Compositional Specification of Behavioral Semantics

Kai Chen
Motorola Labs
Schaumburg, IL, 60646, USA
Kai.Chen@motorola.com

Janos Sztipanovits
ISIS, Vanderbilt University
Nashville, TN, 37203, USA
Sztipaj@isis.vanderbilt.edu

Sandeep Neema
ISIS, Vanderbilt University
Nashville, TN, 37203, USA
Sandeep@isis.vanderbilt.edu

Abstract

An emerging common trend in model-based design of embedded software and systems is the adoption of Domain-Specific Modeling Languages (DSMLs). While abstract syntax metamodeling enables the rapid and inexpensive development of DSMLs, the specification of DSML semantics is still a hard problem. In previous work, we have developed methods and tools for the semantic anchoring of DSMLs. Semantic anchoring introduces a set of reusable "semantic units" that provide reference semantics for basic behavioral categories using the Abstract State Machine (ASM) framework. In this paper, we extend the semantic anchoring framework to heterogeneous behaviors by developing a method for the composition of semantic units. Semantic unit composition reduces the required effort from DSML designers and improves the quality of the specification. The proposed method is demonstrated through a case study.

1. Introduction

Model-based design of embedded software uses formal, composable and manipulable models in the design, implementation and system integration process. An emerging common trend in model-based software and systems design is that modeling languages are domain-specific: they offer software/system developers abstractions and notations that are tailored to characteristics of their application domains.

Model analysis and model-based code generation require the precise specification of DSMLs. This is partly achieved by metamodeling languages and metamodels describing the abstract syntax of DSMLs [4]. While abstract syntax metamodeling has been an important step in model-based design and been used in various model-based design frameworks, explicit and formal specification of behavioral semantics has not received much attention. For instance, the UML SPT [1] does not have precisely defined semantics [2], which creates possibility for semantic mismatch between design models and modeling languages of analysis tools. This is particularly problematic in safety critical real-time and embedded systems domain, where semantic ambiguities may produce conflicting results across different tools.

We started addressing these problems by extending our Model Integrated Computing (MIC) tool suite [6] with a semantic anchoring infrastructure for DSMLs. The Semantic Anchoring infrastructure [7] [8] includes a set of well-defined "semantic units" that capture the behavioral semantics of basic dynamic behavior categories. The semantics of a DSML is defined by specifying the transformation rules between the abstract syntax metamodel of the DSML and that of a selected semantic unit. In this paper we build on the previous results and address the impact of system heterogeneity by developing a method to specify DSML semantics as the composition of semantic units.

The organization of this paper is the following: Section 2 provides a short overview of the concepts of semantic anchoring and semantic units. The core idea for semantic unit composition is explained in Section 3. In Section 4, we demonstrate our approach using a simple case study. Section 5 is our conclusion.

2. Semantic Anchoring and Semantic Units

A DSML can be formally defined as a 5-tuple $L = <A, C, S, M_S, M_C>$ consisting of abstract syntax (A), concrete syntax (C), syntactic mapping (M_C), semantic domain (S) and semantic mapping (M_S) [7]. 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. The DSML semantics are defined in two parts: a semantic domain S and a semantic mapping M_S: $A \rightarrow S$. The semantic domain S is usually defined in some formal, mathematical framework, in

978-3-9810801-2-4/DATE07 © 2007 EDAA

terms of which the meaning of the models is explained. The semantic mapping relates syntactic concepts to those of the semantic domain.

Although DSMLs use many different modeling concepts and notations for accommodating needs of domains and user communities, the scope of well understood behavioral abstractions are more limited. Broad categories of component behaviors can be represented by a finite set of *basic behavioral categories*, such as Finite State Machine (FSM), Timed Automaton (TA) and Hybrid Automaton (HA). Similarly, analyzability requirements and the need for correct-by-construction system composition have led to the emergence of a set of *basic component interaction categories* expressed as Model of Computations such as Synchronous Data Flow (SDF), and Process Networks (PN) [5]. This observation led us to propose a semantic anchoring infrastructure for defining behavioral semantics of DSMLs. The development and use of the semantic anchoring infrastructure includes the following tasks [7]:

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

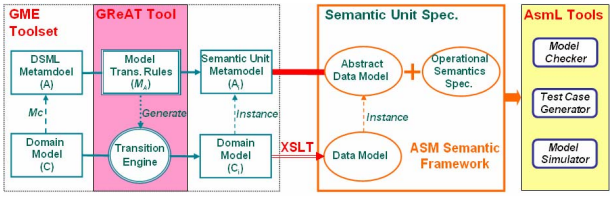

Figure 1. Semantic anchoring tool suite.

Figure 1 shows our semantic anchoring tool suite. It comprises (1) the ASM-based AsmL tool suite [3] [9] for specifying semantic units and (2) the MIC modeling (GME) and model transformation (GReAT) tool suites [6] that support the specification of transformation rules between the DSML metamodels and the Abstract Data Models defined in the semantic units. The behavioral semantics of semantic units are specified as a Control State ASMs [3] using AsmL. Microsoft Research has developed a set of tools to support the simulation, test case generation and model checking for AsmL specifications.

3. Semantic Unit Composition

In the semantic anchoring infrastructure, we define a finite set of semantic units, which capture the semantics of basic behavioral and interaction categories. If the semantics of a DSML can be directly anchored to one of these basic categories, its semantics can be defined by simply specifying the model transformation rules between the metamodel of the DSML and the Abstract Data Model of the semantic unit [7]. However, in heterogeneous systems, the semantics is not always fully captured by a predefined semantic unit. If the semantics is specified from scratch (which is the typical solution if it is done at all) it is not only expensive but we loose the advantages of anchoring the semantics to (a set of) common and well-established semantic units. This is not only loosing reusability of previous efforts, but has negative consequences on our ability to relate semantics of DSMLs to each other and to guide language designers to use well understood and safe behavioral and interaction semantic "building blocks" as well.

Our proposed solution is to define semantics for heterogeneous DSMLs as the composition of semantic units. If the composed semantics specifies a behavior which is frequently used in system design (for example, the composition of SDF interaction semantics with FSM behavioral semantics defines semantics for modeling signal processing systems [5]), the resulting semantics can be considered a *derived semantic unit*, which is built on *primary semantic units*, and could be offered up as one of the set of semantic units for future anchoring efforts. Note that *primary semantic units* refer to the semantic units that capture the semantics of the *basic behavioral categories*, such as FSM, TA and HA. The composition approach we describe in the rest of the paper is strongly influenced by Gossler and Sifakis framework for composition [11] by clearly separating behavior and interaction. In the following we provide a brief overview of the composition approach that will be followed by a case study.

Mathematically, a semantic unit specification can be represented as a 2-tuple $<A, R>$, where A is an Abstract Data Model specifying the abstract syntax of the semantic unit and R represents a set of Operations and Transition Rules (*updates*, in the ASM terminology). We use $M = I(A)$ to denote the set of all instances of A. Then, each $m \in M$ is a well formed Data Model defined by the Abstract Data Model A and R specifies the behavior of each $m \in M$. The behavior in ASM is modeled by a sequence of *steps*, where a

Step in a given state includes the execution *simultaneously* of all Rules whose guard conditions are true (and the updates are consistent) [3]. Since ASM states are mathematical structures (*sets* with basic operations and predicates), it is easy to integrate Abstract Data Models and Rules. The integrated tool suite ensures that the behavior of domain models defined in a DSML is simulated according to their "reference semantics" by automatically transforming them into AsmL Data Models using the transformation rules.

We model semantic unit composition as *structural* and *behavioral compositions* (see Figure 2). An ASM instance includes an *m* data model, the *R* rule set and the *S* dynamic state variables updated during runs. The structural composition defines relationships among selected elements of Abstract Data Models using partial maps. In Figure 2, we demonstrate semantic composition with two semantic units, SU1 and SU2. The composed semantics is also represented as a 2-tuple $<A, R>$. The structural composition yields the composed Abstract Data Model $A = <A_C, A_{SU1}, A_{SU2}, g_1, g_2>$, where g_1, g_2 are the partial maps between concepts in A_C, A_{SU1}, and A_{SU2}.

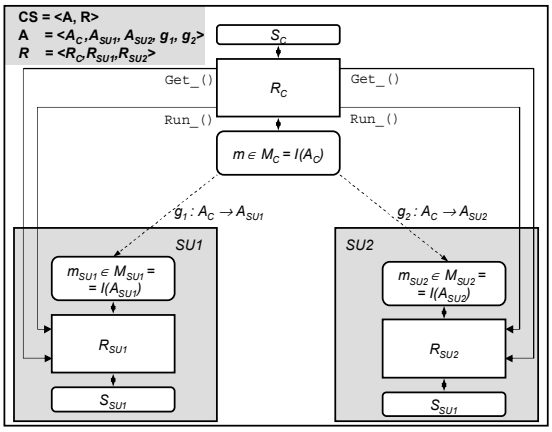

Figure 2. A graphical representation for the semantic unit composition.

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

In fact, the behavioral composition specifies a controller, which restricts the executions of actions. Since the behavior of the embedded semantic units can be described as partial orders on the sets of actions they can perform, the behavioral composition can be modeled mathematically as a composition of the partial orders.

4. Case Study: Semantic Specification for EFSM

EFSM was developed by General Motors Research to specify vehicle motion control software [12]. As a case study, we defined the behavioral semantics of EFSM as the composition of two primary semantic units, Finite State Machine (FSM-SU) and Synchronous Dataflow (SDF-SU). The full semantic specification includes two composition steps: (1) the semantics of EFSM Components are defined as the composition of FSM-SU and SDF-SU; (2) the semantics of SEFSM Systems are then defined as the composition of the semantics of EFSM Components, which can also be considered as a derived semantic unit, called Action Automaton Semantic Unit, and SDF-SU. Due to space limitations, we can only briefly describe the first composition step. The full semantic specifications can be downloaded from [10].

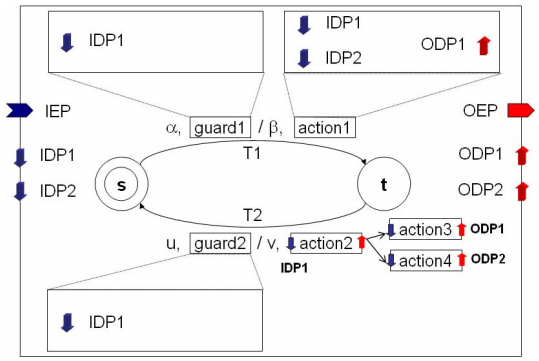

Figure 3. A simple EFSM component model.

4.1. EFSM Components

An EFSM model is a synchronous reactive system including a set of components communicating through event channels and data channels. An EFSM component is an FSM-based model. We use a simple component model shown in Figure 3 as an example to explain the structure and the behavior of EFSM components. The component communicates with other components through *ports*, including a single *input event port* (IEP), an *output event port* (OEP), two *input data ports* (IDP1 and IDP2) and two *output data ports*

(ODP1 and ODP2). A component also includes an FSM, where *transitions* are labeled with a *trigger event*, a *guard*, an *output event* and set of *actions*. Guards and actions are *computational functions* within the component and receive their input data through *input data ports*. The execution of an action (a function) may produce new data, while the execution of a guard only returns a Boolean value for the true or false evaluation.

4.2. Primary Semantic Units Used

4.2.1. FSM-SU Specification. The specification contains two parts: an Abstract Data Model $A_{FSM\text{-}SU}$ and Operations and Transformation Rules $R_{FSM\text{-}SU}$ on the data structures defined in A. The AsmL abstract class *FSM* prescribes the top-level structure of a FSM. All the abstract members of *FSM* are further specified by a concrete FSM, which is an instance of the Abstract State Model. (see detailed examples in [7])

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

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

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

4.2.2. SDF-SU Specification. The AsmL specification of the Abstract Data Model $A_{SDF\text{-}SU}$ is shown below. *Token* is defined as an AsmL structure to package data using the AsmL construct *case*. *Port* and *Channel* are defined as first-class types. The Boolean attribute *exist* of a port indicates whether the port has a valid data token. When all the input ports of a node have valid data tokens, the node is enabled to fire. In the specification, *Fire* is an abstract function. A concrete node will override the abstract function *Fire* with a computational function. The AsmL abstract class *SDF* captures the top-level structure of a model. The abstract property *inputPorts* contains a sequence of the SDF model's input ports that do not belong to any internal nodes. The abstract property *outputPorts* expresses the similar meaning.

```
structure Value
  case IntValue
    v as Integer
  case DoubleValue
    v as Double
  case BoolValue
    v as Boolean
structure Token
  value as Value?
class Port
  var token as Token   = Token (null)
  var exist as Boolean = false
class Channel
  srcPort as Port
  dstPort as Port
abstract class Node
  abstract property inputPorts  as Seq of Port
    get
  abstract property outputPorts as Seq of Port
    get
  abstract Fire ()
abstract class SDF
  abstract property nodes       as Set of Node
    get
  abstract property channels    as Set of Channel
    get
  abstract property inputPorts  as Seq of Port
    get
  abstract property outputPorts as Seq of Port
    get
```

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

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

4.3. Compositional Semantic Specification for EFSM Components

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

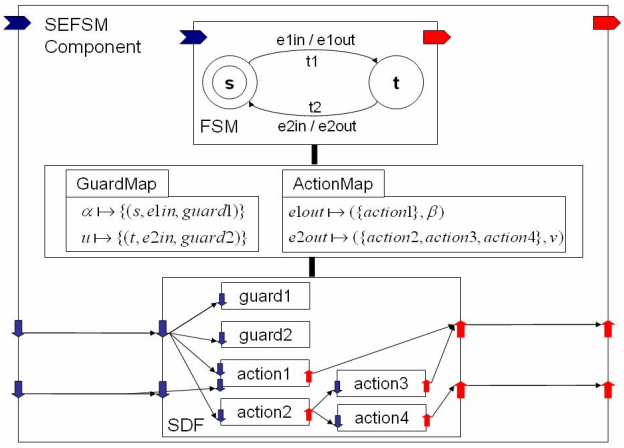

Figure 4. A compositional structure of the EFSM component originally shown in Figure 3.

4.3.1. Structural Composition. The structural composition defines mapping from elements in the Abstract Data Model of the composed semantic unit to elements in FSM-SU and those in SDF-SU. Figure 4 shows the role of FSM-SU and SDF-SU in the EFSM component model by restructuring the example in Figure 3. In the modified structure, the FSM model controls the event-related behaviors, while the SDF model takes charge of the data-related computations. Comparing Figure 3 and 4, we can find that the overall structure of the FSM model closely matches that of the original EFSM component, except for events, guards and actions. The trigger events and the output events in the FSM model are renamed. The guards and actions are represented as nodes in the SDF model. The relationships between the FSM model and the SDF model are specified by two maps: *GuardMap* and *ActionMap*. In this section, we only briefly explain how these two maps help to relate the FSM model with the SDF model. More details will be introduced in the following behavioral composition section.

The new compositional structure is built in a way that each transition in the original component is decomposed into three parts: a transition in the FSM model, a node representing the guard and a node representing the action in the SDF model. In the original component, a transition can be unambiguously located by the combination of the source state, the trigger event, and the guard. In the compositional structure, the information can be expressed by a 3-tuple *(s, e, n)*, where *s* refers a state in the FSM model; *e* is a local trigger event in the FSM model; and *n* represents a node in the SDF model. When a component receives an event, this event is a global event and will not be directly forwarded to the FSM model. The *GuardMap* maps this global event to a set of 3-tuples, each tuple referring to a transition in the original component whose trigger event matches this global event. Using the example in Figure 3 again, the event α is the trigger event only for the transition *T1*. In the compositional structure as shown in Figure 4, the *T1* transition is decomposed into the *t1* transition in the FSM model, whose source state is *s* and trigger event is *e1in*, and the *guard1* and *action1* node in the SDF model. As a result, *GuardMap* assigns the event α to the set *{(s, e1in, guard1)}*.

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

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

The execution of a transition in the original EFSM component can be decomposed into a three-step process: (1) the evaluation of the guard functions for all outgoing transitions from the current state as nodes in the SDF model; (2) the selection of an enabled transition in the FSM model; and (3) the execution of actions of the transition as nodes in the SDF model. The three steps are related to each other by the maps

GuardMap and *ActionMap*. The output event produced by the execution of a transition in the FSM model is a local event. *ActionMap* maps it to a 2-tuple *({n}, e)*, where *{n}* refers to a set of nodes (actions) in the SDF model and *e* refers to a global output event that will be propagated out of the component. For instance, the execution of the *t2* transition of the FSM model in Figure 4 generates a local event *e2out*. Since the *t2* transition corresponds to the *T2* transition in the original component (Figure 3), which is attached with actions, *action2*, *action3* and *action4*, and an output event *v*, the *ActionMap* maps the local event *e1out* to a 2-tuple *({action2, action3, action4}, v)* accordingly.

The rules verbalized above are specified in AsmL as Operations and Transition Rules. The operational rule *Run* of *Component* specifies the top-level component operations as a sequence of *updates*. The AsmL construct *require* asserts that the component's input event port must have a valid event. The rule first consumes the event in the port and checks whether this event triggers further updates in the component. If the event does, the rule *MapToLocalInputEvent* returns the corresponding local event used to trigger the FSM model; if not, a *null* value is returned and the reaction is completed. If a valid local event is returned, it activates the FSM model. The reaction of the FSM model returns a local output event. If the EFSM component produces an output event in this reaction, the rule *MapToGlobalOutputEvent* maps the local event to the global output even, which is then stored in the output port of the component.

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

Furthermore, we observe that this behavioral semantics specification is not limited to the EFSM components. It actually specifies the semantics of a common behavioral category that captures the reactive computation behaviors. Therefore, we can consider the semantic specification for EFSM components as a new derived semantic unit, called Action Automaton Semantic Unit (AA-SU). We can leverage this AA-SU to compositionally specify the semantics of EFSM systems. (Please refer to [10] for the full specification.)

5. Conclusion

Compositional semantic specification is a necessary step for making DSMLs semantically precise and practical. The proposed approach builds on a large body of work on ASM [3], semantics of composition [5], and on our earlier work on semantic anchoring [7] [8]. As a future step we will continue the construction of a library of primary semantic units and will move toward increased automation in semantic unit composition.

6. References

[1] Object Management Group. *UML™ Profile for Modeling and Analysis of Real-Time and Embedded systems*. realtime/05-02-06.

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

[3] E. Boerger and R. Staerk. *Abstract State Machines: A Method for HighLevel System Design and Analysis*. Springer, 2003.

[4] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty. Modelintegrated development of embedded software. *Proceedings of the IEEE*, volume 91, 2003.

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

[6] The MIC Tool Suite. http://www.escherinstitute.org/Plone/tools/suites/mic.

[7] Chen K., Sztipanovits J., Neema S., Emerson M., Abdelwahed S. Toward a Semantic Anchoring Infrastructure for Domain-Specific Modeling Languages, In *Proceedings of the Fifth ACM International Conference on Embedded Software (EMSOFT'05)*, pages 35-44, September 2005.

[8] Chen K., Sztipanovits J., Abdelwahed S., Jackson E. Semantic Anchoring with Model Transformations. In *Proceedings of European Conference on Model Driven Architecture -Foundations and Applications (ECMDA-FA)*, November 2005. LNCS, vol. 3748. pages 115-129, Springer-Verlag, 2005.

[9] The Abstract State Machine Language. www.research.microsoft.com/fse/asml.

[10] The Semantic Anchoring Tool Suite. www.isis.vanderbilt.edu/SAT.

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

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