Semantic Anchoring with Model Transformations*

Kai Chen, Janos Sztipanovits, Sherif Abdelwalhed, and Ethan Jackson

Institute for Software Integrated Systems, Vanderbilt University,
P.O. Box 1829 Sta. B., Nashville, TN 37235, USA
{kai.chen, janos.sztipanovits, sherif.abdelwalhed, ethan.jackson}
@vanderbilt.edu

Abstract. Model-Integrated Computing (MIC) is an approach to Model-Driven Architecture (MDA), which has been developed primarily for embedded systems. MIC places strong emphasis on the use of domain-specific modeling languages (DSML-s) and model transformations. A metamodeling process facilitated by the Generic Modeling Environment (GME) tool suite enables the rapid and inexpensive development of DSML-s. However, the specification of semantics for DSML-s is still a hard problem. In order to simplify the DSML semantics, this paper discusses semantic anchoring, which is based on the transformational specification of semantics. Using a mathematical model, Abstract State Machine (ASM), as a common semantic framework, we have developed formal operational semantics for a set of basic models of computations, called semantic units. Semantic anchoring of DSML-s means the specification of model transformations between DSML-s (or aspects of complex DSML-s) and selected semantic units. The paper describes the semantic anchoring process using the meta-programmable MIC tool suite.

1 Introduction

The Model-Driven Architecture (MDA) advocates a model-based approach for software development. Model-Integrated Computing (MIC) [27,24] is a domain-specific approach to MDA, which has been developed primarily for embedded systems. The MIC approach eases the complicated task of embedded system design by equipping developers with domain-specific modeling languages [25] tailored to the particular constraints and assumptions of their various application domains. A well-made DSML captures the concepts, relationships, integrity constraints, and semantics of the application domain and allows users to program imperatively and declaratively through model construction.

While a metamodeling process enables the rapid and inexpensive development of DSML syntax, the semantics specification for DSML-s remains a challenge problem. Transformational specification of semantics [9], gives us a chance

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

A. Hartman and D. Kreische (Eds.): ECMDA-FA 2005, LNCS 3748, pp. 115–129, 2005.
© Springer-Verlag Berlin Heidelberg 2005

to simplify the DSML semantics design. This paper exploits the transformational semantics specification approach for creating a semantic anchoring infrastructure [22]. This infrastructure incorporates a set of metaprogrammable MIC tools, including: the Generic Model Environment (GME) [4] for metamodeling, the Graph Rewriting and Transformation (GReAT) [2] tool for model transformation, the Abstract State Machines (ASM) [18,12], 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 for semantics specification.

The organization of this paper proceeds as follows: Section 2 describes the background for DSML specifications. Semantic anchoring is summarized in Section 3. In Section 4, we use a simple DSML that captures the finite state machine domain from Ptolemy II [5] as a case study to demonstrate the key steps in the semantic anchoring process. Conclusions and future work appear in Section 5.

2 Background: DSML Specification

A DSML can be formally defined as a 5-tuple $L = \langle A, C, S, M_S, M_C \rangle$ consisting of abstract syntax (A), concrete syntax (C), syntactic mapping (M_C), semantic domain (S) and semantic mapping (M_S) [28]. The syntax of a DSML consists of three parts: an abstract syntax, a concrete syntax, and a syntactic mapping. The abstract syntax A defines the language concepts, their relationships, and well-formedness rules available in the language. The concrete syntax C defines the specific notations used to express models, which may be graphical, textual, or mixed. The syntactic mapping, $M_C : C \rightarrow A$, assigns syntactic constructs to elements in the abstract syntax.

DSML syntax provides the modeling constructs that conceptually form an interface to the semantic domain. The semantics of a DSML provides the meaning behind each well-formed domain model composed from the syntactic modeling constructs of the language. For example, in MIC applications, the semantics of a domain model often prescribes the behavior that simulates an embedded system.

DSML semantics are defined in two parts: a semantic domain S and a semantic mapping $M_S : A \rightarrow S$ [21]. The semantic domain S is usually defined in some formal, mathematical framework, in terms of which the meaning of the models is explained. The semantic mapping relates syntactic concepts to those of the semantic domain. In DSML applications, semantics may be either structural or behavioral. The *structural semantics* 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, are 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. In this paper, we focus on the behavioral semantics of a DSML.

3 Semantic Anchoring

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

1. Define a set of minimal modeling languages $\{L_i\}$ for the basic behavioral abstractions and develop the precise specifications for all components of $L_i = \langle C_i, A_i, S_i, M_{Si}, M_{Ci} \rangle$. We use the term "semantic unit" to describe these basic modeling languages.
2. Define the behavioral semantics of an arbitrary $L = \langle C, A, S, M_S, M_C \rangle$ modeling language transformationally by 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 indicates that the semantics of L is anchored to the S_i semantic domain of the L_i modeling language.

The tool architecture supporting the semantic anchoring process above is shown in Figure 1. The GME tool suite [4] is used for defining the abstract syntax, A, for an $L = \langle C, A, S, M_S, M_C \rangle$ DSML using UML Class Diagrams [7] and OCL as metalanguage [8]. The $L_i = \langle C_i, A_i, S_i, M_{Si}, M_{Ci} \rangle$ semantic unit is defined as an AsmL specification [1] 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 (which is 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 [2]. The abstract syntax A and A_i

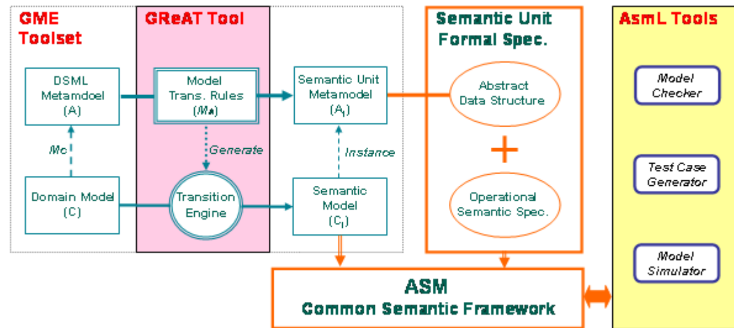

Fig. 1. Tool Architecture for DSML Design throughs Semantic Anchoring

are expressed as metamodels. Connection between the GME-based metamodeling environment and the AsmL environment is provided by a simple syntax conversion. Since the GReAT tool suite generates a model translation engine from the meta-level specification of the model transformation [23], any legal domain model defined in the DSML can be directly translated into a corresponding AsmL data model and can be simulated by using the AsmL native simulator. In the following, we give explanation of our methodology and the involved tools.

3.1 Formal Framework for Specifying Semantic Units

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

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 [12]. 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 [14], SDL [17] and VHDL [13]. In particular, the International Telecommunication Union adopted an ASM-based formal semantics definition of SDL as part of SDL language definition [6].

The Abstract State Machine Language, AsmL [1], developed by Microsoft Research, 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 model checking for AsmL specifications. The fact that there exists plentiful supporting tools for AsmL specifications was a important reason for us to select AsmL over other formal specification languages, such as Z [16], tagged signal model [26] and Reactive Modules [11]. A detailed introduction to ASM and AsmL is beyond the scope of this paper, but readers can refer to other papers [1,12,18].

3.2 Formal Framework for Model Transformation

We use model transformation techniques as a formal approach to specify the $M_A : A \rightarrow A_i$ mapping between the abstract syntax of a DSML and the abstract syntax of the semantic unit. Based on our discussion above, the abstract syntax A 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 an Abstract Data Model

expressed using the AsmL data structure. However, the specification of the M_A transformation requires that the domain and codomain of the transformation is expressed in the same language. In our tool architecture, this common language is the abstract syntax metamodeling language (UML class diagrams and OCL), since the GReAT tool suite is based on this formalism.

This choice requires building a UML/OCL-based metamodeling interface for the Abstract Data Model used in the AsmL specification of the semantic unit. One possible solution is to define a UML/OCL metamodel that captures the abstract syntax of the generic AsmL data structures. The other solution is to construct a metamodel that captures only the exact syntax of the AsmL Abstract Data Model of a particular semantic unit. Each solution has its own advantages and disadvantages. In the first solution, different semantic units can share the same metamodel and the same AsmL generator can be used to generate the data model in the native AsmL syntax. The disadvantage is that the model transformation rules and the AsmL specification generator are more complicated. Figure 2 shows a simplified version of the metamodel of generic AsmL data structures as it appears in the GME metamodeling environment. In the second solution, a new metamodel needs to be constructed for different semantic units, but the transformation rules are simpler and more understandable. Since the metamodel construction is easier compared with the specification of model transformation rules, we selected the second solution in our current work. We will present a metamodel example using this approach in section 4.4.

The $M_A : A \rightarrow A_i$ semantic anchoring is specified by using the Unified Model Transformation (UMT) language of the GReAT tool suite [23]. UMT itself is a

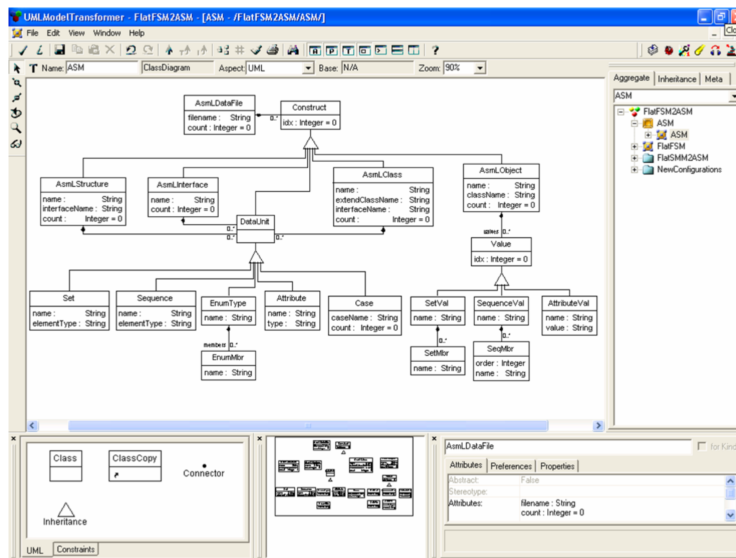

Fig. 2. Metamodel for a Set of AsmL Data Structures

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

4 Semantic Anchoring Case Study: FSM Domain in Ptolemy II

We have applied the semantic anchoring method and tool suite to design several DSML-s, including one patterned after the finite state machine (FSM) domain in Ptolemy II [5], the MATLAB Stateflow [20], and the IF timed automata based modeling language [15]. The detailed implementation can be downloaded from [3]. We use the FSM domain from Ptolemy II as a case study to illustrate the process described above.

4.1 The FSM Domain in Ptolemy

The Ptolemy FSM domain was proposed by Edward Lee with the name *charts [10] in 1999. It 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 *charts and the hierarchical FSMs in Ptolemy II, readers may refer to [5,10].

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 another state machine as a child in the hierarchy.

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,
```

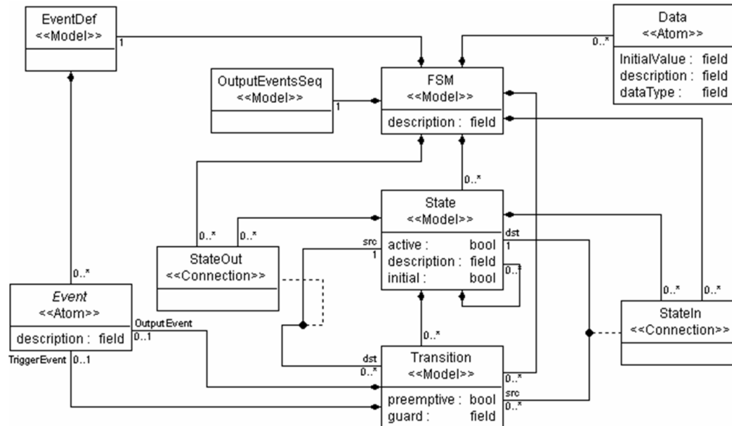

Fig. 3. A UML Class Diagram for the FML Metamodel

is attached to the *FSM* class. It specifies that if a *FSM* object has child states, exactly one child state must be the initial state. This is a constraint in Ptolemy II FSM domain.

Visualizations for instances of classes also need to be specified in the metamodel, so that an icon in a domain model will denote an instance of the corresponding class in the metamodel. In GME, this is usually done by setting a metamodel class's "Icon" attribute to the name of the desired bitmap.

4.3 Semantic Unit Specifications for FML

An appropriate semantic unit for FML should be generic enough to express the behavior of all syntactically correct FSMs. 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.

The semantic unit specification includes two parts: an Abstract Data Model and a Model Interpreter defined as operational rules on the data structures. 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 specifications together with some short explanations. Interested readers can download the full specifications from [3].

Constructing 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. The Abstract Data Model

does not need to capture every details of the FML modeling constructs, since some of them are only semantically-redundant. 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 *structure*. It may consist of one or more fields via the AsmL *case* construct. The keyword *structure* in AsmL declares a new type of compound value. In AsmL, classes contain instance variables and are the only way to share memory. Structures contain fields and do not share memory. Note that each AsmL language construct has its mathematical meaning in ASM. Readers can refer to [29] for their relationships. 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. 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 state machine.

```
structure Event
class FSM
  var outputEvents as Seq of Event
  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 a similar reason, we add the ? modifier after several other types of variable fields.

```
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 srcState       as State
  var dstState       as State
```

Behavioral Semantics for FML. We are now ready to specify the behavioral semantics for FML as operational rules, which can interpret the AsmL data structures defined above. Due to the space limitation, we show only two operational rules here.

Top-Level FSM Operations. A *FSM* instance waits for input events. Whenever an allowed input event arrives, the *FSM* instance reacts in a well-defined manner by updating its data fields and activating enabled transitions. To avoid non-determinism, the Ptolemy II 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 Event) =
  step let s as State = getCurrentState (fsm, e)
  step let pt as Transition? = getPreemptiveTrasition (fsm, s, e)
  step
    if pt <> null then doTransition (fsm, s, pt)
    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, then the rule will invoke the child states of the current state. Next, it checks for enabled non-preemptive transitions from the current state. If one exists, the rule will take this transition and end this reaction. Otherwise, it will do nothing and end this reaction.

Invoke Slaves. The operational rule *invokeSlaves* describes the operations required to invoke the child machine in a hierarchical state. The AsmL construct *require* is used here to assert that this state should be a hierarchical state, and it should have a start state in its child machine. The rule first determines the active state in the child machine. The rest of this operational rule is the same as the *fsmReact* rule. The similarity between the reactions of the top-level state machine and any child machine facilitates the Ptolemy II style composition of different models of computations.

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

4.4 Semantic Anchoring Specifications for FML to the Semantic Unit

Having the abstract syntax of FML and an appropriate semantic unit specified, we are now ready to describe the semantic anchoring specifications for FML. We use UMT, a language supported by the GReAT tool, to specify the model transformation rules between the metamodel of FML (Figure 3) and the metamodel for the semantic unit shown in Figure 4.

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

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

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

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

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

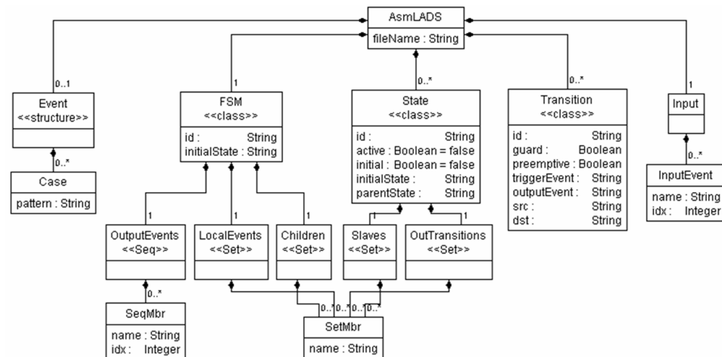

Fig. 4. Metamodel Capturing AsmL Abstract Data Structures for FML

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

Figure 5 shows the top-level transformation rule that consists of a sequence of sub-rules. These sub-rules are linked together through ports within the rule boxes. The connections represent the sequential flow of domain objects to and from rules. The ports *FSMIn*, and *AsmLIn* are input ports, while ports *FSMOut* and *AsmLOut* are output ports. In the top-level rule, *FSMIn* and *AsmLIn* are bound to the top-level state machine in the FSM model that is to be transformed, and the root object (a singleton instance of *AsmLADS*) in the semantic data model that is to be generated, respectively. The four key steps in the transformation algorithm, as described above, are corresponding to the four sub-rules contained in the top level rule.

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

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

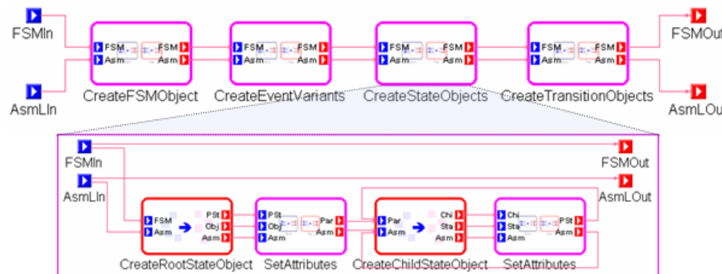

Fig. 5. Top-level model transformation rule

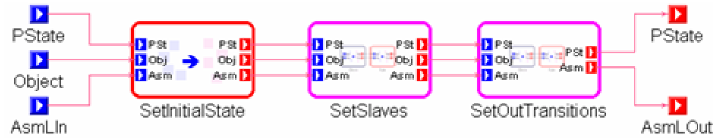

Fig. 6. Model Transformation Rule: *SetAttributes*

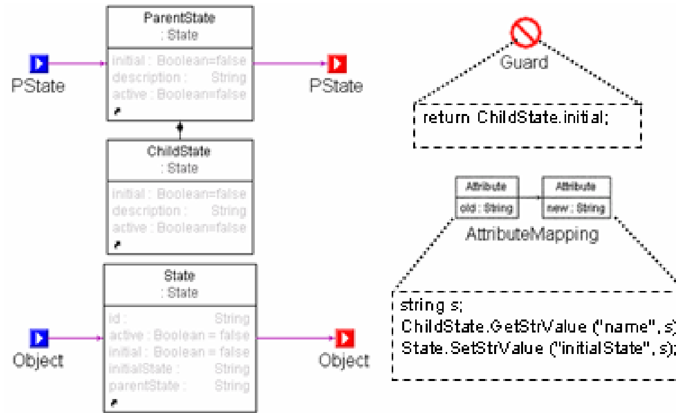

Fig. 7. Model Transition Rule: *SetInitialState*

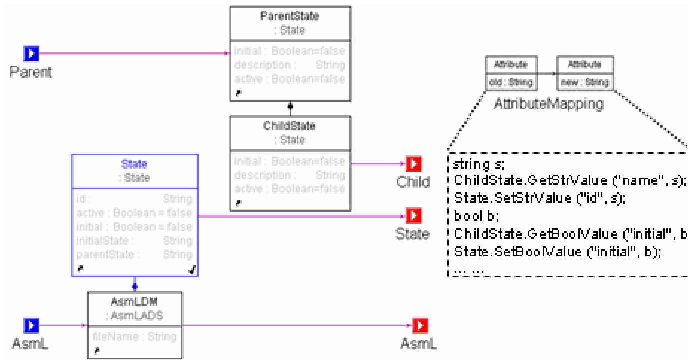

Fig. 8. Model Transition Rule: *CreateChildStateObject*

from the current state are added as members to the attribute *OutTransitions* by the sub-rule *SetOutTransitions*.

The final contents of model transformation rules are pattern graphs that are specified in UML class diagrams. Figure 7 shows a part of the *SetInitial-State* rule, which is a pattern graph. This rule features a GReAT *Guard* code block and a GReAT *AttributeMapping* code block. This rule is executed only

if the graph elements match and the *Guard* condition evaluates to true. The *AttributeMapping* block includes code for reading and writing object attributes.

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

In the semantic anchoring process, the GReAT engine takes a legal FML domain model, executes the model transformation rules and generates an AsmL

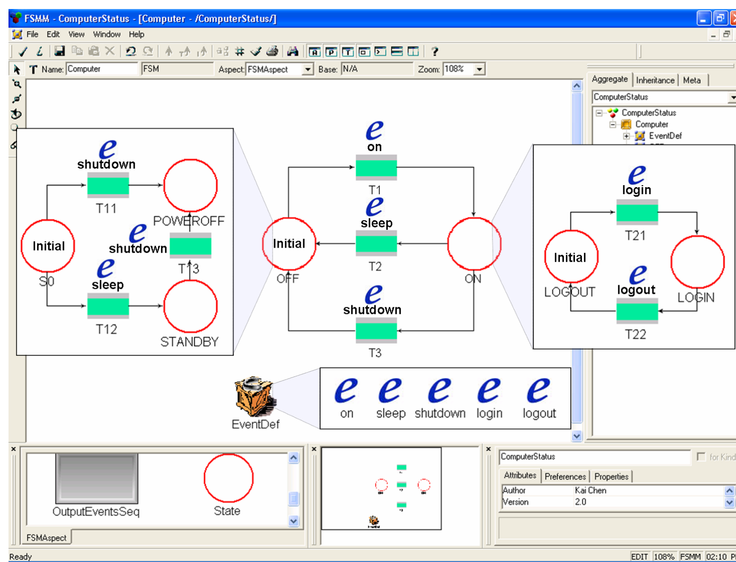

Fig. 9. A Hierarchical FSM model: *ComputerStatus*

```
structure Event
   case on
   case sleep
   case shutdown
   case login
   case logout
ComputerStatus = new FSM([], OFF, {ON,OFF})
OFF       = new State(true, S0, null, {S0, POWEROFF, STANDBY}, {T1})
ON        = new State(false, LOGOUT, null, {LOGOUT, LOGIN}, {T3,T2})
S0        = new State(true, null, OFF, {}, {T12,T11})
POWEROFF  = new State(false, null, OFF, {}, {})
STANDBY   = new State(false, null, OFF, {}, {T13})
LOGOUT    = new State(true, null, ON, {}, {T21})
LOGIN     = new State(false, null, ON, {}, {T22})
T1  = new Transition(true, false, Event.on, null, OFF, ON)
T2  = new Transition(true, false, Event.sleep, null, ON, OFF)
T3  = new Transition(true, false, Event.shutdown, null, ON, OFF)
T11= new Transition(true, false, Event.shutdown, null, S0, POWEROFF)
T12= new Transition(true, false, Event.sleep, null, S0, STANDBY)
T13= new Transition(true, false, Event.shutdown, null,STANDBY,POWEROFF)
T21= new Transition(true, false, Event.login, null, LOGOUT, LOGIN)
T22= new Transition(true, false, Event.logout, null, LOGIN, LOGOUT)
```

Fig. 10. Part of the AsmL Data Model Generated from the *ComputerStatus* Model

data model. As an example, we design a simple hierarchical *FSM* model in the GME modeling environment (Figure 9), which simulates the status of a computer. An XML file storing the AsmL data model is generated through the semantic anchoring process. We developed an AsmL specification generator, which can parse this XML file and generate the data model in native AsmL syntax as shown in Figure 10. 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 model *ComputerStatus*. With these specifications, the AsmL tools can simulate the behavior, do the test case generation and model checks. For more information about the AsmL supported analysis, see [1].

5 Conclusion and Future Work

This paper proposes a rapid and formal DSML design methodology, which integrates the semantic anchoring method and the metamodeling process. 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 specifications. 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. We are now working on specifying a semantic unit that can capture the common semantics for varied real-time system modeling languages. An interesting area for further research is use cases for semantic units. This may include the automatic generation of model translators that confirm the operational semantic captured in the semantic unit and offer semantically well founded tool integration and tool verification technology.

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 Ptolemy II. `www.ptolemy.eecs.berkeley.edu/ptolemyII`.
6. ITU-T recommendation Z.100 annex F: SDL formal semantics definition. International Telecommunication Union, Geneva, 2000.
7. OMG unified modeling language specification version 1.5. Object Management Group document, 2003. formal/03-03-01.
8. UML 2.0 OCL final adopted specification. Object Management Group document, 2003. ptc/03-10-14.
9. A. Maggiolo-Schettini and A. Peron. Semantics of full statecharts based on graph rewriting. In *LNCS*, pages 265–279. Springer-Verlag, 1994.

10. Alain GiraltB, Bilung Lee and E. Lee. Hierarchical finite state machines with multiple concurrency models. *IEEE Transactions On Computer-aided Design Of Integrated Circuits And Systems*, 18(6), 1999.
11. R. Alur and T. A. Henzinger. Reactive modules. *Form. Methods Syst. Des.*, 15(1):7–48, 1999.
12. E. Boerger and R. Staerk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer, 2003.
13. 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.
14. 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.
15. M. Bozga, S. Graf, I. Ober, and J. Sifakis. Tools and applications II: The IF toolset. In *Proceedings of SFM'04, LNCS*, volume 3185. Springer-Verlag, 2004.
16. A. Diller. *Z: an Introduction to Formal Methods*. John Wiley & Sons Ltd., second edition, 1994.
17. U. Glasser and R. Karges. Abstract state machines semantics of SDL. *Journal of University Computer Science*, 3(12):1382–1414, 1997.
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 and B. Rumpe. Meaningful modeling: What's the semantics of "semantics"? *IEEE Computer*, 37(10), 2004.
22. Kai Chen, J. Sztipanovits, S. Neema, M. Emerson and S. Abdelwahed. Toward a semantic anchoring infrastructure for domain-specific modeling languages. In *5th ACM International Conference on Embedded Software (EMSOFT'05)*, 2005.
23. 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.
24. 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.
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. 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.
27. J. Sztipanovits and G. Karsai. Model-integrated computing. *IEEE Computer*, 30(4):110–111, 1997.
28. T. Clark, A. Evans, S. Kent and P. Sammut. The MMF approach to engineering object-oriented design languages. In *Workshop on Language Descriptions, Tools and Applications*, 2001.
29. Yuri Gurevich, Benjamin Rossman and W. Schulte. Semantics essence of AsmL, March 2004. MSR-TR-2004-27.