState** classes correspond to vertices; instances of the **Transition** class are edges. The diagram also declares which vertex types can be connected together, and gives the edge types that can make these connections. The solid lines passing through the connector symbol (●) indicate that edges can be created between vertices, and the dashed line from the connector to the **Transition** class indicates that these edges are instances of type **Transition**. The diagram encodes yet more rules: Lines that end with a diamond (◆) indicate hierarchical containment, e.g. **State** instances can contain other states and transitions. Lines that pass through a triangle (△) identify inheritance relationships, e.g. a **StartState** inherits the properties of **State**.

This example illustrates two important points about metamodeling languages. First, a small metamodel can define a rich domain that may include a non-trivial inheritance hierarchy, a graph class, and other concepts like hierarchical containment and aspects. Metamodels are concise specifications of complex domains. Second, the *meanings* of metamodeling constructs are tedious to define, and the language appears idiosyncratic to users. This problem is compounded by the fact that competing metamodeling languages are “defined” with excessively long standards: The GME manual [41], much of which is devoted to metamodeling, is 224 pages. The Meta Object Facility (MOF) language, an OMG standard used by MDA and UML, requires a 358 page description [77]. These long natural language descriptions mean that tool implementations are likely to differ from the standards, and that the standards themselves are more likely to be inconsistent or ambiguous.

We hope to alleviate some of these problems by formalizing the metamodeling process. A metamodeling language  $L_{meta}$  is a DSML with a special interpretation  $\llbracket \cdot \rrbracket_{meta}$  (called the *metamodeling semantics*) that maps *models* to *domains*:

$$L_{meta} = \langle D_{meta}, (\llbracket \cdot \rrbracket_{meta}) \rangle \quad (\text{III.8})$$

The interpretation  $\llbracket r \rrbracket_{meta}$  maps a model realization  $r$  to a new domain. There is one technical caveat: Interpretations map from models of one domain to models of another domain. In order to make a mapping from models to domains, we need to create a *domain of domains* that provides a structural encoding for domains. A domain of domains is created by first choosing a class of formulas  $\mathcal{F}$  that are to be used for describing the constraint axioms of every domain. Next, we construct a domain  $D_{\mathcal{F}}$  and a bijection  $\delta : \mathbb{Z}_+^{\mathcal{V}} \times \mathbb{Z}_+^{\mathcal{V}} \times \mathcal{P}(\mathcal{F}) \rightarrow D_{\mathcal{F}}$  that maps two signatures and a set of formulas to a model in the special domain  $D_{\mathcal{F}}$ . The notation  $\mathbb{Z}_+^{\mathcal{V}}$  is the set of all partial functions from  $\mathcal{V}$  to  $\mathbb{Z}_+$ , i.e. the set of all signatures. Note that for the domain of domains we will fix a particular  $\Sigma$  and  $\vdash$ . This approach allows us to specify metamodeling languages transformationally, as shown in Figure 27. The domain  $D_{meta}$  represents a metamodeling language

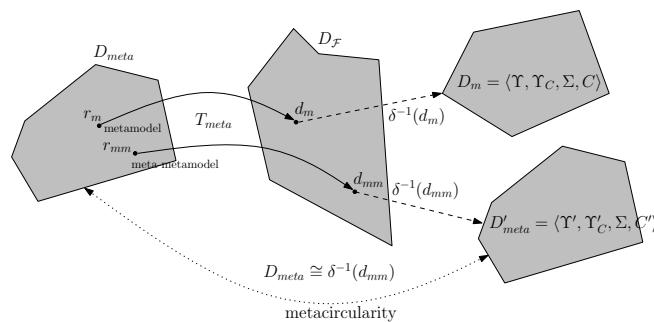


Figure 27. Abstract view of the metamodeling process

with some arbitrary notation for describing domains (e.g. UML).  $T_{meta}$  is a transformation that converts models in  $D_{meta}$  to a structural representation of a domain in  $D_{\mathcal{F}}$ . The transformation  $T_{meta}$  encodes the semantics of the metamodeling language. For example, the metamodel  $r_m$  is transformed to the model  $d_m$  by applying the interpretation  $\llbracket r_m \rrbracket^{T_{meta}}$ . The actual domain defined by a metamodel is recovered by the inverse function  $\delta^{-1}$  that recovers a domain from a model in  $D_{\mathcal{F}}$ . Thus, the domain defined by the metamodel  $r_m$  is discovered by applying  $\delta^{-1}(\llbracket r_m \rrbracket^{T_{meta}})$ . Our formalization also allows us to describe the notion of *metacircularity* precisely. Intuitively, a metamodeling language is metacircular if there exists a metamodel in the language that defines the language. Formally, a metamodeling language is metacircular if there exists a well-formed metamodel  $r_{mm}$  such that  $D_{meta} \cong \delta^{-1}(\llbracket r_{mm} \rrbracket^{T_{meta}})$ . The model  $r_{mm}$  is called the *meta-metamodel*, as shown in Figure 27. This can be imagined geometrically: The set of all well-formed metamodels forms a decision boundary in  $R_{\Upsilon_{meta}}$ . A metamodeling language is metacircular if there exists a metamodel that reconstructs the decision boundary of the metamodeling language.

We have provided a structural semantics for model-based design. Next, we examine the applicability of our approach to existing model-based tool suites. We will apply our methods to the Model-Integrated Computing (MIC) tool suite [43] by producing an algebraic definition for each of its components. These components include a metamodeling facility (MetaGME), a modeling editor (Generic Model Environment (GME)), and a model transformation tool (Graph Rewriting and Transformation (GReAT) tool). Our algebraic specification of these components captures the structural semantics of the tool suite independently from the software implementation of the individual tools. With the algebraic definitions in hand, we can associate objects in the software tools (models, metamodels, and transformations) with mathematical entities in the algebraic formulation of the tool suite.

For the remainder of this chapter we will focus our attention on tool suites. However, this is not the only application of structural semantics. We have extended the algorithms found in logic programming to constructively analyze domains and transformations. Our analysis tool, called *FORMULA*, is based on a nonmonotonic extension of Horn logic. *FORMULA* (FORMal Modeling Using Logic Analysis) is a fully constructive theorem prover for a class of stratified axioms. *FORMULA*'s extension of algorithms for nonmonotonic logic yields a fascinating intersection of modeling and formal methods. More details on *FORMULA* can be found in [89].

### Formalizing Model-Integrated Computing

Before we apply our framework to MIC, we must choose the style of logic for expressing constraints. This is the most important degree of freedom, because it adjusts the expressiveness of the resulting specialization. In particular, the consequence operator affects the algorithmic complexity of checking model well-formedness and calculating the properties of domains and transformations. In order to provide a reasonable degree of expressiveness, while maintaining some analyzability we shall write axioms in an extended form a Horn logic.

First, let us review some basic definitions, beginning with basic Horn logic. *Formulas* are built from terms with *variables* and logical *connectives*. There are different approaches for distinguishing variables from constants. One way is to introduce a new alphabet  $\Sigma_v$  that contains variable names such that  $\Sigma \cap \Sigma_v = \emptyset$ . The terms  $T_{\Upsilon_C}(\Sigma)$  are called ground terms, and contain no variables. This set is also called the *Herbrand Universe* denoted  $\mathcal{U}_H$ . The set of all terms, with or without variables, is  $T_{\Upsilon_C}(\Sigma \cup \Sigma_v)$ , denoted  $\mathcal{U}_T$ . Finally, the set of all *non-ground* terms is just  $\mathcal{U}_T - \mathcal{U}_H$ . A *substitution*  $\phi$  is term endomorphism  $\phi : \mathcal{U}_T \rightarrow \mathcal{U}_T$  that fixes constants. In another words, if a substitution  $\phi$  is applied to a term, then the substitution can be moved to the inside  $\phi f(t_1, t_2, \dots, t_n) = f(\phi t_1, \phi t_2, \dots, \phi t_n)$ . A substitution does not change constants, only variables, so  $\forall g \in \mathcal{U}_H, \phi(g) = g$ . We say two terms  $s, t \in \mathcal{U}_T$  *unify* if there exists substitutions  $\phi_s, \phi_t$  that

make the terms identical  $\phi_s s = \phi_t t$ , and of finite length. (This implies the *occurs check*[90] is performed.) We call the pair  $(\phi_s, \phi_t)$  the unifier of  $s$  and  $t$ . The variables that appear in a term  $t$  are  $vars(t)$ , and the constants are  $const(t)$ .

A *Horn clause* is a formula of the form  $h \Leftarrow t_1, t_2, \dots, t_n$  where  $h$  is called the *head* and  $t_1, \dots, t_n$  are called the *tail* (or body). We write  $T$  to denote the set of all terms in the tail. The head only contains variables that appear in the tail,  $vars(h) \subseteq \bigcup_i vars(t_i)$ . A clause with an empty tail ( $h \Leftarrow$ ) is called a *fact*, and contains no variables. Recall that these clauses will be used *only* to calculate model properties. This is enforced by requiring the heads to use those function symbols that do not encode model structure, i.e. every head  $h = f(t_1, \dots, t_n)$  has  $f \in (\Upsilon_C - \Upsilon)$ . (Proper subterms of  $h$  may use any symbol.) This is similar to restrictions placed on declarative databases[91]. We slightly extend clauses to permit *disequality* constraints. A Horn clause with disequality constraints has the form  $h \Leftarrow t_1, \dots, t_n, (s_1 \neq s'_1), (s_2 \neq s'_2), \dots, (s_m \neq s'_m)$ , where  $s_i, s'_i$  are terms with no new variables  $vars(s_i), vars(s'_i) \subseteq \bigcup_i vars(t_i)$ . We can now define the *meaning* of a Horn clause. The definition we present incorporates the *Closed World Assumption* which assumes all conclusions are derived from a finite initial set of facts (ground terms)  $I$ . Given a set of Horn clauses  $\Theta$ , the operator  $\widehat{F}_\Theta$  is called the *immediate consequence operator*, and is defined as follows:

$$M \widehat{F}_\Theta = M \cup \left\{ \phi(h_\theta) \mid \exists \phi, \theta, \phi(T_\theta) \subseteq M \text{ and } \forall (s_i \neq s'_i)_\theta \in \theta, \phi s_i \neq \phi s'_i \right\}$$

where  $\phi$  is a substitution and  $\theta$  is a clause in  $\Theta$ . It can be proved that  $I \vdash_\Theta I_\infty$  where  $I \widehat{F}_\Theta I_1 \widehat{F}_\Theta \dots \widehat{F}_\Theta I_\infty$ . The new terms derivable from  $I$  can be calculated by applying the immediate consequence operator until no new terms are produced (i.e. the least fixed point). Notice that the disequality constraints force the substitutions to keep certain terms distinct. *Nonrecursive Horn logic* adds the restriction that the clauses of  $\Theta$  can be ordered  $\theta_1, \theta_2, \dots, \theta_k$  such that the head  $h_{\theta_i}$  of clause  $\theta_i$  does not unify with any tail  $t \in T_{\theta_j}$  for all  $j \leq i$ . This is a key restriction; without it, the logic can become undecidable. Consider the recursive axiom  $\Theta = \{f(f(x)) \Leftarrow f(x)\}$ . Then  $\{f(c_1)\} \vdash_\Theta \{f(c_1), f(f(c_1)), \dots, f(f(f(\dots f(c_1) \dots)))\}$  includes an infinite number of distinct terms.

The above definition shows that Horn logic corresponds to a classical consequence operator. It is clearly extensive, isotone, and idempotent. However, it imposes too much of a restriction on the structure of domains. We can extend Horn logic by introducing a pseudo-negation that is commonly called *negation-as-failure* (NAF). The extensions allows terms in the tail to be “negated”, e.g.  $h \Leftarrow \neg t$ . The interpretation is that we may conclude  $h$  if  $t$  cannot be proved. Thus negation is equivalent to the failure of inference of certain terms. It turns out that this small change has a resounding affect on the corresponding theory. The consequence operator loses the isotone property, placing it in the fascinating area of nonmonotonic logic[92][93][94]. In the interest of space, we shall leave the reader with this informal description of negation-as-failure.

Finally, we allow *term restrictions* to be placed on domains. A term restriction forces all well-formed

models to use a finite set of terms of the form  $f(\dots)$  that are explicitly enumerated. For example, if a domain has a signature  $\Upsilon = \{(f, 1), (g, 2)\}$ , and we wish to place term restrictions on  $f$ , then we may write  $\forall m, m \in D(\Upsilon, C) \Rightarrow (m \cap \{f(x) | x \in T_\Upsilon(\Sigma)\}) \subseteq \{f(c_1), f(c_2)\}$ . This restriction indicates that if a model  $m$  is well-formed then every term of the form  $f(x) \in m$  has either  $x = c_1$  or  $x = c_2$  for  $c_1, c_2 \in \Sigma$ . We will simplify this notation by writing<sup>3</sup> **restrict**( $f, \{f(c_1), f(c_2)\}$ ). Note that Horn logic has already been implemented in programming languages like Prolog, usually with the SLD resolution algorithm [95]. For simple problems, like checking model well-formedness, we can directly use these existing tools. However, most of the analysis problems we encounter (e.g. domain equivalence) require more sophisticated tools. The theorem prover FORMULA was developed for analyzing DSMLs. Prolog also includes an implementation of NAF, but it must be used carefully as it may be unsound. However, we haven taken care to ensure that our descriptions can be soundly evaluated by Prolog implementations of NAF.

### MiniMeta: A Formalized MIC Tool Suite

We now present a formalized MIC tool suite called *MiniMeta*. As the name implies, MiniMeta is a scaled down version of MIC. MiniMeta uses a simpler metamodeling language called eMOF, the *essential meta-object facility* [96], but does not include all of the MetaGME features. The eMOF language is the kernel of MetaGME and many other metamodeling languages, including MOF and UML. The architecture of MiniMeta is shown in Figure 28. The boxes correspond to domains, boxes with extruding arrows correspond to transformations.

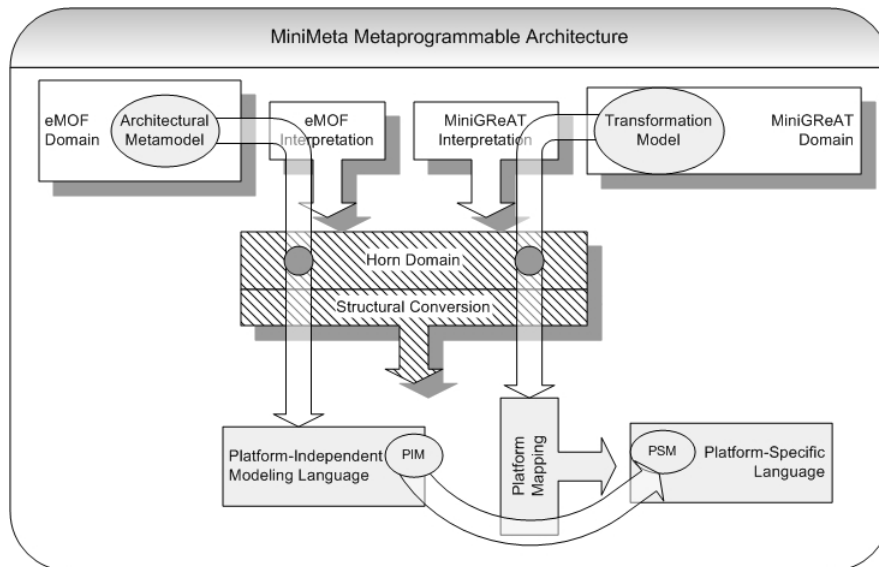


Figure 28. The architecture for the MiniMeta Tool Flow.

<sup>3</sup>We write **constants** in the typewriter font to distinguish them from function symbols.

The ovals are models, and each model is drawn in the domain that contains it. *MiniMeta* contains two special domains, *eMOF* and *MiniGReAT*, as well as two special transformations called the *eMOF Interpretation* and the *MiniGReAT Interpretation*. These four objects constitute the “meta” level of the architecture. The *eMOF* domain defines the set of well-formed metamodels, and the *eMOF Interpretation* converts a metamodel to a model in the domain of domains over Horn formulas, i.e.  $D_{Horn}$ . Similarly, the *MiniGReAT* domain defines the set of well-formed transformation models, and the *MiniGReAT Interpretation* converts a transformation model into a model in  $D_{Horn}$ . The structural conversion operator  $\delta^{-1}$  generates the corresponding domains or transformations from models in  $D_{Horn}$ . In the figure, the Horn domain and the structural conversion operator are crosshatched to indicate that these parts of the framework are generic, and do not depend on the details of *eMOF/MiniGReAT*. The *eMOF* and *MiniGReAT* domains serve as interfaces to the domain and transformation authoring facilities. Though we are free to construct these domains however we wish, we will define the *eMOF* domain in accordance with UML class diagram concepts, and the *MiniGReAT* domain with graph transformation concepts, so that our formalization can be easily linked to existing tools. Also notice that we have not included a “meta-metamodel” for *eMOF*. This is because the *eMOF* domain “bootstraps” the metamodeling facility, so it must be defined purely in Horn notation. Of course, a meta-metamodel for it may exist, but this can only be constructed after the *eMOF* domain and *eMOF* interpretation have been defined. It is possible to write a metamodel for the *MiniGReAT* domain, though this is not necessary.

Once the domain and transformation authoring facilities have been defined, users can apply MIC by constructing DSMLs, transformations, and models of their own. To construct a DSML, the user builds a metamodel, which is a member of the *eMOF* domain. Such a metamodel is shown as the gray oval labeled *Architectural Metamodel* in Figure 28. Though not shown in the diagram, a graphical editor like GME typically helps the user to build such a model. Once complete, the metamodel is converted to a domain via the *eMOF Interpretation*, which transforms the metamodel into a structurally represented set of Horn clauses. The structural conversion operator recovers the actual domain from the Horn model, and this is shown as the *Platform-independent modeling language* in the figure. A similar path occurs for transformations: The user builds a model of a transformation, then transforms this model to the Horn domain. The actual transformation is recovered with a structural conversion, yielding, for example, the *Platform Mapping* transformation. The user then constructs models in the user-defined languages, and applies transformations between these models. In the figure, *PIM* (platform-independent model) is transformed to *PSM* (platform-specific model). The metalevel of the architecture provides the necessary facilities to produce DSMLs, platform stacks, and platform mappings. We now formalize each of these components.

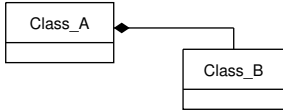
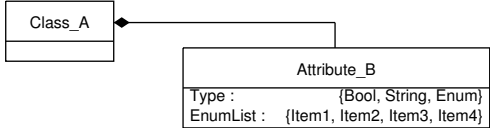
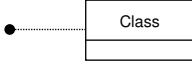
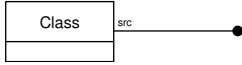
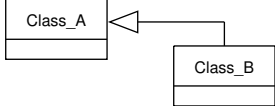
The eMOF notation is based on the Unified Modeling Language (UML), which is also standardized by the OMG. This notation supports the following essential concepts: classes, associations, attributes, containment, and inheritance. Since UML is usually drawn in a graph-like notation, we will imagine some concepts to be annotated vertices and others to be annotated edges. (The actual encoding may be more complicated than just unary and binary relations.) Table 9 lists the vertex-like primitives. The first column describes the

Table 9. Table of Vertex Primitives for MiniUML Metamodels

Vertex Primitives	UML Example
<p style="text-align: center;"><b>Class</b></p> <p>A class is primitive part of a metamodel that can have <i>Containment</i>, <i>Attribute Containment</i>, <i>Association Endpoint</i>, <i>Inheritance</i> edges incident on it. Every class has a <i>name</i>.</p>	
<p style="text-align: center;"><b>Association Class</b></p> <p>An association class is a primitive on which <i>Containment</i>, <i>Association</i>, <i>Attribute Containment</i>, and <i>Inheritance</i> edges can be incident. Every association class has a <i>name</i>.</p>	
<p style="text-align: center;"><b>Attribute Class</b></p> <p>An attribute class is a primitive on which <i>Attribute Containment</i> edges can be incident. Every attribute class has a <i>name</i> and a <i>type</i> which can be <i>boolean</i>, <i>string</i>, or <i>enumeration</i>. Attribute classes of type enumeration may have a list of <i>enumeration values</i>.</p>	
<p style="text-align: center;"><b>Connector</b></p> <p>A connector has exactly two <i>Association Endpoint</i> edges and one <i>Association</i> edge incident on it.</p>	

primitives and any rules dictating the use of those primitives. The second column depicts a typical use-case of the primitives using UML-like notation. Table 10 provides similar data for the edge-like primitives.

Table 10. Table of Edge Primitives for MiniUML Metamodels

Edge Primitives	UML Example
<p style="text-align: center;"><b>Containment</b></p> <p>An edge that must terminate on a <i>Class</i>.</p>	
<p style="text-align: center;"><b>Attribute Containment</b></p> <p>An edge that must start on an <i>Attribute Class</i>.</p>	
<p style="text-align: center;"><b>Association</b></p> <p>An edge that must start on a <i>Connector</i> and end on an <i>Association Class</i>.</p>	
<p style="text-align: center;"><b>Association Endpoint</b></p> <p>An edge that must start on a <i>Connector</i> and end on an <i>Association Class</i>. Association endpoints have a <i>incidence</i> that can be either <i>source</i> or <i>destination</i>.</p>	
<p style="text-align: center;"><b>Inheritance</b></p> <p>An edge that cannot form a directed cycle consisting only of inheritance edges.</p>	



These tables describe how primitives are composed into metamodels, but they do not describe how metamodels encode domains. Thus, even at the metamodeling level, we strictly maintain a separation between the *metamodel structural semantics* (the legal metamodels) and the *metamodeling semantics* (how metamodels encode domains). At this point we are only describing the structural semantics of metamodels, which we do by characterizing the malformed models, i.e. using the *malform* symbol. Some constraints are easy to describe; for example the *attribute* primitive has a *type* that can only be one of the values `{bool, string, enum}`. We encode this constraint by placing a term restriction on the *type* symbol.

$$(type, 1) \in \Upsilon,$$

$$\mathbf{restrict}(type, \{type(\mathbf{bool}), type(\mathbf{string}), type(\mathbf{enum})\})$$

The *incidence* property on an *association endpoint* is another example of a term restriction. A more interesting constraint comes from the acyclic nature of inheritance, or, more precisely, from the nature of cycles themselves. When we say “no cycles”, we really mean an infinite list of Horn constraints: “no self-loops”, “no cycles of length two”, “no cycles of length three”, . . . , ad infinitum. Because we require the logic to be acyclic (no recursion), we cannot faithfully encode properties with an infinite number of equivalence classes (distinctly different structural incarnations). The reader familiar with logic programming might object to this claim, because standard Prolog permits a concise description of cycles using lists; other languages permit descriptions without lists, but using recursion. The key difference is that Prolog reasons about single models of finite size. Thus, the length of lists may be bounded or the recursion may terminate because proofs are made over finite models (closed world assumption). However, when we solve analysis problems, such as “find some well-formed model with size greater than one”, we must find the existence of model within a infinite number of possible models. The acyclic Horn logic ensures that this search procedure only needs to consider a finite number of models from the domain. Thus, the only way to properly encode an acyclic inheritance hierarchy is to approximate the “no cycles” constraint for a finite range of cycle lengths from 1 to  $n$ . Let the symbol *inheritance*( $x, y$ ) denote an inheritance edge from  $x$  to  $y$ . This auxiliary symbol is placed in the constraint signature  $\Upsilon_C$ . Similarly, let *ipath*<sub>3</sub>( $x, y, z$ ) and *ipath*<sub>4</sub>( $x, y, z, w$ ) indicate directed inheritance paths of length three and four. The following axioms allow us to calculate inheritance paths of these lengths:

$$\{(inheritance, 2), (ipath_3, 3), (ipath_4, 4)\} \subset \Upsilon_C$$

$$ipath_3(x, y, z) \Leftarrow inheritance(x, y), inheritance(y, z), (x \neq y \neq z),$$

$$ipath_4(x, y, z, w) \Leftarrow ipath_3(x, y, z), inheritance(z, w), (w \neq x \neq y \neq z).$$

An inheritance path of length three is made up of two inheritance edges across three distinct vertices. The disequality constraints  $x \neq y \neq z$  require the vertices in the path to be distinct. Three such constraints are needed to ensure that all three vertices are distinct. The four-path is defined by the presence of a three-path and a new edge that extends the three-path by one unique vertex. This process can be continued to define any path of finite length. A cycle of length  $n > 2$  is defined by the presence of a path of length  $n$  and an

inheritance edge that connects the end of the path to the beginning. The definitions for  $icycle_1$ ,  $icycle_2$ ,  $icycle_3$  and  $icycle_4$  are shown below:

$$\begin{aligned} & \{(icycle_1, 1), (icycle_2, 2), (icycle_3, 3), (icycle_4, 4)\} \subset \Upsilon_c \\ & icycle_1(x) \Leftarrow inheritance(x, x) \\ & icycle_2(x, y) \Leftarrow inheritance(x, y), (x \neq y) \\ & icycle_3(x, y, z) \Leftarrow ipath_3(x, y, z), inheritance(z, x) \\ & icycle_4(x, y, z, w) \Leftarrow ipath_4(x, y, z, w), inheritance(w, z) \end{aligned}$$

Finally, an inheritance hierarchy is malformed ( $imalform$ ) if there is any such cycle.

$$\begin{aligned} & (imalform, 1) \in \Upsilon_C \\ & imalform(icycle_1(x)) \Leftarrow icycle_1(x) \\ & imalform(icycle_2(x, y)) \Leftarrow icycle_2(x, y) \\ & imalform(icycle_3(x, y, z)) \Leftarrow icycle_3(x, y, z) \\ & imalform(icycle_4(x, y, z, w)) \Leftarrow icycle_4(x, y, z, w) \end{aligned}$$

The rest of the constraint axioms for eMOF are shown in Appendix B.3. Note that this particular encoding of the domain constraints uses much pseudo-negation. We have done this to reduce the number of constraints that must be written. However, for efficiency purposes, it is better to minimize the amount of negation that is used. At this point, we have formalized the eMOF domain with axioms written in an extended form of Horn logic. This constitutes a tool-independent and precise definition of the eMOF domain.

#### The Horn Domain $D_{Horn}$

The next step in the formalization process is to write the transformation  $T_{meta}$  from the eMOF domain to the Horn domain. Before we can do this, we must define the domain of domains for Horn logic. Note that all domains, including  $D_{Horn}$ , will be defined over a fixed alphabet  $\Sigma$  such that  $\mathbb{Z}_+ \subset \Sigma$ . Furthermore, there exists some subset of  $\Sigma$ , called  $\Sigma_F$ , that is bijectively related to the vocabulary of function names  $\mathcal{V}$ , via a bijection  $\delta_f$ . This bijection allows a function symbol to be translated into a constant for the purpose of representation. A similar bijection  $\delta_V$  must exist between a subset of  $\Sigma$ , called  $\Sigma_V$ , and the set of variable names used by the class of Horn formulas  $\mathcal{F}_{Horn}$ . Choose  $\Sigma$  so that  $\Sigma_F \cap \Sigma_V = \emptyset$ . Table 11 lists the function symbols and constraints placed on the Horn domain. The table also informally describes what the inverse of an encoded term yields with respect to signatures and formulas. Note that we have not included a symbol for encoding term restrictions. These will be encoded by axioms with empty tails. The relationship between Horn models and domains is formalized by providing a mapping  $\delta : \mathbb{Z}_+^{\mathcal{V}} \times \mathbb{Z}_+^{\mathcal{V}} \times \mathcal{P}(\mathcal{F}_{Horn}) \rightarrow D_{Horn}$  from a domain (signatures and constraints) to the domain of domains. This mapping is defined with the following structural induction over  $\langle \Upsilon, \Upsilon_C, \Sigma, C \rangle$ :

**Definition 11.** The structural representation function  $\delta$  is given by the following induction:

1.

$$\delta(\Upsilon, \Upsilon_C, C) \mapsto \delta(\Upsilon) \cup \delta(\Upsilon_C) \cup \left( \bigcup_{s \in C} \delta(s) \right)$$

2.

$$\delta(\Upsilon) \mapsto \bigcup_{f \in \text{dom} \Upsilon} \text{def}(\text{sig}, \delta_f(f), \Upsilon(f))$$

3.

$$\delta(\Upsilon_C) \mapsto \bigcup_{f \in \text{dom} \Upsilon} \text{def}(\text{con}, \delta_f(f), \Upsilon(f))$$

4.

$$s_i \in C, s_i = \left( \begin{array}{l} H \Leftarrow \neg L'_1, \dots, \neg L'_m, \\ L_1, \dots, L_n, (v_{j_1} \neq v_{j_2}), \\ \dots, (v_{j_k} \neq v_{j_k}) \end{array} \right), \delta(s_i) \mapsto \left\{ \begin{array}{l} \text{axiom}(i, \delta(H)) \cup \\ \bigcup_{L' \in s_i} \text{tail}(i, \text{neg}(\delta(L'))) \cup \\ \bigcup_{L \in s_i} \text{tail}(i, \delta(L)) \cup \\ \bigcup_{(v \neq u) \in s_i} \text{tail}(i, \text{neg}(\delta(v), \delta(u))) \end{array} \right.$$

5.

$$\delta f(t_1, t_2, \dots, t_n) \mapsto \text{map}_{\Upsilon(f)+1}(\delta_f(f), \delta(t_1), \dots, \delta(t_n))$$

6.

$$\delta(x) \mapsto \text{var}(\delta_v(x)) \text{ where } x \text{ is a variable. } \delta(c) \mapsto c \text{ where } c \text{ is a constant.}$$

The well-formed models in the Horn domain cannot be defined entirely with Horn logic. Three constraints require additional operators that add integers (+) and compute subterms ( $\sqsubseteq$ ). The first constraint states that the arity of a  $\text{map}_n(x, \dots)$  term must match the arity of the function symbol  $x$ .

$$\text{malform}(\text{map}_n(x, y_1, \dots, y_{n-1})) \Leftarrow \left( \begin{array}{l} \text{map}_n(x, y_1, \dots, y_{n-1}), \\ \text{def}(t, x, y), (n \neq y + 1) \end{array} \right)$$

The second constraint requires that variables introduced in the head of a clause must have been introduced in the tail of the clause. To express this constraint we introduce a *subterm* relation  $\sqsubseteq$  such that a term  $t'$  is a subterm of a term  $t$  if  $t'$  appears in an argument of  $t$  or an argument of some subterm of  $t$ .

$$\begin{aligned} \text{vargood}(v, x) &\Leftarrow \text{axiom}(x, h), \text{tail}(x, t), (\text{var}(v) \sqsubseteq h), (\text{var}(v) \sqsubseteq t) \\ \text{malform}(\text{axiom}(x, h)) &\Leftarrow \text{axiom}(x, h), (\text{var}(v) \sqsubseteq h), \neg \text{vargood}(v, x) \end{aligned}$$

The final constraint prohibits pseudo-negation in the head of a clause, as pseudo-negation does not have meaning in the head.

$$\text{malform}(\text{axiom}(x, h)) \Leftarrow \text{axiom}(x, h), (\text{neg}(h') \sqsubseteq h)$$

Though these axioms are not written in the strict Horn logic that we previously defined, they do not significantly impact algorithms that deduce formal properties of domains.

Domains viewed as two signatures and a family of axioms have an equivalence ( $=$ ) between them that only takes into account the equivalence of their notation:  $D_i = D_j$  if  $\Upsilon_i = \Upsilon_j$ ,  $\Upsilon_{C_i} = \Upsilon_{C_j}$ , and  $s_p = s'_p$  for each  $s_p \in C_i, s'_p \in C_j$ . This equivalence is a weak form of equivalence that depends on a common indexing of the axioms for  $D_i$  and  $D_j$ . It holds that  $(D_i = D_j) \Rightarrow (D_i \cong D_j)$ , but the converse does not hold.

**Proposition 12.** *Fix  $\Sigma, \mathcal{V}, \delta_f$ , and  $\delta_V$ . The representation function  $\delta$  is a one-to-one and onto function from domains with Horn axioms to the set of well-formed Horn models  $D_{Horn}(\Upsilon_{Horn}, C_{Horn})$ .*

We briefly sketch the proof. Any well-formed Horn model corresponds to a well-formed (possibly empty) set of signatures and to a well-formed set of extended Horn formulas. Thus,  $\delta$  is an onto function. Given two domains  $D_i$  and  $D_j$ , if  $\delta(D_i) = \delta(D_j)$ , then the domains must have the same signatures and a common indexing of identical axioms, therefore  $D_i = D_j$ . Thus,  $\delta$  is a bijection and there exists an inverse  $\delta^{-1}$  that maps Horn models to domains defined with Horn formulas.

Table 11. Encoding of concepts in the Horn Domain.

**The Horn Domain**

<p><b>Define.</b> <math>def(x, y, z)</math> defines a function symbol <math>y</math> with arity <math>z</math> in a signature <math>x</math>. If the same symbol appears in multiple signatures, then the arity of the symbol must be the same in every signature. There may be two signatures, one for <math>\Upsilon(\mathbf{sig})</math> and one for <math>\Upsilon_C(\mathbf{con})</math>.</p> <p><math>\{(def, 3), (sigtype, 1)\} \subset \Upsilon_{Horn}</math>  <b>restrict</b><math>(sigtype, \{sigtype(\mathbf{sig}), sigtype(\mathbf{con})\})</math>  <math>malform(def(x, y, z)) \Leftarrow def(x, y, z) \wedge def(x', y, z'), (z \neq z')</math>  <math>malform(x, y, z) \Leftarrow def(x, y, z), \neg sigtype(x)</math></p> <p>The inverse representation function <math>\delta^{-1}</math> of a term <math>def(x, y, z)</math> yields a symbol definition of the form <math>(y, z) \in \Upsilon_x</math>.</p>
<p><b>Map<sub>n</sub>.</b> <math>map_n(x, y_1, y_2, \dots, y_{n-1})</math> converts an <math>n</math>-ary term to <i>prefix form</i>. The symbol name <math>x</math> must be defined with a <i>def</i>. The domain definition provides a finite number <math>k</math> of <i>map</i> symbols.</p> <p><math>\{(map_2, 2), \dots, (map_k, k)\} \subset \Upsilon_{Horn}</math>  <math>malform(map_2(x, y_1)) \Leftarrow map_2(x, y_1), \neg def(t, x, z)</math>  <math>\dots</math>  <math>malform(map_k(x, y_1, \dots, y_{k-1})) \Leftarrow map_k(x, y_1, \dots, y_{k-1}), \neg def(t, x, z)</math></p> <p>The inverse representation function <math>\delta^{-1}</math> on <math>map_n(x, y_1, \dots, y_{n-1})</math> yields a literal of the form <math>\delta_f^{-1}(x)(\delta^{-1}(Y_1), \dots, \delta^{-1}(y_1))</math>.</p>
<p><b>Axiom/Tail.</b> <i>axiom</i> defines the head of an axiom and assigns it a unique identifier. <i>tail</i> adds a tail literal to an axiom by referring to the axiom's unique identifier. Each <i>tail</i> must be added to an axiom that has been defined with <i>axiom</i>. Every axiom identifier must be unique.</p> <p><math>\{(axiom, 2), (tail, 2)\} \in \Upsilon_{Horn}</math>  <math>malform(tail(x, y)) \Leftarrow tail(x, y), \neg axiom(x, z)</math>  <math>malform(axiom(x, y)) \Leftarrow axiom(x, y), axiom(x, z), (y \neq z)</math></p> <p>Given <math>axiom(x, h)</math> and tails <math>tail(x, t_1), \dots, tail(x, t_m)</math>, the inverse representation function <math>\delta^{-1}</math> yields a clause with the corresponding head and all tails conjuncted together: <math>\delta^{-1}(h) \Leftarrow \delta^{-1}(t_1), \dots, \delta^{-1}(t_m)</math>.</p>
<p><b>Neg/Neq/Var.</b> <math>neg(x)</math> indicates the negation of literal <math>x</math>. <math>neq(x, y)</math> indicates the disequality <math>x \neq y</math>. <math>var(x)</math> indicates that <math>x</math> is a variable.</p> <p><math>\{(neg, 1), (neq, 2), (var, 1)\} \subset \Upsilon_{Horn}</math></p> <p>The inverse representation function <math>\delta^{-1}</math> of <math>neg(x)</math> yields the negated term <math>\neg \delta^{-1}(x)</math> and <math>neq(x, y)</math> yields the disequality <math>\delta^{-1}(x) \neq \delta^{-1}(y)</math>. Finally, <math>\delta^{-1}(var(x))</math> yields a variable <math>\delta_v^{-1}(x)</math>.</p>

A metamodeling semantics is defined by a transformation from eMOF metamodels to Horn models:  $T_{meta} = \langle \Upsilon_{eMOF}, \Upsilon_{D_{Horn}}, \tau_{meta} \rangle$ . For the purpose of illustration, we will make this transformation as simple as possible: Every *class* named  $x$  in the input metamodel becomes a unary function symbol  $x(n)$  in the Horn model, where the single argument  $n$  indicates that an object called  $n$  is an instance of  $x$ . For example, the automata metamodel in Figure 26 contains a class called **state** (i.e. the term  $class(\mathbf{state})$ ). This term will be translated to a term in the Horn model that adds a unary function symbol  $state$  to the domain signature:  $def(\mathbf{sig}, \mathbf{state}, 1)$ . Similar to classes, *association classes* become ternary function symbols, where the first argument is the name of association instance, the second is the name of the source object, and the third is the name of the destination object. Attribute classes become binary function symbols, where the first argument identifies the object that contains the attribute instance, and the second argument identifies the value of the attribute instance. The transformation contains the following clauses:

$$\tau_{meta} \supset \left\{ \begin{array}{l} def(\mathbf{sig}, x, 1) \Leftarrow class(x) \\ def(\mathbf{sig}, x, 2) \Leftarrow attribute(x, y) \\ def(\mathbf{sig}, x, 3) \Leftarrow assocClass(x) \end{array} \right.$$

Converting metamodeling concepts to function symbols is the simple part of the transformation. The core of the transformation must produce domain constraints that faithfully implement the metamodel. For example, a model is malformed if it assigns an improper value to an enumeration attribute. In order to introduce this constraint, we generate a function symbol  $enumvalue$ , that contains all the enumeration values for all enumeration attributes using term restrictions. Additionally, for each enumeration attribute, we generate a constraint consisting of a head and two tail terms that requires each attribute instance to use one of the enumerated values. These are generated with the following transformation:

$$\begin{aligned} & def(\mathbf{sig}, \mathbf{enumvalue}, 2) \Leftarrow attribute(x, \mathbf{enum}). \\ & axiom \left( enum(x, y), map_3(\mathbf{enumvalue}, x, y) \right) \Leftarrow enum(x, y) \wedge attribute(x, \mathbf{enum}). \\ & axiom \left( attribute(x, \mathbf{enum}), malformed(map_3(x, var(\mathbf{y}), var(\mathbf{z}))) \right) \Leftarrow attribute(x, \mathbf{enum}). \\ & tail \left( attribute(x, \mathbf{enum}), map_3(x, var(\mathbf{y}), var(\mathbf{z})) \right) \Leftarrow attribute(x, \mathbf{enum}). \\ & tail \left( attribute(x, \mathbf{enum}), neg(map_3(\mathbf{enumvalue}, x, var(\mathbf{z}))) \right) \Leftarrow attribute(x, \mathbf{enum}). \end{aligned}$$

Assume that the **IsAndState** attribute of Figure 26 is an enumeration containing the constants **andState** and **orState**. The metamodeling transformation would produce the following encoding of the this attribute in the Horn model:

```

def(sig, IsAndState, 2), def(sig, enumvalue, 2),
axiom(enum(IsAndState, andState), map3(enumvalue, IsAndState, andState)),
axiom(enum(IsAndState, orState), map3(enumvalue, IsAndState, orState)),
axiom(attribute(IsAndState, enum), malform(map3(IsAndState, var(y), var(z)))),
tail(attribute(IsAndState, enum), map3(IsAndState, var(y), var(z))),
tail(attribute(IsAndState, enum), neg(map3(enumvalue, IsAndState, var(z))))

```

Finally, applying the inverse representation function yields the following axioms:

$$\text{restrict} \left( \begin{array}{c} \{(IsAndState, 2), (enumvalue, 2)\} \subset \Upsilon \\ enumvalue, \left\{ \begin{array}{l} enumvalue(IsAndState, andState), \\ enumvalue(IsAndState, orState) \end{array} \right\} \end{array} \right) \\
malform(IsAndState(y, z)) \Leftarrow IsAndState(y, z), \neg enumvalue(IsAndState, z)$$

Appendix B.1 includes some additional components of the eMOF transformation. This completes the full formalization of the eMOF metamodeling facility. We would have to repeat a similar formalization for the graph transformation language Mini-GRaT. However, formalizing MiniGRaT does not require any new techniques, so we omit its description.

### Implementing MiniMeta with GME/GRaT

In this section we construct an implementation of MiniMeta using existing tools. The implementation will be performed so that each object constructed within a software tool can be mapped to a formal entity in the MiniMeta tool suite. The first feature that we need to implement is a model editor for constructing eMOF metamodels, and for checking well-formedness of metamodels. GME already provides a metamodeling facility, called *MetaGME*, that converts a metamodel into a domain-specific model editor. GME is not formalized and its informal semantics may change from version to version, but it represents a decade of work with over 200,000 lines of C++ code. Thus, there is significant motivation to reuse these tools. This reuse can be done safely by informally defining the eMOF domain with a MetaGME metamodel. This MetaGME description is shown in Figure 30. GME will generate an eMOF model editor that allows us to construct models that “look like” eMOF. Figure 31.a shows an example of an eMOF metamodel constructed within GME.

At this point we can construct eMOF-like objects, but we have not defined how they are mapped to true eMOF models (which are subsets of the eMOF term algebra). We compensate for this by extending GME with an analysis component that converts a GME eMOF model into a set of ground terms. The ground terms are loaded into an embedded Prolog engine along with the formal definition of the eMOF domain (transcribed into Prolog syntax). The SLD resolution procedure of Prolog is used to formally prove that the GME model

is well-formed. Prolog works well for this task, because it only has to reason about a single model, as opposed to an entire domain. Additionally, GME includes a COM-based extension mechanism, so it is trivial to add this component to the model editor. Prolog provides a simple foreign language interface, so it is also trivial to utilize the proving engine. Figure 29 shows the implementation of our formal metamodeling facility. Figure

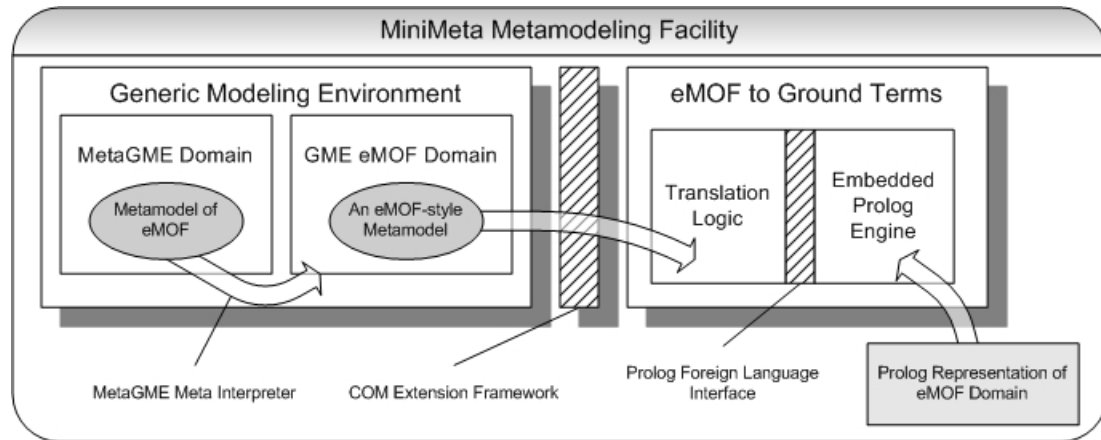


Figure 29. Detailed view of MiniMeta metamodeling facility implementation using MetaGME, GME, and embedded Prolog Engine.

31.b shows the analysis component. The *Translation to definite clauses* list shows the conversion of the eMOF metamodel into ground terms (also called *definite clauses* in Prolog). Each of the translated ground terms is added to the knowledge base of the Prolog engine. The proving engine is then queried to prove `malform(X)`. If this cannot be proved, then the GME model is well-formed. If `malform(X)` can be proved then the model is malformed. In this case, all solutions are displayed to this user; each solution identifies some problem in the model. In Figure 31.b, the engine is unable to prove `malform(x)`, so the model is well-formed. We modified the metamodel of Figure 31.a, and added an inheritance edge from `Interface` to `Input`, thereby creating an inheritance cycle of length 2. Figure 32 shows how this inheritance cycle was correctly detected by the analysis component. This implementation illustrates how existing tools can be readily used to build a formal metamodeling facility. Additionally, our tool architecture introduces a new level of flexibility not found in the existing metaprogrammable tools, because the eMOF domain specification is not hard coded into the tool and can be easily modified.



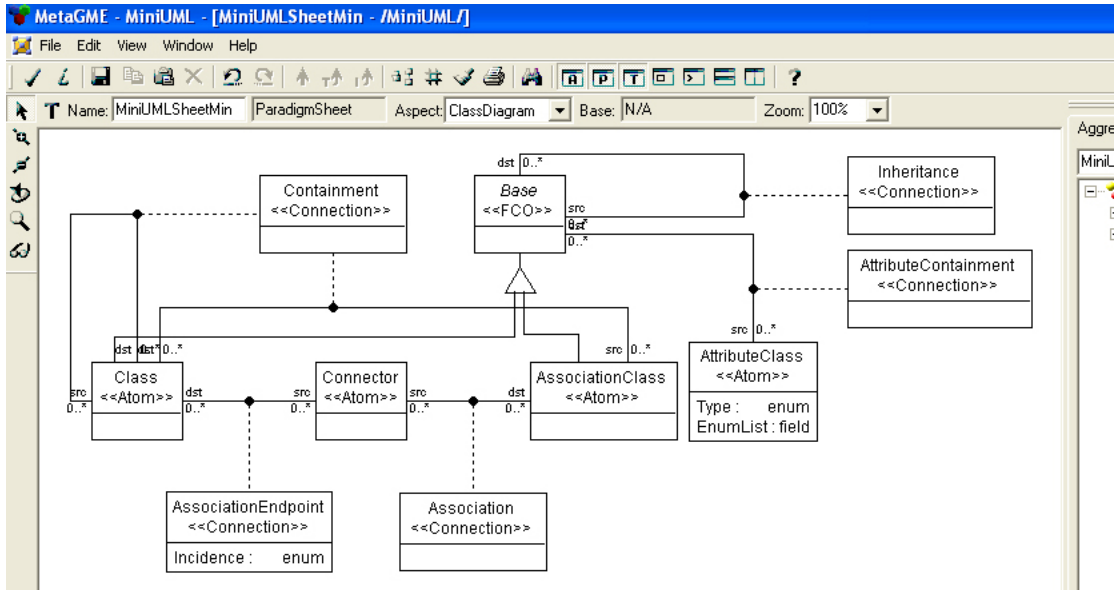


Figure 30. A MetaGME metamodel of the eMOF Domain.

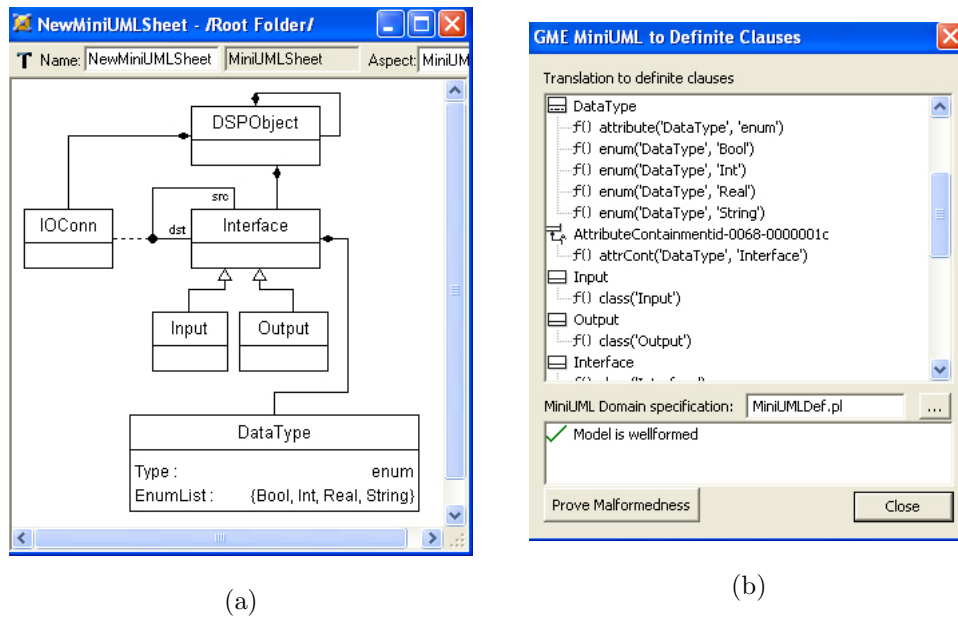


Figure 31. (a) An eMOF metamodel in GME of DSP domain. (b) Translation of metamodel to definite clauses and verification that metamodel is well-formed, using a Prolog engine.



Figure 32. Results of check after inheritance cycle is added to DSP metamodel.

The next feature to implement is the generation of new domains and model editors from eMOF metamod-els. This functionality is also implemented with the help of an embedded Prolog engine, but is necessarily more complex. Figure 33 shows the implementation of the domain generation facility. Assume that a user has constructed an eMOF metamodel of domain  $X$  in GME. This metamodel is shown in the GME eMOF domain in the figure. The domain generator component (upper-right) translates the eMOF metamodel onto ground terms and loads these into another embedded Prolog engine. Additionally, the eMOF transformation axioms (transcribed into Prolog syntax) are loaded into the engine, and a forward chaining procedure deduces the Horn model. The Horn model is extracted from the Prolog engine, and the inverse representation function  $\delta^{-1}$  is applied. This results in the signatures and constraint axioms for domain  $X$ . The signatures and axioms are automatically converted to Prolog syntax and saved to an external file. At this point, a formal procedure has generated a formal definition of domain  $X$ , but we still need to generate a model editor. A model editor is generated by applying a graph transformation (written in GReAT) to the eMOF metamodel.

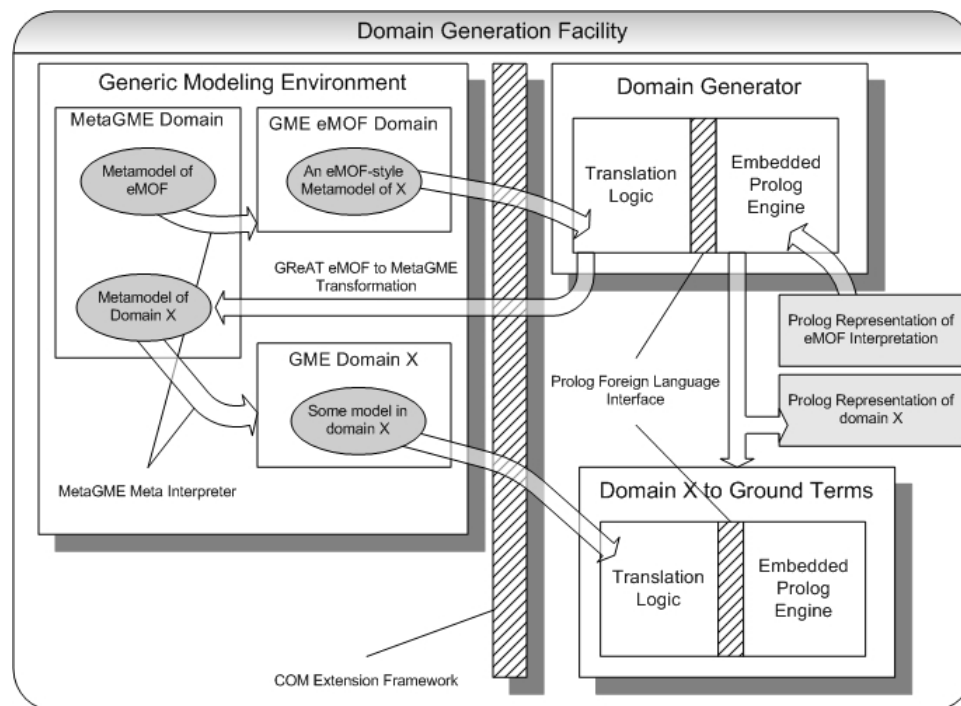


Figure 33. Implementation of the domain generation facility using MetaGME, GME, GReAT, and an embedded Prolog engine.

This graph transformation converts the eMOF metamodel to a similar MetaGME metamodel. The converted metamodel is shown in the MetaGME domain in the figure. Next, the MetaGME interpreter generates the modeling environment for domain  $X$  from the MetaGME metamodel. At this point, we can construct models in GME that look like domain  $X$  models, but we must also be able to check these models

against the formal definition generated by the domain generator. This is accomplished by automatically generating a conversion tool that converts GME models in domain  $X$  to ground terms, as shown in the lower-right hand side of Figure 33. This tool loads the ground terms from an  $X$  model into a Prolog engine, along with the generated Prolog description of domain  $X$ , and then proves well-formedness of a model. This closes the loop and reconnects the tool suite to the formal definitions. Though the details of this process are involved, users need only click one button and all the steps are automatically carried out. Figure 34 shows the domain generation component. The list labeled *Input Model* shows the translation of the input metamodel onto ground terms. Below this list are options for selecting the types of objects generated by the component. If all of the options are checked, then the domain generator performs the entire process described above. The lists on the right-hand side show the terms in the generated Horn model. Appendix B.2 shows the generated domain definition of the eMOF metamodel in Figure 31.a. This metamodel describes a language for constructing hierarchical dataflow graphs.

After the domain generator completes, domain models can be constructed using GME. Figure 35 shows a DSP model created with GME under the DSML generated from Figure 31.a. As was the case with metamodeling, a well-formedness checking tool is attached to the newly generated GME domain. This tool loads the formal definition of the domain into a Prolog engine and converts the domain model into definite clauses, which are then checked for well-formedness. Figure 36 shows the result of activating the well-formedness checking tool on the DSP model of Figure 35. The tree-view titled *Translation to Definite Clauses* shows the translation of each model element into a corresponding set of ground terms. The tree is organized according to the model hierarchy. Interestingly, the tool reports that the model is malformed. This occurs because we augmented the DSP metamodel of Figure 31.a with an additional constraint that disallows a short-circuit of an input with an output. The connection from  $I2$  to  $O2$  is such a short-circuit. Our eMOF domain allows complex constraints to be directly annotated on a metamodel. Figure 37 shows the annotated constraint in DSP metamodel. After the domain generator converts the diagrammatic part of the metamodel into Horn clauses, it adds any annotations to the domain definition. Our approach combines the advantages of metamodeling with constraint languages in a consistent fashion. All of the features get translated into the same underlying formalism.

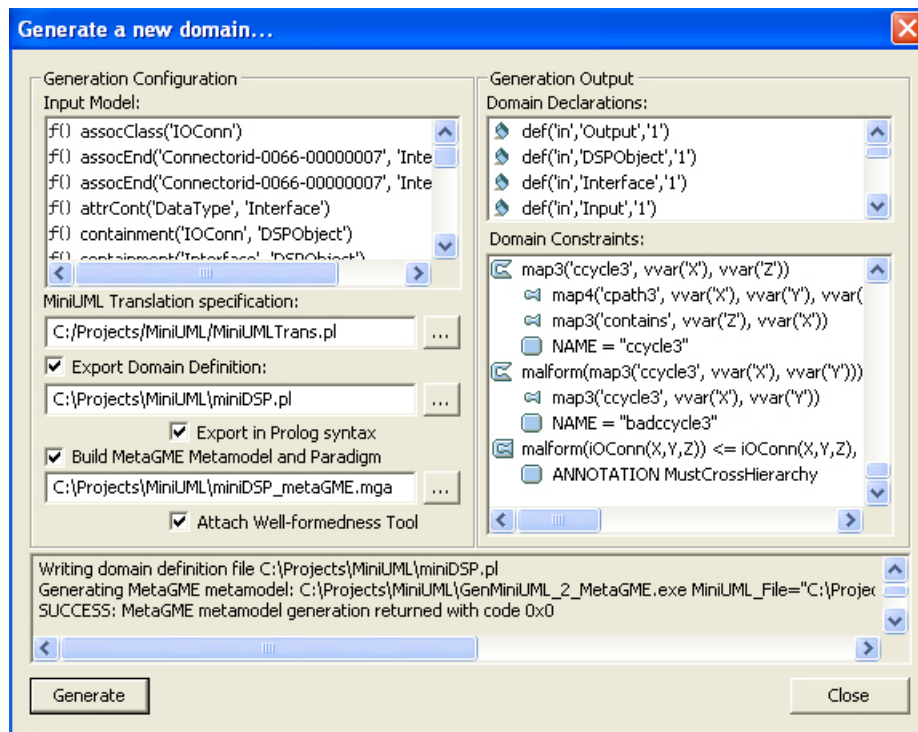


Figure 34. Invocation of the domain generator component.

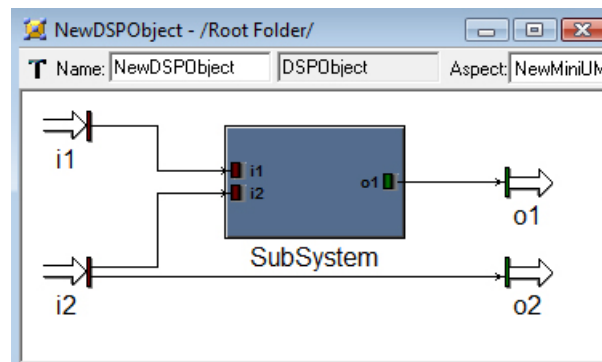


Figure 35. An example DSP model created using the DSML generated in Figure 31.a

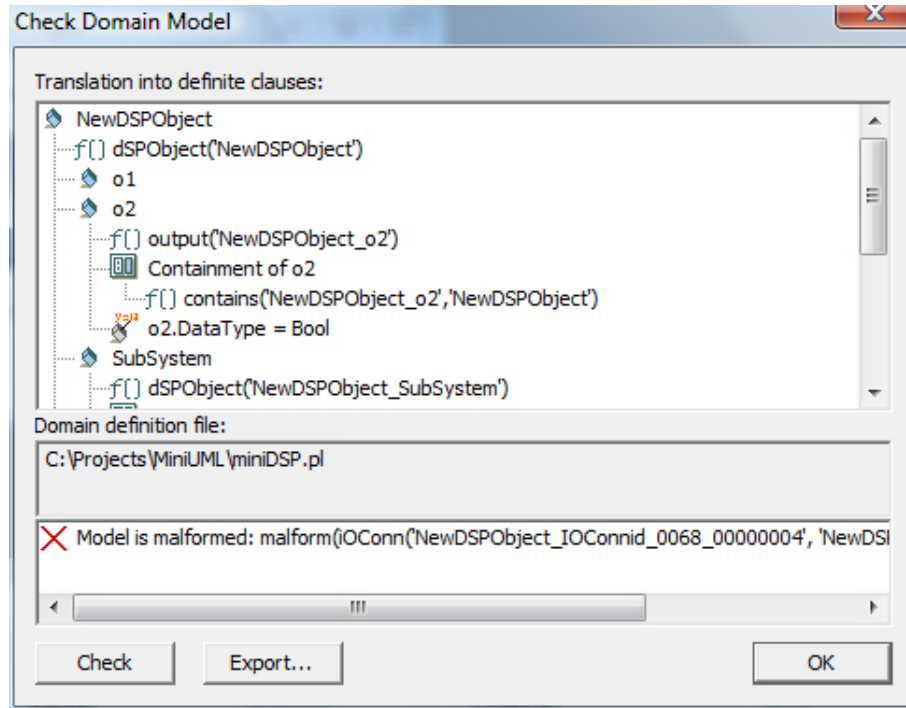


Figure 36. Results of checking the DSP model against the formal domain definition

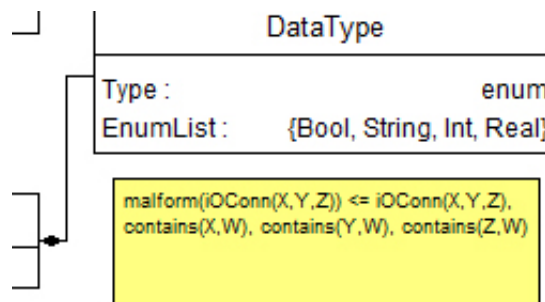


Figure 37. Annotation of the DSP metamodel with an additional constraint

## Discussion and Conclusion

In this work we have explored the structural foundations of model-based design. We developed a generic structural semantics and applied this semantics to a specific model-based framework. Beyond this, there are other interesting directions to explore. For example, domains restricted to Horn constraints can be analyzed with algorithms based on backwards chaining and constructive negation[92]. We have developed a tool called FORMULA (Formal Modeling Using Logic Analysis) that implements Horn-based analysis algorithms for domains. FORMULA is fed a set of domains (signatures and Horn constraints) and may then be queried to prove a property  $\theta$ . By property, we mean a set of terms with variables from  $T_{\mathcal{R}_C}(\Sigma \cup \Sigma_V)$ . A model  $r$  satisfies the property if there exists a substitution  $\phi$  such that  $r \vdash \phi(\theta)$ . For a given property  $\theta$ , FORMULA will construct a model  $r$  and substitution  $\phi$  such that  $r \vdash \phi(\theta)$ , or it will report that no such model exists. This can be used to prove domain equivalence and structure preserving maps. Since the proof procedure is constructive, it may also be useful for generating well-formed models from smaller, possibly malformed, submodels (embeddings). The user may construct a malformed submodel, and then use the tool to find a well-formed version of this submodel.

Our formal structural semantics has some interesting implications on current model-based tool suites. It is well-known that the basic concepts (classes, associations, etc...) in metamodeling languages are not sufficient to encode more intricate structural constraints. For example, imagine a dataflow language where it is illegal to directly short system inputs with system outputs within the same dataflow graph. This type of constraint cannot be specified using only eMOF constructs. The typical solution used by metamodeling tools is to annotate metamodels with a constraint language like OCL (Object Constraint Language). With our approach, metamodels are translated into constraints, so additional constraints can be injected directly into the resulting interpretation of the metamodel. This provides a consistent view of metamodels and constraint annotations: They are just two different ways of describing domain constraints.

To varying degrees, descriptions of model-based frameworks (e.g. MDA, UML, MOF, and MetaGME) use the term meta-metamodel synonymously with the definition of the metamodeling language. However, we have shown that a meta-metamodel is not a definition of the metamodeling semantics. Rather, it is a consequence of the metamodeling semantics, and this is why it can be automatically discovered. This recognition is more profound than just misuse of terminology, because many metaprogrammable modeling tools are hard-coded with a particular metamodeling language. If the metamodeling semantics is viewed as just another model transformation, then there is no reason to hard-code a tool around a particular “meta-metamodel”. The fundamental concepts that should be fixed are the way primitives are composed into models (e.g. via a term algebra) and the style of logic used to write constraints and transformations. Tools built up from this foundation could simultaneously support many different metamodeling languages, and new metamodeling languages could be arbitrarily created without rewriting the tool. Even without rebuilding

tool infrastructure, metamodels should be viewed as formal entities, and as such, it should be possible to migrate them across different tools while preserving their structural semantics.

### **Acknowledgments**

Special thanks to Dr. Constantine Tsinakis. His expertise in Universal Algebra and his clarity of explanation helped to improve the algebraic presentation of this work. This work was supported by NSF grants CCR-0225610 and CCR-0424422.



## CHAPTER IV

### AUTOMATED MODEL CONSTRUCTION AND ANALYSIS

#### Preliminaries - Metamodels, Domains, and Logic

This chapter describes constructive techniques, similar to those found in logic programming, for reasoning about *domain-specific modeling languages* (DSMLs) defined with metamodels. Before we proceed, we must describe how a metamodel can be viewed as a formal object that characterizes the well-formed models adhering to that metamodel. We will refer to the models that adhere to metamodel  $X$  as *the models of metamodel  $X$* . In order to build some intuition for this view, consider the simple *DIGRAPH* metamodel of Figure 38. The models of *DIGRAPH* consist of instances of the *Vertex* and *Edge* classes such that *Edge*

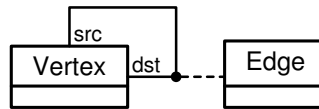


Figure 38. DIGRAPH: A simple metamodel for labeled directed graphs

instances “connect” *Vertex* instances. In another words, *DIGRAPH* characterizes a class of labeled directed graphs. Thus, a model might be formalized as a pair  $G = \langle V \subseteq \Sigma, E \subseteq V \times V \rangle$ , where  $\Sigma$  is an alphabet of vertex labels. If  $\Sigma$  is fixed, then the set  $\mathcal{G}$  of all models of *DIGRAPH* is:  $\mathcal{G} = \{(V, E) | V \subseteq \Sigma, E \subseteq V^2\}$ . This is the classic description of labeled digraphs, and at first glance it might appear possible to extend this description to characterize the models of arbitrary metamodels. Unfortunately, UML-like metamodels[77][40] contain a number of constructs that deny a simple extension of graph-based descriptions. The *UNSAT* metamodel of Figure 39 illustrates some of these constructs. First, classes may have non-trivial internal

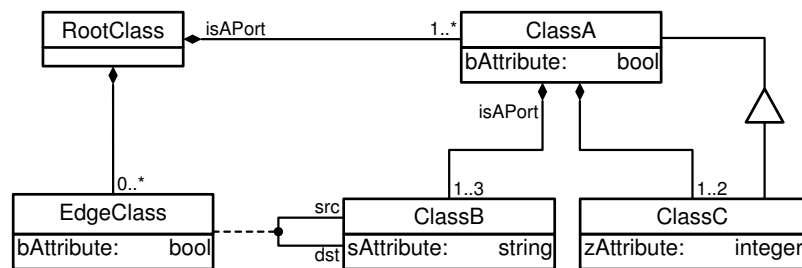


Figure 39. UNSAT: A complex metamodel with no finite non-trivial models

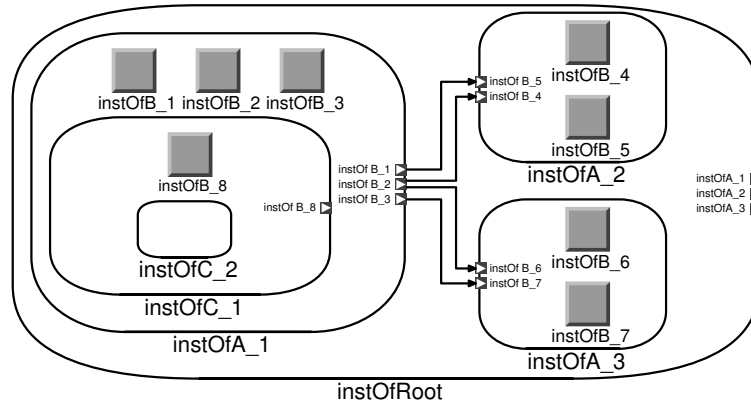


Figure 40. Model that (partially) adheres to the UNSAT metamodel

structure. For example, classes of *UNSAT* have typed member fields (called *attributes*). An instance of *ClassA* has a **boolean** field named *bAttribute*. Classes also inherit this structure, e.g. an instance of *ClassC* has two attributes, *bAttribute* and *zAttribute*, via inheritance. Instances may contain other instances with constraints on the type and number of contained instances. An instance of *ClassA* must contain between 1 and 3 instances of *ClassB*. Second, internal instance structure can be “projected” onto the outside of an instance as *ports*. The containment relation from *ClassA* to *RootClass* has the *isAPort* rolename, requiring that all contained instances of *ClassA* appear as interfaces on the outside of the containing instance of *RootClass*. Figure 40 shows a model with containment and ports. The hollow oblong shapes denote instances that can contain other instances, and the small squares with white arrows on the oblongs’ borders denote ports. For example, the outermost container *instOfRoot* is an instance of the *RootClass* and contains three instances of *ClassA*. Each *ClassA* instance appears as a port on the far right-hand side of *instOfRoot*. Containment and ports are a useful form of information hiding, but they also complicate matters because ports permit edges to cross hierarchy. For example, the edges in Figure 40 connect instances of *ClassB* together even though these instances are not contained in the same parent. Furthermore, the edges are actually contained in the *RootClass* instance, even though the endpoints are not. The third major complication arises because edges are not simple binary relations. In UNSAT, edges are instances of *EdgeClass*, and so each edge has a member field named *bAttribute*. In general, edges must be distinguishable (i.e. labeled), otherwise it would not be possible to reliably determine the values of member fields. In fact, the UML-notation (correctly) implies that edges are ternary associations between an edge label, source label, and destination label.

Graph-based formalisms have been used extensively by the model transformation community, and provide reasonable approximations of model structure for the purpose of transformation. However, in this work we

do not focus on model transformation, but rather we explore techniques for reasoning about all the details of metamodel and model structure. One approach to characterizing realistic model structure might be to combine all existing graph extensions and consider models to be *hierarchical*[82], *typed, attributed*[81] *hypergraphs* with labeled edges. However, even this would not handle all aspects of modern metamodeling languages, and it would produce a brittle and unwieldy formalism. In [39] we present an alternative approach to model structure based on formal logic, which we briefly outline now. In order to present our view, we begin with the concept of a *domain* (in the sense of *domain-specific modeling languages*). A domain  $D = \langle \Sigma, \Upsilon, \Upsilon_C, C \rangle$  is a quadruple where  $\Sigma$  is an (infinite) alphabet for distinguishing model elements,  $\Upsilon$  is a finite signature for encoding model concepts,  $\Upsilon_C$  is a finite signature for encoding model properties, and  $C$  is a set of logical statements (constraints) for deriving model properties. A *model realization* is set of terms from the *term algebra*[84]  $T_\Upsilon(\Sigma)$  over signature  $\Upsilon$  generated by  $\Sigma$ . The set of all possible model realizations is  $\mathcal{P}(T_\Upsilon(\Sigma))$ , i.e. all subsets of terms. We will use the notation  $(f, n) \in \Upsilon$  to indicate that *function symbol*  $f$  of arity  $n$  is a member of the signature  $\Upsilon$ .

**Example 13.** *The domain of labeled digraphs  $D_g$  has the model realizations given by the signature  $\Upsilon = \{(v, 1), (e, 2)\}$  and a countably infinite alphabet ( $|\Sigma| = |\mathbb{N}_0|$ ). These two symbols encode the concepts of vertex and edge. Vertices are encoded using the unary function symbol  $v$  and edges are encoded using the binary function symbol  $e$ . Some model realizations include:*

1.  $M_1 = \{ v(\mathbf{c}_1), v(\mathbf{c}_2), e(\mathbf{c}_1, \mathbf{c}_2) \}$ , a 2-path from a vertex  $\mathbf{c}_1$  to a vertex  $\mathbf{c}_2$ .
2.  $M_2 = \{ v(\mathbf{c}_3), e(\mathbf{c}_3, \mathbf{c}_4) \}$ , a dangling edge starting at vertex  $\mathbf{c}_3$ .
3.  $M_3 = \{ v(e(\mathbf{c}_5, \mathbf{c}_6)), v(v(\mathbf{c}_7)) \}$ , a structure that is not a graph at all.

where the symbols written in `typewriter` font indicate members of the alphabet.

The term algebra easily captures arbitrary  $n$ -ary concepts and permits concepts to be combined in complex ways. Example 13-3 shows that function symbols can be arbitrarily nested. This example also shows that not all model realizations combine the modeling concepts in ways that match our intended meaning of the symbols. Example 13-1 describes a simple 2-path, but 13-2 describes a dangling edge because vertex  $\mathbf{c}_4$  is not in the model. Example 13-3 does not correspond to a graph in any obvious way, but is still a legal member of  $\mathcal{P}(T_\Upsilon(\Sigma))$ .

The set of model realizations of a domain contains all possible ways that the concepts can be used together. In fact, with a single operator  $f$  of arity greater than or equal to one, and an alphabet with at least one element, a countably infinite number of terms can be generated. (Consider a successor operation *succ* and  $\Sigma = \{0\}$ .) Thus, for all non-trivial cases the number of possible model realizations is uncountably infinite. Therefore  $\mathcal{P}(T_\Upsilon(\Sigma))$  will typically contain many model realizations that use the function symbols contrarily to our intentions. In order to counteract this, we must define a set of model properties, characterized by another signature  $\Upsilon_C$ , and a set  $C$  of logical statements for deriving model properties. For simplicity, we assume that

$\Upsilon_C$  simply extends the signature of  $\Upsilon$  (i.e.  $\Upsilon_C \supset \Upsilon$ ). For example, the property of directed paths could be captured by:  $\Upsilon_C = \{(v, 1), (e, 2), (path, 2)\}$  and  $C = \{\forall x, y, z (e(x, y) \vee path(x, y)) \wedge e(y, z) \Rightarrow path(x, z)\}$ . The symbol  $path(\cdot, \cdot)$  encodes the concept of a directed path between two vertices. The single logical statement in  $C$  defines how to derive the paths in a digraph. The keyword *derive* is important, and there are some subtle points to be made about derivation.

Classically, the notion of a derivation is represented by a *consequence operator*, written  $\vdash$ , which maps sets of terms to sets of terms  $\vdash: \mathcal{P}(T_{\Upsilon_C}(\Sigma)) \rightarrow \mathcal{P}(T_{\Upsilon_C}(\Sigma))$ . A consequence operator encapsulates the inference rules of a particular style of logic, and may make use of additional *axioms* to derive terms. In our framework, the set  $C$  is the set of axioms that the consequence operator may use. Given a model  $M$  (i.e., a set of terms),  $M \vdash_C M'$  denotes the set of terms  $M'$  that can be discovered from the terms  $M$  and the axioms  $C$ . A term  $t$  can be derived from a model  $M$  if  $t \in M'$ . We will simply write  $M \vdash t$  to denote that  $t \in M'$ . Notice that the consequence operator generalization does not require terms to be viewed as predicates. For example, given the simple graph  $M_1$  of Example 13-1, we can derive the term  $path(c_1, c_2)$ , but this term does not evaluate to a boolean value. Classical consequence operators, in the sense of Tarski, correspond to *closure operators* and are *extensive*, *isotone*, and *idempotent*[84]. Later, we will discuss the consequence operators of nonmonotonic logics where the isotone property does not hold. The history of mathematical logic is rich and diverse; we will not summarize it here. Instead, we will focus on particular applications and limit our discussion to those applications. For the reader unfamiliar with this area, it suffices to remember these two points: First, consequence operators capture the derivation of terms. Second, terms are not predicates.

Among the properties that can be encoded using  $\Upsilon_C$  and  $C$ , we require at least one property to be defined that characterizes if a model is well-formed. We permit well-formedness to be defined either positively or negatively. A *positive domain* includes the function symbol  $wellform(\cdot)$  in  $\Upsilon_C$ , and a model  $M$  is well-formed if  $\exists x \in T_{\Upsilon_C}(\Sigma), M \vdash_C wellform(x)$ . In another words, a model is well-formed if a term of the form  $wellform(x)$  can be derived for some  $x$ . A *negative domain* is characterized by the function symbol  $malform(\cdot)$  such that a model is well-formed if  $\forall x \in T_{\Upsilon_C}(\Sigma), M \not\vdash_C malform(x)$ . In another words, a model is well-formed if it is not possible to prove  $malform(x)$  for any  $x$ . At first glance it may appear that the positive domains have weaker definitions than negative domains. In fact, this depends on the expressiveness of the underlying logic of  $\vdash$ . For example, if the logic has a “negation” (which is not the usual propositional negation) then we can define  $wellform(x) \Leftrightarrow \forall y \neg malform(y)$  for some arbitrary  $x$ . On the other hand, if the logic is restricted, then the positive domains may be strictly weaker than the negative domains.

A domain captures the set of possible model realizations and provides a mechanism to discern the good models from the bad ones. From this perspective, the set of all metamodels also defines a domain  $D_{meta}$  that characterizes all well-formed metamodels. Let the set  $\mathcal{V}$  be a fixed vocabulary of function symbols and

the sets  $\Sigma$  and  $\Sigma_v$  be two fixed disjoint countably infinite alphabets. Let  $SIG(\mathcal{V}) = \{\Upsilon | \Upsilon : \mathcal{V} \rightarrow \mathbf{Z}_+\}$ , be the set of all partial functions from  $\mathcal{V}$  to the positive integers, i.e., the set of all possible signatures. Finally, let  $\mathcal{F}(\Upsilon, \Upsilon_C)$  be the set of all formulas that can be defined over terms composed from function symbols of  $\Upsilon, \Upsilon_C$  with constants from  $\Sigma$  and variables from  $\Sigma_v$ . These parameters allow us to characterize the set of all domains  $\Delta_{\mathcal{F}}$  that can be defined with a particular style of logic<sup>1</sup>:

$$\Delta_{\mathcal{F}} = \bigcup_{\Upsilon \in SIG(\mathcal{V})} \bigcup_{\Upsilon_C \in SIG(\mathcal{V})} \bigcup_{C \subseteq \mathcal{F}(\Upsilon, \Upsilon_C)} (\Sigma, \Upsilon, \Upsilon_C, C)$$

A *metamodeling language* is a pair  $(D_{meta}, \tau_{meta})$  where  $\tau_{meta} : D_{meta} \rightarrow \Delta_{\mathcal{F}}$  maps metamodels to domains. In [39] we show how the mapping can be constructed for realistic metamodel languages. With this approach, we can extract a precise set of domain concepts and constraints from a metamodel by applying the mapping  $\tau_{meta}$ . Here we overload the notation  $D$  to also represent the set of all well-formed models characterized by the domain  $D$ .

Given these preliminaries, we now turn our attention to the analysis of domains. For example, we might like to know: *Does a domain contain any non-trivial finite models?* It turns out that this fundamental question is difficult to answer for UML-like metamodels. Consider the *UNSAT* metamodel of Figure 39. If a model of *UNSAT* contains anything at all, then it contains an instance of *RootClass*. However, an instance of *RootClass* must contain at least one instance of *ClassA*, which in turn must contain at least one instance of *ClassC*. So far the constraints pose no problem. However, the inheritance operator declares that *ClassC* is a subclass of *ClassA*, so *ClassC* inherits the property that each instance must also contain at least one instance of *ClassC*. This leads to an infinite regress, so there exists no non-trivial finite model of *UNSAT*. This can be seen in Figure 40, which is a finite model that almost adheres to *UNSAT*, except that the instance *instOfC\_2* does not contain another instance of *ClassC*. The degree to which we can reason about metamodels depends on the expressiveness of the constraint logic. We now turn our attention to a well-known decidable subset of first-order logic, Horn Logic.

### Analysis of Nonrecursive Horn Domains

The simplest class of logic we examine is nonrecursive Horn logic[94]. Admittedly, this class is too small for characterizing most realistic domains, but the algorithms for manipulating this logic serve as a foundation for the more expressive logic that we describe in the next section. We begin by recalling some definitions. *Formulas* are built from terms with *variables* and logical *connectives*. There are different approaches for distinguishing variables from constants. One way is to introduce a new alphabet  $\Sigma_v$  that contains variable names such that  $\Sigma \cap \Sigma_v = \emptyset$ . The terms  $T_{\Upsilon_C}(\Sigma)$  are called ground terms, and contain no variables. This

<sup>1</sup>Technically, we should include the property that all  $\Upsilon_C$  signatures contain *wellform*( $\cdot$ ) or *malform*( $\cdot$ ). We have left this out as it unnecessarily complicates the definition of  $\Delta_{\mathcal{F}}$

set is also called the *Herbrand Universe* denoted  $\mathcal{U}_H$ . The set of all terms, with or without variables, is  $T_{\Upsilon_C}(\Sigma \cup \Sigma_v)$ , denoted  $\mathcal{U}_T$ . Finally, the set of all *non-ground* terms is just  $\mathcal{U}_T - \mathcal{U}_H$ . A *substitution*  $\phi$  is term endomorphism  $\phi : \mathcal{U}_T \rightarrow \mathcal{U}_T$  that fixes constants. In another words, if a substitution  $\phi$  is applied to a term, then the substitution can be moved to the inside  $\phi f(t_1, t_2, \dots, t_n) = f(\phi t_1, \phi t_2, \dots, \phi t_n)$ . A substitution does not change constants, only variables, so  $\forall g \in \mathcal{U}_H, \phi(g) = g$ . We say two terms  $s, t \in \mathcal{U}_T$  *unify* if there exists substitutions  $\phi_s, \phi_t$  that make the terms identical  $\phi_s s = \phi_t t$ , and of finite length. (This implies the *occurs check* is performed.) We call the pair  $(\phi_s, \phi_t)$  the unifier of  $s$  and  $t$ . The variables that appear in a term  $t$  are  $vars(t)$ , and the constants are  $const(t)$ .

A *Horn clause* is a formula of the form  $h \Leftarrow t_1, t_2, \dots, t_n$  where  $h$  is called the *head* and  $t_1, \dots, t_n$  are called the *tail* (or body). We write  $T$  to denote the set of all terms in the tail. The head only contains variables that appear in the tail,  $vars(h) \subseteq \bigcup_i vars(t_i)$ . A clause with any empty tail ( $h \Leftarrow$ ) is called a *fact*, and contains no variables. Recall that these clauses will be used *only* to calculate model properties. This is enforced by requiring the heads to use those function symbols that do not encode model structure, i.e. every head  $h = f(t_1, \dots, t_n)$  has  $f \in (\Upsilon_C - \Upsilon)$ . (Proper subterms of  $h$  may use any symbol.) This is similar to restrictions placed on declarative databases[91]. We slightly extend clauses to permit *disequality* constraints. A Horn clause with disequality constraints has the form  $h \Leftarrow t_1, \dots, t_n, (s_1 \neq s'_1), (s_2 \neq s'_2), \dots, (s_m \neq s'_m)$ , where  $s_i, s'_i$  are terms with no new variables  $vars(s_i), vars(s'_i) \subseteq \bigcup_i vars(t_i)$ . We can now define the *meaning* of a Horn clause. The definition we present incorporates the *Closed World Assumption* which assumes all conclusions are derived from a finite initial set of facts (ground terms)  $I$ . Given a set of Horn clauses  $\Theta$ , the operator  $\widehat{F}_\Theta$  is called the *immediate consequence operator*, and is defined as follows:

$$M \widehat{F}_\Theta = M \cup \left\{ \phi(h_\theta) \mid \exists \phi, \theta, \phi(T_\theta) \subseteq M \text{ and } \forall (s_i \neq s'_i)_\theta \in \theta, \phi s_i \neq \phi s'_i \right\}$$

where  $\phi$  is a substitution and  $\theta$  is a clause in  $\Theta$ . It can be proved that  $I \vdash_\Theta I_\infty$  where  $I \widehat{F}_\Theta I_1 \widehat{F}_\Theta \dots \widehat{F}_\Theta I_\infty$ . The new terms derivable from  $I$  can be calculated by applying the immediate consequence operator until no new terms are produced (i.e. the least fixed point). Notice that the disequality constraints force the substitutions to keep certain terms distinct. *Nonrecursive Horn logic* adds the restriction that the clauses of  $\Theta$  can be ordered  $\theta_1, \theta_2, \dots, \theta_k$  such that the head  $h_{\theta_i}$  of clause  $\theta_i$  does not unify with any tail  $t \in T_{\theta_j}$  for all  $j \leq i$ . This is a key restriction; without it, the logic can become undecidable. Consider the recursive axiom  $\Theta = \{f(f(x)) \Leftarrow f(x)\}$ . Then  $\{f(c_1)\} \vdash_\Theta \{f(c_1), f(f(c_1)), \dots, f(f(f(\dots f(c_1) \dots)))\}$  includes an infinite number of distinct terms. Let  $\mathcal{F}_{NH}(\Upsilon, \Upsilon_C)$  be the set of all sets of Horn clauses defined over signatures  $\Upsilon, \Upsilon_C$  with alphabets  $\Sigma, \Sigma_v$ .

We call domains specified with formulas from  $\mathcal{F}_{NH}$  *nonrecursive Horn domains* (abbreviated NHD). The first problem we wish to solve is the membership problem for positive NHDs.

**Definition 14.** The membership problem for positive NHDs: Given a positive NHD  $D$ , does there exists a finite model  $M \subset \mathcal{U}_H(D)$  such that  $M \vdash_C wellform(x)$  for some  $x$ . The notation  $\mathcal{U}_H(D)$  indicates the set

of ground terms defined by the signature  $\Upsilon$  of  $D$ .

The membership problem for positive NHDs is the easiest problem to solve. We will solve it by actually constructing a model  $M$  for which a *wellform*( $\cdot$ ) term can be derived. This is possible because nonrecursive Horn logic has an important property called *monotonicity*: If a model  $M$  derives terms  $M'$ , and another model  $N$  contains  $M$ , then  $N$  must derive at least  $M'$ . In symbols,  $M \subseteq N$  and  $M \vdash_{\Theta} M'$ ,  $N \vdash_{\Theta} N'$ , then  $M' \subseteq N'$ . This property implies that an algorithm only needs to examine the “smallest” models that could derive a *wellform*( $\cdot$ ) term. Our algorithms are similar to those found in logic programming, but with some necessary augmentations. Typically, logic programs are provided with a set of initial facts that form the closed world. Our problem is to figure out the set of facts such that if the logic program were initialized with these facts, then the desired outcome (e.g. deriving a *wellform*( $\cdot$ ) term) would occur. This distinction means that our algorithms cannot rely on the fact that the closed world contains a finite number of ground terms, because these terms are not yet known. It turns out that although there are an infinite number of “small” models, these models can be partitioned into a finite number of equivalence classes; these classes can be exhaustively examined.

We have developed a theorem prover called *FORMULA* (FORmal Modeling Using Logic Analysis) which implements these techniques. Figure 41.a shows a positive NHD of directed graphs, called *CYCLE*, using *FORMULA* syntax. Line 1 declares the two function symbols  $v$  (for vertex) and  $e$  (for edge). The keyword **in** marks these as input symbols, i.e. elements of the signature  $\Upsilon$ . The remaining symbols are used to calculate properties of an input model, and are marked **priv** for private symbols, i.e. elements of  $\Upsilon_C$ . The theorem prover will never return a model that contains a private symbol. Well-formed models of the *CYCLE* domain must contain either a directed 3-cycle or 4-cycle. Lines 7,8 define the properties of 3-cycles and 4-cycles based on the properties of 3-paths and 4-paths. For example, a 3-cycle exists if there is a 3-path on vertices  $X, Y, Z$  and there is an edge from  $Z$  to  $X$ . (Note that the variable names are local to each clause.) Notice the use of disequality constraints in the definition of 3-paths and 4-paths in Lines 10-13. These constraints ensure that the paths contain unique vertices. Finally, Lines 16-18 define the derivation of *wellform*( $\cdot$ ) terms.

The first step towards generating a well-formed model is to determine the derivation steps that lead to *wellform*( $\cdot$ ) terms. This is done via an augmented form of *backwards chaining*. First, some definitions are necessary. We call two terms  $s, t$  *isomorphic* if there exists a substitution  $\phi$  such that  $\phi$  is a term monomorphism (one-to-one map),  $\phi s = t$ , and  $\phi^{-1}$  is also a substitution. Clearly it holds that  $s = \phi^{-1} t$ . Given a set of terms  $T$ , let  $I_T$  be an equivalence relation on terms such that  $(s, t) \in I_T$  if  $s$  and  $t$  are isomorphic. It is easy to see that  $I_T$  is an equivalence relation, because composition of monomorphisms yields another monomorphism. A *goal term*  $g$  is a term (with variables), and a solution  $M$  is a set of ground terms such  $M \vdash_{\Theta} M'$  and  $\exists \phi, \exists t \in M' (\phi g = t)$ . In another words, a solution is a model that derives a ground term unifying with the goal. The terms derived from the solution  $M$  are all ground terms, so,

---

```

1: in  $v$  arity 1; in  $e$  arity 2;
2: priv  $path3$  arity 3; priv  $path4$  arity 4;
3: priv  $cycle3$  arity 3; priv  $cycle4$  arity 4;
4: priv  $useless$  arity 1; priv  $useless2$  arity 1;
5: priv  $wellform$  arity 1;
6:
7:  $cycle3(X,Y,Z) \leq path3(X,Y,Z), e(Z,X)$ ;
8:  $cycle4(X,Y,Z,W) \leq path4(X,Y,Z,W), e(W,X)$ ;
9:
10:  $path3(X,Y,Z) \leq e(X,Y), e(Y,Z),$ 
11:  $!=(X,Y), !=(X,Z), !=(Z,Y)$ ;
12:  $path4(X,Y,Z,W) \leq path3(X,Y,Z), e(Z,W),$ 
13:  $!=(W,X), !=(W,Y), !=(Y,Z)$ ;
14:
15:  $useless(X) \leq useless2(X)$ ;
16:  $wellform(X) \leq useless(X), e(X,Y)$ ;
17:  $wellform(X) \leq cycle3(X,Y,Z)$ ;
18:  $wellform(X) \leq cycle4(X,Y,Z,W)$ ;

```

---

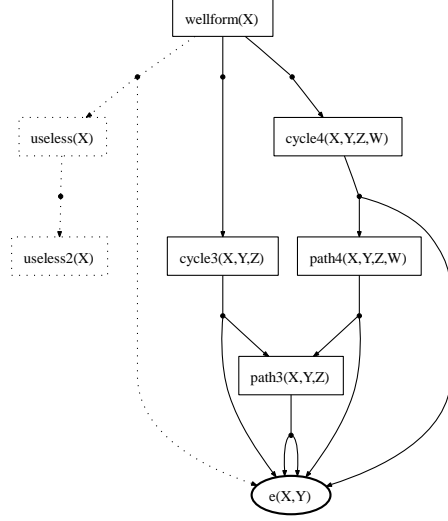


Figure 41. (Left) CYCLE: a positive NHD in FORMULA syntax. (Right) Backwards chaining graph generated from goal  $wellform(X)$ .

without lost of generality, it can be assumed that the unifier is  $(\phi, id_{\mathcal{U}_T})$ . Let  $terms(D)$  be the union of all terms in the domain definition, (i.e. union of all heads and tails). Given a set of goals  $G$  and a domain  $D$ , let  $[t]$  be the equivalence class of  $t$  in  $I_{terms(D) \cup G}$ . A *backwards chaining graph*  $B(G)$  over a set of goal terms  $G$  is defined inductively as follows:

1. For each  $g \in G$ ,  $[g] \in V_{B(G)}$ .
2. For all clauses  $h_{\theta_i} \Leftarrow t_1, \dots, t_m$  in  $\Theta$  such that  $[h_{\theta_i}] \in V_{B(G)}$ , then  $[t_i]_{1 \leq i \leq m} \in V_{B(G)}$  and there exists a directed “AND” edge  $([h_{\theta_i}], \{[t_i]\}_{1 \leq i \leq m}) \in E_{B(G)}$ .
3. For all clauses  $h_{\theta_i} \Leftarrow t_1, \dots, t_m$  in  $\Theta$  such that  $h_{\theta_i}$  unifies with some tail  $t_{\theta_j}$  and  $[t_{\theta_j}] \in V_{B(G)}$  then  $[h_{\theta_i}] \in V_{B(G)}$  and there exists a directed edge  $([t_{\theta_j}], [h_{\theta_i}]) \in E_{B(G)}$ .

The right-hand side of Figure 41 shows the backwards chaining graph generated by the single goal term  $wellform(X)$ . There are significantly fewer vertices in the graph than terms in the domain definition, because many terms are isomorphic.  $B(G)$  has several properties, though we will not prove them here.  $B(G)$  is finite because the domain  $D$  has a finite number of clauses, and  $B(G)$  is acyclic because  $D$  is nonrecursive. Unlike typical backwards chaining, the sinks in the graph are not ground terms, because they are none, but are terms with function symbols completely in  $\Upsilon$ . Any sinks without this property are pruned from the graph. For example, the  $useless(\cdot)$  and  $useless2(\cdot)$  terms are pruned, because there are no ways to derive these terms from  $\Upsilon$  terms. The vertices and edges in dotted lines are the pruned part of the graph. If a solution exists then there must be a directed path from every  $[g]_{g \in G}$  vertex to a non-pruned sink using non-pruned



edges. This holds because a solution contains only ground terms, which impose stronger restrictions on the unifier morphisms, than those imposed by the construction of  $B(G)$ .

The backwards chaining graph captures the various paths from the goal to possible solutions, and each path must be walked until a solution is found or it is confirmed that no solution exists. A path can be “unrolled” one at a time (as in SLD resolution[97]), or a tree can be constructed capturing every possible walk. We choose the latter in order to support other uses of FORMULA. Figure 42 shows the unrolling of the Figure 41 into a *solution tree*. The tree has a root with a single AND edge having an endpoint on each goal term  $g$ . Every goal term  $g$  attached to the root receives an edge for each  $v \in B(G)$  such that  $g$  unifies with  $v$ . For example, Figure 42 shows the vertex  $wellform(V0)$  connected to the  $wellform(V1)$ . This edge indicates that  $wellform(V0)$  unifies with  $wellform(V1)$ . The tree construction algorithm always *standardizes apart* unifying terms by instantiating them with unique variables.  $wellform(V1)$  has two distinct paths in the backwards chaining graph, and each of these are unrolled into two subtrees of the  $wellform(V1)$  vertex. If a clause has disequality constraints, then these appear as constraints on the edges in the solution tree.

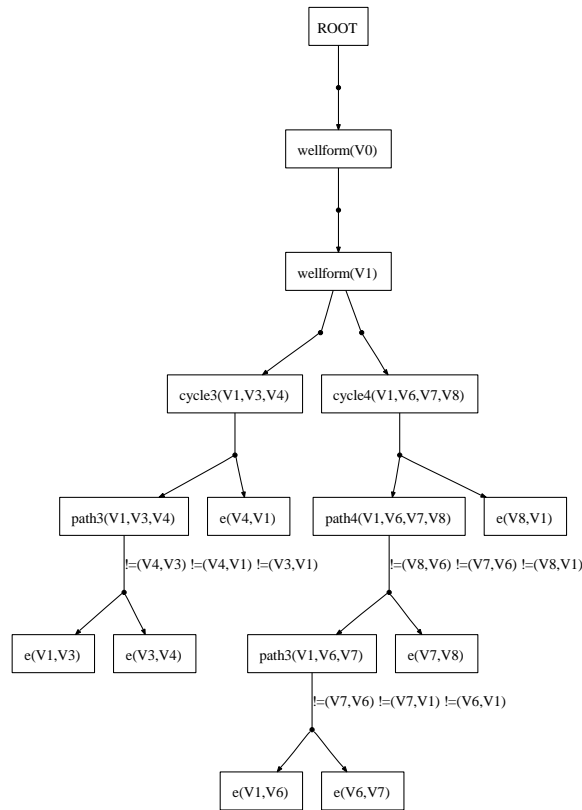


Figure 42. Solution tree generated from backwards chaining graph of Figure 41

The solution tree is viewed as a constraint system over terms. As the tree is walked, equations concerning terms are collected. A unification of terms  $s, t$  can be converted to a system of equations over variables. For example  $g(X, Y)$  unifies with  $g(Z, Z)$  if  $X = Y = Z$ . Clearly any unifier  $(\phi_s, \phi_t)$  must have  $\phi_s(X) = \phi_s(Y) = \phi_t(Z)$ . The correct equations are calculated by an inductive procedure as motivated in [97]. The constraint system is represented as a forest of union-find trees; a unification  $s, t$  yields a set of equations  $\{s_i = t_i\}$ , which is converted to operations on the forest: for each equation  $s_i = t_i$  perform  $join(find(s_i), find(t_i))$  where the  $find(x)$  operation creates the vertex labeled  $x$  if  $x$  does not already exist. For example, there is one non-trivial union-find tree in Figure 43 resulting from the unification of  $wellform(V0)$  with  $wellform(V1)$ , which joins  $V0$  and  $V1$ . As terms are added to the forest, so are their subterms. Dependency links are maintained between vertices, where a term  $t$  is dependent on a term  $s$  if  $s$  is a subterm of  $t$ . An operation fails if the dependency edges form a cycle, essentially indicating that a multi-step unification fails. The dependency edges in Figure 43 are gray and labeled “depends”. Disequality constraints are implemented as “Not equal” edges between vertices. Notice that all terms in the same union-find tree share the same constraints and dependencies. As trees are joined, all the constraints are moved up to the root. For example, in Figure 43 all constraint edges terminate on the *JOIN* vertex. Thus, a disequality constraint fails if a vertex is unequal to itself, or a join operation moves the source and destination of a disequality edge onto the same join vertex. As the algorithm walks the solution tree, it performs operations on the constraint system. As soon as the constraint system becomes inconsistent, the algorithm restarts on an unexplored combination of subtrees. FORMULA maintains all possible restart configurations, and only fails after all restarts have been tried. Let  $W$  be the sequence of vertices visited in a walk of the solution tree. Then  $CS(W)$  is the constraint system produced by that walk.

After a consistent walk  $W$  has been found, the constraint system  $CS(W)$  can be converted into a set of ground terms. Notice that the sinks (ignoring disequality edges) in the constraint system are those terms for which all other terms are dependent. In fact, our construction guarantees that the sinks are just variables or ground terms. Let  $sinks(CS)$  be the sinks of a consistent constraint system  $CS$  defined as follows: A union-find tree  $T \in CS$  is a *sink tree* if the root has no outgoing edges, or only has outgoing disequality edges. If no leaves of the sink tree are ground, then pick a leaf and place it in  $sinks(CS)$ . Choose any substitution  $\phi_{min}$  such that  $\phi_{min}(X) \mapsto c_X \in (\Sigma - const(D))$ , where  $c_X$  is a unique constant not appearing anywhere in the domain definition. If a variable  $X$  is in the same union-find tree as a ground term  $t_g$ , then  $\phi_{min}(X) \mapsto t_g$ . The values of all other variables are calculated transitively to form the full substitution  $\phi_{sol}$ . Finally, the candidate solution  $M_W$  for walk  $W$  is

$$M_W = \left( \bigcup_{v \in W} \phi_{sol}(t_v) \right) \cap T_{\Upsilon}(\Sigma)$$

where  $t_v$  is the term of a vertex  $v$  in the walk  $W$  of the solution tree.  $M_W$  is a *proper solution* if no model terms of the form  $f(t_1, \dots, t_n)$ , where  $f \in \Upsilon$ , are removed by the intersection with  $T_{\Upsilon}(\Sigma)$ . Such a term would

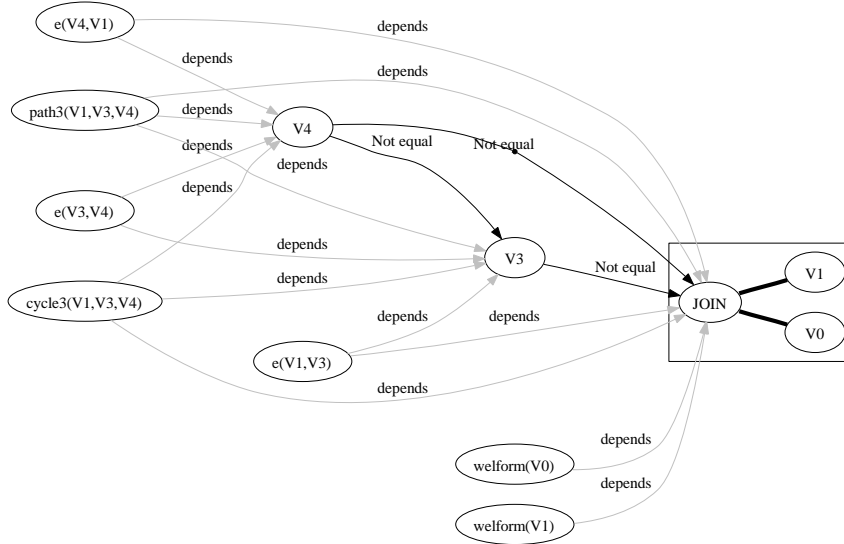


Figure 43. Constraint system shown as a forest of union-find trees.

be thrown out if it contains a subterm  $t_i$  built from symbols of  $\Upsilon_C - \Upsilon$ . In this case, the candidate solution is discarded and another walk through the solution tree is attempted. Applying this algorithm to the constraint system of Figure 43 gives  $sinks(CS) = \{V0, V3, V4\}$ . Let  $\phi_{min}(V0) \mapsto c_0, \phi_{min}(V3) \mapsto c_1, \phi_{min}(V4) \mapsto c_2$ . By transitivity,  $\phi_{sol}(V1) \mapsto c_0$ , and all variables are accounted for. Applying  $\phi_{sol}$  to each vertex on the left-hand walk of Figure 42 gives a candidate model  $M_W = \{e(c_0, c_1), e(c_1, c_2), e(c_2, c_0)\}$ , which is a correctly constructed 3-cycle. It is not difficult to prove:

**Theorem 15.** *A positive NHD has a non-trivial finite model iff there exists a walk  $W$  such that  $CS(W)$  is consistent and the candidate model  $M_W$  is proper.*

These algorithms can also be used to construct well-formed models with particular *embeddings*. Let  $\gamma : \mathcal{U}_H \mapsto \mathcal{U}_H$  be a term endomorphism (i.e. a homomorphism over model terms). A model  $M'$  can be *embedded* into a model  $M$ , written  $M' \leq M$ , if there exists a one-to-one term endomorphism (i.e. a monomorphism) such that  $\gamma(M') \subseteq M$ . Constructive techniques that can produce embeddings allow us to sketch a model that might be malformed, but produce a well-formed version that still contains the original model. This can be quite useful for users who do not understand all of the particular constraints of a modeling language, and would like the computer to correct mistakes. Consider the top-left graph of Figure 44. This *star* graph ( $S_4$ ) is malformed with respect to the *CYCLE* domain, because it contains neither a 3-cycle nor 4-cycle. However, with a slight modification to the algorithms above, a new model can be built that is well-formed and contains an embedding of the *star* graph. Let  $D$  be a domain and let an *input model*

$M_I$  be a finite subset of model terms  $T_{\Upsilon}(\Sigma)$ . Choose any one-to-one map  $\alpha : \Sigma \rightarrow \Sigma_v$  that uniquely relates

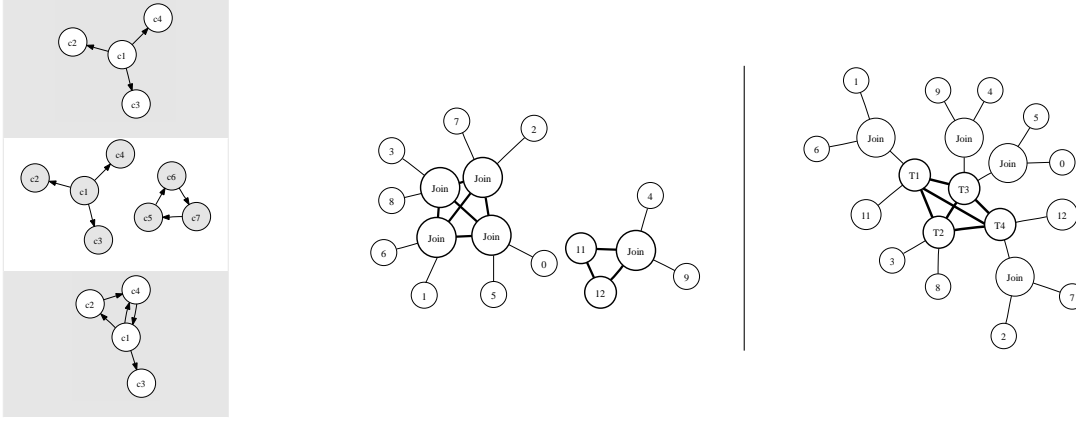


Figure 44. (Left) A malformed input model (top), a well-formed embedding, and a minimal embedding (bottom). (Middle) Initial constraint system showing only sink trees and disequality constraints. (Right) Minimized constraint system.

constants to variables in  $\Sigma_v$ . Clearly  $\alpha$  induces a monomorphism  $\phi_\alpha : T_{\Upsilon}(\Sigma) \rightarrow T_{\Upsilon}(\Sigma_v)$  from terms without variables to terms that only have variables. We will use this monomorphism to encode the input model as a Horn clause. Pick any function symbol  $f \notin \Upsilon_C$  and add it  $\Upsilon_C$  with arity  $|const(M_I)|$ , i.e. the arity of  $f$  is equal to the number of constants in the input model  $M_I$ . Add the following clause  $\theta_{M_I}$  to  $D$ :

$$\theta_{M_I} = f(\alpha(c_1), \alpha(c_2), \dots, \alpha(c_n)) \Leftarrow \bigwedge_{t_m \in M_I} \phi_\alpha(t_m) \bigwedge_{i \neq j} (\alpha(c_i) \neq \alpha(c_j))$$

where  $1 \leq i, j \leq n = |const(M_I)|$ . Recall from the previous algorithms, that a solution is constructed by defining a substitution  $\phi_{sol}$  that is determined by the sink variables  $sinks(CS(W))$ . Consider any solution to any goal set  $G$  where  $f(\alpha(c_1), \dots, \alpha(c_n)) \in G$ . By construction, the restriction of  $\phi_{sol}$  to  $sinks(CS(W))$  yields a one-to-one map. In the construction above, all pairs of variables induced by  $M_I$  have disequality constraints, so  $\alpha(const(M_I)) \subseteq sinks(CS(W))$  for any consistent walk  $W^2$ . Therefore, the restriction of any  $\phi_{sol}$  to the terms  $T_{\Upsilon}(\alpha(const(M_I)))$  must be a monomorphism. Thus,  $\gamma = (\phi_{sol} \circ \phi_\alpha)$  gives the embedding of  $M_I$  in any proper solution  $M_W$  for a consistent walk  $W$ .

**Theorem 16.** *Given an input model  $M_I$  and a positive NHD  $D$ , augmented with  $f$  and  $\theta_{M_I}$ . Any proper solution to a goal set  $G$ , where  $f(\alpha(c_1), \dots, \alpha(c_n)) \in G$ , contains an embedding of  $M_I$ .*

In particular, let the goal set  $G = \{f(\alpha(c_1), \dots, \alpha(c_n)), wellform(X)\}$ , where the variable  $X$  is not in the image of  $\alpha$ , then any solution to  $G$  contains  $M_I$  and is well-formed. The middle-left graph of Figure 44 shows FORMULA's construction of a well-formed version of the star graph in the *CYCLE* domain.

The default embedding produced by FORMULA is not particularly elegant. It contains a star juxtaposed with a 3-cycle. This solution was constructed because  $\phi_{sol}$  assigns a unique constant to each sink variable,

<sup>2</sup>This is a slight simplification. There will be some representative sink variable for each variable in the image of  $\alpha$ .

yielding a *maximal solution* with respect to the number of constants. A smaller solution can be found by manipulating the final constraint system  $CS(W)$  so that the number of sink variables are reduced. This can be accomplished by merging sink trees, which is legal if the trees do not have disequality constraints between them. The middle graph of Figure 44 shows the sink trees of the constraint system after producing the middle-left embedding. The root of each tree is in bold, and disequality constraints between trees are shown as bold edges. These are the only types of edges between trees, because sink trees do not have dependency edges between them. A *minimal solution* can be formed by partitioning the root vertices into a minimal number of independent sets. This is a computationally hard optimization problem related to the *independent set problem*. The right side of Figure 44 shows the optimized constraint system, which contains only four trees (and four sink variables). The roots of the optimized constraint system form a clique, therefore no further optimization is possible. The bottom-left graph shows the optimized solution generated by FORMULA, wherein the star and 3-cycle have been merged in an ideal fashion. Note that this process yields a minimal, but not necessarily *minimum* model. Finding a minimum model requires minimizing all possible consistent walks of the solution tree.

### Extensions, Tools, and Future Directions

We have shown that the constructive reasoning of UML-like metamodels is a rich area of study, both theoretically and algorithmically. In the interest of space we have used directed graphs as our toy example. However, these techniques can be applied to much more complicated metamodels, and with practical applications: *Metamodel composition* is the process of constructing new domain-specific languages by combining existing metamodels. Two metamodels,  $mm_1$  and  $mm_2$ , can be syntactically combined with an operator  $\circ$ , such as class equivalence[98], and the syntactic composition can be converted into a domain  $D_{comp} = \tau_{meta}(mm_1 \circ mm_2)$ . The membership problem for the domain can then be solved, thereby deciding if the metamodel composition is *semantically* meaningful. Other problems, like the construction of embeddings, correspond to the automatic construction of useful models that satisfy the domain constraints. Model transformations can also be incorporated into our framework, and then constructive techniques can be used to prove that the transformation always produces well-formed output models from well-formed input models. This is the weakest form of correctness one could imagine, but checking these properties has remained mostly open. There is already precedent for the use of Prolog engines to transform a particular input model  $M_I$  to an output model  $M_O$ , as is done by Viatra2[73]. A particular input/output pair  $(M_I, M_O)$  can then be compared to check for mutual consistency (e.g. via bisimulation). However, checking properties of the overall transformation is more difficult, though our approach can handle it as long as the transformation is restricted to an appropriate class of logic. The verification goal resembles Hoare's notion of a *verifying compiler*[48].

This brings us to questions of expressiveness. How expressive is Horn logic and how far can it be taken? This question has driven our development of FORMULA, which we now summarize. Positive NHDs are not particularly expressive, but they are an essential starting point for developing constructive techniques for more expressive domains. The next step in the progression is to solve the membership problem for *negative* NHDs. Recall that negative domains characterize the malformed models with the symbol  $malform(\cdot)$ , and a model  $M$  is wellformed if  $\forall x, M \not\vdash_C malform(x)$ . Negative NHDs can express domains not representable by positive NHDs, because of the universal quantification over  $malform(x)$ . Notice that the solution tree for a goal  $G = \{malform(x)\}$  contains all equivalence classes of malformed models, and the *malformedness* property is monotonic in models. With these observations, the membership problem can be solved by repeating this procedure: Prune all leaves in  $B(\{malform(x)\})$ , except for one symbol  $f \in \Upsilon$ . If  $malform(x)$  can be proved on the corresponding pruned solution tree, then by monotonicity, no wellformed model can contain a term unifying with  $f$ . If  $malform(x)$  cannot be proved, then a wellformed model  $M = \{f(\cdot, \dots, \cdot)\}$  has been found. This test is repeated (at least once) for each  $f \in \Upsilon$ ; due to unification issues, it may be repeated multiple times for non-unifying  $f$ -terms. This procedure is also implemented in FORMULA.

A further increase in expressiveness can be obtained by extending the Horn logic so that a tail can contain a “negated” term  $\neg t_i$ . (For example, the *UNSAT* domain (Figure 39) can be defined with this extension.) Loosely, a negated term is a constraint requiring that a solution  $M \not\vdash_C t_i$ . Theoretically, this extension approximates the power of full first order logic, but remains decidable (under additional restrictions on its use). It turns out that this simple extension corresponds to a *nonmonotonic* logic, and has deep theoretical and algorithmic repercussions. Our major challenge has been the development of constructive techniques for domains written in Horn logic extended with negation. These techniques are also implemented in FORMULA, and extend existing work on nonmonotonic inference[94][93] to deal with the particulars of UML-like metamodels. Theoretically, these extensions must be handled carefully in order to maintain the soundness and completeness of the theorem prover. Algorithmically, our approach combines the aforementioned algorithms with state-of-the-art SAT solvers to construct models. In conclusion, a reasonable level of expressiveness can be obtained.

A common criticism of theorem proving is the requirement of the user to understand the underlying mathematics. We have addressed this issue by developing an automated conversion from metamodels to domain definitions. This approach is described in [39], and supports metamodeling in the well-known Generic Modeling Environment (GME) toolsuite[43]. Furthermore, because the theorem prover is constructive, the results of the prover are concrete models that can be automatically imported back into the GME modeling environment. This closes the loop, providing constructive reasoning about models and metamodels without leaving the comfort of the modeling toolsuite (for most of the common queries). Our future work is to apply

these techniques to analyze model transformations, including those specified with the Graph Rewriting and Transformation (GReAT) language[72] that is also part of the GME toolsuite.

## CHAPTER V

### STRUCTURAL INTERFACES FOR ADAPTIVE SYSTEMS

#### Introduction

Frameworks based on *domain-specific modeling languages* (DSMLs)[43] and *platform-based design*[99] have a history of success in the embedded software realm. DSMLs/platforms capture, among many things, the computation and communication mechanisms of a class of systems. A well-defined class of systems can then be used to disambiguate software specifications. For example, a typical software specification might contain:

*Refresh the client's data from the server every 1s.*

Without definitions to disambiguate this specification, the system is ill-defined: Does the client contact the server after exactly one second has passed or at least one second? How long is a second? Is it an approximate second with respect to the discrete clock ticks of the client? Perhaps a global continuous timeline is assumed. The answers to these questions influence implementation decisions (e.g. “Do we need a time synchronization algorithm?”), and so they indirectly determine whether or not the final implementation is correct. Loosely speaking, a *DSML* or *platform* is a set of parameters that removes the ambiguities from system specifications. An engineer's first task in the design process is to select the appropriate DSML/platform; only then can specification begin. (We shall use the term *DSML* from hereon, though *platform* can be used interchangeably.)

Besides precision, there are other major advantages to the DSML approach: Each DSML provides a rich set of formal methods, simulation engines, verification tools, and code generators. By choosing a DSML, the engineer inherits a tool set tailored to the problem domain. Also, DSMLs can be ordered by the degree to which details are abstracted away. Highly abstract domains are useful for system specification, while less abstract domains are closer to the final implementation. A system is initially specified with a highly abstract domain, and then incrementally migrated to less abstract domains, via *model transformations* or *platform mappings*, until a final implementation is produced. We shall refer to any description of a system in the context of a DSML as a *model*, regardless of the degree of abstraction.

The typical software life-cycle begins with an abstract model  $M_n$  belonging to an abstract DSML  $X_n$ . This abstract model is analyzed and simulated, usually to check functional requirements. A model transformation  $[[ ]_{n-1}^n$  maps models from  $X_n$  to the less abstract DSML  $X_{n-1}$ , where  $M_{n-1} \doteq [[M_n]]_{n-1}^n$  is a more concrete model. This process repeats until an implementation model  $M_0$  is produced. Finally, code is generated from  $M_0$  into a language like  $C$ , the result of which is compiled and deployed.

The model-generate-deploy (MGD) life-cycle works well for a number of embedded systems domains,



but not all. For example, the topology of an ad hoc sensor network changes. Adaptive control systems adiabatically evolve essential control parameters. Modal and adaptive software systems switch their running behavior between a number of predefined scenarios. Interestingly, we can still view these systems through a model-based lens in the following way: A snapshot of the system at a point in time corresponds to some high-level model, but not necessarily the initial high-level model of the system. Thus, a system transitions through a number configurations, each of which is concisely described by some high-level model belonging to the DSML.

We combine our model-based view with ideas from adaptive software systems[100] to create a reusable mechanism for implementing adaptive systems. First, notice that a single “pass” through the MGD tool flow yields an implementation from the perspective of a single model. We shall denote a single pass by the composite map  $\llbracket \cdot \rrbracket_0^n = \llbracket \cdot \rrbracket_0^1 \circ \llbracket \cdot \rrbracket_1^2 \circ \dots \circ \llbracket \cdot \rrbracket_{n-1}^n$ . Thus  $\llbracket M \rrbracket_0^n$  denotes the system generated from model  $M$  by transforming it through  $n$  levels of abstraction plus a final transformation to code (i.e. the zeroth level). Our goal is to compress the entire MGD tool flow so that it fits onto an embedded platform, allowing the embedded system to dynamically calculate  $\llbracket M \rrbracket_0^n$  for any model provided to it. This provides a substrate for adaptation: A *controller*  $C$  (called the *adaptor*) monitors the state of the embedded platform  $P$ , and decides when the platform (called the *adaptee*) should modify its implementation. When  $C$  decides to adapt the platform, it sends a new model  $M'$  to  $P$ , and  $P$  calculates  $\llbracket M' \rrbracket_0^n$  to determine its new implementation. In this way, existing tools and specification techniques can be reused to implement adaptation.

Our discussion has oversimplified the problem somewhat. Figure 45 shows a more complete picture of an embedded platform extended with adaptive capabilities. We shall describe this system in terms of *components*[101], which utilize the object oriented concepts of *encapsulation* and *interface*[102]. A component is a black box that encapsulates functionality, except for a number of exposed interfaces. From the outside of a component, an interface provides a set of *ports* for pushing data into the system and extracting data from the system. For example, the right side of Figure 45 shows the *Data\_Interface* as a white box with a dashed border. Each port of the interface has a name and a direction (given by an arrow). Ports that lay outside of the shaded component boundary are exposed, and arrows that point towards (away) from the component consume (produce) data with respect to the external environment. Thus, data may be pushed into the component via the *Push-Token-In* port or extracted from the component via the *Get-Token-Out* port. Interfaces also expose ports on the inside of the component. From the inside of the component, the *Get-Token-In* and *Num-Tokens-In* ports provide access to the data acquired through external interactions. An interface may perform non-trivial mediation between the external and internal sides of the interface. For example, the *Data\_Interface* might manage a queue that stores pushed tokens.

Interfaces mediate interactions for a number of purposes, e.g. causality[22] and timing[103] interfaces were recently presented. In this chapter we introduce a novel type of interface, called a *structural interface*,

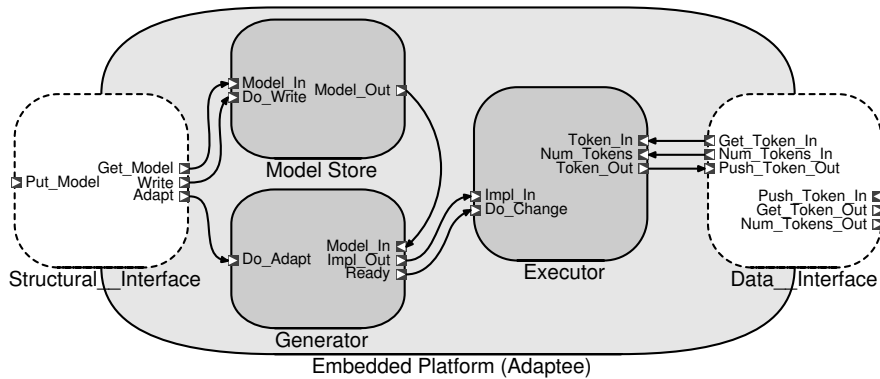


Figure 45. Interfaces and subcomponents of an adaptive component.

for mediating the exchange of model structure between components. The structural interface is an essential part of our adaptive substrate, as shown on the left-hand side of Figure 45. The interface accepts arbitrary model structure through the exposed *Put\_Model* port. On the inside, the interface passes only *well-formed models*, via the *Get\_Model* port, to a *Model Store* subcomponent that stores the model. It also provides internal notification signals for triggering an overwrite of the store (*Write*) and generation of a new implementation (*Adapt*). The *Generator* subcomponent implements the generation facility. Finally, the *Executor* subcomponent executes the current implementation and accesses the data interface. These interfaces and subcomponents implement the map  $\llbracket \rrbracket_0^n$  using well-studied design principles.

The structural interface guards the adaptive component from dangerous interactions. For example, feeding the component a meaningless model would destroy its functionality. The interface protects the component by transforming a model  $M$ , put in from the outside of the interface, into a well-formed model  $M'$ , given to the inside of the component. This is a non-trivial form of mediation provided by the structural interface. Section V.2 introduces the formal foundations of DSML structural semantics presented last year[39]. This formalization is key to generating structural interfaces from DSMLs specified using *metamodels*. The mediation performed by a structural interface depends on the intended style of adaptation. In this paper we develop the concept of a structural interface. We proceed as follows: In Section V.3 we show that interfaces can be denotationally characterized with a general trace semantics, and we describe an important class of interfaces implementing scenario-based adaptation. Section V.4 proves that a DSML has a unique interface with respect to this scenario-based adaptation. Additionally, we show how to generate the interface for structural semantics defined with a nonmonotonic Horn logic. We conclude in Section V.5.

## Structural Semantics

The work presented here relies heavily on our earlier presentation of the structural semantics of DSMLs. The structural semantics provides the basic mathematical framework for defining the objects that pass through structural interfaces. Traditional programming languages are characterized by the *syntax* and the *semantics* of the language. Historically, syntax has been viewed from the perspective of *regular* or *context-free* languages. Interestingly, most languages are neither regular nor context-free because a number of common constructs cannot be represented with these formalisms. For example, the language  $\mathcal{L} = \{wcv \mid w \in (a|b)^*\}$  is neither regular nor context-free[34]. This language corresponds to the restriction that an identifier  $w$  must be declared before its use. (The first  $w$  is the declaration,  $c$  is some code, and the second  $w$  is the use of the identifier.) This fact has not bothered designers of traditional languages, because more complex constraints can be checked during later phases of the compiler. For example, type-systems manipulate objects that are more complex than finite strings. Thus, many complex constraints can be pushed into the type-system.

However, this piecemeal approach to structure is not effective for model-based design. Figure 46 shows a prototypical model of an embedded system. Semantically, the boxes might represent  $n$ -ary maps, concurrent actors, or sequential functions. The arrows might be composition of maps, queues, or function calls. Structurally, the model is akin to a directed graph, but far more complicated. Vertices are compound objects with multiple labeled connection points. Vertices and edges are typed and can contain data members (attributes). DSMLs also utilize a number of relational concepts beyond graph-like edges, including *containment relations*, *aspect membership*, and *hierarchy-crossing edges*. Furthermore, many DSMLs necessarily impose non-trivial constraints on the arrangement of structure. For example, if we interpret the arrows of Figure 46 as instantaneous data dependencies, then a cycle in the model implies deadlock. From this perspective, Figure 46 is a structurally malformed model. Constraints such as these cannot be captured with context-free languages, or have cumbersome translations into context-free languages. A piecemeal approach is also insufficient, because this does not support sound reasoning about model languages and model transformations.

In [39] we presented an approach to formalizing the *structural semantics* of arbitrary DSMLs. We use this as a foundation for structural interfaces. A *domain* is the set of all well-formed model structures that belong

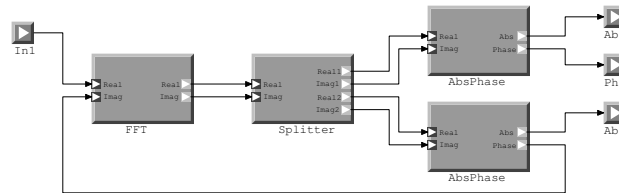


Figure 46. An example of model structure.

to a DSML. An arbitrary domain is characterized by three pieces of information: **(1)** The set  $\Upsilon$  of ingredients for building structure. **(2)** The set  $R_\Upsilon$  of all structures that can be built from the basic ingredients. **(3)** A set of constraints  $C$  that defines the subset of  $R_\Upsilon$  containing only well-formed models. We call the elements of  $R_\Upsilon$  *model realizations*, which may or may not be well-formed. Notice that traditional language syntax is also characterized by this same information. For example, take  $\Upsilon = \Sigma$  to be a finite alphabet. Then  $R_\Upsilon = \Sigma^*$  is the Kleene closure of  $\Sigma$ , and  $C = A$  is a deterministic finite automaton or pushdown automaton that accepts only well-formed strings. Thus, our approach differs by how this information is specified.

Instead of a finite alphabet as the basic ingredient, we use a *signature* of an algebra and a countably infinite set of constants  $\Sigma$ . The details of the signature depend on the DSML. Let  $\mathcal{V}$  be a countably infinite set of *function symbols*, then a signature  $\Upsilon$  is a partial map  $\Upsilon : \mathcal{V} \rightarrow \mathbb{Z}_+$  that assigns an arity to each function symbol in  $\mathbf{dom} \Upsilon$ . We can also write a signature as a set enumerating the symbol-arity pairs. For example, a simple domain of directed graphs (digraphs) might have  $\Upsilon = \{(v, 1), (e, 2)\}$ , where the function symbol  $v$  takes one argument and the function symbol  $e$  takes two arguments. Informally, a *term* is an arbitrarily deep mixing of function symbols and constants that respects arities. A model realization is a set of *terms*. For example, a digraph with vertices  $c_1$  and  $c_2$  and edge  $(c_1, c_2)$  could be represented as  $R = \{v(c_1), v(c_2), e(c_1, c_2)\}$ . The constants of  $\Sigma$  (e.g.  $c_1, c_2$ ) stand for distinguishable elements of a model, and the function symbols endow these elements with information. Thus, by writing  $v(c_1)$  we have denoted that  $c_1$  stands for a vertex. The  $n$ -ary relations of a domain are easily captured by selecting the correct arities. For example, given a model realization  $R$ , the vertex set is  $V = \{x | v(x) \in R\}$  and the edge set (a binary relation) is  $E = \{(x, y) | e(x, y) \in R\}$ . Notice that the values returned by  $v(\cdot)$  or  $e(\cdot, \cdot)$  are not explicitly defined. Formally, these functions are implicitly defined by the *term algebra*[38] over  $\Upsilon$  generated by  $\Sigma$ . This algebra is written  $T_\Upsilon(\Sigma)$  and has the following properties:

1.  $\Sigma$  *generates* the algebra:  $T_\Upsilon(\Sigma) = \langle \Sigma \rangle$ .
2. Functions never return a member of  $\Sigma$ :  $\forall f \in \mathbf{dom} \Upsilon, \mathbf{im} f \cap \Sigma = \emptyset$ .
3. If applications of  $f$  and  $g$  yield the same value, then  $f$  and  $g$  are the same function symbols applied to the same arguments:  $f(t_1, \dots, t_{\Upsilon(f)}) = g(s_1, \dots, s_{\Upsilon(g)}) \Rightarrow (f = g, t_i = s_i)$ .

Formally, a term is a member of the term algebra, and every term can be uniquely identified by an application of functions and constants. This *unique readability* property means that the actual values of the functions are irrelevant and every term algebra  $T_\Upsilon(\Sigma)$  is isomorphic to every other such algebra. With this in mind, the set of all model realizations is the set of all possible sets of terms:  $R_\Upsilon = \mathcal{P}(T_\Upsilon(\Sigma))$ .

The term algebra allows all function symbols and constants to be combined in every conceivable way. This makes it an ideal construction, because all well-formed models are guaranteed to be in  $R_\Upsilon$  regardless of how we intend to encode models. There is also a downside. There are many members of  $R_\Upsilon$  that use the function symbols contrarily to our intentions. For example,  $R = \{v(e(c_1, c_2))\}$  is also a member of the term

algebra for the digraph domain, but it does not correspond to a digraph. Thus, we need a set of constraints over  $R_{\Upsilon}$  that determines the subset of  $R_{\Upsilon}$  containing all well-formed models.

Choosing a constraint language is essential, because it can make or break the decidability of structural properties. An elegant choice would utilize our encoding of models as terms. Fortunately, formal logic provides a fully compatible framework via *consequence operators*. A consequence operator  $\vdash_{\Theta}$  is a mapping from sets of terms to sets of terms  $\vdash_{\Theta}: \mathcal{P}(T_{\Upsilon}(\Sigma)) \rightarrow \mathcal{P}(T_{\Upsilon}(\Sigma))$  where  $\Theta$  is a set of *axioms* and  $R \vdash_{\Theta} R'$  denotes the set of all terms  $R'$  that can be derived from  $R$  using the axioms  $\Theta$ . We will write  $R \vdash_{\Theta} t$  to denote that a term  $t \in R'$  can be derived from  $R$ . We use consequence operators in the following way: Call a signature  $\Upsilon'$  an *extension* of  $\Upsilon$  if  $\Upsilon'$  is also a signature and  $\Upsilon \subseteq \Upsilon'$ . Constraints are given by a pair  $(C, \Upsilon_C)$  where  $C$  is a set of axioms and  $\Upsilon_C$  is an extension of  $\Upsilon$  that contains a unary function symbol not in  $\Upsilon$ . If this symbol is *wellform*( $\cdot$ ), then a model realization  $R$  is well-formed if  $\exists x \in T_{\Upsilon_C}(\Sigma), R \vdash_C \text{wellform}(x)$ . A model is well-formed if it is possible to derive any term of the form *wellform*( $\cdot$ ). Contrarily,  $\Upsilon_C$  may contain the symbol *malform*( $\cdot$ ) and  $R$  is well-formed if it is impossible to derive a *malform*( $\cdot$ ) term:  $\forall x \in T_{\Upsilon_C}(\Sigma), R \not\vdash_C \text{malform}(x)$ . We can tune the expressiveness of the structural semantics by choosing the consequence operator. Algorithmically, consequence operators correspond to repeated applications of inference procedures, which are usually well-known and readily available.

The combination of algebra and logic proves useful for formalizing other aspects of model-based design. For example, a model transformation  $\tau$  corresponds to a map between the model realizations of two domains  $\tau: R_{\Upsilon} \rightarrow R_{\Upsilon'}$ . Additionally,  $\tau$  can be defined with a set of transformation axioms, and then the consequence operator calculates the transformation. From there, metamodeling can be described as a special transformation  $\tau_{meta}$  from a metamodel domain to the domain of all domains. Thus, a formal basis can be constructed for a large portion of a model-based tool framework. We have implemented this using the *Generic Modeling Environment* (GME)[43]. Using GME, a user can construct a metamodel annotated with constraints, and then the resulting formal description of the domain can be automatically generated. Additionally, we have developed an automated theorem prover called *FORMULA* (FORmal Modeling Using Logic Analysis) for proving properties of domains and transformations.

Metamodeling is a common method for defining the structural semantics of DSMLs. We will now describe, in detail, a metamodel that will be used as a running example throughout this chapter. Figure 47 shows an example metamodel *DF*, written in the notation of UML class diagrams. *DF* defines the structure of a digital signal processing (DSP) language. Figure 46 is an example model belonging to the *DF* domain. (Actually, *DF* allows cycles, so Figure 46 is well-formed with respect to this metamodel.) The box labeled *DSPObject* represents the class of dataflow processing elements. Instances of *DSPObject* are shown as blocks (e.g. the block labeled *FFT*) in Figure 46. Dataflow elements provide an interface with input ports for accepting data, and output ports for emitting data. Structurally, interfaces appear as sets of labeled

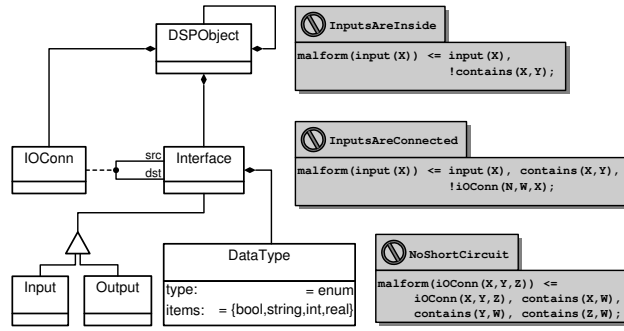


Figure 47. *DF* - A dataflow metamodel with constraint annotations

connection points contained within *DSPObj* instances. The ports are shown as white arrows on the left and right hand sides of the *DSPObj* instances (Figure 46). The *DF* metamodel defines ports using three classes: *Interface*, *Input*, and *Output*. Instances of *Input* (*Output*) correspond to input (output) ports. The triangle symbol in *DF* declares that *Interface* is a superclass of *Input* and *Output*. Edges terminating in black diamonds indicate containment, so *Interface* instances are contained in *DSPObj* instances. This rule also applies to the subclasses of *Interface*. Similarly, *DSPObj* instances can contain more *DSPObj* instances, permitting arbitrarily deep hierarchy in the dataflow systems.

Ports are usually typed to restrict the data values that can be received or produced. Structurally, types correspond to additional information tagged onto port instances. These data tags (or class fields) are represented by attribute classes, of which *DataType* is an example. The data encapsulated by attributes are also typed, and so the *DataType* class contains a *type* field describing the type of the data encapsulated by instances. (This requires some meta-level thinking.) In this example, a *DataType* instance encapsulates a type identifier from a known list of types. Thus, the type of *DataType* is *enumeration*, written `type: = enum`. The *items* field contains the possible values of the *DataType* enumeration: *bool*, *string*, *int*, or *real*. Finally, ports can be connected together, as shown by the arrows of Figure 46. The metamodel explains that instances of the *IOConn* class are edges that start and end on instances of *Interface*, and the actual edge is contained in a *DSPObj* instance. The shaded annotations (right-hand side) add more constraints to the definition of well-formed *DF* models. We shall describe these constraints in more detail later.

A metamodel is a concise mechanism for specifying the complex structural semantics of DSMLs, such as those found in embedded systems. Without a formal foundation for these diagrams, it would be difficult to effectively reason about realistic DSMLs. Our approach converts this diagram into a model signature  $\Upsilon$ , a constraint signature  $\Upsilon_C$  extending  $\Upsilon$ , and a set of constraints  $C$  for deciding if a model is well-formed. Constraint annotations are viewed as added axioms, which are combined with the constraints generated from the diagram. Thus, we can utilize existing descriptions of DSML structure in an automated fashion.

## Time-Model Dynamics

Using the structural semantics, we can now be more precise about the mediation capabilities of a structural interface. Figure 48 zooms in on the structural interface of Figure 45. At every point in time, the structural interface contains an encoding of the high-level model that describes the current configuration of the component. At time  $t_0$  this model is the initial high-level model  $M^{t_0}$ . At some point in time  $t_i$ , an adaptor requests a change in implementation by putting a new model  $M^a$  into the interface. In Figure 48, the initial model is a triangle graph ( $K_3$ ), but the adaptor puts a star graph ( $S_4$ ) into the interface. Notice that the term-algebraic encoding of the model is used as a communication protocol. In fact, our approach generates a customized protocol from a metamodel, reducing communication overhead. The actual data transmitted is shown under the star graph.

In the most general sense, the interface implements a mapping  $\mathcal{I} : R_{\Upsilon} \times R_{\Upsilon} \rightarrow R_{\Upsilon}$ . The model provided to the inside of the component at time  $t_i$  is  $M^{t_i} = \mathcal{I}(M^{t_{i-1}}, M^a)$ , where  $M^a$  is the model put into the interface by the adaptor, and  $M^{t_{i-1}}$  is the previous configuration. Additionally, the interface guards the inside of the component from malformed models. Assuming  $M^{t_{i-1}}$  is well-formed, then the interface must return a well-formed model  $M^{t_i}$  to the inside of the component, even if  $M^a$  is malformed. This implies that if the initial model is well-formed, then the component's configuration will always be reflected by some well-formed model. We write  $R \models C$  (read “ $R$  satisfies  $C$ ”) if a model realization  $R$  satisfies the well-formedness constraints  $C$ .

**Definition 17.** Let  $D = \langle \Sigma, \Upsilon, \Upsilon_C, C \rangle$  be a domain. A structural interface  $\mathcal{I}_D$  has the following properties:

1.  $\mathcal{I}_D : R_{\Upsilon} \times R_{\Upsilon} \rightarrow R_{\Upsilon}$
2.  $\forall X, Y \in R_{\Upsilon}, (X \models C) \Rightarrow (\mathcal{I}_D(X, Y) \models C)$

In Figure 48, the interface  $\mathcal{I}$  returns the complete graph  $K_4$  to the inside of the component, i.e.  $\mathcal{I}(K_3, S_4) = K_4$ . Presumably,  $K_3$  and  $K_4$  are well-formed models for this domain.

Notice that  $M^{t_i}$  may contain many of the same terms as  $M^{t_{i-1}}$ . Practically, this occurs often, so it is useful to rephrase the interface in terms of the model elements that changed between  $M^{t_i}$  and  $M^{t_{i-1}}$ . We call this a  $\Delta$ -interface. Given an interface  $\mathcal{I}$ , the associated  $\Delta$ -interface assigns a pair  $(\Delta^+(X, Y), \Delta^-(X, Y))$  to each input pair  $(X, Y)$  according to the following:

$$\Delta^+(X, Y) = \mathcal{I}(X, Y) - X \tag{V.1}$$

$$\Delta^-(X, Y) = X - \mathcal{I}(X, Y) \tag{V.2}$$

where the subtraction operation is set-subtraction. The  $\Delta^+$  set contains all new terms added to  $X$ , and the  $\Delta^-$  set contains all terms taken away from  $X$ . It is easy to see that

$$\mathcal{I}(X, Y) = [X \cup \Delta^+(X, Y)] - \Delta^-(X, Y)$$

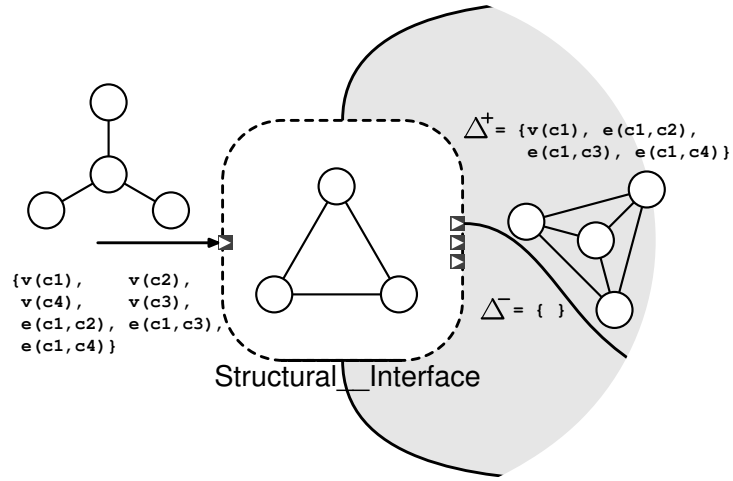


Figure 48. A general  $\Delta$ -interface acting on an input model

holds, so no information is lost by using the  $\Delta$ -interface. Figure 48 shows the two sets  $\Delta^+$ ,  $\Delta^-$  for the adaptation from  $K_3$  to  $K_4$ . These sets can be viewed as control inputs to the generator and model store subcomponents of the adaptee.

The main objective is to actually calculate an interface  $\mathcal{I}_D$  from a domain  $D$ . Clearly, there is not a unique interface  $\mathcal{I}_D$ . In fact, different interfaces capture different assumptions about how components respond to adaptation requests. Thus, the interface-generation problem is under-specified until details about the permissible *time-model dynamics* are provided. This brings us to the usual problem: How do we describe the *time-model dynamics* independently of the interface? We use the well-known approach of *traces*. Traces provide a denotational mechanism for describing dynamics, while interfaces provide an operational mechanism. An interface is “correct” with respect to traces if the set of traces generated by an interface matches the “correct” trace set. Let us be more specific.

**Definition 18.** Given a domain  $D$ , a *model trace*  $\rho$  is an infinite sequence of well-formed models from  $D$ :

1.  $\rho : \mathbb{Z}_+ \rightarrow R_{\Upsilon}$
2.  $\forall i \in \mathbb{Z}_+, \rho(i) \models C$

We call  $\mathcal{T}$  a *trace semantics* or a *set of traces*, if for every well-formed model  $R$  in  $D$ , there is some trace that begins at  $R$ .

**Lemma 19.** A domain  $D$  and a structural interface  $\mathcal{I}_D$  defines a trace semantics  $Tr(\mathcal{I}_D)$  according to the following induction:

1.  $Tr(\mathcal{I}_D)_0 = \{\rho \mid \mathbf{dom} \rho = [0, 0] \text{ and } \rho(0) \models C\}$



$$\begin{aligned}
2. \mathcal{Tr}(\mathcal{I}_D)_{i>0} &= \left\{ \rho \left| \begin{array}{l} \mathbf{dom} \rho = [0, i], \\ \rho \upharpoonright_{[0, i-1]} \in \mathcal{Tr}(\mathcal{I}_D)_{i-1}, \\ \exists Y \in R_{\Upsilon} \rho(i) = \mathcal{I}_D(\rho(i-1), Y) \end{array} \right. \right\} \\
3. \mathcal{Tr}(\mathcal{I}_D) &= \mathcal{Tr}(\mathcal{I}_D)_{\infty}
\end{aligned}$$

We also call the set  $\mathcal{Tr}(\mathcal{I})$  the *set of traces generated by  $\mathcal{I}$* . This set contains all the possible sequences of adaptations that can be reached by starting from any well-formed initial model, and putting any model realization into the structural interface. The base case of the induction, the set  $\mathcal{Tr}(\mathcal{I})_0$ , contains all sequences of length 1 that start at a well-formed model. From there, sequences of length 2 are those maps defined on the interval  $[0, 1]$  such that the *restriction* of the map to the interval  $[0, 0]$  (written  $\rho \upharpoonright_{[0, i-1]}$ ) is in  $\mathcal{Tr}(\mathcal{I})_0$ . Also, the value of the sequence at location 1 is given by  $\mathcal{I}(\rho(0), Y)$  for some  $Y \in R_{\Upsilon}$ . In general, the sequences are grown by finding all the traces of length  $i$ , and then collecting all the possible extensions to length  $i + 1$ . The lemma is easily proved, because by Definition 17  $\mathcal{I}(\rho(0), Y)$  is well-formed. By induction, every element of a sequence is well-formed. Therefore, the set  $\mathcal{Tr}(\mathcal{I})_{\infty}$  is a set of traces according to Definition 18. With these definitions, we can describe the time-model dynamics for a domain  $D$  by characterizing the trace set  $\mathcal{T}_D$ . This denotational definition does not immediately yield an interface. However, an interface that implements these dynamics has the property that  $\mathcal{Tr}(\mathcal{I}_D) = \mathcal{T}_D$ . We will now describe the time-model dynamics for a number of important system classes.

### Scenario-based Adaptation

Many system classes define adaptation with respect to a predefined set of scenarios. These classes include *modal systems*, *hierarchically layered alternatives*, and *adaptive software systems*. From the adaptor/adaptee perspective, systems differ by **(1)** how the adaptors pick the current scenario, **(2)** how scenarios overlap in the adaptee. Figure 49 shows a typical example of scenario-based adaptation for the purposes of fault mitigation. The dataflow graph in the lower-half of Figure 49 captures the computational structure of an embedded platform. The platform intentionally contains a certain amount of redundancy. Under “normal operation” the platform uses the computational path modeled by dataflow elements  $\{a_0, a_1, a_2, a_3\}$ . The upper-half of the figure shows an automata-based fault adaptor. This adaptor monitors the behavior of the adaptee for indications that a subcomponent has failed. If failure event  $e_1$  is detected, then dataflow element  $a_0$  or  $a_1$  has failed. The fault adaptor invokes a predefined scenario  $M1$  that mitigates the fault by reconfiguring the adaptee to use the path  $\{b_0, b_1, a_2, a_3\}$ . In this case, the scenarios are not disjoint with respect to model structure. Simpler forms of modality are accomplished by defining the  $i^{th}$  mode (or scenario) to be a unique submodel  $R_i \in R_{\Upsilon}$ . The overall system model  $R$  is the disjoint union of the scenarios,  $R = \dot{\bigcup}_i R_i$ . Assume that adaptees are initialized with the overall system model  $M^{all}$  containing the possible scenarios. The time-model dynamics always puts the adaptee into a model strictly smaller than

$M^{all}$ . Thus, we have our first requirement on the model traces for scenario-based adaptation:

**Definition 20.** A domain  $D$  exhibits a *scenario-based* time-model dynamics if the allowed model traces satisfy:

$$\forall \rho \in \mathcal{T}, \forall i \in \mathbb{Z}_+, \rho(i) \leq \rho(0)$$

where models are ordered by set inclusion. This definition correlates precisely with the notion of a scenario. In fact, if the time-model dynamics of a domain satisfies this property, then the scenarios can be directly reconstructed from the traces. Assuming the models  $M^{all}$  are finite, it must be that  $\mathbf{im} \rho \subseteq \{R' | R' \models C, R' \leq \rho(0)\}$ . Practically, model structures are always finite, hence the number of possible scenarios is finite. (Of course, the state space associated with the *behavioral semantics* of a model is often infinite. But, we are dealing with structure here.)

Though concise, this basic trace semantics does not reflect how interfaces react to particular inputs. We now extend the trace semantics to model the response of the interface to sequences of adaptation requests. Let  $\alpha$  be a sequence of adaptation requests, i.e.  $\alpha : \mathbb{Z}_+ \rightarrow R_{\mathcal{Y}}$ . This sequence represents the models put into the interface, which may be malformed. Let  $(\alpha, \rho)$  be a pair such that  $\alpha(i)$  denotes the adaptation request provided to the platform at time  $t_i$ , and  $\rho(i+1)$  denotes the result of that request. Call a set  $\mathcal{T}_{IO}$  of extended traces, the *IO trace semantics*. (“IO” stands for input/output.) Notice that the basic trace semantics can be recovered by a projection operator  $\pi_{\rho}$ . Let  $\pi_{\rho}(\mathcal{T}_{IO}) = \{\rho | (\alpha, \rho) \in \mathcal{T}_{IO}\}$ . Sometimes it is convenient to extract all the traces that start at a particular initial condition. Let  $\pi_{\rho(0)=R}(\mathcal{T}_{IO}) = \{(\alpha, \rho) | \rho(0) = R, (\alpha, \rho) \in \mathcal{T}_{IO}\}$ . The projections  $\pi_{\alpha}$  and  $\pi_{\alpha(0)=R}$  are defined analogously.

**Definition 21.** A set of extended traces  $\mathcal{T}_{IO}$  is an *IO trace semantics* with respect to domain  $D$  if:

1.  $\pi_{\rho}(\mathcal{T}_{IO})$  is a trace semantics.

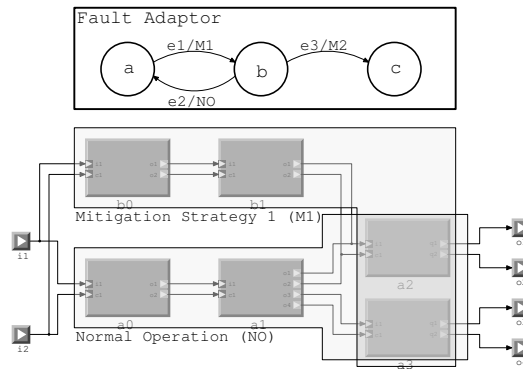


Figure 49. Example of scenario-based adaptation

$$2. \forall R \models C, \forall \alpha : \mathbb{Z}_+ \rightarrow R_{\Upsilon}, \alpha \in \pi_{\alpha}(\pi_{\rho(0)=R}(\mathcal{T}_{IO}))$$

The first property requires the projection of the extended semantics to be a proper trace semantics. The second property requires the extended trace semantics to define the response of the system to every possible adaptation input sequence, starting from every possible initial condition.

Most scenario-based systems also make the following assumption about adaptation history: Whenever an adaptor submits the same adaptation request, the adaptee configures itself to the same model. For example, in Figure 49, if the adaptor demands fault mitigation  $M1$  multiple times, then the adaptee complies the same way by configuring itself to  $\{b_0, b_1, a_2, a_3\}$ . Again, we do not mean that the system loses its state, but that the commands to the reconfiguration engine, as provided by the structural interface, are the same each time the same model is pushed into the interface. We call this property *adaptation-repeatable*.

**Definition 22.** Given a domain  $D$  and an IO trace semantics  $\mathcal{T}_{IO}$  over  $D$ , then trace semantics is *adaptation-repeatable* if  $\mathcal{T}_{IO}$  satisfies:

$$\forall R \models C, \forall (\alpha_1, \rho_1), (\alpha_2, \rho_2) \in \pi_{\rho(0)=R}(\mathcal{T}_{IO}), \alpha_k(i) = \alpha_{k'}(j) \Rightarrow \rho_k(i+1) = \rho_{k'}(j+1),$$

where  $i, j \geq 0, k, k' \in \{1, 2\}$

Pulling all of these properties together, we now define the traces semantics found in most styles of scenario-based adaptation.

**Definition 23.** Given a domain  $D$ , a trace semantics is *scenario-regular* if the following hold:

1.  $\mathcal{T}_{IO}$  over  $D$  is adaptation-repeatable
2. Whenever a request corresponds to a known scenario, the system chooses that scenario.
3. Whenever a request  $M^a$  corresponds to an unknown scenario, there exists a greatest known scenario  $M^k$ , such that  $M^k \leq M^a$ .

We have already discussed the first property. The second property is intuitive: Imagine that the fault adaptor in Figure 49 puts the submodel given by  $M1$  into the interface. In this case,  $M1$  is a known scenario, and is the model that the interface should provide to the inside of the component. Unfortunately, we do not have space to describe all the existing frameworks that satisfy these properties. In the authors' opinion, these proceed naturally from the intended meaning of a "scenario" or "mode", but presented from our model-based perspective. For tool examples, see the *modal models* of *Ptolemy II* [30], the modes of *Giotto* [17], the hierarchical alternatives of *DESERT* [104], and the fault adaptors of *ARMORs* [105].

Though the first two properties capture existing intuition, the third property is new. It explains how a scenario should be selected when the adaptee is provided with an unknown adaptation request. Existing tools do not deal with this eventuality, but our structural interfaces can. Property 3 provides the skeleton for deriving meaningful structural interfaces for scenario-based adaptation. In the next section we show how to calculate an interface that satisfies these properties.

## Calculating Scenario-Regular Interfaces

In this section we show how to construct an interface  $\mathcal{I}$  such that  $\mathcal{T}r_{IO}(\mathcal{I}) = \mathcal{T}r_{IO}$ , where  $\mathcal{T}r_{IO}$  is the denotational semantics of a scenario-regular model dynamics<sup>1</sup>. In order to simplify the proofs, we will make the assumption that the scenarios given by a model  $M^{all}$  are all the well-formed submodels (as opposed to just some of the well-formed submodels). Generally, the structural semantics of a domain can be defined to force this. We now show that a scenario-regular trace semantics has a nice underlying algebraic structure.

**Lemma 24.** *Let  $\mathcal{T}r_{IO}$  be an adaptation-repeatable semantics over domain  $D$ . Let  $R_{\Upsilon}^*$  be the set of all well-formed models of  $D$ . Then, there exists a function  $\Gamma(X, Y) : R_{\Upsilon}^* \times R_{\Upsilon} \rightarrow R_{\Upsilon}$  such that:*

$$\forall(\alpha, \rho) \in \mathcal{T}r_{IO}, \forall i \geq 0, \rho(i+1) = \Gamma(\rho(0), \alpha(i))$$

*Proof.* For every  $(\alpha, \rho) \in \mathcal{T}r_{IO}$  define  $\forall i \geq 0, \Gamma(\rho(0), \alpha(i)) = \rho(i+1)$ . The trace semantics defines the response for every possible adaptor starting from every possible well-formed model  $M^{t_0}$ . Thus, for every well-formed model  $M^{t_0}$  and every model realization  $R$ , there is a trace  $(\alpha, \rho)$  with  $\alpha(0) = R$  and  $\rho(0) = M^{t_0}$ . Hence,  $\Gamma$  is defined over the proper domain. Assume  $\Gamma$  is not well-defined. It is possible that there exists  $(\alpha, \rho)$  and  $(\alpha', \rho')$  such that  $\rho(0) = \rho'(0)$ ,  $\alpha(i) = \alpha'(j)$  and  $\rho(i+1) \neq \rho'(j+1)$  for some  $i, j$ . However, this is impossible as  $\mathcal{T}r_{IO}$  is adaptation-repeatable. Thus,  $\Gamma$  is well-defined.  $\square$

This lemma tells us that a scenario-regular semantics is completely determined by a function  $\Gamma$  parameterized by the initial model of the trace. Actually, this result only uses the property of adaptation-repeatable. When we include the definition of scenario-regular (Definition 23), we find that  $\Gamma$  defines a family of *interior operators* over  $R_{\Upsilon}$ . Interior operators are foundational in modern algebra, so this is an important result.

**Lemma 25.** *Let  $\mathcal{T}r_{IO}$  be a scenario-regular semantics over domain  $D$ . For each  $R \models C$ , define  $\Gamma_R(X) = \Gamma(R, X)$  to be a unary function formed by fixing the first coordinate of  $\Gamma$  at  $R$ . Then, every  $\Gamma_R(X)$  is an interior operator with range in the interval  $[R, \emptyset]$ , and hence satisfies:*

1.  $\Gamma_R(\Gamma_R(X)) = \Gamma_R(X)$  (Idempotent)
2.  $\Gamma_R(X) \leq X$  (Contractive)
3.  $X \leq Y \Rightarrow \Gamma_R(X) \leq \Gamma_R(Y)$  (Isotone)
4.  $\emptyset \leq \Gamma_R(X) \leq R$  (Bounded)

*Proof.* By construction of  $\Gamma(X, Y)$ , the unary map  $\Gamma_R(X)$  takes  $X$  to a known scenario. Thus,  $\Gamma_R(\Gamma_R(X)) = \Gamma_R(X)$ , because by Definition 23, a known scenario is mapped to itself. Also, by Definition 23, if  $X$  is an unknown scenario, it is mapped to a smaller known scenario;  $\Gamma_R$  is contractive. Furthermore, since  $X$  is mapped to itself or  $X$  is mapped to a greatest well-formed model below  $X$ , the maps are isotone. Finally, by Definition 20,  $\Gamma_R(X) \leq R$ , and  $\Gamma_R(X)$  cannot be smaller than  $\emptyset$ .  $\square$

<sup>1</sup>We extend  $\mathcal{T}r(\mathcal{I})$  to  $\mathcal{T}r_{IO}(\mathcal{I})$  in the obvious way, without further discussion.

Combining these facts, the scenarios of a model  $R$  define a lattice  $L_R = \langle [R, \emptyset], \wedge, \vee, \top, \perp \rangle$  where the meet ( $\wedge$ ) and join ( $\vee$ ) operations are defined in the usual way from the underlying partial order  $\leq$ . The interior operator  $\Gamma_R$  takes any model realization  $X \in R_\Upsilon$  and pushes it into this lattice. In fact, Definition 23 requires that  $X$  be pushed to a *unique* greatest element in the lattice. This puts an interesting restriction on the structure of  $L_R$ . First, if  $X$  has no intersection with the interval, then the only solution is to map it to  $\emptyset$ . Similarly, if  $X$  is a known scenario, then it is mapped to itself. However, if  $X \in [R, \emptyset]$ , but is an unknown scenario, then the least upper bound of all known scenarios under  $X$  must be unique:  $\bigvee \{Y | \Gamma_R(Y) = Y, Y \leq X\} \leq X$ . Figure 50 illustrates the properties of a such a lattice.

The rectangle represents the set of all model realizations  $R_\Upsilon$ , and the gray “bag” in the center represents the lattice generated by some well-formed model  $R$ . The black circles are the known scenarios of  $R$  and the white circles are unknown scenarios. The dotted lines indicate the result of  $\Gamma_R$ . Consider the unknown scenario labeled  $C$ .  $\Gamma_R(C)$  takes  $C$  to the greatest known scenario under  $C$ , which is unique for this lattice. However, the gap in the lattice, labeled  $D$ , has a number of equally large known scenarios directly under it. This violates the requirement that the greatest known scenario exists and is below the unknown scenario. (In this case, the greatest scenario exists, but is incomparable to  $D$ .) Thus, the example lattice does not

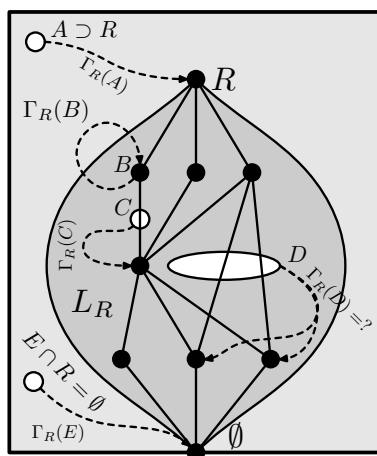


Figure 50. Lattice generated by the adaptation scenarios of  $R$

correspond to scenario-regular adaptation. Additionally, if the lattice is scenario-regular, then  $\Gamma_R$  has no degrees of freedom, and  $\Gamma(X, Y)$  is uniquely determined by the structural semantics of  $D$ .

**Theorem 26.** *Given a domain  $D$ , there exists at most one scenario-regular semantics  $\mathcal{I}_{IO}$  where the scenarios are the well-formed submodels of a model. Furthermore,  $\Gamma(X, Y)$  is unique.*

This theorem provides the foundation for an automatic construction of structural interfaces. First, the user defines a domain  $D$  by constructing a metamodel  $M_{meta}$  with existing modeling tools. Second, the user creates a model  $M^{t_0}$  belonging to  $D$ , capturing the adaptation scenarios. Again, this can be done with

existing tools. At this point, we apply a series of analyses to create the structural interface:

1. Extract the structural semantics from the metamodel  $M_{meta}$  by applying  $\tau_{meta}(M_{meta})$ . The result of this extraction is a tool-independent definition of the domain  $D$ .
2. Check that the initial model  $M^{t_0}$  is well-formed with respect to  $D$ .
3. Prove that the  $L_R$  lattices of  $D$  have the proper structure.
4. Generate the interface  $\mathcal{I}(X, Y) = \Gamma_{M^{t_0}}(Y)$  implementing the interior operator for the lattice  $L_{M^{t_0}}$ . By construction of  $\mathcal{I}$  and uniqueness of the semantics  $\mathcal{Tr}_{IO}(\mathcal{I}) = \pi_{\rho(0)=M^{t_0}}(\mathcal{S}_{IO})$ .

Steps 1 and 2 have been described in[39]. Step 3 is a more difficult step, and depends on the consequence operator used to define well-formedness. Step 4 generates an interface specific to the embedded system modeled by  $M^{t_0}$ . It can also be viewed as a partial evaluation[106] of  $\Gamma(X, Y)$  by fixing  $X = M^{t_0}$ . We now show how to perform steps 3 and 4 for the cases where well-formedness is defined with a nonmonotonic consequence operator corresponding to an extension of Horn logic.

### Interfaces from Horn Logic with Negation

In our previous work on structural semantics we used an extension of Horn logic to capture the *nonmonotonic* nature of many modeling languages. Informally, nonmonotonic logics have the property that new facts can be proved from less information. This does not happen in classical logic. Given some information  $X \subseteq Y$ , then the facts provable from  $X$  are a subset of the facts provable from  $Y$ , i.e.  $X \vdash X'$ ,  $Y \vdash Y'$  then  $X' \subseteq Y'$ . Consider, once again, the DSP system of Figure 46. It is well-formed with respect to the metamodel of Figure 47, which means that we cannot prove any term of the form  $malform(\cdot)$  from the encoding of this model into a set of terms  $Y$ .  $\forall t, Y \not\vdash malform(t)$ . However, if we delete the edges going from the *FFT* element to the *Splitter* element, then the model becomes malformed. The *Splitter* element will never receive any data, and the system is not schedulable. The encoding of Figure 46 minus these edges yields a strictly smaller set of terms  $X \subset Y$ . Nonetheless,  $\exists t, X \vdash malform(t)$ , so a new fact is proved from fewer facts. A well-studied form nonmonotonic logic is Horn logic extended with a pseudo-negation. Before we describe this, let us recall some definitions of Horn logic using function symbols.

Well-formedness rules are described with axioms, which are *formulas* built from terms with *variables* and logical *connectives*. There are different approaches for distinguishing variables from constants. One way is to introduce a new alphabet  $\Sigma_v$  that contains variable names such that  $\Sigma \cap \Sigma_v = \emptyset$ . The terms  $T_{\Upsilon_C}(\Sigma)$  are called ground terms, and contain no variables. This set is also called the *Herbrand Universe* denoted  $\mathcal{U}_H$ . The set of all terms, with or without variables, is  $T_{\Upsilon_C}(\Sigma \cup \Sigma_v)$ , denoted  $\mathcal{U}_T$ . Finally, the set of all *non-ground* terms is just  $\mathcal{U}_T - \mathcal{U}_H$ . A *substitution*  $\phi$  is term endomorphism  $\phi : \mathcal{U}_T \rightarrow \mathcal{U}_T$  that fixes constants. In another words, if a substitution  $\phi$  is applied to a term, then the substitution can be moved to

the inside  $\phi f(t_1, t_2, \dots, t_n) = f(\phi t_1, \phi t_2, \dots, \phi t_n)$ . A substitution does not change constants, only variables, so  $\forall g \in \mathcal{U}_H, \phi(g) = g$ . We say two terms  $s, t \in \mathcal{U}_T$  *unify* if there exists substitutions  $\phi_s, \phi_t$  that make the terms identical  $\phi_s s = \phi_t t$ , and of finite length. (This implies the *occurs check* is performed.) We call the pair  $(\phi_s, \phi_t)$  the unifier of  $s$  and  $t$ . The variables that appear in a term  $t$  are  $\text{vars}(t)$ , and the constants are  $\text{const}(t)$ .

A *Horn clause* is a formula of the form  $h \Leftarrow t_1, t_2, \dots, t_n$  where  $h$  is called the *head* and  $t_1, \dots, t_n$  are called the *tail* (or body). We write  $T$  to denote the set of all terms in the tail. The head only contains variables that appear in the tail,  $\text{vars}(h) \subseteq \bigcup_i \text{vars}(t_i)$ . A clause with an empty tail ( $h \Leftarrow$ ) is called a *fact*, and contains no variables. Recall that these clauses will be used *only* to calculate model properties. This is enforced by requiring the heads to use those function symbols that do not encode model structure, i.e. every head  $h = f(t_1, \dots, t_n)$  has  $f \in (\Upsilon_C - \Upsilon)$ . (Proper subterms of  $h$  may use any symbol.) This is similar to restrictions placed on declarative databases[91]. We slightly extend clauses to permit *disequality* constraints. A Horn clause with disequality constraints has the form  $h \Leftarrow t_1, \dots, t_n, (s_1 \neq s'_1), (s_2 \neq s'_2), \dots, (s_m \neq s'_m)$ , where  $s_i, s'_i$  are terms with no new variables  $\text{vars}(s_i), \text{vars}(s'_i) \subseteq \bigcup_i \text{vars}(t_i)$ . We can now define the *meaning* of a Horn clause. The definition we present incorporates the *Closed World Assumption* which assumes all conclusions are derived from a finite initial set of facts (ground terms)  $I$ . Given a set of Horn clauses  $\Theta$ , the operator  $\widehat{F}_\Theta$  is called the *immediate consequence operator*, and is defined as follows:

$$I \widehat{F}_\Theta = I \cup \left\{ \phi(h_\theta) \mid \exists \phi, \theta, \phi(T_\theta) \subseteq I \text{ and } \forall (s_i \neq s'_i)_\theta \in \theta, \phi s_i \neq \phi s'_i \right\}$$

where  $\phi$  is a substitution and  $\theta$  is a clause in  $\Theta$ . It can be proved that  $I \vdash_\Theta I_\infty$  where  $I \widehat{F}_\Theta I_1 \widehat{F}_\Theta \dots \widehat{F}_\Theta I_\infty$ . The new terms derivable from  $I$  can be calculated by applying the immediate consequence operator until no new terms are produced (i.e. the least fixed point). Notice that the disequality constraints force the substitutions to keep certain terms distinct. *Nonrecursive Horn logic* adds the restriction that the clauses of  $\Theta$  can be ordered  $\theta_1, \theta_2, \dots, \theta_k$  such that the head  $h_{\theta_i}$  of clause  $\theta_i$  does not unify with any tail  $t \in T_{\theta_j}$  for all  $j \leq i$ . This is a key restriction; without it, the logic can become undecidable. Consider the recursive axiom  $\Theta = \{f(f(x)) \Leftarrow f(x)\}$ . Then  $\{f(c_1)\} \vdash_\Theta \{f(c_1), f(f(c_1)), \dots, f(f(f(\dots f(c_1) \dots)))\}$  includes an infinite number of distinct terms.

Clearly,  $\vdash$  is monotonic for classical Horn logic. We now extend the logic with negation to support nonmonotonicity. We should mention that a proper mathematical formulation of negation was one of the most important problems in logic programming. In this discussion we will avoid many complications by only describing a restricted form of nonmonotonicity corresponding to *stratified logic programs*. For more details see[94]. Consider the following Horn clause with negation:

$$\text{malform}(x) \Leftarrow f(x), \neg g(x)$$

where  $f, g \in \Upsilon$  are model symbols. Informally, this clause is satisfied if we find an occurrence of  $f(x)$ , but

we cannot find a corresponding occurrence of  $g(x)$ . It is essential to know that no other clauses generate terms of the form  $g(x)$ , otherwise we might conclude  $malform(x)$ , which could lead to the conclusion  $g(x)$ , which invalidates the fact that  $g(x)$  could not be found. Also notice that only tail elements can be negated. Let  $\Theta$  be a set of clauses in the following *normal form*:

$$h \Leftarrow t_1, \dots, t_n, (s_1 \neq s'_1), \dots, (s_m \neq s'_m), \neg r_1, \dots, \neg r_p$$

All the terms in the tail  $t_i = f(\dots)$ ,  $r_j = f'(\dots)$  have function symbols  $f, f' \in \Upsilon$ , and there is at least one positive term in the tail ( $n \geq 1$ ). Additionally, the variables in  $h$  occur in nonnegated terms, and  $h = g(\dots)$  is a constraint term:  $g \in (\Upsilon_C - \Upsilon)$ . Let  $T_\Theta^+ = \{t_i\}$  and  $T_\Theta^- = \{r_j\}$ . For this form, the immediate consequence operator can be rewritten to include negation.

$$I \widehat{\vdash}_\Theta = I \cup \left\{ \phi(h_\theta) \mid \begin{array}{l} \exists \phi, \theta, \forall \phi' \phi(T_\Theta^+) \subseteq I \text{ and } (\phi' \circ \phi)(T_\Theta^-) \cap I = \emptyset \text{ and} \\ (\phi' \circ \phi)s_i \neq (\phi' \circ \phi)s'_i \end{array} \right\}$$

A valid substitution  $\phi$  is one that does not fix any negative variables, but after  $\phi$  is applied to  $T_\Theta^-$ , there exists no other substitution  $\phi'$  that maps a negative term to a known fact. This reasoning is sound due to our previous requirements on the form of  $\Theta$ .

Given these definitions, consider the following extended Horn clauses:

1. Let  $\theta_1$  be  $malform(x) \Leftarrow f(x), g(x), h(x)$
2. Let  $\theta_2$  be  $malform(x) \Leftarrow f(x), g(x), \neg h(x)$
3. Let  $\theta_3$  be  $malform(x) \Leftarrow f(x), \neg g(x), \neg h(x)$

where  $\Upsilon = \{(f, 1), (g, 1), (h, 1)\}$  and  $(malform, 1) \in \Upsilon_C$ . If the constraint set is  $C_1 = \{\theta_1\}$ , then the domain  $D_1$  is monotonic in the *malform* property. Therefore, for any initial model  $M^{t_0}$  that is well-formed, all of its submodels are well-formed. Hence,  $D_1$  trivially satisfies the uniqueness property. Next, consider  $C_2 = \{\theta_2\}$ . The initial model  $M^{t_0} = \{f(c_1), g(c_1), h(c_1)\}$  is well-formed, but the submodel  $X = \{f(c_1), g(c_1)\}$  is malformed, because the lack of  $h(c_1)$  activates the malformedness rule. There are two equally sized models smaller than  $X$  that are well-formed ( $M^1 = \{f(c_1)\}$ ,  $M^2 = \{g(c_1)\}$ ), so the uniqueness property is not satisfied. An scenario-regular interface cannot be generated from domain  $D_2$ . Finally, let  $C_3 = \{\theta_3\}$  and consider  $M^{t_0} = \{f(c_1), f(c_2), h(c_1), h(c_2)\}$ . This model is well-formed, but the submodel  $X = M^{t_0} - \{h(c_1)\}$  is not. However, there is a greatest well-formed model under  $X$ , namely  $X' = M^{t_0} - \{f(c_1), h(c_1)\}$ , because the smallest corrective action is to remove  $f(c_1)$ . This domain is scenario-regular.

The problematic domains have malformedness clauses containing two or more positive terms and at least one negative term. However, there are cases where these clauses are not problematic. Consider  $C_4 = \{\theta_2, \theta_3\}$ . In this domain, there is not a unique least upper bound below the malformed model  $\{f(c_1), g(c_2)\}$ .  $\theta_2$  suggests that we remove either the  $f$  or  $g$  term. However, if the  $g$  term is removed, then  $\theta_3$  activates,



requiring removal of the  $f$  term. This leaves the empty model, which is strictly smaller than the model obtained by removing the  $f$  term. Intuitively, clause  $\theta_3$  breaks the ambiguity introduced by  $\theta_2$ .

This intuition can be formalized into a procedure that checks if a domain is scenario-regular. The notation  $(M, \phi) \models \Theta$  indicates that  $\phi$  is a substitution that satisfies clause  $\Theta$  for model  $M$ .

**Definition 27.** Given clauses  $\theta_1$  and  $\theta_2$  in normal form,  $\theta_2$  covers  $\theta_1$  if:

1.  $\exists \emptyset \subset S^+ \subset T_1^+$  such that  $\forall (M, \phi)$  it holds that
2.  $(M, \phi) \models \theta_1 \Rightarrow \exists \phi' (M - \phi(S^+), \phi') \models \theta_2$

When this property holds, a larger upper bound can be produced by removing  $\phi'(T_2^+)$ . This property can be calculated using unification and subsumption algorithms. Though there is not space to prove it here, the following theorem checks if a domain is scenario-regular:

**Theorem 28.** A domain  $D$ , with  $\Theta = C$  in normal form, is scenario-regular iff  $\forall \theta \in \mathbf{reduce}(\Theta)$ ,  $|T_\theta^+| = 1$ .

The map  $\mathbf{reduce}(\Theta)$  is defined inductively:

1.  $\mathbf{reduce}_0(\Theta) = \Theta$ , and  $\mathbf{reduce}(\Theta) = \mathbf{reduce}_\infty(\Theta)$ .
2. if  $\theta \in \mathbf{reduce}_i$  and  $|T_\theta^-| = 0$ , then  $\mathbf{reduce}_{i+1}(\Theta) = \mathbf{reduce}_i(\Theta) - \{\theta\}$ .
3. if  $\theta_1, \theta_2 \in \mathbf{reduce}_i$  and  $\theta_2$  covers  $\theta_1$ , then  $\mathbf{reduce}_{i+1}(\Theta) = \mathbf{reduce}_i(\Theta) - \{\theta_1\}$ .

This algorithm eliminates all malformedness rules that are purely monotonic, and all rules that are disambiguated by the covering relation. If the resulting set is empty, or every clause has exactly one positive term (e.g.  $\theta_3$  in the example), then the domain is scenario-regular. Additionally, the set  $\mathbf{reduce}(\Theta)$  can be directly converted into a  $\Delta$ -interface, where only the  $\Delta^-$  set is non-empty. If the reduction is non-empty, then every remaining clause has the form  $\mathit{malform}(\dots) \Leftarrow t, \neg r_1, \dots, \neg r_n$ . This can be converted into the clause  $\neg t \Leftarrow t, \neg r_1, \dots, \neg r_n$  where  $\neg t$  indicates that  $t$  is removed from the original set of facts. The immediate consequence operator can be adjusted to represent removal. Call  $\neg\mathbf{reduce}(\Theta)$  the set of clauses rewritten in “removal form”. Then we have the following:

$$\Delta^-(M^{t_0}, Y) \mapsto Y', \text{ where } (Y \cap M^{t_0}) \vdash_{\neg\mathbf{reduce}(\Theta)} Y' \quad (\text{V.3})$$

$$\mathcal{I}(M^{t_0}, Y) \mapsto (M^{t_0} - \Delta^-(M^{t_0}, Y)) \quad (\text{V.4})$$

Hence  $\mathcal{I}(M^{t_0}, Y) = \Gamma_{M^{t_0}}(Y)$  and correctly implements the denotational semantics of a scenario regular interface. Returning to the metamodel of Figure 47, this represents a scenario-regular domain: If a dataflow edge is deleted, and that edge is connected to an *Input*, then the *DSPObject* and all of its contents are deleted. This may leave some outgoing edges dangling, which are also deleted. The process continues iteratively, until the model is well-formed. The consequence operator  $\vdash_{\neg\mathbf{reduce}(\Theta)}$  implements precisely this iteration. Thus, we have a formal basis for robustly implementing the scenario-based adaptation of Figure 49 using structural interfaces.

## Conclusions and Future Work

We outlined an approach for automatically constructing an adaptive substrate for embedded systems. The general approach involves miniaturizing the tools for model-based design so that existing mechanisms of system construction can be used for on-the-fly for adaptation. An essential piece of this approach is a structural interface for pushing new configurations into the adaptive component. This interface must guard the delicate insides of a component from accidental or malicious attempts at misconfiguration. In order to accomplish this we have introduced a number of novel concepts relating DSML structure, defined via metamodels, to structural interfaces. First, model structure can be converted to a communication protocol for interacting with the interface. Second, DSML structure can be analyzed to determine if a theoretically sound form of mediation exists with respect to the intended time-model dynamics (i.e. style of adaptation). Third, we have thoroughly analyzed this approach for a common form of adaptation based on scenarios.

Unfortunately, this has left little space to describe our implementation techniques, which we summarize now. First, communication can be made more efficient by encoding sets of terms with a compact representation. For example, function symbols can be indexed, so that only indices are exchanged. More complex compression can be used if the embedded platform has resources to perform the encoding/decoding. Second, the clauses of  $\text{--reduce}(\Theta)$  take a restricted form, permitting rapid calculation of the well-formed submodel. For example, the RETE-based algorithms are effective for these types of clauses[107]. RETE networks do make a space/time tradeoff, and this can be adjusted depending on the embedded platform. Third, we have implemented our analysis algorithms with the FORMULA theorem prover. This prover can convert domain definitions into the necessary normal form, but this (as usual) is computationally hard. However, it only needs to be done once and off-line.

Our work is continuing in two directions. First, we are studying other forms of adaptation that are not scenario-based. For example, the work on service-oriented architectures (SOA) for wireless sensor nets exhibits interesting dynamics. Second, we are developing techniques for implementing the remaining components of the adaptive substrate. There exist two extremes to this subproblem. First, we may know nothing about how changes in model structure are reflected during code generation. If this is the case, then with scenario-based adaptation, we can partially evaluate the generated code for a number of scenarios. We analyze the various implementations for commonalities, and package them so that size and cost of reconfiguration is minimized. In the other extreme, we may know that smaller scenarios access strictly smaller amounts of the same code base. In this case, the code generated from the overall model can be instrumented with additional control structure to efficiently implement adaptation.

## APPENDIX A

### TRANSFORMATION FROM FSA TO C

#### Transformation rules

This appendix completes the FSA to C transformation example. The next step in the transformation is to create an assignment that initializes the *currentState* variable to the initial state. Figure 51 shows the rule that accomplishes this. It locates the initial state in the input FSA and then uses the crosslink created in Figure 14 to extract the associated enumeration element. It then creates a new assignment in the *SimLoop* that initializes the current state variable to the initial state enumeration element.

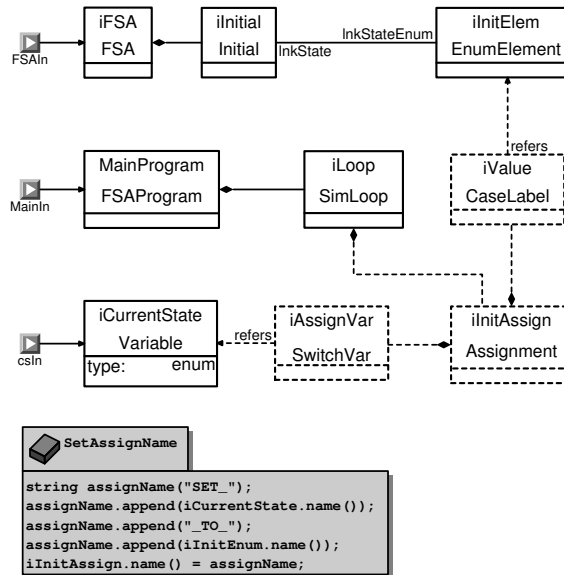


Figure 51. Finds the initial state and creates an assignment that initializes the *currentState* variable

The next step in the transformation is to create a unique *case* block for each instance of *State*. Figure 52 shows the transformation logic. An empty *switch* block is placed inside each of the new *case* blocks. This *switch* block will contain a *case* for each instance of *Transition* that starts on the corresponding state. The empty *switch* is passed out for processing by other rules. This rule also uses the previously created crosslinks to find the *EnumElement* linked to a *State*.

At this point in the transformation, every state has a corresponding *case* block, and inside of this *case* is an empty *switch* that switches on the *currentEvent* variable. Given a *State* instance  $s$  and the corresponding *Switch* instance  $w$ , we wish to find every transition  $s \xrightarrow{e} s'$ . For each of these, we must add a *case* to  $w$  with label  $e$ , and this *case* must assign  $s'$  to the *currentState* variable. Figure 53 shows the transformation

that accomplishes this. Because of the semantics of GREAT, this rule only works for *non-loop transitions*, i.e.  $s \neq s'$ . Figure 54 shows the rule that handles loop transitions. Note that these two rules use a new construct called a *guard*. A guard is boolean expression that can be attached to a rule; GREAT discards all matches that do not evaluate to *true* with respect to all guard expressions. The guard permits GREAT to find the *Event* instance  $e$  corresponding the trigger of a *Transition*. Recall that a *Transition* instance has a string attribute (also) called *guard*. Thus, we must locate the *Event* instance with the same name as the string attribute on the *Transition* instance. A guard expression allows us to filter matches so that we only consider *Transition-Event* pairs where the string attribute on the guard of the *Transition* matches the name of the *Event*. Notice that by subclassing *State* into *Initial* and *Acceptor*, we avoid a number of guard expressions. We could have added an enumeration attribute to *State* that would capture these distinctions, but the GREAT rules would have been more complicated.

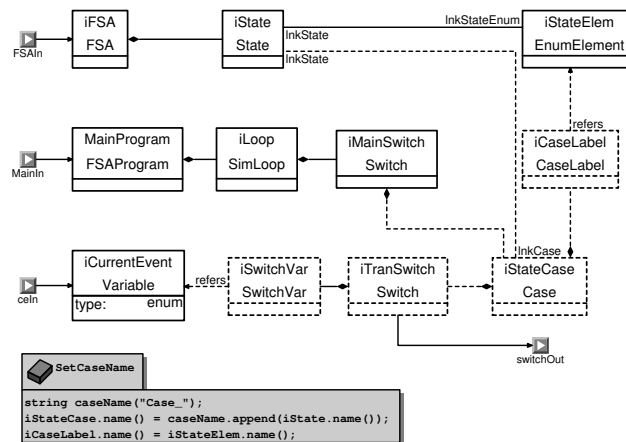


Figure 52. Creates a *case* and empty *switch* block for each *State* instance.

The final rule is one of the simplest. It introduces user feedback into the generated code. Whenever the automaton transitions to an acceptor state, a simple message is displayed to the user. The message declares that the input sequence was accepted and identifies the accepting state. This rule relies on the *lnkState-lnkCase* crosslink (created in Figure 52) to find the *case* block associated with each acceptor state, i.e. instance of *Acceptor*. Figure 55 shows this rule.

Figure 56 shows the sequencing of the GREAT rules. Recall that if a rule contains smaller rules, then the smaller rules complete before downstream rules execute. This semantics is important, because the rules inside of the *BuildSimulator* rule depend on the fact that the state and event *Enumeration* instances have already been constructed. This is ensured by placing the *BuildStateEnum* and *BuildEventEnum* rules inside of the *BuildDeclarations* rule.

Figure 57 shows an example FSA. State  $S1$  is the initial state, while state  $S3$  is an acceptor state. The

FSA contains two events, named  $a$  and  $b$ . Figure 58 shows the result of applying the transformation to the FSA in Figure 57. This tree shows the hierarchy of the generated model. When viewed from this perspective, it is easy to see how close the generated model is to the final C code. The actual code produced by the code generator is shown in the next section. In the interest of space the *EventStream* class is not shown. Finally, Figure 59 shows the result of executing the FSA with some input. Note that the “accept” message is delayed by one event.

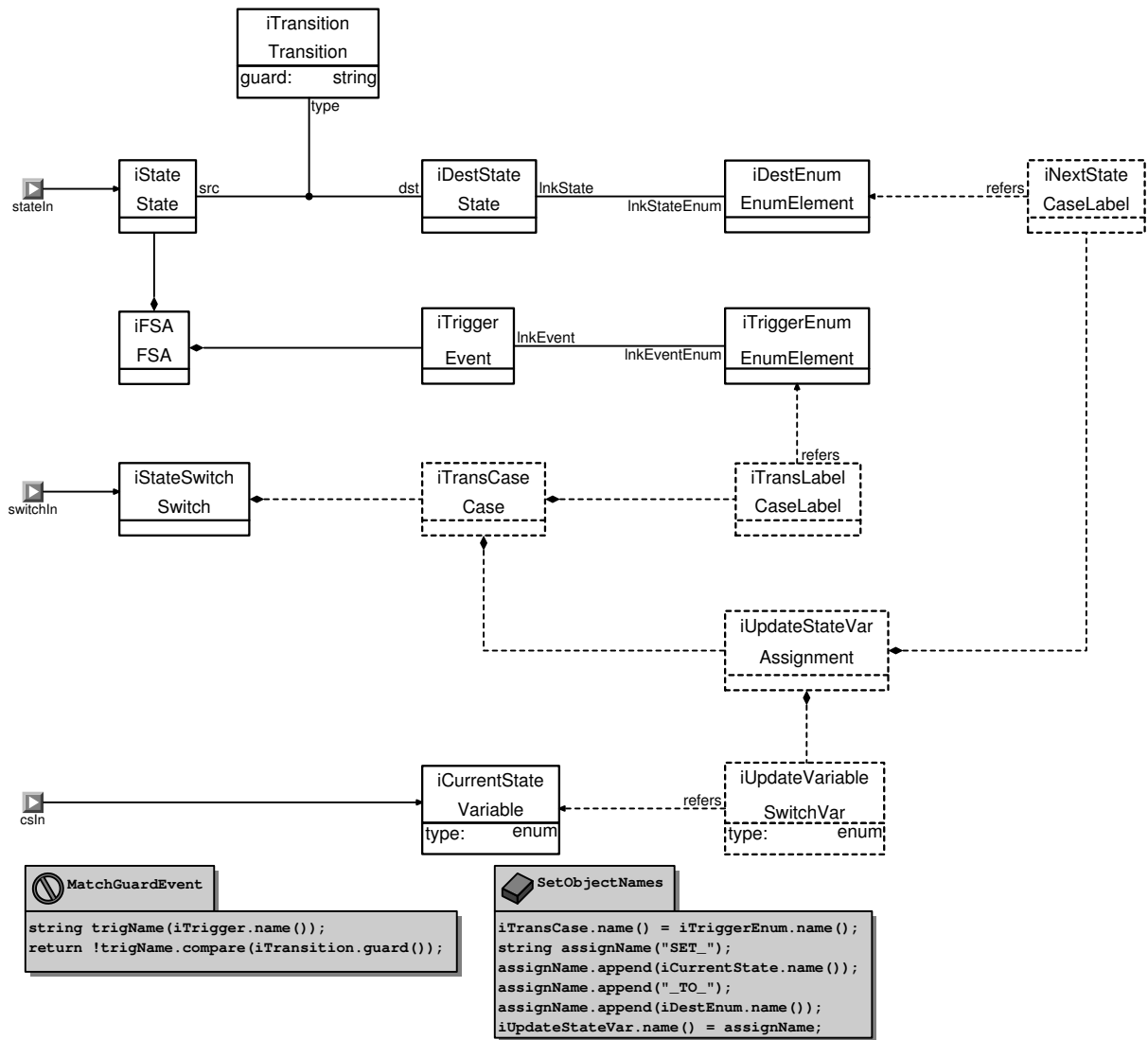


Figure 53. Creates non-loop transitions

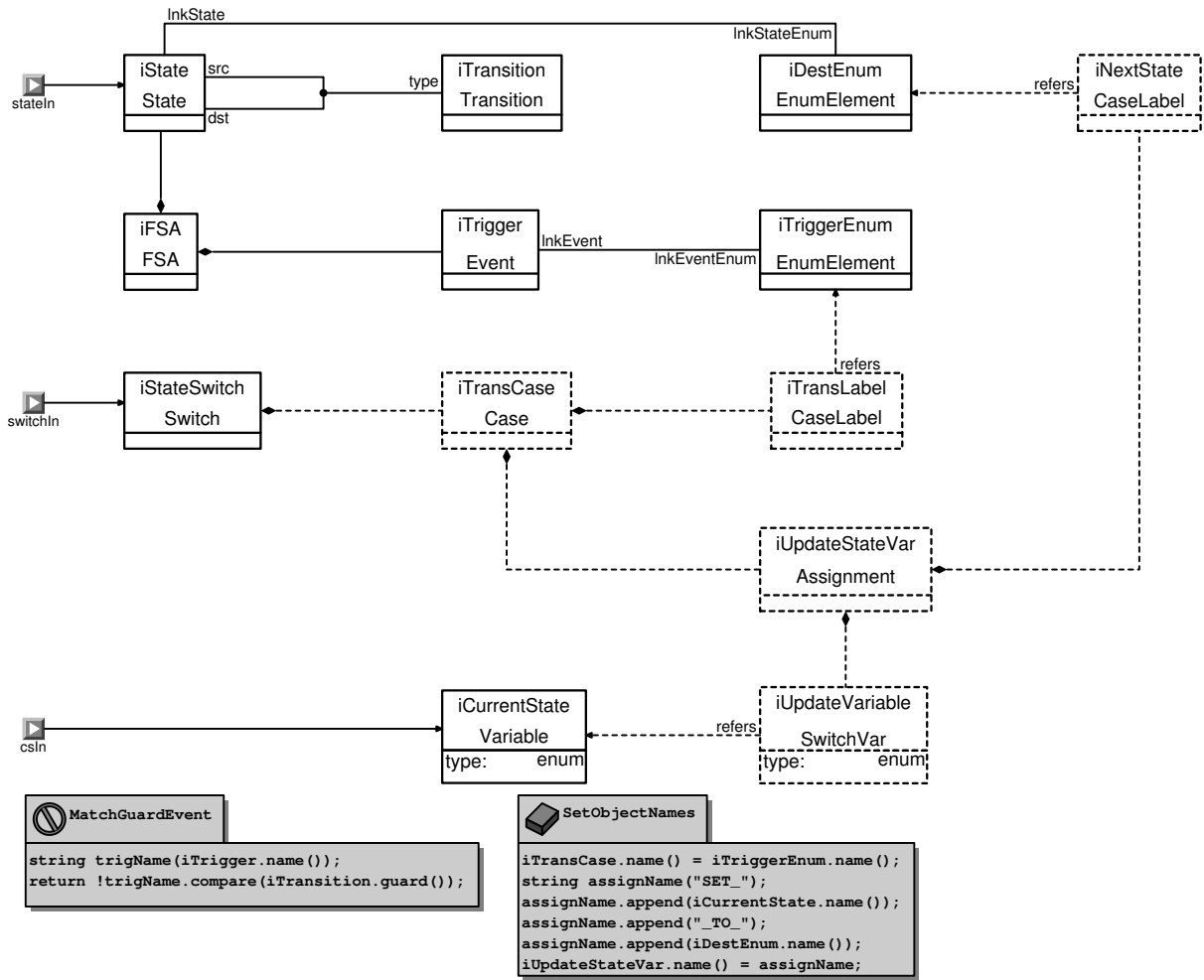


Figure 54. Creates loop transitions

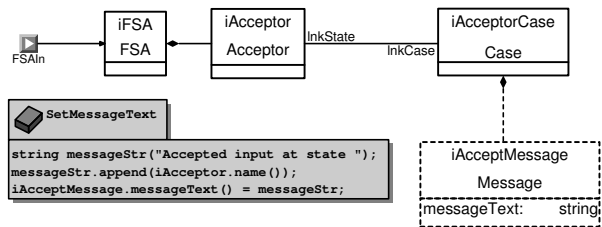


Figure 55. Builds feedback messages

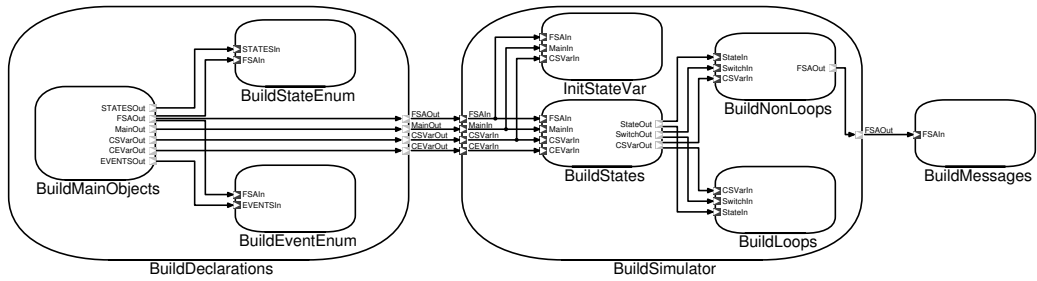


Figure 56. Sequencing of transformation rules

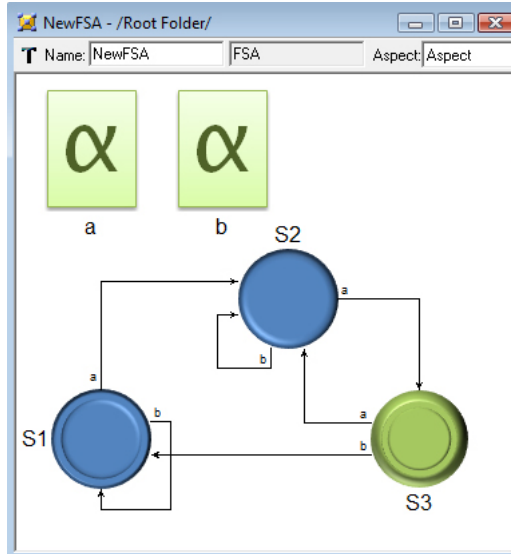


Figure 57. Example FSA acceptor



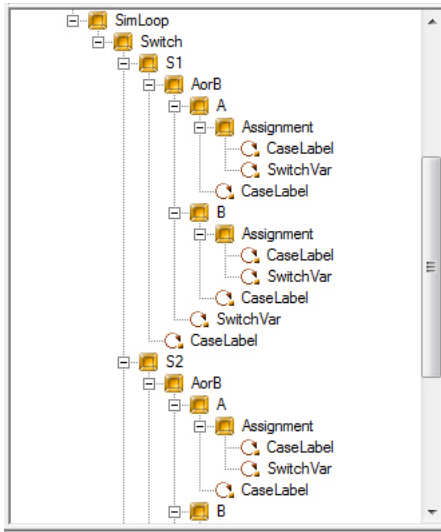


Figure 58. Partial C abstract syntax tree generated from example FSA.

```

Input event [0 to 1]: b
Input event [0 to 1]: b
Input event [0 to 1]: b
Input event [0 to 1]: a
Input event [0 to 1]: b
Input event [0 to 1]: b
Input event [0 to 1]: a
Input event [0 to 1]: b
Accepted input at state S3
Input event [0 to 1]: a
Input event [0 to 1]: a
Input event [0 to 1]: b
Accepted input at state S3
Input event [0 to 1]:

```

Figure 59. Simulation of generated FSA model

## C Code generated from FSA

```
#include "EventStream.h"
// ----- FSA Enumerations ----- //
enum STATES { STATE_S1 = 0, STATE_S3,
              STATE_S2, STATES_Count };
enum EVENTS { EVENT_a = 0, EVENT_b,
              EVENTS_Count };

// ----- FSA Variables ----- //
int currentEvent;
EventStream eventStream(EVENTS_Count);
int currentState;

// --- FSA Simulation Loop ---- //
void main() {
    currentState = STATE_S1;
    while(eventStream.read(currentEvent)) {
        switch (currentState) {
            case STATE_S2:
                switch (currentEvent) {
                    case EVENT_a:
                        currentState = STATE_S3;
                        break;
                    case EVENT_b:
                        currentState = STATE_S2;
                        break;
                    default:
                        return;
                        break; }
                break;
            case STATE_S3:
                printf("Accepted input at
state S3\n");
                switch (currentEvent) {
                    case EVENT_a:
                        currentState = STATE_S2;
                        break;
                    case EVENT_b:
                        currentState = STATE_S1;
                        break;
                    default:
                        return;
                        break; }
                break;
            case STATE_S1:
                switch (currentEvent) {
                    case EVENT_b:
                        currentState = STATE_S1;
                        break;
                    case EVENT_a:
                        currentState = STATE_S2;
                        break;
                    default:
                        return;
                        break; }
                break;
            default:
                return;
                break; }
        }
    }
    printf("HALT\n"); }
```

Figure 60. C code generated from AST model of an FSA

## APPENDIX B

### SPECIFICATION OF META-LEVEL COMPONENTS

#### Partial eMOF Transformation

The equations below calculate which types can contain other types, including containment capabilities endowed by inheritance edges. These equations handle containment of both classes and association classes.

$$\text{axiom}(\text{containment}(x, \text{ipath}(h, y)), \text{map}_3(\text{cancontain}, \text{var}(\mathbf{x}), \text{var}(\mathbf{y}))) \Leftarrow \text{containment}(x, y, \text{ipath}(h, y)).$$

$$\text{tail}(\text{containment}(x, \text{ipath}(h, y)), \text{map}_2(x, \text{var}(\mathbf{x}))) \Leftarrow \text{containment}(x, y, \text{class}(x), \text{ipath}(h, y)).$$

$$\text{tail}(\text{containment}(x, \text{ipath}(h, y)), \text{map}_4(x, \text{var}(\mathbf{x}), \text{var}(\mathbf{z}), \text{var}(\mathbf{w}))) \Leftarrow \text{containment}(x, y, \text{assocClass}(x), \text{ipath}(h, y)).$$

$$\text{tail}(\text{containment}(x, \text{ipath}(h, y)), \text{map}_2(h, \text{var}(\mathbf{y}))) \Leftarrow \text{containment}(x, y, \text{class}(h), \text{ipath}(h, y)).$$

These equations are similar to the inheritance cycle equations of the eMOF domain. They are added to the domain description to prevent containment cycles.

$$\text{axiom}(\text{cloop}, \text{malform}(\text{map}_3(\text{contains}, \text{var}(\mathbf{x}), \text{var}(\mathbf{x})))) \Leftarrow \text{true}.$$

$$\text{tail}(\text{cloop}, \text{map}_3(\text{contains}, \text{var}(\mathbf{x}), \text{var}(\mathbf{x}))) \Leftarrow \text{true}.$$

$$\text{axiom}(\text{ccycle2}, \text{malform}(\text{map}_3(\text{contains}, \text{var}(\mathbf{x}), \text{var}(\mathbf{y})))) \Leftarrow \text{true}.$$

$$\text{tail}(\text{ccycle2}, \text{map}_3(\text{contains}, \text{var}(\mathbf{x}), \text{var}(\mathbf{y}))) \Leftarrow \text{true}.$$

$$\text{tail}(\text{ccycle2}, \text{map}_3(\text{contains}, \text{var}(\mathbf{y}), \text{vvar}(\mathbf{x}))) \Leftarrow \text{true}.$$

The entire eMOF transformation is specified with about 200 transformation axioms. These axioms handle all the ways that containment, associations, attributes, and inheritance interact.

## Example of Generated Domain in Prolog

```
%% Signature symbols
:- dynamic
    output/1,    dSPObject/1,  interface/1,  input/1,
    malform/1,   enumvalue/2,  dataType/2,  contains/2,
    ccycle3/2,   cancontain/2,  iOConn/3,    cpath3/3,
    canconn/3.

%% Enumeration attribute values
enumvalue('DataType', 'Bool').
enumvalue('DataType', 'String').
enumvalue('DataType', 'Int').
enumvalue('DataType', 'Real').

%% Domain constraints
%% Attributes
malform(dataType(Y,V)) :- dataType(Y,V), dataType(Y,W), (V \== W).
malform(dataType(Y,V)) :- dataType(Y,V), \+interface(Y), \+input(Y), \+output(Y).
malform(iOConn(N,X,Y)) :- iOConn(N,X,Y), \+canconn('IOConn',X,Y).

%% Connection rules
canconn('IOConn',X,Y) :- interface(X), interface(Y).
canconn('IOConn',X,Y) :- input(X), interface(Y).
canconn('IOConn',X,Y) :- output(X), interface(Y).
canconn('IOConn',X,Y) :- interface(X), input(Y).
canconn('IOConn',X,Y) :- interface(X), output(Y).
canconn('IOConn',X,Y) :- input(X), input(Y).
canconn('IOConn',X,Y) :- input(X), output(Y).
canconn('IOConn',X,Y) :- output(X), input(Y).
canconn('IOConn',X,Y) :- output(X), output(Y).

%% Containment rules
malform(contains(X,Y)) :- contains(X,Y), \+cancontain(X,Y).
cancontain(X,Y) :- iOConn(X,Z,W), dSPObject(Y).
cancontain(X,Y) :- interface(X), dSPObject(Y).
cancontain(X,Y) :- dSPObject(X), dSPObject(Y).
cancontain(X,Y) :- input(X), dSPObject(Y).
cancontain(X,Y) :- output(X), dSPObject(Y).
malform(dataType(Y,Z)) :- dataType(Y,Z), \+enumvalue('DataType',Z).
malform(contains(X,X)) :- contains(X,X).
malform(contains(X,Y)) :- contains(X,Y), contains(Y,X).
cpath3(X,Y,Z) :- contains(X,Y), contains(Y,Z), (X \== Y), (X \== Z), (Y \== Z).
ccycle3(X,Z) :- cpath3(X,Y,Z), contains(Z,X).
malform(ccycle3(X,Y)) :- ccycle3(X,Y).

%% Additional annotations
malform(iOConn(X,Y,Z)) :- iOConn(X,Y,Z), contains(X,W), contains(Y,W), contains(Z,W).
```

## eMOF Domain

Table 12. Encoding of eMOF vertex primitives

<p><b>Class.</b> <math>class(x)</math> denotes a class named <math>x</math>.</p> <p><math>(class, 1) \in \Upsilon</math></p>
<p><b>Association Class.</b> <math>assocClass(x)</math> denotes an association class named <math>x</math>.</p> <p><math>(assocClass, 1) \in \Upsilon</math></p>
<p><b>Attribute Class.</b> <math>attribute(x, y)</math> denotes an attribute named <math>x</math> of type <math>y</math>. <math>enum(x, y)</math> indicates that attribute <math>x</math> can take the enumerated value <math>y</math>.</p> <p><math>\{(attribute, 2), (enum, 2)\} \subset \Upsilon</math>  <b>restrict</b><math>(type, \{type(\mathbf{bool}), type(\mathbf{string}), type(\mathbf{enum})\})</math></p> <p>An attribute must have a proper type. Also, an attribute cannot have an enum list, unless it is an enum attribute.</p> <p><math>malform(attribute(x, y)) \Leftarrow attribute(x, y), \neg type(y)</math>  <math>malform(enum(x, y)) \Leftarrow enum(x, y), \neg attribute(x, z)</math>  <math>malform(enum(x, y)) \Leftarrow enum(x, y), attribute(x, z), (z \neq \mathbf{enum})</math></p>
<p><b>Connector.</b> <math>connector(x)</math> denotes a connector named <math>x</math>. A connector is good (<math>conngood(x)</math>) if it has the appropriate edges, and every connector must be good.</p> <p><math>(connector, 1) \in \Upsilon, (conngood, 1) \in \Upsilon_C</math>  <math>conngood(w) \Leftarrow \left( \begin{array}{l} connector(x), assocEnd(x, y, \mathbf{src}), \\ assocEnd(x, z, \mathbf{dst}), association(x, w) \end{array} \right)</math>  <math>malform(connector(x)) \Leftarrow connector(x), \neg conngood(x)</math></p>

Table 13. Encoding of eMOF edge primitives

<p><b>Containment.</b> <math>containment(y, x)</math> denotes the containment relationship of <math>y</math> in <math>x</math>. A Containment edge must terminate on a <i>Class</i>. It may begin on another <i>Class</i> or <i>Association Class</i>.</p> <p><math>(containment, 2) \in \Upsilon</math>  <math>malform(containment(y, x)) \Leftarrow containment(y, x), \neg class(x)</math>  <math>malform(containment(y, x)) \Leftarrow containment(y, x), \neg class(y), \neg assocClass(y)</math></p>
<p><b>Attribute Containment.</b> <math>attrCont(y, x)</math> indicates an attribute containment relationship of <math>y</math> in <math>x</math>. An attribute containment edge must begin on an <i>Attribute</i>. It may terminate on a <i>Class</i> or <i>Association Class</i>.</p> <p><math>(attrCont, 2) \in \Upsilon</math>  <math>malform(attrCont(y, x)) \Leftarrow attrCont(y, x), \neg attribute(y)</math>  <math>malform(attrCont(y, x)) \Leftarrow attrCont(y, x), \neg class(x), \neg assocClass(x)</math></p>
<p><b>Association.</b> <math>association(x, y)</math> denotes an association relationship from <i>Connector</i> <math>x</math> to <i>Association Class</i> <math>y</math>. This is the only relationship allowed.</p> <p><math>(association, 2) \in \Upsilon</math>  <math>malform(association(x, y)) \Leftarrow association(x, y), \neg connector(x)</math>  <math>malform(association(x, y)) \Leftarrow association(x, y), \neg assocClass(y)</math></p>
<p><b>Association Endpoint.</b> <math>assocEnd(x, y, z)</math> indicates an association endpoint relation from <i>Connector</i> <math>x</math> to <i>Class</i> <math>y</math> with incidence <math>z</math>. <math>z</math> must a member of the closed unary relation <i>incidence</i>.</p> <p><math>\{(assocEnd, 3), (incidence, 1)\} \subset \Upsilon</math>  <math>\mathbf{restrict}(incidence, \{incidence(\mathbf{src}), incidence(\mathbf{dst})\})</math>  <math>malform(assocEnd(x, y, z)) \Leftarrow assocEnd(x, y, z), \neg connector(x)</math>  <math>malform(assocEnd(x, y, z)) \Leftarrow assocEnd(x, y, z), \neg class(y)</math>  <math>malform(assocEnd(x, y, z)) \Leftarrow assocEnd(x, y, z), \neg incidence(z)</math></p>
<p><b>Inheritance.</b> <math>inheritance(y, x)</math> indicates the inheritance relationship <math>y</math> inherits from <math>x</math>. An inheritance relationship can start and end on a <i>Class</i> or <i>Association Class</i>. There should be no directed cycles consisting only of inheritance edges.</p> <p><math>(inheritance, 2) \in \Upsilon, (imalform, 1) \in \Upsilon_C</math>  <math>malform(inheritance(y, x)) \Leftarrow inheritance(x, y), \neg class(x), \neg assocClass(x)</math>  <math>malform(inheritance(y, x)) \Leftarrow inheritance(x, y), \neg class(y), \neg assocClass(y)</math>  <math>malform(imalform(x)) \Leftarrow imalform(x)</math></p> <p>See Section III.4 for the definition of axioms concerning <i>imalform</i>.</p>

## BIBLIOGRAPHY

- [1] Dijkstra, E.W.: The humble programmer. *Commun. ACM* **15**(10) (1972) 859–866
- [2] D. Masys, D. Baker, A.B., Cowles, K.: Giving patients access to their medical records via the internet: the pcsso experience. *Journal of the American Medical Informatics Association* **9**(2) (2002) 181–191
- [3] Lee, E.A., Neuendorffer, S., Wirthlin, M.J.: Actor-oriented design of embedded hardware and software systems. *Journal of Circuits, Systems, and Computers* **12**(3) (2003) 231–260
- [4] Danvy, O.: An analytical approach to programs as data objects. Doctor scientarum thesis, BRICS, Department of Computer Science, University of Aarhus (2006)
- [5] Lee, E.A.: Absolutely positively on time: What would it take? *IEEE Computer* **38**(7) (2005) 85–87
- [6] C. Tomlin, S. Boyd, I.M.A.B.M.J., Xiao, L. In: Computational Tools for the Verification of Hybrid Systems. John Wiley (2003) In Software-Enabled Control, T. Samad and G. Balas (eds.).
- [7] Sastry, S., Sztipanovits, J., Bajcsy, R., Gill, H.: Scanning the issue - special issue on modeling and design of embedded software. *Proceedings of the IEEE* **91**(1) (2003) 3–10
- [8] Stärk, R., Schmid, J., Börger, E.: Java and the Java Virtual Machine - Definition, Verification, Validation. Springer (2001)
- [9] Samelson, K., Bauer, F.L.: Sequential formula translation. *Commun. ACM* **3**(2) (1960) 76–83
- [10] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1995)
- [11] Jackson, E.K., Sztipanovits, J.: Using separation of concerns for embedded systems design. *Proceedings of the Fifth ACM International Conference on Embedded Software (EMSOFT'05)* (September 2005) 25–34
- [12] Object Management Group: Data distribution service for real-time systems specification. Technical report (2005)
- [13] Srivastava, A., Eustace, A.: Atom: a system for building customized program analysis tools. In: PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation, New York, NY, USA, ACM Press (1994) 196–205
- [14] Heckmann, R., Langenbach, M., Thesing, S., Wilhelm, R.: The influence of processor architecture on the design and the results of wcet tools. *Proceedings of the IEEE* **91**(7) (2003) 1038–1054
- [15] Alur, R., Dill, D.L.: A theory of timed automata. *Theor. Comput. Sci.* **126**(2) (1994) 183–235
- [16] Henzinger, T.A., Kirsch, C.M.: A typed assembly language for real-time programs. In: EMSOFT. (2004) 104–113
- [17] Henzinger, T.A., Horowitz, B., Kirsch, C.M.: Giotto: a time-triggered language for embedded programming. *Proceedings of the IEEE* **91**(1) (2003) 84–99
- [18] Agha, G.: Actors: a model of concurrent computation in distributed systems. MIT Press, Cambridge, MA, USA (1986)
- [19] Kahn, G.: The semantics of simple language for parallel programming. In: IFIP Congress. (1974) 471–475
- [20] Lee, E.A., Parks, T.M.: Dataflow process networks. (May 1995) 773–799

- [21] G.E. Allen, B.E.: Real-time sonar beamforming on workstations using process networks and posix threads. In: *IEEE Transactions on Signal Processing*. Volume 48. (2000) 921–926
- [22] Zhou, Y., Lee, E.A.: A causality interface for deadlock analysis in dataflow. In: *EMSOFT*. (2006) 44–52
- [23] Gasieniec, L., Pelc, A., Peleg, D.: The wakeup problem in synchronous broadcast systems. *SIAM J. Discrete Math.* **14**(2) (2001) 207–222
- [24] Plotkin, G.D.: A structural approach to operational semantics. *J. Log. Algebr. Program.* **60–61** (2004) 17–139
- [25] Gurevich, Y.: *Evolving algebras 1993: Lipari guide*. (1995) 9–36
- [26] Barnett, M., Schulte, W.: *The abcs of specification: asml, behavior, and components*. *Informatica (Slovenia)* **25**(4) (2001)
- [27] K. Chen, J. Sztipanovits, S.N.M.E., Abdelwahed, S.: Toward a semantic anchoring infrastructure for domain-specific modeling languages. In: *Proceedings of the Fifth ACM International Conference on Embedded Software (EMSOFT’05)*. (September 2005)
- [28] Lee, E.A., Messerschmitt, D.G.: Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Computers* **36**(1) (1987) 24–35
- [29] Christopher Brooks, Edward A. Lee, X.L.S.N.Y.Z., Zheng, H.: *Heterogeneous concurrent modeling and design in java (volume 3: Ptolemy ii domains)*. Technical Report UCB/EECS-2007-9, EECS Department, University of California, Berkeley (January 11 2007)
- [30] Christopher Brooks, Edward A. Lee, X.L.S.N.Y.Z., Zheng, H.: *Heterogeneous concurrent modeling and design in java (volume 1: Introduction to ptolemy ii)*. Technical Report UCB/EECS-2007-7, EECS Department, University of California, Berkeley (January 11 2007)
- [31] Lee, E.A., Xiong, Y.: A behavioral type system and its application in ptolemy ii. *Formal Asp. Comput.* **16**(3) (2004) 210–237
- [32] Schach, S.R.: *Object-Oriented and Classical Software Engineering*. McGraw-Hill Pub. Co. (2001)
- [33] Peterson, J.L.: *Petri Net Theory and the Modeling of Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA (1981)
- [34] Aho, A.V., Ullman, J.D.: *Principles of Compiler Design (Addison-Wesley series in computer science and information processing)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1977)
- [35] Dijkstra, E.W.: Letters to the editor: go to statement considered harmful. *Commun. ACM* **11**(3) (1968) 147–148
- [36] Ogden, W.F.: A helpful result for proving inherent ambiguity. *Mathematical Systems Theory* **2**(3) (1968) 191–194
- [37] Kozen, D.: Partial automata and finitely generated congruences: An extension of Nerode’s theorem. In Crossley, J.N., Rammel, J.B., Shore, R.A., Sweedler, M.E., eds.: *Logical Methods: In Honor of Anil Nerode’s Sixtieth Birthday*. Birkhäuser, Ithaca, New York (1993) 490–511
- [38] Burris, S.N., Sankappanavar, H.P.: *A course in universal algebra*. Springer-Verlag (1981)
- [39] Jackson, E.K., Sztipanovits, J.: Towards a formal foundation for domain specific modeling languages. *Proceedings of the Sixth ACM International Conference on Embedded Software (EMSOFT’06)* (October 2006) 53–62



- [40] Object Management Group: Unified modeling language: Superstructure version 2.0, 3rd revised submission to omg rfp. Technical report (2003)
- [41] Institute For Software Integrated Systems: Gme 5 user’s guide. Technical report, Vanderbilt University (2005)
- [42] Object Management Group: Object constraint language v2.0. Technical report (2006)
- [43] G. Karsai, J. Sztipanovits, A.L.T.B.: Model-integrated development of embedded software. Proceedings of the IEEE **91**(1) (January 2003) 145–164
- [44] E. Magyari, A. Bakay, A.L.T.P.A.V.A.A.G.K.: Udm: An infrastructure for implementing domain-specific modeling languages. In: In the 3rd OOPSLA Workshop on Domain-Specific Modeling, OOPSLA 2003, Anaheim, California. (2003)
- [45] Emmelmann, H., Schröder, F.W., Landwehr, L.: Beg: a generation for efficient back ends. In: PLDI ’89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation, New York, NY, USA, ACM Press (1989) 227–237
- [46] Bozga, M., Graf, S., Ober, I., Ober, I., Sifakis, J.: The if toolset. In: SFM. (2004) 237–267
- [47] Behrmann, G., David, A., Larsen, K.G.: A tutorial on uppaal. In: SFM. (2004) 200–236
- [48] Hoare, T.: The verifying compiler: A grand challenge for computing research. J. ACM **50**(1) (2003) 63–69
- [49] Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G., eds.: Handbook of graph grammars and computing by graph transformation: vol. 2: applications, languages, and tools. World Scientific Publishing Co., Inc., River Edge, NJ, USA (1999)
- [50] Mens, T., Gorp, P.V., Varró, D., Karsai, G.: Applying a model transformation taxonomy to graph transformation technology. Electr. Notes Theor. Comput. Sci. **152** (2006) 143–159
- [51] Csertán, G., Huszerl, G., Majzik, I., Pap, Z., Pataricza, A., Varró, D.: Viatra - visual automated transformations for formal verification and validation of uml models. In: ASE. (2002) 267–270
- [52] Varró, D., Varró-Gyapay, S., Ehrig, H., Prange, U., Taentzer, G.: Termination analysis of model transformations by petri nets. In: ICGT. (2006) 260–274
- [53] Neema, S., Karsai, G.: Software for automotive systems: Model-integrated computing. In: ASWSD. (2004) 116–136
- [54] Sprinkle, J., Agrawal, A., Levendovszky, T., Shi, F., Karsai, G.: Domain model translation using graph transformations. In: ECBS. (2003) 159–167
- [55] Wolf, W.: Computers as components: principles of embedded computing system design. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2001)
- [56] Gschwind, M., Hofstee, H.P., Flachs, B.K., Hopkins, M., Watanabe, Y., Yamazaki, T.: Synergistic processing in cell’s multicore architecture. IEEE Micro **26**(2) (2006) 10–24
- [57] Stein, L.: Challenging the computational metaphor: Implications for how we think (1999)
- [58] Rong Chen, Marco SgROI, G.M.L.L.A.S.V.J.R.: Embedded system design using uml and platforms. In: Proceedings of Forum on Specification and Design Languages 2002 (FDL’02). (September 2002)
- [59] Douglas Densmore, Roberto Passerone, A.S.V.: A platform-based taxonomy for esl design. IEEE Design and Test of Computers **23**(5) (September 2006) 359– 374

- [60] Balarin, F., Watanabe, Y., Hsieh, H., Lavagno, L., Passerone, C., Sangiovanni-Vincentelli, A.L.: Metropolis: An integrated electronic system design environment. *IEEE Computer* **36**(4) (2003) 45–52
- [61] Lee, E.A., Neuendorffer, S.: Actor-oriented models for codesign: Balancing re-use and performance. *Formal Methods and Models for Systems*, Kluwer (2004)
- [62] F. Balarin, Y. Watanabe, H.H.L.L.C.P.A.L.S.V.: Metropolis: an integrated electronic system design environment. *IEEE Computer* **36**(4) (April 2003)
- [63] Lee, E.A.: The problem with threads. Technical Report UCB/EECS-2006-1, EECS Department, University of California, Berkeley (January 10 2006)
- [64] A. L. Sangiovanni-Vincentelli, L. Carloni, F.D.B., Sgroi, M.: Benefits and challenges for platform-based design. In: *Proceedings of the Design Automation Conference (DAC'04)*. (June 2004)
- [65] J. Burch, R. Passerone, A.S.V.: Modeling techniques in design-by-refinement methodologies. In: *Integrated Design and Process Technology*. (June 2002)
- [66] Lee, E.A., Messerschmitt, D.G.: Synchronous data flow. *Proceedings of the IEEE* **75**(9) (1987) 1235–1245
- [67] Bezivin, J., et al.: Towards a precise definition of the omg/mda framework (2001)
- [68] Plotkin, G.D.: A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus (1981)
- [69] Hamon, G., Rushby, J.: An operational semantics for stateflow. *Proceedings of the conference on Fundamental Approaches to Software Engineering (FASE'04) Volume 2984 of LNCS* (2005) 229–243
- [70] Gurevich, Y.: Evolving algebra: An attempt to discover semantics. *EATCS* **43** (1991) 264–284
- [71] Hamon, G.: A denotational semantics for stateflow. *Proceedings of the Fifth ACM International Conference on Embedded Software (EMSOFT'05)* (September 2005) 164–172
- [72] G. Karsai, A. Agrawal, F.S.: On the use of graph transformations for the formal specification of model interpreters. *Journal of Universal Computer Science* **9**(11) (November 2003) 1296–1321
- [73] Csertan, G., Huszerl, G., Majzik, I., Pap, Z., Pataricza, A., Varro, D.: Viatra: Visual automated transformations for formal verification and validation of uml models (2002)
- [74] Kuehne, T.: Matters of (meta-) modeling. *Software and Systems Modeling (SoSyM)* **5**(4) (December 2006) 369–385
- [75] Harel, D., Rumpe, B.: Meaningful modeling: What's the semantics of "semantics"? *IEEE Computer* **37**(10) (2004) 64–72
- [76] The Eclipse Modeling Framework: [www.eclipse.org/emf/](http://www.eclipse.org/emf/). Technical report
- [77] Object Management Group: Meta object facility specification v1.4. Technical report (2002)
- [78] Object Management Group: Meta object facility (mof) 2.0 query/view/transformation specification. Technical report (2005)
- [79] Microsoft DSL Toolkit: <http://msdn2.microsoft.com/en-us/vstudio/aa718368.aspx>. Technical report
- [80] Heckel, R., Kuster, J., Taentzer, G.: Confluence of typed attributed graph transformation systems (2002)

- [81] Ehrig, H., Prange, U., Taentzer, G.: Fundamental theory for typed attributed graph transformation. In: ICGT. (2004) 161–177
- [82] Drewes, F., Hoffmann, B., Plump, D.: Hierarchical graph transformation. *J. Comput. Syst. Sci.* **64**(2) (2002) 249–283
- [83] Guttag, J.: Abstract data types and the development of data structures. *Commun. ACM* **20**(6) (1977) 396–404
- [84] Burris, S.N., Sankappanavar, H.: *A Course in Universal Algebra*. Springer-Verlag (1981)
- [85] E. Lee, A.S.V.: A unified framework for comparing models of computation. *IEEE Trans. on Computer Aided Design of Integrated Circuits and Systems* **17**(12) (December 1998) 1217–1229
- [86] T. A. Henzinger, C. M. Kirsch, M.A.S., Pree, W.: From control models to real-time code using giotto. *Control Systems Magazine* **2**(1) (2003) 50–64
- [87] de Alfaro, L., Henzinger, T.: Interface-based design. In: *In Engineering Theories of Software Intensive Systems, proceedings of the Marktoberdorf Summer School*, Kluwer, Kluwer (2004)
- [88] A. Benveniste, P. Caspi, S.E.N.H.P.L.G., de Simone, R.: The synchronous languages twelve years later. *Proceedings of the IEEE* **91**(1) (2003) 64–83
- [89] Jackson, E.K., Sztipanovits, J.: Constructive techniques for meta- and model-level reasoning. In: under review. (2007)
- [90] Weijland, W.P.: Semantics for logic programs without occur check. In: *ICALP*. (1988) 710–726
- [91] Minker, J.: Logic and databases: A 20 year retrospective. In: *Logic in Databases*. (1996) 3–57
- [92] Chan, D.: An extension of constructive negation and its application in corouting. In *Proceedings of NACLPL*, The MIT Press (1989) 447–493
- [93] Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: *ICLP/SLP*. (1988) 1070–1080
- [94] Dantsin, E., Eiter, T., Gottlob, G., Voronkov, A.: Complexity and expressive power of logic programming. *ACM Comput. Surv.* **33**(3) (2001) 374–425
- [95] AĭT-KACI, H.: *Warren’s Abstract Machine: A Tutorial Reconstruction*. MIT Press (1991)
- [96] Object Management Group: *Meta object facility (mof) 2.0 core specification*. Technical report (2003)
- [97] Chan, D.: Constructive negation based on the complete database. In: *Proc. Int. Conference on LP’88*, The MIT Press (1988) 111–125
- [98] Emerson, M., Sztipanovits, J.: Techniques for metamodel composition. *OOPSLA 2006 Domain-Specific Languages Workshop* (2006)
- [99] Rong Chen, Marco Sgroi, G.M.L.L.A.S.V.J.R.: Embedded system design using uml and platforms. In: *Proceedings of Forum on Specification and Design Languages 2002 (FDL’02)*. (September 2002)
- [100] Laddaga, L.: Creating robust software through self-adaptation. *IEEE Intelligent Systems and Their Applications* **14** (May-June 1999) 26–29
- [101] Gößler, G., Sifakis, J.: Composition for component-based modeling. *Sci. Comput. Program.* **55**(1-3) (2005) 161–183
- [102] de Alfaro, L., Henzinger, T.A.: Interface theories for component-based design. In: *EMSOFT*. (2001) 148–165

- [103] Thiele, L., Wandeler, E., Stoimenov, N.: Real-time interfaces for composing real-time systems. In: EMSOFT. (2006) 34–43
- [104] Neema, S., Lédeczi, Á.: Constraint-guided self-adaptation. In: IWSAS. (2001) 39–51
- [105] Whisnant, K., Kalbarczyk, Z., Iyer, R.K.: A system model for dynamically reconfigurable software. IBM Systems Journal **42**(1) (2003) 45–59
- [106] Bondorf, A., Palsberg, J.: Generating action compilers by partial evaluation. J. Funct. Program. **6**(2) (1996) 269–298
- [107] Gupta, A., Forgy, C., Newell, A.: High-speed implementations of rule-based systems. ACM Trans. Comput. Syst. **7**(2) (1989) 119–146