The Software Engineering of Domain-Specific Modeling Languages: A Survey Through Examples

Ethan K. Jackson
ejackson@isis.vanderbilt.edu

Technical Report ISIS-07-807

Institute For Software Integrated Systems

Vanderbilt University, Nashville, TN

March 2007

Abstract— **This paper presents the fundamental concepts of model-based design to the broader software engineering community. We examine model-based design from the perspective of *domain-specific modeling languages* (DSMLs). DSMLs capture the structure, behavioral characteristics, and abstractions of complex problem domains. Model transformations defined between language syntaxes serve as high-level specifications of domain-specific compilers. Additionally, transformations are used to change abstraction levels. This paper is example driven and includes examples from a number of tools including ASML [1], Ptolemy II [2], GME [3], and GReAT [4].**

Index Terms— **model-based design, domain-specific modeling languages, structural semantics, model of computation, model transformations**

I. 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 shear scale of software continues to grow, both on a per node basis, and in 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 [5]. 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 generic 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 [2] argue that all of these perspectives are essentially the same. Nonetheless, it is useful to think it terms of one (or more) of these metaphors. In this paper we focus on the view that an application context is 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. The purpose of this paper 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 extensions to 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 3 presents the formal foundations of DSML semantics. Section 4 describes how DSMLs programs can be executed (simulated) on traditional machines. Section 5 examines syntax and compiler construction. We conclude in Section 6. Finally, we emphasize concrete code examples, providing the reader with tangible snapshots of a number of model-based tools.

II. 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. In this view, programming languages allows us to *represent* complex dynamical systems in a compact form as syntactically correct data [6].

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 [7]. 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 [8]. 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 this problem 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" [9]. 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 procedure: 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 [10]. 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 [11], and patterns, e.g. the visitor pattern [12], 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 Djikstra and Hoare, argued for both these properties, and history bares their mark [13]. The model-based community uses syntaxes to filter out behaviorally incorrect models, or as a first-pass before verification [14]. The trade-off is often made that syntaxes with high parsing complexity parsing 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.

III. 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 [15]):

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 delived 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 requirement 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.

A. 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 [16]. 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 [17]. For example, the precise cache replacement policy affects 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. A successful approach has been to change the underlying machine model to include a precise notion of time. Programs, which are just data interpreted by the machine, define a dynamical system 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 Σ_i and output alphabet Σ_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 $s \overset{\alpha}{\rightarrow} s'$ indicates that the system transitions to s' when it is in state s and observes α.
5) $\Pi : Q \rightarrow \Sigma_o$ is a mapping from states to observations.

In this case, we can imagine that the input alphabet Σ_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 $s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_n$ and we observe a dynamical system that looks like the sequence $\Pi(s_0), \Pi(s_1), \ldots, \Pi(s_n)$.

The timed-automaton extends this model, allowing states to modulate *clocks*, which count the passage of time [18]. 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 $(v(c)\ \mathbf{op}\ q)$ is valid in $\mathcal{R}$, where $\mathbf{op} \in \{<, \leq, >, \geq\}$. 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 Σ_i and output alphabet Σ_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) $E \subseteq V \times \Sigma_i^\epsilon \times \Phi(X) \times \mathcal{P}(X) \times V$ 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) $\Pi : V \to \Sigma_o$ 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) $\mathbf{add}\ R_i, R_j, R_k$ causes $R_k \leftarrow R_i + R_j$
2) $\mathbf{mul}\ R_i, R_j, R_k$ causes $R_k \leftarrow R_i \times R_j$
3) $\mathbf{load}\ R_i, C$ causes $R_i \leftarrow C$, where C is a data value

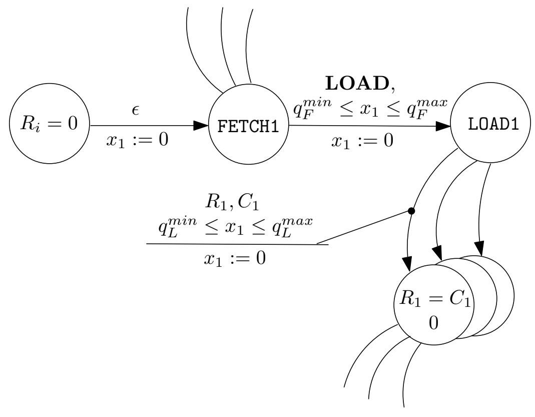


Fig. 1. An abstract machine with a precise notion of time.

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 $R_1 = C_1, R_{i \neq 1} = 0$.

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^{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 ($f \equiv d_0, d_1, \ldots, d_n$), and then augment the basic **CALL** instruction with a requested timing range: $\mathbf{CALL}\ f\ [t^{min}, t^{max}]$. 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 [19]. The E machine includes assembly instructions that start periodic tasks ($\mathbf{schedule}\ j$, for a task j) and suspend tasks for a specified amount of time ($\mathbf{future}\ n, a_j$, 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 [20].

B. 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 [21]. 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

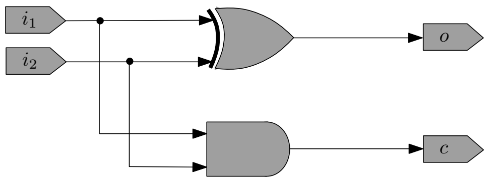


Fig. 2. Example of concurrency in hardware notations.

arrive on the inputs 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.) Systems like these 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 [22]. 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 processes 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 [23]. Let Σ be an alphabet containing the possible data values that appear on connections. The set Σ^* contains all finite sequences of data (*Kleene closure* of Σ), and the set $\Sigma^{\mathcal{Z}_+} = \{f | f : \mathcal{Z}_+ \rightarrow \Sigma\}$ contains all infinite sequences of data. Let $S = \Sigma^* \cup \Sigma^{\mathcal{Z}_+}$ be the set of all finite and infinite sequences of data tokens. A process $P : S \rightarrow S$ maps sequences to sequences.

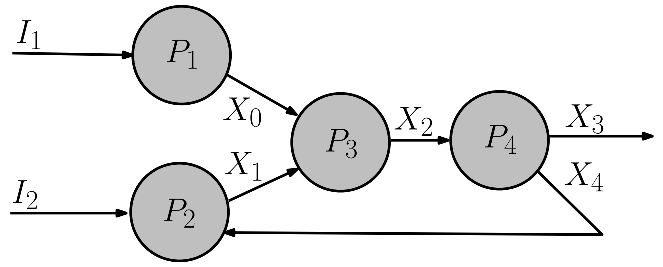
The internal state of a process can be completely abstracted away by defining the mapping appropriately. Consider the classic example of a system that remembers if it has seen an even or odd number of a particular input a. An automaton would do this using at least two states. A process has access to 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^{th} element in the sequence $P_{eo}(S)$ is a if there are an even number of a tokens in the input subsequence $[s_0, s_1, \ldots, s_i]$. Otherwise, the i^{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 $s_i = s'_i, 0 \leq i < len(S)$. [1] The empty sequence $\perp$ is a prefix of every sequence. A process P is *monotonic* if $X \sqsubseteq Y$, 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 $C = \{X_0 \sqsubseteq X_1 \sqsubseteq \ldots\}$ 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 process) $\pi_{i,n} : S^n \rightarrow S$ that extracts the i^{th} sequence from an n-tuple of sequences, i.e. $\pi_{i,n}((S_0, S_1, \ldots, S_{n-1})) \mapsto S_i$. With these definitions, we can view a network of interacting processes as just a set of constraints over the sequences that the processes produce. The particular input sequences are fixed, and the solution to the network is a set of internal/output sequences $\{X_0, X_1, \ldots, X_{n-1}\}$ that satisfy the constraints. Figure 3 shows an example of a process network 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 and $\mathbf{X} = \{X_0, X_1, \ldots, X_{n-1}\}$.

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



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

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

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. $\mathbf{X}^{j+1} = F(\mathbf{I}, \mathbf{X}^j)$. 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 $\mathbf{X} = F(\mathbf{I}, \mathbf{X})$. Amazingly, this constructive process can be implemented by concurrently running processes that send data across the channels until the entire network stabilizes [24]. 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 [25].

Comparing process networks with automata shows that there are some advantanges 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 even is instantaneously observed by all other automata in the system. Implementing synchronous broadcast requires sophisticated distributed algorithms [26]. 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.

IV. 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 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 bare 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.

A. 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 [27]. 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) Γ 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'.

In Plotkin's influential notes on *structural operational semantics*(SOS) [27], he enumerates a number of such reformulations. Interestingly, reformulating an arbitrary structure into a transition system requires generalizing the notation of state. For example, Plotkin points out that an FSA can made into a transition system if $\Gamma = Q \times \Sigma_i^*$, where Σ_i^* is the set of all finite strings over Σ_i. Squeezing an FSA in a TS yields a TS with an infinite number of configurations ($|\Gamma| = |Q| + |\aleph_0|$), even though Q is a finite set. Correctly defining the notion of configuration is essential to applying the TS formalism. 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) [28], generalized the notion of configuration so that it could encompass many different structures. Specifically, he proposed that configurations should be *algebras* over a fixed signature Υ, 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 Υ is called the *signature* of the algebra. A signature names a set of operations (function symbols) $f_1, f_2, \ldots, f_n$, 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 Υ. An abstract state machine A over (U, Υ) 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 $t \in U^{arity(f_i)}$ 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* [1] allows ASMs to be characterized by a set of statements of the form:

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

If the current state is s, and $f_i^s(x_1, x_2, \ldots, x_{arity(f_i)})$ 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 $(y_1, y_2, \ldots, y_{arity(f_j)})$ 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 $l = (y_1, y_2, \ldots, y_{arity(f_j)})$, then:

$$(f_i^s(x_1, x_2, \ldots, x_{arity(f_i)}) = \mathbf{true}) \Rightarrow (s, s') \in \rightarrow,$$
$$\text{where } s' = \begin{cases} f^{s'}_{k \neq j} = f^s_k \\ f^{s'}_j(l' \neq l) = f^s_j(l') \\ f^{s'}_j(l) = u \end{cases} \text{, and } s' \in \Gamma \qquad (1)$$

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

[2]Actually, the type `Double` is 64-bit floating point.

```
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
```

Spec 1. Extending the universe U for timed-automata

Finally, the *Clocks* subuniverse contains names for clock variables. We call these *reserviors*, 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 **Z**. However, we can rely on U to contain the needed elements. Our usage of the term reservior is similar in spirit to the usage of the term *reserve* in [28]. A reserve contains names for objects that may be dynamically introduced into a running ASM.

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 evaluations 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 Υ. 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 Υ. The valuation v is a unary function that

```
12: var v = { clki → 0.0 | clki in clocks }
13: var crnt = any qi | qi in q0
14: const time = new Transition(empty,e,{→},{},empty)
```

Spec 2. Extending the signature Υ for timed-automata

maps each clock to a value, capturing the temporal state of the system. Line 12 initializes v to map 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 ϵ. 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 functions v and *crnt* change. 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

```
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 := { clki → v(clki)+epsilon | clki in clocks }
23:     else crnt := takeTrans.lp
24:        v := { clki → 0.0 | clki in takeTrans.r } union
25:        { clki → v(clki) | clki in clocks − takeTrans.r }
```

Spec 3. Kernel of simulation engine

enabled transitions, 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 (α, τ) 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 (α, τ) 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 ϵ 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*.

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

Spec 4. Main simulation loop

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 product 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 1 at once. Thus, the specification does not require discrete state 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 on 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 [29], this file is called the *abstract data model*. Data Model 1 lists the data model for the simple automaton of Figure 4. Lines

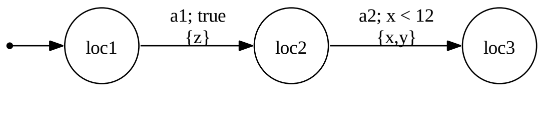


Fig. 4. Timed-automaton represented by data model 1.

```
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 }
```

Model 1. The data model for a simple timed-automaton

9-11 define the *UpdateGuardMaps* for this specific data model. Otherwise, the data model[3] is quite close to the original mathematical definition of a timed-automaton in Section II.B. After a data model and simulator have been combined, the model can be immediately simulated. Figure 3 shows a single partial simulation trace.

```
{ 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
```

Fig. 5. Simulating Data Model 1 with ASML specification

[3]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.

B. 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. 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 to 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 [30]. Statically schedulable networks correspond to networks where the actual data values do not significantly impact how much data the processes consume and produce. Contrarily, *dynamically schedulable* networks may adjust how many 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 [31].

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. For example, [23] gives 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, but programmatically it is not easily specified. The heterogeneous modeling and simulation framework *Ptolemy II* addresses these issues by providing an *abstract semantics* for process network simulation [2][4]. 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, $(r_0, r_1, \ldots, r_m)$ where m is the number of inputs exposed by an actor. A rule is satisfied by an m-tuple of input sequences $(I_0, I_1, \ldots, I_m)$ 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 $\mathbf{R} = ([*,*], \bot, [1])$. 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 $\{\mathbf{R}_1, \mathbf{R}_2, \ldots, \mathbf{R}_k\}$ 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 [32]. This class introduces a number of important methods that a simulator can use to gather information about the actor. The *initialize* method 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 interfaces of actors 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));
```

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 without the actor remembering these tests. Unfortunately, not all actors can be

[4]Ptolemy II also supports the simulation of discrete and hybrid automata.

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 [33] 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* [34]. 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 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 is 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 $\{floor_{min}, \ldots, floor_{max}\}$. 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.

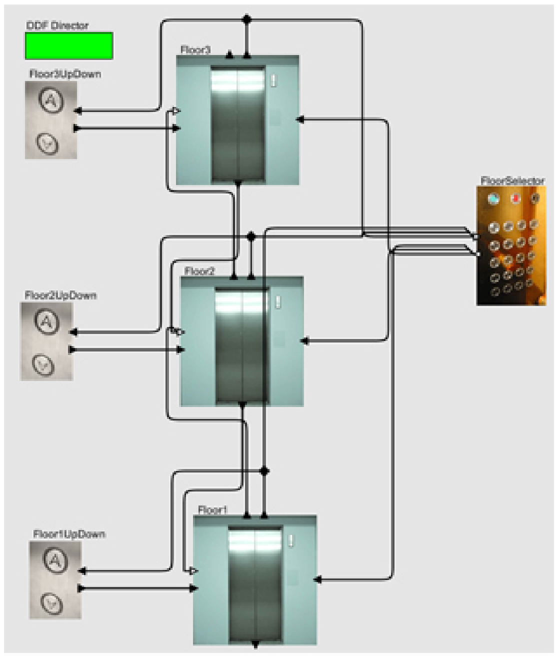


Fig. 6. The Elevator problem in Ptolemy II

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 $[dir, f_1, f_2, \ldots, f_m]$. 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 last went. 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 most come from below. The firing rule for this state is $(\bot, [*])$. If the elevator was seeing going up, then the next time it will come from about, so the firing rule is $([*], \bot)$. If the floor has the elevator, then it will send it out with no inputs, i.e. $(\bot, \bot)$. The elevator starts at the bottom floor, so initially the first firing rule will always apply. We subclass the $intialize$ member to contain:

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

Similary, 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 presssed", "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 the same way as the up/down panels. We connect a single floor selector actor the 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 request 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[5] 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

[5]This is the dynamic data flow (DDF) director.

```
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
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
```

Fig. 7. Simulating the elevator problem in Ptolemy II

[35], which can be considered to be a class of process networks with a particular firing rule.

V. 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 [36].

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.

A. 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 [13]. 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 used by mainstream languages. Furthermore, restrictions on the BNF grammars lead to efficient parser implementations (e.g. LALR, shift-reduce parsers) [36]. These technologies have solidified themselves as the de facto approach for syntax design. Consequently, most language evolution occurs on the semantic side. For example, even Djikstra's famous argument against `goto` statements is an argument on the semantics of `goto`; not its syntactic representation [37].

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

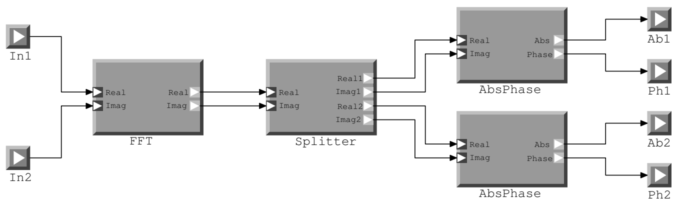


Fig. 8. Example of a model represented as a directed graph

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 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, an consider strings that list the edges of a graph in an arbitrary order. Figure 9 shows

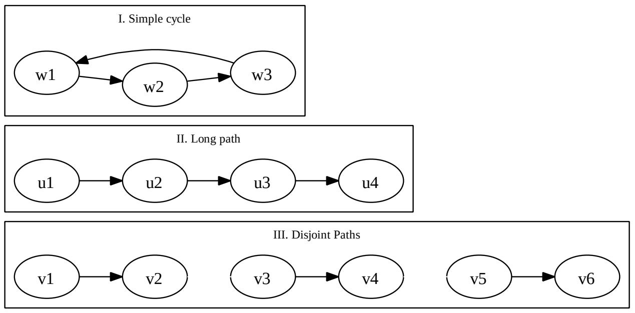


Fig. 9. Several digraphs; graph I is not in the language, but II and III are in $\mathcal{L}_{DAG}$.

several example graphs. A digraph is encoded as a string by arbitrarily ordering the edge relation, and then listing the vertices incident on each edge. For example, the simple cycle of 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 $v_1v_2\ v_3v_4 \ldots v_{n-1}v_n$ corresponds so a set of disjoint 2-paths; 9.III is an example of such a graph. Let V be a set of vertices, a 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) $\forall u \in \mathcal{L}_{DAG}(V),\ 2 \,\big|\, |u|$
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 $E(u) = \bigcup_{i=1}^{\frac{|u|}{2}} (u_{2i-1}, u_{2i})$.

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

It is easy to see that this language will *not* fail the Pumping Lemma, or extensions thereof [38]. 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 acylic graph with more than 3 edges ($n = 6$) into three parts: $x = u_1u_2$, $y = u_3u_4$, $z = u_5 \ldots u_{|u|}$. In this case, duplicating the edge u_3u_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. There are several ways this can be done. 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 [39]. In order to capture the language of all finite directed acyclic graphs, we choose the vertex set to be a countably infinite set Δ ($|\Delta| = |\aleph_0|$). 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 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. 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 Σ is an algebra with universe U whose elements are generated by Σ, 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 ϵ called the *identity element*, and satisfies the following axioms:

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

In this case, ϵ 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* [40] of an arbitrary signature Υ. Without delving into the technical details, we associate a set of operators with the concepts of the language [41]. For example, directed graphs utilize vertices and an 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 Σ. 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 as a string.

Just as in 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 push-down 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.

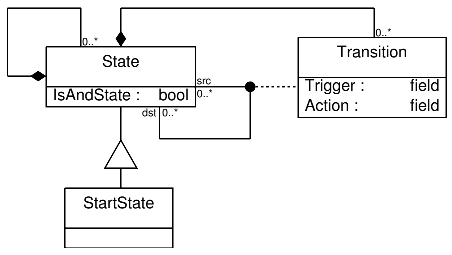


Fig. 10. Example metamodel for hierarchical finite state machines.

These extensions provide a formal underpinning for the syntax of models, but they do *not* suggest a 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* [42]. 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* [43]. 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 (●) 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 (◇) 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`.

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 [44]. 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 passed out, 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 [3]. Figure 11.I shows the

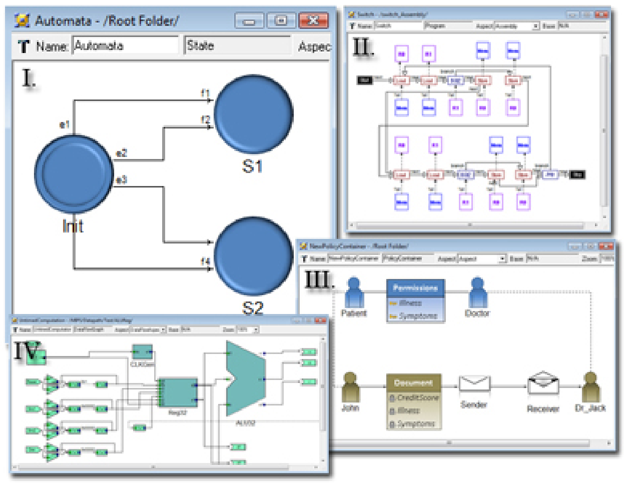


Fig. 11. Example models in the metaprogrammable modeling environment GME: I. FSA model II. Assembly code model III. Access control model IV. Synchronous dataflow model

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.

B. 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 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/interpreter. 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 projects model elements onto instances of three classes: *Atom, Model,* and *Connection*. (BON also provides classes for additional structural features, but these are beyond the scope of this paper.) 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-type 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.

```
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);
   }
}
```

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 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) [45]. 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 [46] 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 this setting 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 [47] or Uppaal [48]. 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 important 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 [49]. 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 (syntax). A model transformation is a mapping $\tau : \mathcal{M} \rightarrow \mathcal{M}'$. Notice that a compiler S can be viewed as a model transformation τ_S. If the compiler S generates C code, then the codomain of τ_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 [46]. 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 [50]. 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 [51].

A model transformation may be viewed as a set of graph rewriting rules. A mapping τ 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. Some approaches to analysis of graph rewriting systems can be found in [52] [53].

Model transformations have been successfully applied to a number of DSMLs. A particularly relevant example was presented in [54] where the authors developed a domain-specific compiler from the well-established Stateflow/Simulink paradigm 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 [4], which was used by the authors of [54]. 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 all the elements of a particular FSA. These elements are 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 Σ. The attribute `guard` of a transition is a textual field that names one of the `Event` instances.

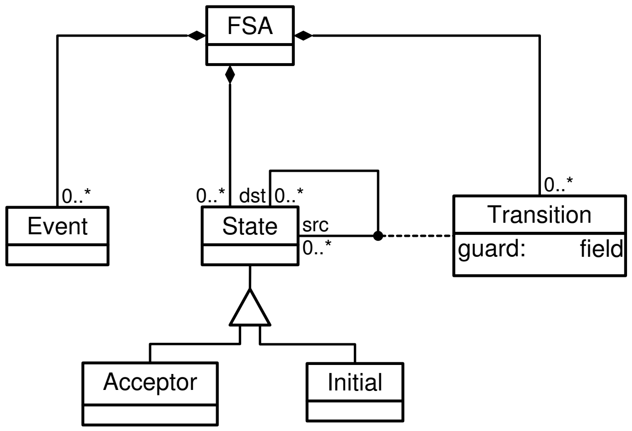


Fig. 12. Metamodel of the input language; a finite state acceptor (FSA) languge.

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]. Algorithm 1 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 4). The program must have a mechanism to read events from the environment;

```
 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: }
```

Algorithm 1. Pseudocode for executing a finite state acceptor.

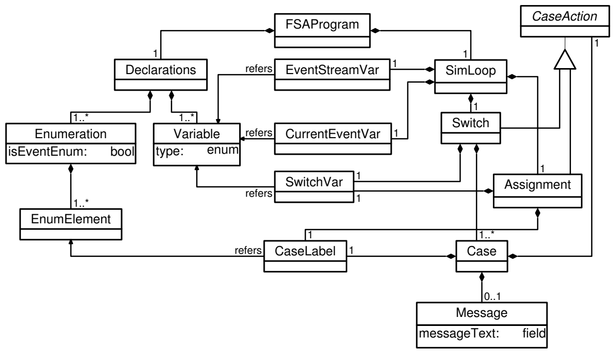
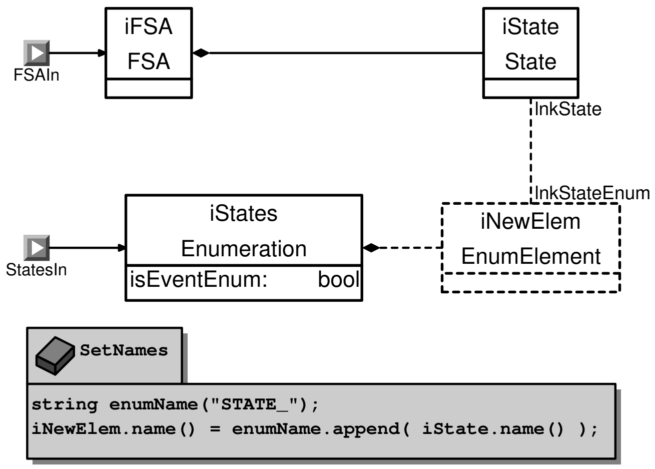


Fig. 13. Metamodel of the output language; a structured subset of C.

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 $S1 \overset{Ei}{\rightarrow} Si$, then the code would have the inner **case** shown in Line 11. This case correctly updates the *currentState* to the new state *Si* (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 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



```
string enumName("STATE_");
iNewElem.name() = enumName.append( iState.name() );
```

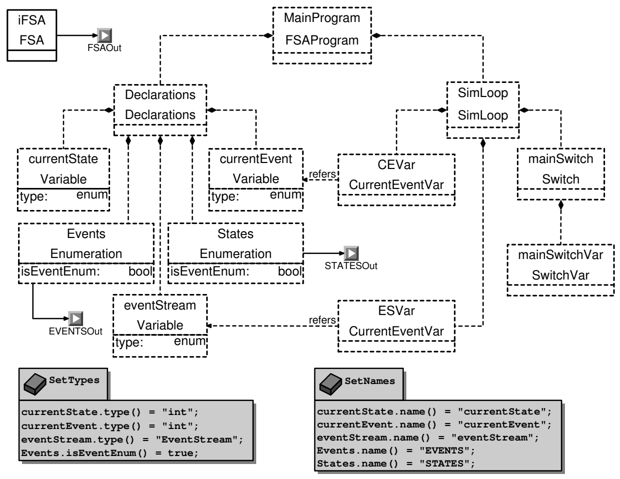
Fig. 14. Graph rewriting rule that creates the STATE enumeration elements.

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 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 instances 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 drawn in dotted lines represent objects that are added to input/output graphs. In this case, for each state in the FSA a corresponding *EnumElement* is instantiated an 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 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 *Build-*



```
currentState.type() = "int";
currentEvent.type() = "int";
eventStream.type() = "EventStream";
Events.isEventEnum() = true;
```

```
currentState.name() = "currentState";
currentEvent.name() = "currentEvent";
eventStream.name() = "eventStream";
Events.name() = "EVENTS";
States.name() = "STATES";
```

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

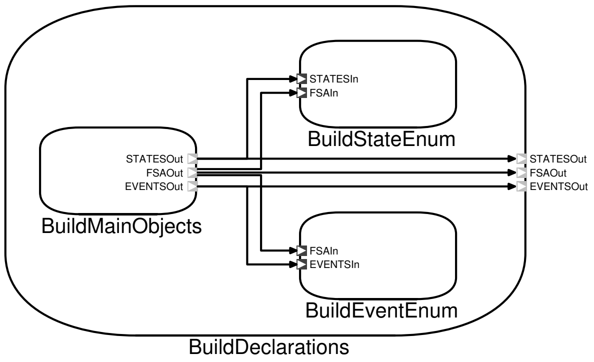


Fig. 16. Sequencing and encapsulation of the graph rewriting rules as a block.

MainObjects 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 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 the appendix.

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 *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. The appendix 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 $\tau_{A,B} : \mathcal{M}_A \to \mathcal{M}_B$, $\tau_{B,C} : \mathcal{M}_B \to \mathcal{M}_C$, we may construct a new map $\tau_{A,C} : \mathcal{M}_A \to \mathcal{M}_C$ 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.

VI. 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, spread sheets) 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, the 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] projects 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 an *Nack*. Similarly, after the sender has sent the

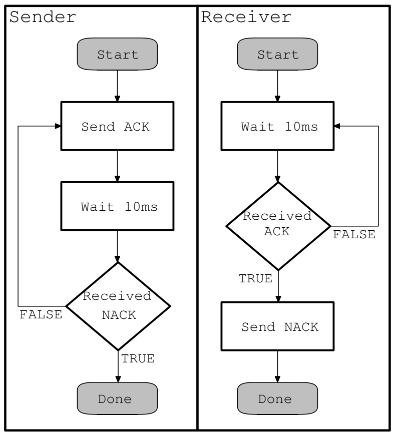


Fig. 17. Simple example of a communication protocol.

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 receive both reach the *Done* box 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 is first technique is doomed to 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 (HFSMs) provide a more precise characterization of the concurrency in the model. This might lead us to diagram in Figure 18.

This HFSM 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, and not one, 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 more or less correct. 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 HFSM 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 (syntax) 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

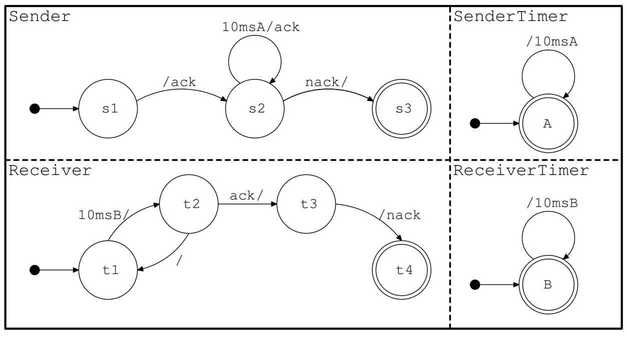


Fig. 18. One possible representation of the protocol using HFSMs.

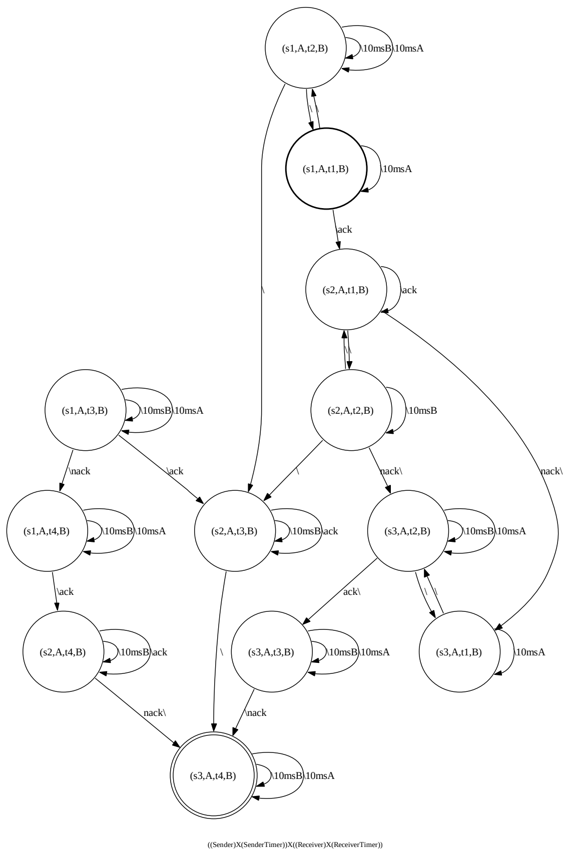


Fig. 19. Interpretation of the specification assuming synchronous product.

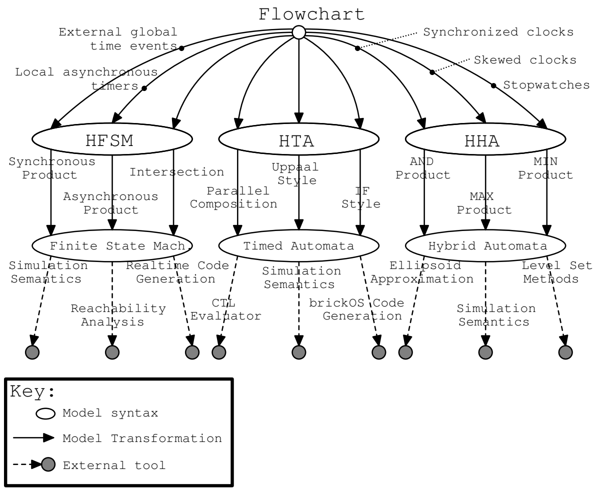


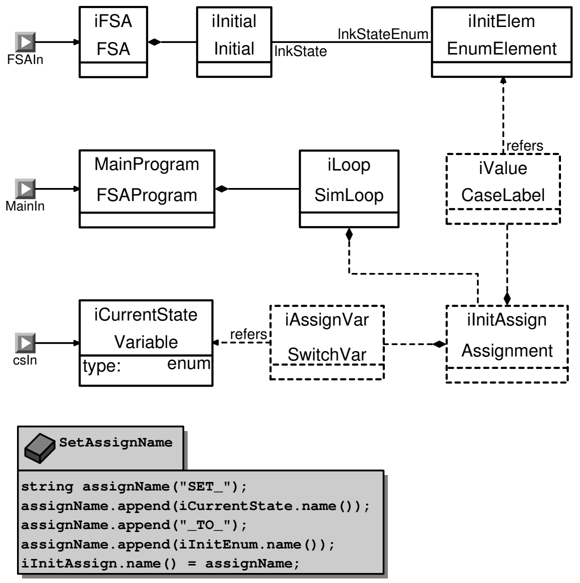
Fig. 20. Model-based view of the meta-level design choices.

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, the 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 hierarhical 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 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 can explore the impact of meta-level choices by choosing a sequence of transformations $\tau_1, \tau_2, \ldots, \tau_n$ where the domain of τ_1 is the flowchart language and the codomain of τ_n is the syntax of simulation instructions for a simulator. Then, by composition, $(\tau_n \circ \ldots \circ \tau_2 \circ \tau_1)m$ 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 τ_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 useful landscape of the related abstraction layers.

Appendix
Transformation from FSA to C

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 intial state. Figure 21 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.

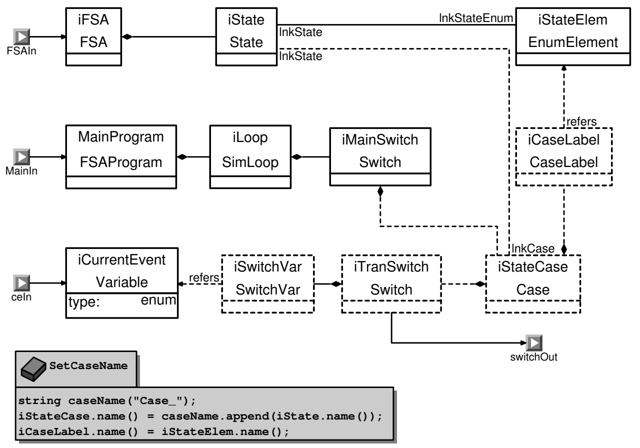


```
string assignName("SET_");
assignName.append(iCurrentState.name());
assignName.append("_TO_");
assignName.append(iInitEnum.name());
iInitAssign.name() = assignName;
```

Fig. 21. 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 22 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*

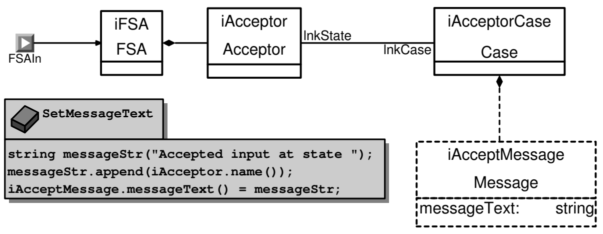


```
string caseName("Case_");
iStateCase.name() = caseName.append(iState.name());
iCaseLabel.name() = iStateElem.name();
```

Fig. 22. Creates a *case* and empty *switch* block for each *State* instance.

must assign s' to the *currentState* variable. Figure 27 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 28 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.

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 22) to find the *case* block associated with each acceptor state, i.e. instance of *Acceptor*. Figure 23 shows this rule.



```
string messageStr("Accepted input at state ");
messageStr.append(iAcceptor.name());
iAcceptMessage.messageText() = messageStr;
```

Fig. 23. Builds feedback messages

Figure 29 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 in *BuildStateEnum* and *BuildEventEnum* rules inside of the *BuildDeclarations* rule.

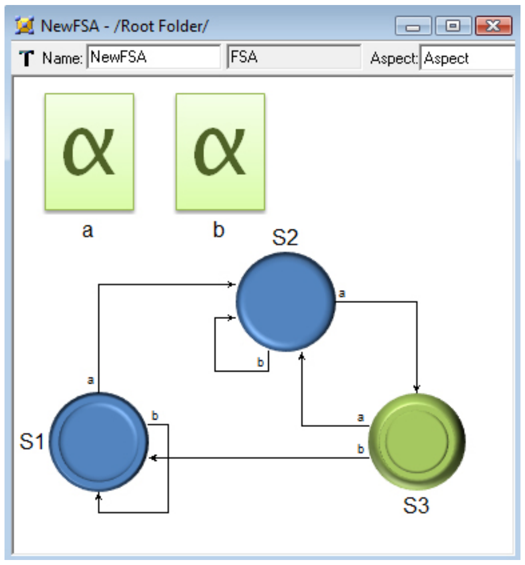


Fig. 24. Example FSA acceptor

Figure 24 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 25 shows the result of applying the transformation to the FSA in Figure 24. 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 subsection. In the interest of space the *EventStream* class is not shown. Finally, Figure 26 shows the result of executing the FSA with some input. Note that the "accept" message is delayed by one event.

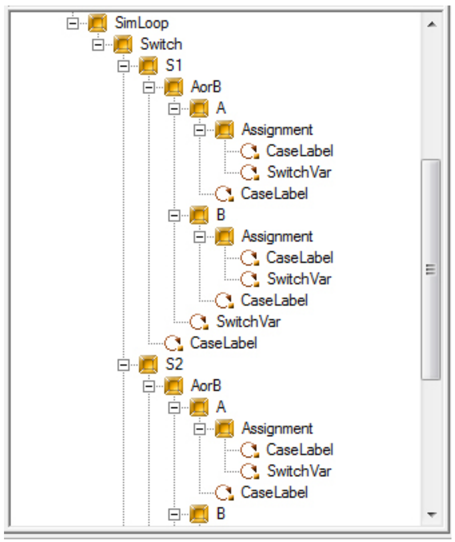


Fig. 25. Partial C abstract syntax tree generated from example FSA.

```
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]:
```

Fig. 26. Simulation of generated FSA model

A. C Code generated from FSA

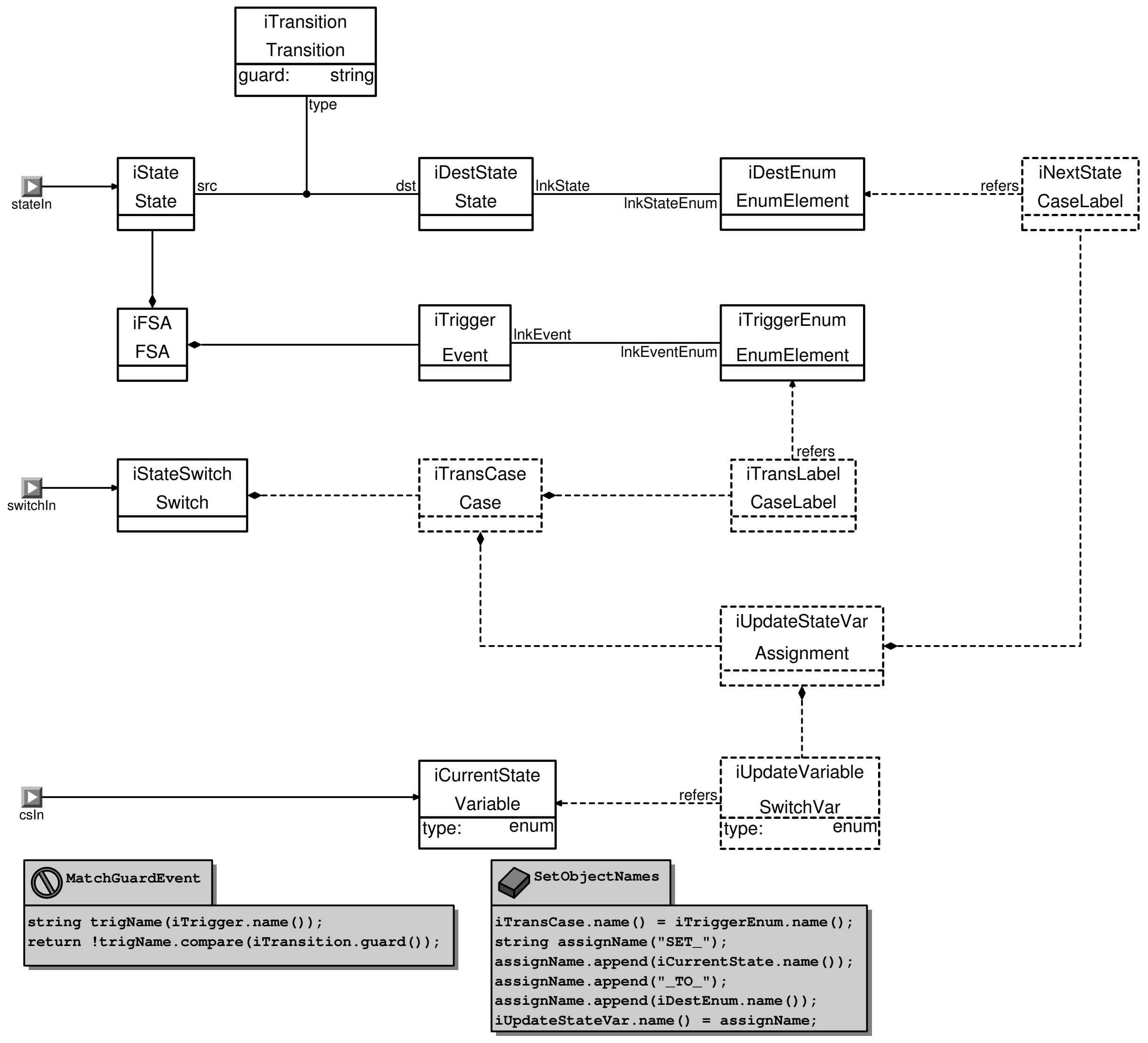
```
#include "EventStream.h"

// ------- 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) {
```

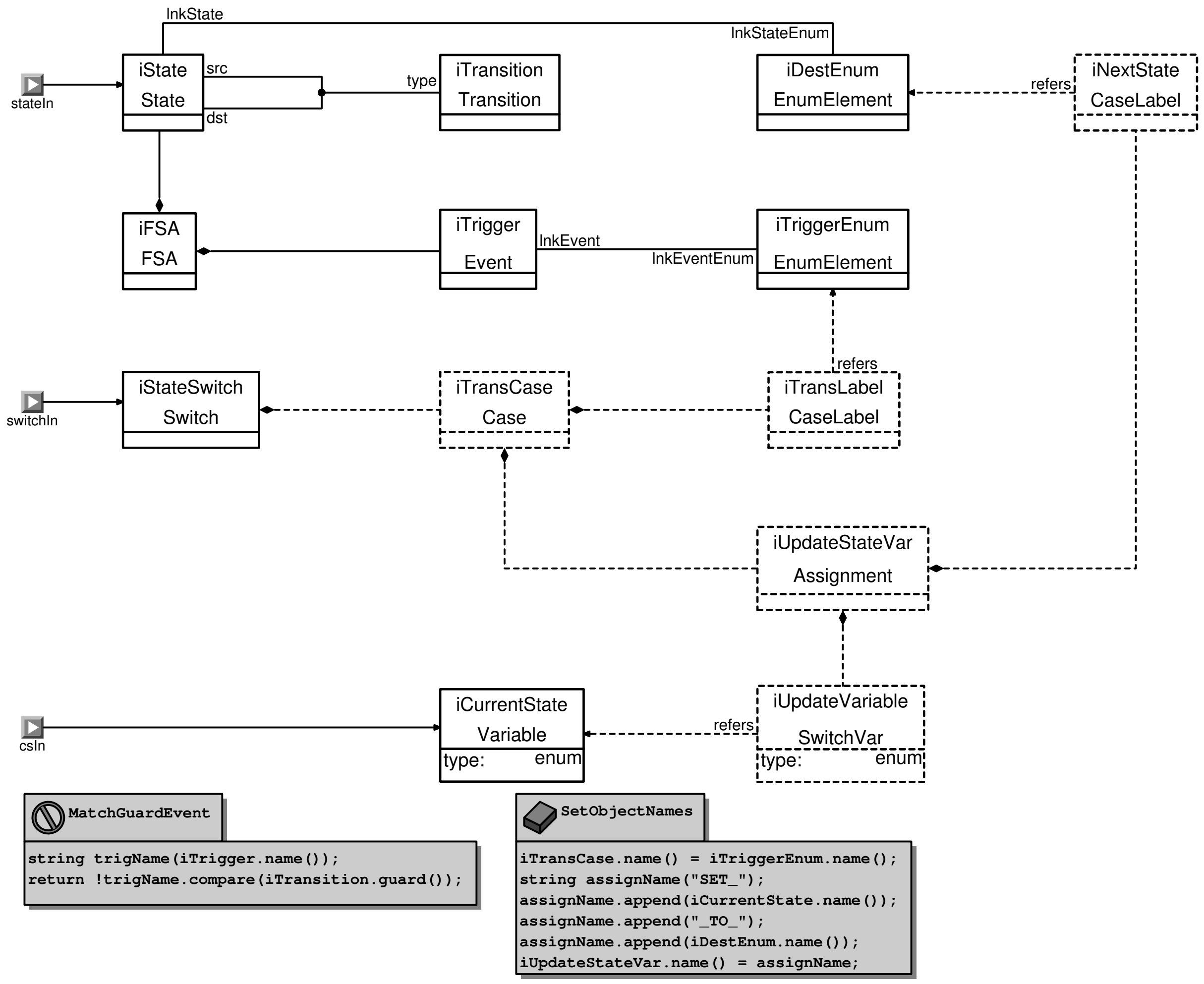
```
case EVENT_a:
 currentState = STATE_S3;
 break;
case EVENT_b:
 currentState = STATE_S2;
 break;
default:
 return;
 break;
}
   break;
  case STATE_S3:
   printf("Accepted input at state S3\n");
switch (currentEvent) {
case EVENT_a:
 currentState = STATE_S2;
 break;
case EVENT_b:
 currentState = STATE_S1;
 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");
}
```



```
string trigName(iTrigger.name());
return !trigName.compare(iTransition.guard());
```

```
iTransCase.name() = iTriggerEnum.name();
string assignName("SET_");
assignName.append(iCurrentState.name());
assignName.append("_TO_");
assignName.append(iDestEnum.name());
iUpdateStateVar.name() = assignName;
```

Fig. 27. Creates non-loop transitions



```
string trigName(iTrigger.name());
return !trigName.compare(iTransition.guard());
```

```
iTransCase.name() = iTriggerEnum.name();
string assignName("SET_");
assignName.append(iCurrentState.name());
assignName.append("_TO_");
assignName.append(iDestEnum.name());
iUpdateStateVar.name() = assignName;
```

Fig. 28. Creates loop transitions

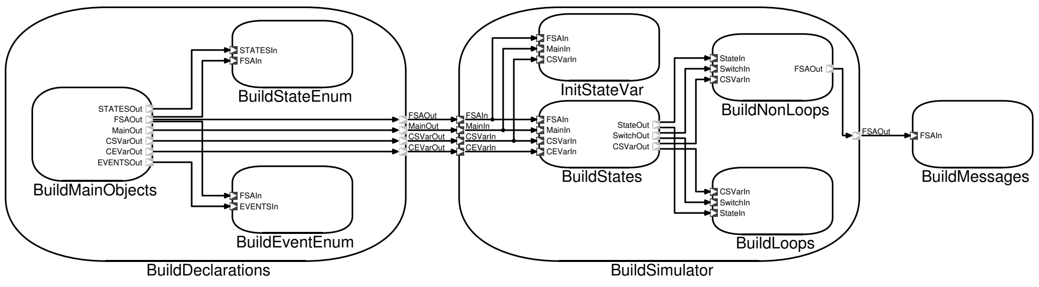


Fig. 29. Sequencing of transformation rules

REFERENCES

[1] M. Barnett and W. Schulte, "The abcs of specification: asml, behavior, and components." *Informatica (Slovenia)*, vol. 25, no. 4, 2001.

[2] E. A. Lee, S. Neuendorffer, and M. J. Wirthlin, "Actor-oriented design of embedded hardware and software systems." *Journal of Circuits, Systems, and Computers*, vol. 12, no. 3, pp. 231–260, 2003.

[3] A. L. T. B. G. Karsai, J. Sztipanovits, "Model-integrated development of embedded software," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 145–164, January 2003.

[4] J. Sprinkle, A. Agrawal, T. Levendovszky, F. Shi, and G. Karsai, "Domain model translation using graph transformations." in *ECBS*, 2003, pp. 159–167.

[5] A. B. D. Masys, D. Baker and K. Cowles, "Giving patients access to their medical records via the internet: the pcasso experience," *Journal of the American Medical Informatics Association*, vol. 9, no. 2, pp. 181–191, 2002.

[6] O. Danvy, "An analytical approach to programs as data objects," BRICS, Department of Computer Science, University of Aarhusy," Doctor Scientarum Thesis, 2006. [Online]. Available: http://www.brics.dk/~danvy/DSc/00_dissertation.pdf

[7] E. A. Lee, "Absolutely positively on time: What would it take?" *IEEE Computer*, vol. 38, no. 7, pp. 85–87, 2005.

[8] I. M. A. B. M. J. C. Tomlin, S. Boyd and L. Xiao, *Computational Tools for the Verification of Hybrid Systems*. John Wiley, 2003, in Software-Enabled Control, T. Samad and G. Balas (eds.).

[9] S. Sastry, J. Sztipanovits, R. Bajcsy, and H. Gill, "Scanning the issue - special issue on modeling and design of embedded software." *Proceedings of the IEEE*, vol. 91, no. 1, pp. 3–10, 2003.

[10] R. Stärk, J. Schmid, and E. Börger, *Java and the Java Virtual Machine - Definition, Verification, Validation*. Springer, 2001. [Online]. Available: citeseer.ist.psu.edu/470832.html

[11] K. Samelson and F. L. Bauer, "Sequential formula translation." *Commun. ACM*, vol. 3, no. 2, pp. 76–83, 1960.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[13] E. W. Dijkstra, "The humble programmer." *Commun. ACM*, vol. 15, no. 10, pp. 859–866, 1972.

[14] E. K. Jackson and J. Sztipanovits, "Using separation of concerns for embedded systems design," *Proceedings of the Fifth ACM International Conference on Embedded Software (EMSOFT'05)*, pp. 25–34, September 2005.

[15] Object Management Group, "Data distribution service for real-time systems specification," Tech. Rep., 2005. [Online]. Available: http://www.omg.org/docs/formal/05-12-04.pdf

[16] A. Srivastava and A. Eustace, "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, pp. 196–205.

[17] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm, "The influence of processor architecture on the design and the results of wcet tools." *Proceedings of the IEEE*, vol. 91, no. 7, pp. 1038–1054, 2003.

[18] R. Alur and D. L. Dill, "A theory of timed automata." *Theor. Comput. Sci.*, vol. 126, no. 2, pp. 183–235, 1994.

[19] T. A. Henzinger and C. M. Kirsch, "A typed assembly language for real-time programs." in *EMSOFT*, 2004, pp. 104–113.

[20] T. A. Henzinger, B. Horowitz, and C. M. Kirsch, "Giotto: a time-triggered language for embedded programming." *Proceedings of the IEEE*, vol. 91, no. 1, pp. 84–99, 2003.

[21] G. Agha, *Actors: a model of concurrent computation in distributed systems*. Cambridge, MA, USA: MIT Press, 1986.

[22] G. Kahn, "The semantics of simple language for parallel programming." in *IFIP Congress*, 1974, pp. 471–475.

[23] E. A. Lee and T. M. Parks, "Dataflow process networks," May 1995, pp. 773–799. [Online]. Available: citeseer.ist.psu.edu/lee95dataflow.html

[24] B. E. G.E. Allen, "Real-time sonar beamforming on workstations using process networks and posix threads," in *IEEE Transactions on Signal Processing*, vol. 48, no. 3, 2000, pp. 921–926.

[25] Y. Zhou and E. A. Lee, "A causality interface for deadlock analysis in dataflow." in *EMSOFT*, 2006, pp. 44–52.

[26] L. Gasieniec, A. Pelc, and D. Peleg, "The wakeup problem in synchronous broadcast systems." *SIAM J. Discrete Math.*, vol. 14, no. 2, pp. 207–222, 2001.

[27] G. D. Plotkin, "A structural approach to operational semantics." *J. Log. Algebr. Program.*, vol. 60-61, pp. 17–139, 2004.

[28] Y. Gurevich, "Evolving algebras 1993: Lipari guide," pp. 9–36, 1995.

[29] S. N. M. E. K. Chen, J. Sztipanovits and S. Abdelwahed, "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.

[30] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing." *IEEE Trans. Computers*, vol. 36, no. 1, pp. 24–35, 1987.

[31] X. L. S. N. Y. Z. Christopher Brooks, Edward A. Lee and H. Zheng, "Heterogeneous concurrent modeling and design in java (volume 3: Ptolemy ii domains)," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2007-9, January 11 2007. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-9.html

[32] ——, "Heterogeneous concurrent modeling and design in java (volume 1: Introduction to ptolemy ii)," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2007-7, January 11 2007. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-7.html

[33] E. A. Lee and Y. Xiong, "A behavioral type system and its application in ptolemy ii." *Formal Asp. Comput.*, vol. 16, no. 3, pp. 210–237, 2004.

[34] S. R. Schach, *Object-Oriented and Classical Software Engineering*. McGraw-Hill Pub. Co., 2001.

[35] J. L. Peterson, *Petri Net Theory and the Modeling of Systems*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1981.

[36] A. V. Aho and J. D. Ullman, *Principles of Compiler Design (Addison-Wesley series in computer science and information processing)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1977.

[37] E. W. Dijkstra, "Letters to the editor: go to statement considered harmful." *Commun. ACM*, vol. 11, no. 3, pp. 147–148, 1968.

[38] W. F. Ogden, "A helpful result for proving inherent ambiguity." *Mathematical Systems Theory*, vol. 2, no. 3, pp. 191–194, 1968.

[39] D. Kozen, "Partial automata and finitely generated congruences: An extension of Nerode's theorem," in *Logical Methods: In Honor of Anil Nerode's Sixtieth Birthday*, J. N. Crossley, J. B. Remmel, R. A. Shore, and M. E. Sweedler, Eds. Ithaca, New York: Birkhäuser, 1993, pp. 490–511.

[40] S. N. Burris and H. P. Sankappanavar, *A course in*

universal algebra. Springer-Verlag, 1981. [Online]. Available: citeseer.ist.psu.edu/article/burris81course.html

[41] E. K. Jackson and J. Sztipanovits, "Towards a formal foundation for domain specific modeling languages," *Proceedings of the Sixth ACM International Conference on Embedded Software (EMSOFT'06)*, pp. 53–62, October 2006.

[42] Object Management Group, "Unified modeling language: Superstructure version 2.0, 3rd revised submission to omg rfp," Tech. Rep., 2003. [Online]. Available: http://www.omg.org/docs/ad/00-09-02.pdf

[43] Institute For Software Integrated Systems, "Gme 5 user's guide," Vanderbilt University, Tech. Rep., 2005. [Online]. Available: http://www.isis.vanderbilt.edu/Projects/gme/GMEUMan.pdf

[44] Object Management Group, "Object constraint language v2.0," Tech. Rep., 2006. [Online]. Available: http://www.omg.org/docs/formal/06-05-01.pdf

[45] A. L. T. P. A. V. A. A. G. K. E. Magyari, A. Bakay, "Udm: An infrastructure for implementing domain-specific modeling languages," in *In the 3rd OOPSLA Workshop on Domain-Specific Modeling, OOPSLA 2003, Anahiem, California*, 2003.

[46] H. Emmelmann, F.-W. Schröer, and L. Landwehr, "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, pp. 227–237.

[47] M. Bozga, S. Graf, I. Ober, I. Ober, and J. Sifakis, "The if toolset." in *SFM*, 2004, pp. 237–267.

[48] G. Behrmann, A. David, and K. G. Larsen, "A tutorial on uppaal." in *SFM*, 2004, pp. 200–236.

[49] T. Hoare, "The verifying compiler: A grand challenge for computing research," *J. ACM*, vol. 50, no. 1, pp. 63–69, 2003.

[50] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, Eds., *Handbook of graph grammars and computing by graph transformation: vol. 2: applications, languages, and tools*. River Edge, NJ, USA: World Scientific Publishing Co., Inc., 1999.

[51] T. Mens, P. V. Gorp, D. Varró, and G. Karsai, "Applying a model transformation taxonomy to graph transformation technology." *Electr. Notes Theor. Comput. Sci.*, vol. 152, pp. 143–159, 2006.

[52] G. Csertán, G. Huszerl, I. Majzik, Z. Pap, A. Pataricza, and D. Varró, "Viatra - visual automated transformations for formal verification and validation of uml models." in *ASE*, 2002, pp. 267–270.

[53] D. Varró, S. Varró-Gyapay, H. Ehrig, U. Prange, and G. Taentzer, "Termination analysis of model transformations by petri nets." in *ICGT*, 2006, pp. 260–274.

[54] S. Neema and G. Karsai, "Software for automotive systems: Model-integrated computing." in *ASWSD*, 2004, pp. 116–136.

[55] W. Wolf, *Computers as components: principles of embedded computing system design*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001.

[56] M. Gschwind, H. P. Hofstee, B. K. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki, "Synergistic processing in cell's multicore architecture." *IEEE Micro*, vol. 26, no. 2, pp. 10–24, 2006.

[57] L. Stein, "Challenging the computational metaphor: Implications for how we think," 1999. [Online]. Available: citeseer.ist.psu.edu/stein99challenging.html

[58] G. M. L. L. A. S.-V. J. R. Rong Chen, Marco Sgroi, "Embedded system design using uml and platforms," in *Proceedings of Forum on Specification and Design Languages 2002 (FDL'02)*, September 2002. [Online]. Available: http://www.gigascale.org/pubs/314.html

[59] A. S.-V. Douglas Densmore, Roberto Passerone, "A platform-based taxonomy for esl design," *IEEE Design and Test of Computers*, vol. 23, no. 5, pp. 359– 374, September 2006. [Online]. Available: http://www.gigascale.org/pubs/932.html

[60] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. L. Sangiovanni-Vincentelli, "Metropolis: An integrated electronic system design environment." *IEEE Computer*, vol. 36, no. 4, pp. 45–52, 2003.