Toward a Semantic Anchoring Infrastructure for Domain-Specific Modeling Languages*

Kai Chen
Institute for Software Integrated Systems
Vanderbilt University,
Nashville, TN, 37205
chenk@isis.vanderbilt.edu

Janos Sztipanovits
Institute for Software Integrated Systems
Vanderbilt University,
Nashville, TN, 37205
janos@isis.vanderbilt.edu

Sandeep Neema
Institute for Software Integrated Systems
Vanderbilt University,
Nashville, TN, 37205
sandeep@isis.vanderbilt.edu

ABSTRACT

Metamodeling facilitates the rapid, inexpensive development of domain-specific modeling languages (DSML-s). However, there are still challenges hindering the wide-scale industrial application of model-based design. One of these unsolved problems is the lack of a practical, effective method for the formal specification of DSML semantics. This problem has negative impact on reusability of DSML-s and analysis tools in domain specific tool chains. To address these issues, we propose a formal well founded methodology with supporting tools to anchor the semantics of DSML-s to precisely defined and validated "semantic units". In our methodology, each of the syntactic and semantic DSML components is defined precisely and completely. The main contribution of our approach is that it moves toward an infrastructure for DSML design that integrates formal methods with practical engineering tools. In this paper we use a mathematical model, Abstract State Machines, a common semantic framework to define the semantic domains of DSML-s.

Categories and Subject Descriptors

D.2.2 [**Software**]: Software Engineering—*Design Tools and Techniques*; D.3.1 [**Software**]: Programming Language—*Formal Definitions and Theory*

General Terms

Language, Design

Keywords

domain-specific modeling language, Model-Integrated Computing, abstract syntax, semantic anchoring.

*This research was supported by the NSF ITR Grant CCR-0225610 "Foundations of Hybrid and Embedded Software Systems".

1. INTRODUCTION

Model-based design uses models, which are formal, composable and manipulable during the design process [30]. The modeling languages are domain-specific, offering designers modeling concepts and notations that are tailored to characteristics of their application domain. Domain-specific modeling languages (DSML-s) represent the structural and behavioral aspects of embedded software and systems. Their semantics capture concurrency, communication abstractions, temporal and other physical properties. For example, a DSML framework (i.e. a set of related modeling aspects) for embedded systems might represent physical processes using ordinary differential equations, signal processing using dataflow models, decision logic using finite-state machines, and resource management using synchronous models.

DSML-s are convenient tools for the design and implementation of embedded software and systems (ESSs). A well-made DSML captures the concepts, relationships, integrity constraints, and semantics of the application domain and allows users to program declaratively through model construction. Domain experts can easily master a DSML, since the domain concepts with which they are already familiar are incorporated in the modeling language . The applications of DSML-based tools, such as Simulink/Stateflow [5], or metaprogrammable tool chains, such as the Model-Integrated Computing (MIC) tools [4], range from non critical systems (e.g. cell phones) to safety critical systems (e.g. medical systems and drive-by-wire controllers for cars). However, adoption of DSML-s and model-based design has been slowed down by the following concerns:

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

applications. Semantic mismatch between design models and modeling languages of analysis tools may result in ambiguity in safety analysis or may produce conflicting results across different tools.

Some of these concerns have been alleviated by the appearance of metamodeling languages [8] representing the abstracts syntax of DSML-s, and metaprogrammable tools [25] that can be adapted easily and inexpensively to different domains. However, abstract syntax metamodeling - although an essential step - does not solve the problem caused by the lack of precise and explicit specification of DSML semantics. There has been much effort in the research community to define semantics of modeling languages by means of informal mathematical text (see e.g. the specification of the Hybrid Interchange Interchange Format, HSIF, [28]) or using formal mathematical notations (see e.g. the operational semantics specification for StateFlow in [20]). In either case, precise specification of semantics requires significant effort, which increases the cost of the specification and adoption of DSML-s. In addition, practical applications of DSML-s require their evolution as the users' understanding of the domain changes.

In this paper we argue that we can reach a reliable, safe and affordable technology for model-based design by developing an infrastructure for *semantic anchoring* of DSML-s via completing the following agenda:

1. Development of precise specification for a set of well-defined "*semantic units*" that capture the semantics of basic models of computations in a formal framework.

2. Development of modeling language front-end for the semantic units and specification of their abstract syntax by using metamodeling.

3. Development of an infrastructure for the transformational specification of DSML semantics by defining the mapping between the abstract syntax metamodels of DSML-s and that of the semantic units.

The proposed approach has direct relationship to and builds on the following model-based design concepts:

1. In platform-based design [29] the concept of *common semantic domain* plays essential role in mapping functional models to architecture platforms. A semantic unit (or an integrated group of semantic units) forms a common semantic domain for those DSML-s, which are anchored to it.

2. *Abstract semantics* [6] represents the common semantic features of families of models of computation, and realize models of computation through specialization of this abstract semantics. We define semantic units such that they can be concretized into a family related models of computations.

3. *Multiple-aspect modeling* [23] enables the managing of complexity of DSML-s by composing them into interrelated aspects. Semantic anchoring is used for the transformational specification of different DSML aspects.

The primary contribution of this paper is the description of key steps of the semantic anchoring process using finite state automata modeling as an example. We use in the process existing MIC tools: the Generic Model Environment (GME) [4] for metamodeling, the Graph Rewriting and Transformation (GReAT) tool [2] for model transformation, the Abstract State Machines (ASM) [11, 18], as a common semantic framework to define the semantic domain of DSML-s, and AsmL [1] - a high-level executable specification language based on the concepts of ASM.

The organization of this paper proceeds as follows: Section 2 describes our methodology to support the DSML design process. The concept of Semantic units and semantic anchoring are defined in Section 3. In Section 4, we use a simple DSML capturing the finite state machine domain from Ptolemy [6] as a case study to demonstrate key steps in the semantic anchoring process. Our conclusions and future work appear in Section 5.

2. BACKGROUND: DSML SPECIFICATION

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

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

The concrete syntax C defines the specific notation used to express models, which may be graphical, textual or mixed. The abstract syntax A defines the concepts, relationships, and integrity constraints available in the language. The semantic domain S is usually defined in some formal framework in terms of which the meaning of the models is explained. The syntactic mapping $M_C : C \rightarrow A$ mapping assigns syntactic constructs (graphical, textual or both) to the elements of the abstract syntax. The semantic mapping $M_S : A \rightarrow S$ semantic mapping relates syntactic concepts to those of the semantic domain.

The languages that are used for defining components of DSML-s are called *metalanguages* and the formal specifications of DSML-s are called *metamodels*. The specification of the abstract syntax of DSML-s requires a meta-language that can express concepts, relationships, and integrity constraints. The specification of the semantic domain and semantic mapping is more complicated, because models might have different interesting interpretations; therefore DSML-s might have several semantic domains and semantic mappings associated with them. For example, the *structural semantics* of a modeling language describes the meaning of the models in terms of the structure of model instances: all of the possible sets of components and their relationships, which are consistent with the well-formedness rules is defined by the abstract syntax. Accordingly, the semantic domain for structural semantics is defined by a *set-valued semantics*. The behavioral semantics may describe the evolution of the state of the modeled artifact along some time model. Hence, the behavioral semantics is formally captured by a mathematical framework representing the appropriate form of dynamics. (It is interesting to note that since DSML-s specify structural and behavioral invariants, which will be satisfied by all domain-specific models (DSMs), the concept of DSML - or more accurately its metamodel - is equivalent to the concept of domain architecture.)

The fact that DSML-s may have several semantic domains underlines the differences between defining semantics for programming languages and for DSML-s. Still, approaching the issues in the conceptual framework of "languages" instead of the conceptual framework "domain architectures" has the advantage of drawing from the rich theoretical and engineering background of language design and specification.

3. SEMANTIC ANCHORING

Our approach to the construction of an infrastructure for semantic anchoring of DSML-s is based on the following observations:

1. DSML-s are opportunistically created according to the needs of domains.

2. There is a well-defined, finite set of Models of Computations (MoCs) [29], which describe canonical interaction patterns among physical and computational components of embedded systems. These MoCs can be defined by a set of : $L_i = \{C_i, A_i, S_i, M_{Si}, M_{Ci}\}$ of minimal languages, where the intuitive meaning of "minimality" is the simplest modeling language required to describe a selected type of behavior.

Semantic anchoring of an arbitrary $L = \{C, A, S, M_S, M_C\}$ modeling language to an $L_i = \{C_i, A_i, S_i, M_{Si}, M_{Ci}\}$ model of computation means specifying the $M_A : A \rightarrow A_i$ mapping. The $M_S : A \rightarrow S$ semantic mapping of L is defined by the $M_S = M_{Si} \circ M_A$ composition, which means that the semantics of L is anchored to the S_i semantic domain of the L_i model of computation.

To develop an infrastructure for semantic anchoring requires the completion of the following tasks:

1. Selection of formal framework(s) for the mathematically precise specification of the ingredients of the languages (abstract syntax, etc.).

2. Selection of the canonic MoC-s and their specifications in the formal frameworks.

3. Development of methods and tools for specifying mapping between modeling languages.

Figure 1 shows our experimental tool architecture that supports the semantic anchoring of DSML-s. The GME tool suit is used to define the abstract syntax, A, for a $L = \{C, A, S, M_S, M_C\}$ DSML using UML Class Diagrams and OCL as metalanguage [9]. The $L_i = \{C_i, A_i, S_i, M_{Si}, M_{Ci}\}$ MoC is defined as an AsmL specification. In order to emphasize the central role of these fully specified modeling languages capturing fundamental MoC-s, we will call them semantic units. In this paper we specify their operational semantics: the semantic unit is defined in terms of (a) an AsmL Abstract Data Model (which corresponds to the A_i, abstract syntax specification of the modeling language defining the semantic unit in the AsmL framework), (b) the S_i, semantic domain (which is implicitly defined by the ASM mathematical framework), and (c) the M_{Si}, semantic mapping, defined as a model interpreter written in AsmL.

The $M_A : A \rightarrow A_i$ semantic anchoring of L to L_i is defined as a model transformation using the GReAT tool suite. The abstract syntax A and A_i are expressed as metamodels. Connection between the GME-based metamodeling environment and the AsmL environment is provided by a syntax conversion. Since the GReAT tool suit generates a model translator from the metalevel specification of the model transformation, any domain model written in the DSML can be directly translated into AsmL and can be simulated using the AsmL simulator. In the followings, we give detailed explanation about our methodology and the involved tools.

3.1 Abstract Syntax Modeling

The Generic Modeling Environment (GME) [4, 25] is a meta-programmable tool that supports the OMG four-layer metamodeling architecture. GME allows users both to design and to model with domain-specific modeling environments. The GME modeling environments may be created by using GME itself through a process of metamodeling. The GME metamodel is based on UML class diagrams [8] and OCL.

3.2 Specification of Semantic Units

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

- TLA^+ is a formal specification language developed by Lamport, which is based on the Temporal Logic of Actions [24]. TLA was designed to specify a wide class of systems, ranging from discrete to hybrid dynamics. Because of this generality and because TLA^+ is a complete language with a precise syntax and formal semantics, it is a good candidate for our purpose.

- The tagged signal model [27] developed by Lee and Sangiovanni-Vincentelli represents behavioral properties of concurrent systems using set-valued semantics. It clarifies, for example, the relationship between the synchronous models of computation used in synchronous languages and asynchronous models such as process networks and dataflow. It has been applied to construct a formal semantics for discrete-event systems.

- Reactive Modules are a formal model developed by Alur and Henzinger [10] for concurrent systems. The model is able to represent synchronous and asynchronous component interactions in a unified framework and supports compositional verification.

- Abstract State Machine (ASM), formerly called Evolving Algebras [18], is a general, flexible and executable modeling structure with well-defined semantics. General forms of behavioral semantics can be encoded as (and simulated by) an abstract state machine [11]. ASM is able to cover a wide variety of domains: sequential, parallel, and distributed systems, abstract-time and real-time systems, and finite- and infinite-state domains. ASM has been successfully used to specify the semantics of numerous languages, such as C [19], Java [13], SDL [16] and VHDL [12]. In particular, the International Telecommunication Union adopted an ASM-based formal semantics definition of SDL as part of SDL language definition [7].

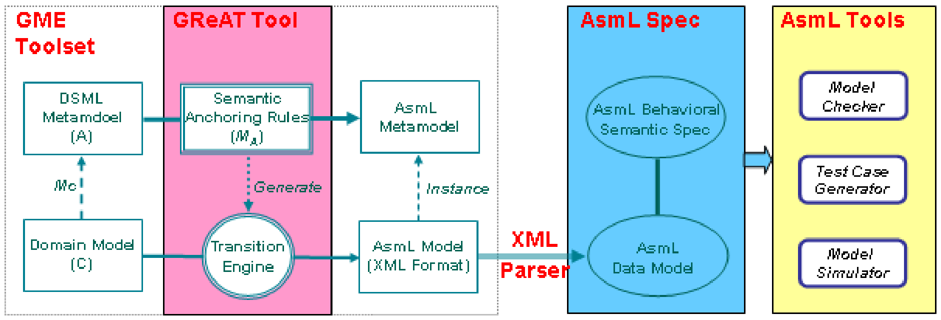


Figure 1: Experimental Tool Suite for Semantic Anchoring

Based on practical considerations, we decided to use ASM as a formal framework for the specification of semantic units. One reason for the decision was the Abstract State Machine Language, AsmL [1], developed by Microsoft Research that makes writing ASM specifications easy within the .NET environment. AsmL specifications look like pseudo-code operating on abstract data structures. As such, they are easy for programmers to read and understand. A set of tools is also provided to support the compilation, simulation, test case generation and verification of AsmL specifications. A detailed introduction to ASM and AsmL is beyond the scope of this paper, but readers can refer to other papers [1,11,17,18].

3.3 Semantic Anchoring Using Model Transformation

We use model transformation techniques as formal approach for specifying the $M_A : A \rightarrow A_i$ mapping between the abstract syntax of a DSML and the abstract syntax of the modeling language used as semantic unit. Based on our discussion above, the A abstract syntax of the DSML is defined as a metamodel using UML class diagrams and OCL, and the A_i abstract syntax of the semantic unit is as an Abstract Data Model expressed using AsmL data structure. However, specification of the M_A transformation between the two abstract syntax specifications requires using the same language. In our tool architecture, this common language is the abstract syntax metamodeling language (UML class diagrams and OCL), since the GReAT tool suite is based on this formalism. Accordingly, we defined using UML class diagrams and OCL the AsmL data structures. In Figure 2, we present a simplified version of this metamodel. Model transformation techniques can now be applied to specify the mapping between a DSML abstract syntax and these AsmL Abstract Data Models.

The $M_A : A \rightarrow A_i$ semantic anchoring is specified by using the Unified Model Transformation (UMT) language of the GReAT tool suite [2]. UMT is defined as a DSML and the M_A transformation can be specified graphically using the GME tool. The transformation rules between the source and the target metamodels form the semantic anchoring of a DSML. The GReAT tool can execute these transformation rules and transform any allowed domain model to an AsmL model. The generated AsmL model is then parsed to generate data model in the native AsmL syntax.

4. CASE STUDY: SEMANTIC ANCHORING FOR THE FSM DOMAIN IN PTOLEMY

We tested our semantic anchoring method and experimental tool suite using several DSML-s, including one patterned after the finite state machine (FSM) domain in Ptolemy, the MATLAB Stateflow and the IF timed automata based modeling language [14]. The detailed implementation can be downloaded from [3]. We use the FSM domain from Ptolemy as a case study to illustrate how to use our DSML design methodology in general.

4.1 The FSM domain in Ptolemy

Finite State Machines (FSMs) have long been used to model the control logic of reactive systems. However, conventional FSM models lack hierarchy and thus have a key weakness: the complexity of the model increases dramatically as the number of states increases. In 1987, David Harel proposed the Statecharts model [21], which extends the conventional FSM by supporting the hierarchical composition of states and concurrency. In 1999, Edward Lee proposed *charts [15,26], which allows the composition of hierarchical FSMs with a variety of concurrency models.

For simplicity, we define a DSML called the FSM Modeling Language (FML) which only supports Ptolemy-style hierarchical FSMs. For a detailed description of the hierarchical FSMs in Ptolemy, readers may refer to [26].

4.2 The Abstract Syntax Definition for FML

Figure 3 shows a UML class diagram for the FML metamodel as represented in GME. The classes in the UML class diagram define the domain modeling concepts. For example, the *State* class denotes the FSM domain concept of state. Instances of the *State* class can be created in a domain model to represent the states of a specific FSM. Note that the *State* class is hierarchical - each State object can contain an entire child machine. The *LocalEvent* class and the *ModelEvent* class represent respectively the local event and model event concepts in Ptolemy FSM domain. Local events are only visible within a single FSM model, whereas model events are globally visible.

A set of OCL constraints is added to the UML class diagram to specify well-formedness rules. For example, the constraint,

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

is attached to the *State* class. It specifies that if a *State*

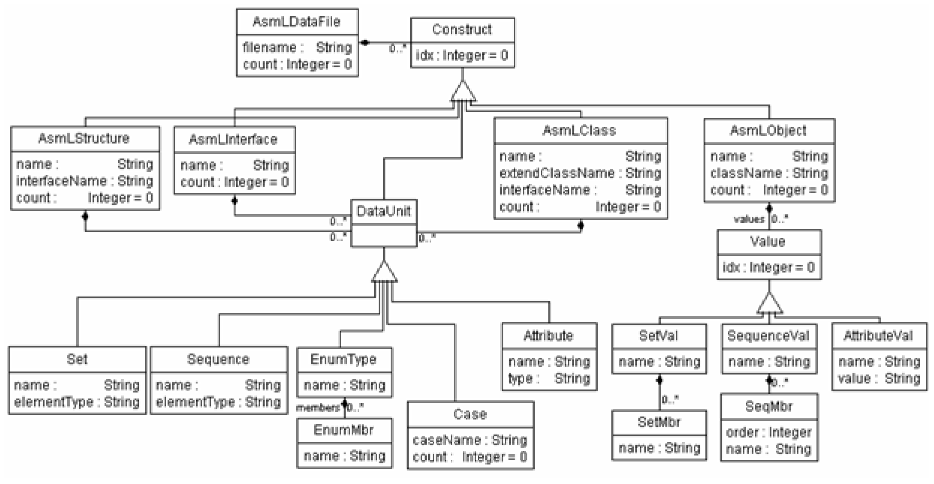


Figure 2: Metamodel for a Set of AsmL Data Structures

object has slaves (hierarchically-contained child states), exactly one slave should be the initial state.

4.3 A Semantic Unit Specification for FML

An appropriate semantic unit for FML should be generic enough to express the behavior of FML models. Since our purpose in this paper is restricted to demonstrate the key steps in semantic anchoring, we do not investigate the problem of identifying a generic semantic unit for hierarchical state machines. We simply define a semantic unit, which is rich enough for FML, but neglects the semantically meaningless elements to reach minimality.

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

4.3.1 AsmL Abstract Data Model for FML

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

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

```
interface Event
structure ModelEvent implements Event
structure LocalEvent implements Event
class FSM
  var outputEvents as Seq of ModelEvent
  var localEvents  as Set of LocalEvent
  var initialState as State
  var children     as Set of State
```

State and *Transition* are defined as first-class types. Note that the variable field *initalState* of the *State* class records the start state of any child machine contained within a given *State* object. The *initalState* will be undefined whenever a state has no child states. This possibility forces us to add the ? modifier to express that the value of the field may be either a *State* instance or the AsmL *undef* value. For the similar reason, we add the ? modifier after several other types of variable fields.

4.3.2 Behavioral Semantics for FML

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

4.3.2.1 Top-level FSM Operations.

A *FSM* instance waits for input events. Whenever an allowed input event arrives, the *FSM* instance reacts in a well-

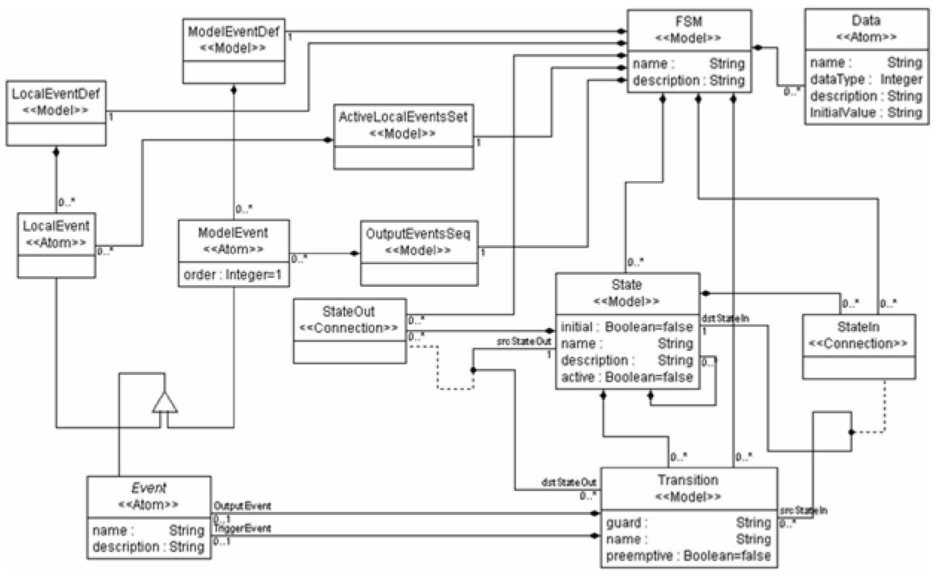


Figure 3: A Class Diagram for the FML Metamodel

```
class State
  var active as Boolean = false
  var initial as Boolean
  var initialState as State?
  var parentState as State?
  var slaves as Set of State
  var outTransitions as Set of Transition
class Transition
  var guard as Boolean
  var preemptive as Boolean
  var triggerEvent as Event?
  var outputEvent as Event?
  var src as State
  var dst as State
```

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

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

First, the rule determines the current state, which might be an initial state. Next, it checks for enabled preemptive transitions from the current state. If one exists, then the machine will take this transition and end the reaction. Otherwise, the rule will first determine if the current state has any child states. If it does, the rule will invoke the slaves (child states) of the current state. Next, it checks for enabled non-preemptive transitions from the current state. If one exists, then the rule will take this transition and end this reaction. Otherwise, it will do nothing and end this reaction.

4.3.2.2 *Do Transition.*

The operational rule *doTransition* specifies the steps through which a machine takes an enabled transition. We use the AsmL *require* construct to assert that the source state of the transition must be the current active state. First, exit the source state of the transition. Next, if the current transition mandates an output event, perform an emit event operation. Finally, make the destination state of the transition an active state.

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

4.3.2.3 *Activate State.*

The operational rule *activateState* describes the operations required to activate a state. The rule first sets the *active* field of the state. Then, it determines whether the state is a hierarchical state. If it is not, then the rule attempts to find an enabled instantaneous transition out of the current state. In Ptolemy, an instantaneous transition is defined as any transition that is outgoing from an atomic state and lacks a trigger event. An instantaneous transition must be taken immediately after entering its source state. If such a transition exists, the rule forces this transition and returns.

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

4.3.2.4 Get Instantaneous Transition.

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

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

4.4 The Semantic Anchoring of FML to the Semantic Unit

Having the abstract syntax of FML and an appropriate semantic unit specified, we are now ready to define the mapping, which provides the semantic anchoring for FML. We use UMT, which is supported by the GReAT tool, to specify the mapping rules between the metamodel for FML (Figure 3) and the metamodel previously defined for a subset of AsmL data structures (Figure 2). In the transformation process, the GReAT engine takes a FML model, executes the mapping rules and generates an AsmL data model.

The mapping specifications in UMT consist of a sequence of mapping rules. Each mapping rule is specified using pattern graphs. A pattern graph is defined using associated instances of the modeling constructs defined in the source and destination metamodels. Objects in a pattern graph can play three different roles as follows: bind, delete and new.

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

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

1. *Handle Events*: Match the model event and local event definitions in the input FML model and create the corresponding variants through the *Case* construct in *Event*.

2. *Handel State Machine*: Locate the top-level state machine in the input FML model; create an AsmL *FSM* object and set its attribute values appropriately.

3. *Handle States*: Navigate through the FML *FSM* object; map its child *State* objects into instances of AsmL *State* class, and set their attribute values appropriately.

4. *Handle Transition*: Navigate the hierarchy of the input model; create an AsmL *Transition* object for each matched FML *Transition* object and set its attribute values appropriately.

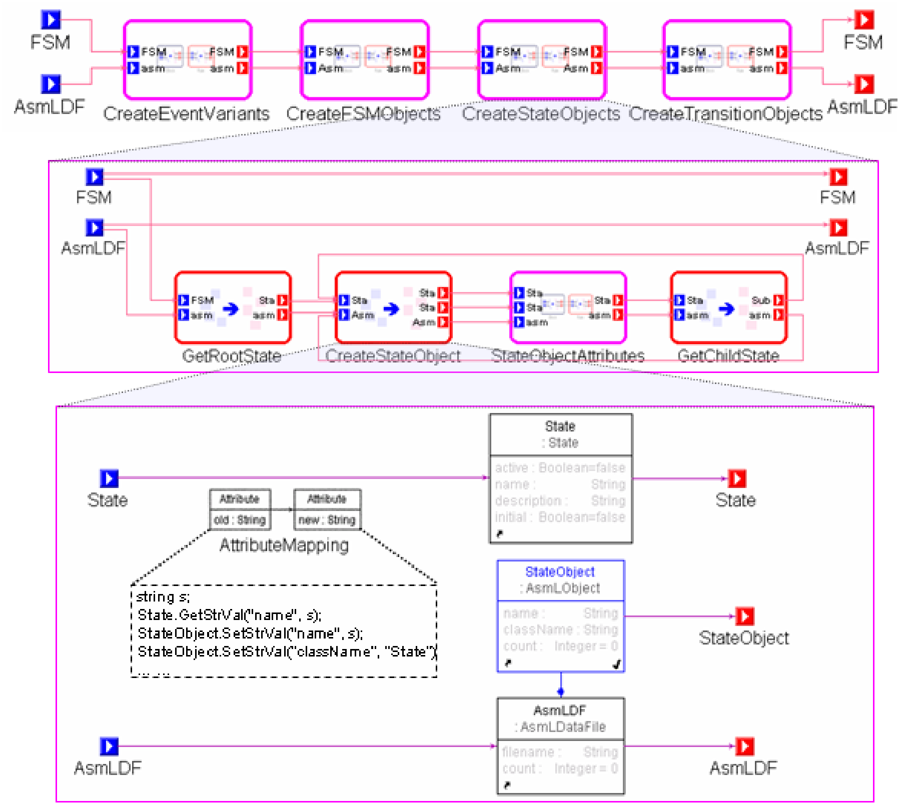
Figure 4 shows the hierarchical structure of the model transformation rules. The top-level rule consists of a sequence of sub-rules. Each of these sub-rules maps entities from FML to the corresponding entity in the semantic unit data structures. A sub-rule may be further decomposed into a sequence of sub-rules. The four key steps in the transformation algorithm, as described above, are corresponding to the four sub-rules contained in the top level rule. The final contents of these rules are pattern graphs that are specified in UML class diagrams. A pattern graph may also include *AttributeMapping* code block that includes code for reading and writing object attributes. For more information about GME and GReAT, please refer to [2, 4, 22, 25].

After we have specified the semantic mapping rules, the GReAT engine can execute these mapping rules and transform any legal FSM domain model directly into an AsmL data model. Figure 5 shows an example hierarchical FSM model which performs checksum calculations.

A XML file storing the AsmL data model is generated through the semantic mapping process. Our AsmL specification generator can parse this XML file and generate data model in native AsmL syntax as shown in Figure 6. The newly created AsmL data model plus the previously-defined AsmL semantic domain specifications compose an abstract state machine that gives the semantics for the FSM Checksum Machine model. With the specifications, the AsmL tools can simulate the behavior, do the test case generation and model checks. For more information about the AsmL supported analysis, readers can refer to [1].

5. CONCLUSION AND FUTURE WORK

The work on establishing a semantic anchoring infrastructure for DSML-s is in early stage. As the example showed, combining operational specification of semantic units with the transformational specification of DSML-s has the potential for improving significantly the precision of DSML spec-



```
string s;
State.GetStrVal("name", s);
StateObject.SetStrVal("name", s);
StateObject.SetStrVal("className", "State");
```

Figure 4: Top-level Mapping Rule for the FML Semantic Mapping specifications

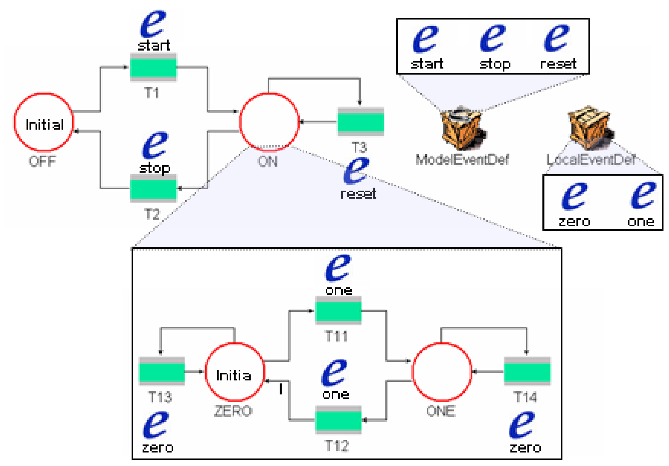


Figure 5: Hierarchical FSM Model for Checksum Machine

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

Figure 6: Part of the AsmL Data Model Generated from the Checksum Machine Model

ifications. The admittedly ad-hoc “semantic unit” we defined for the case study to show semantic anchoring for the FML can be reused to anchor other dialects and variations of hierarchical or flat state machines. The only component which needs to be changed is the mapping, which anchors a new DSML to the ”semantic unit”. We expect that substantial further effort is required to identify the appropriate set of semantic units and the best formal framework, which is general enough to cover a broad range of models of computations and can integrate both operational and denotational semantic specifications. An interesting area for further research is use cases for semantic units, which include the automatic generation of model translators that confirm the operational semantics captured in the semantic unit and semantically well founded tool integration and tool verification technology.

6. ACKNOWLEDGMENT

The authors want to thank for the useful discussions with Edward Lee, George Pappas, Alberto Sangiovanni-Vincentelly and Tom Henzinger in forming the semantic anchoring concept.

7. ADDITIONAL AUTHORS

Matthew Emerson (Institute for Software Integrated Systems, Vanderbilt Univ.) and Sherif Abdelwahed (Institute for Software Integrated Systems, Vanderbilt Univ.).

8. REFERENCES

[1] The abstract state machine language. `www.research.microsoft.com/fse/asml`.

[2] Graph rewriting and transformation. `www.isis.vanderbilt.edu/Projects/mobies`.

[3] Link for semantic anchoring tool suite. `www.isis.vanderbilt.edu/SAT`.

[4] The Generic Modeling Environment: GME. `www.isis.vanderbilt.edu/Projects/gme`.

[5] The MathWorks Simulink. `www.mathworks.com/products/simulink`.

[6] The Ptolemy Project. `www.ptolemy.eecs.berkeley.edu`.

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

[8] OMG unified modeling language specification version 1.5. Object Management Group document, 2003. formal/03-03-01.

[9] UML 2.0 OCL final adopted specification. Object Management Group document, 2003. ptc/03-10-14.

[10] R. Alur and T. A. Henzinger. Reactive modules. *Form. Methods Syst. Des.*, 15(1):7–48, 1999.

[11] E. Boerger and R. Staerk. *Abstract State Machines: A Method for High-Level System Design and Analysis.* Springer, 2003.

[12] E. Borger, U. Glasser, and W. Muller. *Formal Semantics for VHDL*, chapter Formal Definition of an Abstract VHDL'93 Simulator by EA-Machines, pages 107–139. Kluwer Academic Publishers, 1995.

[13] E. Borger and W. Schulte. A programmer friendly modular definition of the semantics of java. In *Formal Syntax and Semantics of Java, LNCS*, volume 1523, pages 353–404. Springer-Verlag, 1999.

[14] M. Bozga, S. Graf, I. Ober, I. Ober, and J. Sifakis. Tools and applications II: The IF toolset. In *Proceedings of SFM'04, LNCS*, volume 3185. Springer-Verlag, 2004.

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

[16] U. Glasser and R. Karges. Abstract state machines semantics of SDL. *Journal of University Computer Science*, 3(12):1382–1414, 1997.

[17] Y. Gurevich. Asm guide 97. CSE Technical Report CSE-TR-336-97.

[18] Y. Gurevich. *Specification and Validation Methods*, chapter Evolving Algebras 1993: Lipari Guide, pages 9–36. Oxford University Press.

[19] Y. Gurevich and J. Huggins. The semantics of the C programming languages. In *Computer Science Logic'92*, pages 274–308. Springer-Verlag, 1993.

[20] G. Hamon and J. Rushby. An operational semantics for stateflow. In *Fundamental Approaches to Software Engineering: 7th International Conference*, pages 229–243. Springer-Verlag, 2004.

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

[22] G. Karsai, A. Agrawal, and F. Shi. On the use of graph transformations for the formal specification of model interpreters. *Journal of Universal Computer Science*, 9(11):1296–1321, 2003.

[23] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty. Model-integrated development of embedded software. In *Proceedings of the IEEE*, volume 91, pages 145–164, 2003.

[24] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley, 2002.

[25] A. Ledeczi, A. Bakay, M. Maroti, P. Volgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai. Composing domain-specific design environments. *IEEE Computer*, 34(11):44–51, 2001.

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

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

[28] MoBIES Group. HSIF semantics. Internal document, The University of Pennsylvania, 2002.

[29] A. Sangiovanni-Vincentelli and G. Martin. Platform-based design and software design methodology for embedded systems. *IEEE Des. Test*, 18(6):23–33, 2001.

[30] J. Sztipanovits and G. Karsai. Model-integrated computing. *IEEE Computer*, 30(4):110–111, 1997.