Addressing Cross-Tool Semantic Ambiguities in Behavior Modeling for Vehicle Motion Control

Sandeep Neema[2], Sushil Birla[1], Shige Wang[1], and Tripti Saxena[2]

[1] General Motors Corporation, Warren, MI 48090
[2] Vanderbilt University, Nashville, TN 37203

Abstract. Emerging model-based development methods in the Automotive Vehicle Motion Control (VMC) domain are using different tools at various stages of the engineering process. Behavioral models created in various forms of finite state machines have to be exchanged across these tools, but semantic unknowns in modeling environments and semantic variations across tools preclude automated correct interpretation. This research presents an approach to address this issue through an unambiguous, math-based, tool-neutral extended finite state machine metamodel (eFSM) for behavior specifications in the automotive VMC domain. The semantics of the metamodel are anchored to formal specifications in a mathematical framework. Our approach requires modeling with commercial tool environments conforming to the eFSM. The conformance is enforced by exporting the tool native models into eFSM-conformant models and checking them against the well-formed rules encoded as OCL constraints in the eFSM. We have performed "proof of concept" exercises with two commercial tools in transforming their native models into eFSM-conformant forms, and have been able to show that certain ambiguities in both tools can be prevented through the eFSM, promising higher confidence software engineering for the VMC domain.

1 Introduction

High integrity functions in Automotive Vehicle Motion Control (VMC) software are becoming increasingly complex as more functions are being realized in software. Factors contributing to the rising complexity include increasing number of interactions, distribution across many electronic control units (ECU-s) and buses, number of different suppliers, and number of engineering stages spread across different disciplines and different tool environments. To add to the complexity, VMC functions are tightly constrained in timing interrelationships, combining discrete and continuous control in ways that are difficult to analyze. The size and complexity of VMC systems have grown beyond the ability to assure their correctness through exhaustive testing and simulation. These difficulties motivate the need for VMC systems engineering processes that prevent errors from the earliest stage and provide work products that are correct by construction [1].

In order to improve engineering quality, industry has been shifting effort from program code level activity towards model based control and software engineering [2,3]. However, it is not possible to transfer model data unambiguously from the tool of one engineering stage to that of another. In other words, different tools are not able to interpret the model with the same meaning. Although tools popular in the VMC engineering process, such as MathWorks Stateflow, I-Logix Rhapsody, and ETAS ASCET, support modeling in the finite state machine (FSM) paradigm, there are semantic unknowns and variations in their FSM-s. Therefore, model data has to be manually interpreted, manipulated and transferred from one engineering stage to the next, imposing penalties in integrity, quality, cost, and time.

Many research and industrial endeavors have addressed cross-tool model exchange issues. Industrial efforts include various standardization activities. ISO 10303 AP 233 [4] extends STEP, the international standard for exchange of product data, to support exchange of behavioral models of various kinds, including the FSM. The Object Management Group (OMG) SysML [5] is developing a standardized system modeling language, as a profile of UML 2.0 [6]. However, UML 2.0 does not have a strong mathematical foundation, e.g., it does not specify constraints on relationships such as generalization-specialization. Thus UML 2.0 does not support unambiguous model transformation and exchange. SAE Analysis Architecture Description Language (AADL) [7] is a modeling language to model system architecture for analysis. AADL focuses on structure and para-functional properties, and is not suitable for systems engineering activities such as requirement specifications. EAST-ADL [8] was developed as a modeling language for electronic architecture with similar objectives, but does not provide unambiguous semantic support for behavior specification. EAST-ADL relied on external tools and languages for behavioral specifications. In parallel, researchers have endeavored to formally specify the semantics of commercial tools. For example, the formal operational semantics for Stateflow by Hamon and Rushby [9] and the operation semantics of Stateflow in BSpec notation by Kestrel Technologies [10] are two of a dozen published Stateflow semantics. However, the formal semantics defined by these research activities are "reverse engineered", without support from the tool vendors, based on the behaviors observed over a set of examples. Conformance to vendor-implemented semantics is demonstrated in most cases by comparing traces with a few tests. It does not provide adequate confidence for the VMC domain.

While the international standards and commercial tools seek breadth of application to enlarge their market, we seek disambiguation of model data for a narrowly defined domain of applications, VMC, where integrity is paramount. The scope is limited to statically configured systems with statically defined deterministic behaviors. The typical behavior of a VMC application can be described in a finite state machine with the continuous closed loop control functions embedded in its action elements. The scope of data exchanges includes VMC systems engineering processes such as requirement specification, functional design, analysis of various types, specification of the distributed platform, allocation of application functions and interactions to platform elements, code generation, integration, verification at every step, and overall validation. Behaviors of this kind can be metamodeled as an extended finite state machine (eFSM). In this paper, we propose an eFSM, with unambiguous semantics anchored in a mathematical foundation, as a well-suited medium for interchange across different stages of the engineering process mentioned above. The proposed approach requires modeling constraints, which are VMC domain-specific to enable a correct-by-construction process.

The rest of this paper is organized as follows. Section 2 explains the language requirements for unambiguous model exchange in the context of the systems engineering process for future VMC development. Section 3 presents the math-based eFSM and its support for unambiguous model exchange. Section 4 describes model export exercises with two commercial tools, using the eFSM, and the lessons learned from these exercises. The concluding Section 5 recapitulates the approach, in the context of planned future work.

2 Disambiguation Needs in a High Integrity Process

Elimination of errors in interpreting model data transferred across engineering process stages is the primary objective of this research. The eFSM is required as an enabler for the process framework shown in Fig. 1 and Fig. 2. The objectives of the process framework are to provide correct by construction products at the minimum feasible life cycle cost, time-to-market, and execution complexity. Vehicle-specific construction, verification and certification constitute a significant part of vehicle cost and time to market. Other supporting requirements are discussed during the following overview of the process framework.

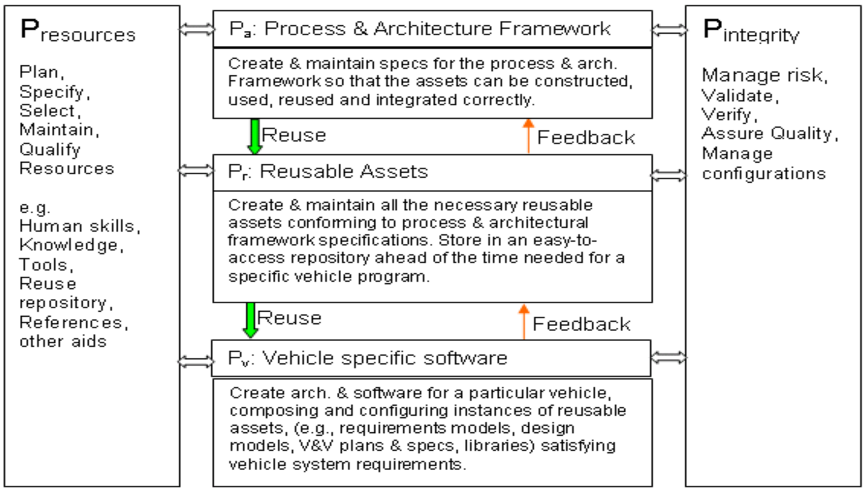

Presources
Plan, Specify, Select, Maintain, Qualify Resources
e.g. Human skills, Knowledge, Tools, Reuse repository, References, other aids
Pa: Process & Architecture Framework
Create & maintain specs for the process & arch. Framework so that the assets can be constructed, used, reused and integrated correctly.
Reuse
Feedback
Pr: Reusable Assets
Create & maintain all the necessary reusable assets conforming to process & architectural framework specifications. Store in an easy-to-access repository ahead of the time needed for a specific vehicle program.
Reuse
Feedback
Pv: Vehicle specific software
Create arch. & software for a particular vehicle, composing and configuring instances of reusable assets, (e.g., requirements models, design models, V&V plans & specs, libraries) satisfying vehicle system requirements.
Pintegrity
Manage risk, Validate, Verify, Assure Quality, Manage configurations

Fig. 1. Systems engineering process framework for vehicle motion control development

The process framework shown in Fig. 1 is based on principles of domain engineering [11] and product line engineering [12]. It has five major groups of processes or sub-frameworks, designated and related as follows. P_v are processes specific to a vehicle and are based on reusing assets created and maintained through processes, P_r, which in turn, follow the process and architectural framework specifications created in the processes, P_a. The eFSM is a part of P_a. $P_{integrity}$, the processes of risk management, configuration management (CM), quality assurance (QA), and V&V are applied to processes, P_v, P_r, and P_a. $P_{resources}$, the processes to plan, specify, select, maintain and qualify resources (e.g. human skills, knowledge, tools, reuse repository, and other aids), are also applied to processes, P_v, P_r, and P_a. $P_{resources}$ also include processes for upgrade, growth, and adaptation. Thus, VMC systems are specified and created with the reuse of proven elements and in ways proven to "plug & play" (compose) correctly. In other words, P_v should be a correct-by-construction process that guarantees the resultant models satisfy the specified system requirements. This process imposes a requirement that the asset be reusable correctly in future VMC applications yet unknown, thus requiring unambiguous semantics. As in the systems engineering process framework, the software development processes include the resource-related processes, $P_{resource}$, the development processes, P_a, P_r, and P_v, and the integrity processes, $P_{integrity}$.

With formal reuse of a full complement of assets from a library or repository, system conceptualization becomes a composition process rather than the traditional decomposition process. Formal reuse begins with formal fine-grained requirements models (in the form of extended finite state machines or automata) from which vehicle-specific requirements are composed, following pre-defined composition rules, applied to the requirements space (see layer 2 in Fig. 2). The process requires that incorrect compositions, including unwanted interactions, be prevented.

While the formal external behavioral models will endure over time, it is expected that the concrete realizations, for example layer 3 and greater in Fig. 2, will change as implementation technologies change. Referring to P_v in Fig. 1, the system conceptual architecture evolves bottom-up from the fine-grained requirements models by searching for matching design & implementation (D & I) entities in the reusable assets library in P_r. Search and matching criteria include not only the functional requirements model, but also associated parafunctional requirements and D & I constraints. When the best match is determined, the requirements are "allocated" to the matching "D & I" entities. The process is iterated until all requirements are allocated.

For provably correct transformation of requirements into concrete realizations or implementations, certain process and architectural constraints are imposed. Referring to the layer 2-3 transformation, shown as ${}^2T_3$ in Fig. 2, the process of functional design is constrained to be an elaboration (or refinement) of the requirements specification automata. More than one requirement automaton may be allocated to a functional design unit. The refinement constraint assures that the functional design inherently conforms to the specification. When the

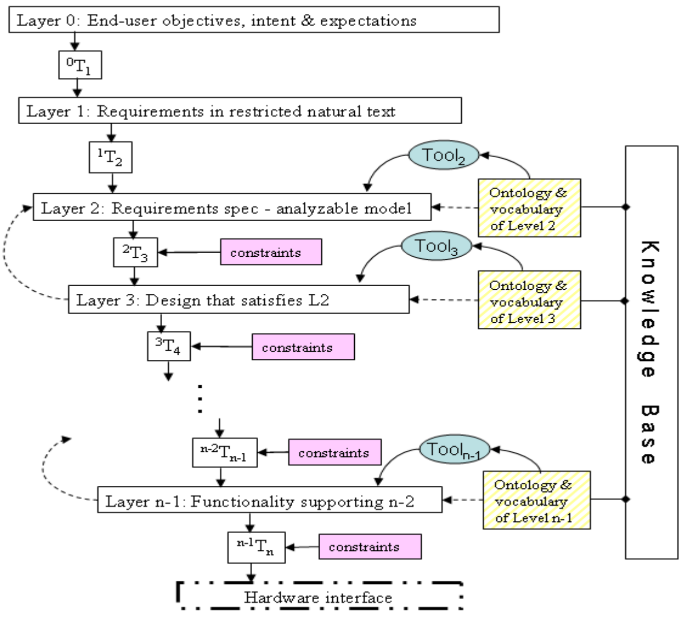

Layer 0: End-user objectives, intent & expectations
$^0T_1$
Layer 1: Requirements in restricted natural text
$^1T_2$
Tool$_2$
Ontology & vocabulary of Level 2
Layer 2: Requirements spec - analyzable model
$^2T_3$
constraints
Tool$_3$
Layer 3: Design that satisfies L2
Ontology & vocabulary of Level 3
$^3T_4$
constraints
$^{n-2}T_{n-1}$
constraints
Tool$_{n-1}$
Ontology & vocabulary of Level n-1
Layer n-1: Functionality supporting n-2
$^{n-1}T_n$
constraints
Hardware interface
Knowledge Base

Fig. 2. Multi-stage model transformation in the systems engineering process

engineering process progresses to the stage of identifying a component, for example in layer 3-4 transformation — $^3T_4$, the external interface of the functional design (which is the same as the requirements specification) maps on to the component, and becomes a part of the components external interface (through its service access points). Thus the component is also a composable automaton. More than one functional design unit may be allocated to a component.

The framework provides for a number of stages or layers of transformation, $^{i-1}T_i$, (beyond layer 3 in Fig. 2), depending on the complexity of the system, in order to localize and isolate the effects of implementation decisions. Each transformation stage uses previously proven transformation rules, mappings, metamodels and ontologies defining the language of each stage, collectively shown as the knowledge base. It is a combination of work products from processes P_a and P_r. The language of each layer defines the universe of services available from that layer, specified as composable automata.

As engineering, denoted as P_v in Fig. 1, progresses beyond the $^3T_4$ transformation (Fig. 2), the modeling frameworks on both sides of the transformation, $^{i-1}T_i$, must be compatible, i.e., the semantics of elements in layers (i-1) and i be unambiguous, the transformed elements have a defined correspondence from layer (i-1) to layer i, and the process, $^{i-1}T_i$, be semantic-preserving. Each transformation, $^{i-1}T_i$, requires a combination of tool automation for the rule-driven part and human effort for the creative part of the work. It should be possible to use the best-in-class tool for each $^{i-1}T_i$. The reusable assets and work products of P_v should be protected from obsolescence due to changes in the tools. It should be possible to reposit the work products of every stage in a form independent of the tools producing or consuming the work products. Upon a change in implementation (layers greater than 3 in Fig. 2), the systems engineering process is reapplied to the affected composable entity. If there is no other change in the system, with formal reuse and conformance to the specified architecture, only the changed component has to be re-verified against its specification. When compositions of formally proven components (assets created in P_r) are created, using rules defined in P_a, the components do not have to be re-verified. This is an important requirement on the modeling framework, because it reduces vehicle-specific costs of verification, rectification, and certification.

3 A Language for Modeling Finite State Machines

To enable unambiguous exchange of FSM-based behavioral models in the conceptual process framework for high integrity VMC software, we have developed a modeling language, eFSM, with rules and constraints, for describing FSM models unambiguously. The eFSM is represented in a tool-neutral metamodel form and consists of a set of modeling elements whose semantics and composition rules are formally anchored to a math-based specification for unambiguous interpretation and processing. Fig. 3 is an overview of the mathematical framework for defining and interpreting the eFSM. The basic elements of a FSM, such as states, events, transitions, and actions, and their relationships are defined in the eFSM and are linked to additional mathematical specifications (ontologies) which may be expressed in other mathematical languages and processed by their respective mathematical engines to reason about the model. The semantics of eFSM have been developed as a composition of multiple semantic domains using ASML. A detailed description of eFSM language, examples, and semantics are presented in [13]. Multiple mathematical languages are accommodated by means of language transformation, based on their respective metamodels and cross-language transformation rules.

Adopted from the Mealy machine definition [14], the transition in the eFSM, F, is a mathematical function defined as follows:

$$F : S \times \Sigma \rightarrow S \times \Gamma \tag{1}$$

where S is a finite, non-empty set of states. Σ is the input alphabet for a finite, non-empty set of symbols, and Γ is the output alphabet for a finite, non-empty set of symbols. We choose to formalize transition as function which precludes non-determinism, owing to the high-integrity needs of the VMC domain. The tradition formalization of transition as a relation allows for non-determinism, which is not suitable for the VMC domain. The arrow labeled T_{ij} in Fig. 3 shows an example transition from state S_i to state S_j.

The elements in the input alphabet and output alphabet are called events (e_{in} and e_{out} in Fig. 3). The FSM function uses "saturated" expressions for input and output events [15]. Events may have associated parameters when needed for input to the function performed during a transition or for the output

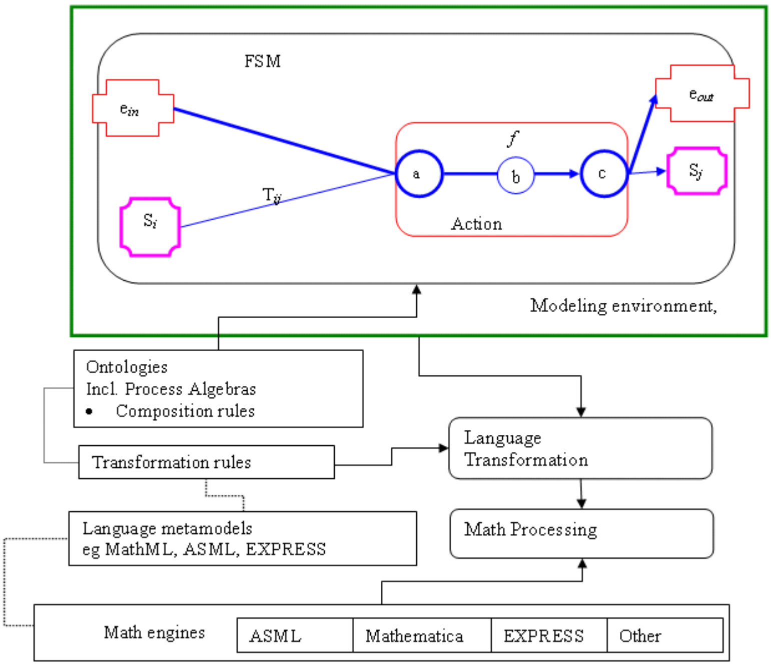


Fig. 3. eFSM for unambiguous cross-tool model exchange

generated by the function during the transition. The function performed during a transition, called an action, is defined mathematically as follows:

$$f : \sigma \rightarrow \gamma$$

where f is an action function (b in Fig. 3) with σ being a set of inputs related to the parameters associated with an input event (a in Fig. 3) in Σ and γ being a set of outputs related to the parameters associated with an output event (c in Fig. 3) in Γ. The guard condition, modeled as a function with a Boolean output, can also be associated with an input event, the truth-value of which is interpreted as presence or absence of the event. With this definition, a FSM can itself be treated as a mathematical function that maps an input alphabet into an output alphabet. In the eFSM, a FSM is a specialization of a mathematical function where each transition is also a function, whose input alphabet is a combination of event, guard, and state, and output alphabet, a combination of event and state.

3.1 Composition Rules

Compositions of FSM-s, transitions, and actions in eFSM are all constrained to mathematical function composition with one-to-one or many-to-one mapping from input to output. Complex functions in a model are composed from primitive functions or less complex functions. If a composite function involves control flow, e.g., branching or forking, the control flow must be modeled explicitly in a FSM conforming to the eFSM. Otherwise, a composite action is modeled as a sequential composition of functions. Interacting FSM-s are also composed mathematically to form a composite FSM. For sequential composition of FSM-s, the outputs of a FSM in the composition are the inputs of its immediate successor with the rules [16] specified as follows:

$$\begin{aligned} \Sigma_{F_1*F_2} &= \Sigma_{F_1} \cup \Sigma_{F_2} - (\Gamma_{F_1} \cap \Sigma_{F_2}), \\ \Gamma_{F_1*F_2} &= \Gamma_{F_1} \cup \Gamma_{F_2} - (\Gamma_{F_1} \cap \Sigma_{F_2}), \\ S_{F_1*F_2} &= S_{F_1} \cup S_{F_2}, \end{aligned} \tag{2}$$

where $F_1 * F_2$ is sequential composition of FSM F_1 and F_2 (as defined in Eq.(1)), as shown below:

$$F_1 * F_2 = \bigcup \begin{array}{l} \{f | f \in F_1, f(\gamma) \notin \Sigma_{F_2}\} \\ \{f | f \in F_2, f(\sigma) \notin \Gamma_{F_1}\} \\ \{f = f_1 * f_2 | f_1 \in F_1, f_2 \in F_2, f_1(\gamma) = f_2(\sigma)\} \end{array} \tag{3}$$

For parallel compositions, the inputs and outputs of constituent FSM-s obey the rules [17] specified as follows:

$$\begin{aligned} \Sigma_{F_1||F_2} &= \Sigma_{F_1} \cup \Sigma_{F_2}, \\ \Gamma_{F_1||F_2} &= \Gamma_{F_1} \cup \Gamma_{F_2}, \\ S_{F_1||F_2} &= S_{F_1} \times S_{F_2} \end{aligned} \tag{4}$$

where $F_1||F_2$ is the parallel composition of FSM F_1 and F_2, as shown below:

$$F_1||F_2 = \bigcup \begin{array}{l} \{f | f \in F_1, f(\sigma) \notin \Sigma_{F_2}\} \\ \{f | f \in F_2, f(\sigma) \notin \Sigma_{F_1}\} \\ \{f = f_1||f_2 | f_1 \in F_1, f_2 \in F_2, f_1(\sigma) = f_2(\sigma), f(\gamma) = \{f_1(\gamma)\} \cup \{f_2(\gamma)\}\} \end{array} \tag{5}$$

3.2 Constrained Generalization-Specialization Relationship

The eFSM supports extensions through a generalization-specialization relationship. It utilizes the generalization-specialization relationship from the object-oriented modeling paradigm as a technique to unify related concepts and thereby support integration and reuse. The eFSM constrains specialization to be performed by restriction and extension only, in order to eliminate ambiguity and to reduce computational complexity. Specialization by restriction may be performed by restricting the type of at least one element used in the more general model type to a specialization (e. g., subtype) of the corresponding element in the more general model. Restriction of the type of an element representing some value may also be performed by limiting the range of eligible values (i.e. the domain of a function), often expressed with an addition of constraint clauses. Extension of a model type is performed through addition of elements. For example, a FSM model type may be specialized to derive another FSM model type through the addition of a state or a transition.

A model type can also be specialized through a composition of elements from multiple more general model types, if the elements in these general types are mutually exclusive. An element in the specialized model type can be either of the same type as the element in its corresponding more general model type or a specialization of the element in the more general model type. The object-oriented analogy of this type of specialization would be a class C composed of class A and class B, is specialized to a class C' that is composed of A' and B where A' is a specialization of A.

Multiple levels of specialization are possible to form a generalization-specialization chain. With multi-level specialization, the eFSM can support the creation of reusable assets (types) through recursively-chaining type instances. At the first level of reusability are the types defined in the metamodeling environment, which include the FSM, State, Event, Transition, etc. as mentioned earlier. These elements in effect constitute the modeling constructs for the first level. When creating a FSM model, i.e., an instance of a FSM model type, as many instances of these modeling elements are created and assigned values as required to model the intended behavior. The behavior specification thus created in a FSM model can also be made a "type" (say level-2 type), and a collection of level-2 types constitutes the "type library" for the second level of reuse. At the second level, instances of level-2 types are created for a particular vehicle (or a vehicle product family). The level-2 types could also be specialized and placed in the type library, and then instantiated for a particular vehicle. A third, or in general n^{th}, level of reuse will include in its "type library" elements created in the $(n-1)^{th}$ level.

3.3 Ontology

The ontologies used in the eFSM are based on mathematical languages. The fundamental modeling elements in the eFSM have a one-to-one correspondence with mathematical concepts in the externally-defined mathematical specifications. Common elements include the mathematical function, its domain (including co-domain hereafter), and set. Domains include numbers with quantities extensible to physical quantities. The language elements for specifying constraints in the metamodeling environment map into first order logic (FOL). Structures of modeling elements, constructed in the modeling environment, can also be represented in FOL. To achieve semantics-preserving, unambiguous model transformation, we adopt a rule-based model transformation method with a set of formally-defined, isomorphic transformation rules. The mathematical reasoning can then be performed by established mathematical engines external to the modeling tool. Examples of such math-processing engines include Abstract State Machine Language (ASML), Mathematica, and ISO 10303-11, EXPRESS.

As an example of domain-specific specialization, definitions of Action related elements in the eFSM utilize an ontology which defines the basic knowledge of rigid body motion in VMC, related to displacement, time, velocity, acceleration, and jerk, and includes units of measurement, their dimensions, and unit balancing rules. When this ontology is specified as a constraint-set on functions relating these physical quantities in a composition, $f_1(f_2(x))$, the codomain of f_2, must match the domain of f_1, i.e., be the same set of physical quantities. Thus, this constraint-set is used to assure that the specification is unambiguous and consistent with the physics of the controlled process. This check on the specification prevents errors from propagating and multiplying in the D&I stages of the engineering process.

3.4 Language Transformation

In the systems engineering process, commercial tools are commonly used as the engineering interface for creation of work products at each stage. The work products, typically in the form of models, must have unambiguous semantics for (re)use across the stages. As the modeling languages adopted by different commercial tools are different, unambiguous semantics of models across tools and engineering stages cannot be achieved by directly using the tool native modeling languages. The models created in the tools must then be transformed to the eFSM-conformant models for exchanges with unambiguous interpretation and processing. This implies the semantic domain of a commercial tool must be mapped to the semantic domain of the eFSM. Such mapping is the metamodel-level transformation of modeling languages, and it requires the modeling constructs in the tool native modeling language have semantically equivalent modeling constructs in the eFSM. This can be achieved through constraining tool native modeling language followed by mapping of allowed tool native modeling language constructs to the eFSM modeling constructs unambiguously. The feasibility of such a mapping is assured if the semantic domain of the eFSM is wider than that of the tool's restricted native modeling language, though we do not provide a formal proof. To meet this condition and avoid introducing the ambiguity when modeling using a tool, only those modeling elements of the tool-specific modeling language, whose semantics can be unambiguously mapped to those of the eFSM elements, are allowed to be used in modeling. This requires applying domain-specific constraints to the tool native modeling languages. The modeling elements with tool-specific, implicit semantics, such as priorities of transitions captured in graphical layout, must be explicated before they can be used for modeling. To ensure the resultant model in a tool is eFSM-conformant, the constraints must be checked inside the tool, if the tool-APIs support user-specified constraints, or during exportation of the model. Models satisfying the constraints are eFSM-conformant.

The constrained tool-specific modeling elements can be transformed into semantic equivalent eFSM modeling elements (simple or composite) through a one-to-one mapping. The semantic, one-to-one mapping between the constrained tool-specific modeling elements and the eFSM modeling elements can be defined as transformation rules. Some tool-specific semantics defined unambiguously within a tool, which may be ambiguous across tools, such as data retention during event processing, can also be captured in the transformation rules. Additional semantic well-formedness rules, such as the presence of required elements and matches of their types, can also be defined as rules incorporated in the eFSM. The transformation of a model created natively in a tool to an eFSM-conformant model is then realized by applying transformation rules. The rules are encoded and operationalized using techniques such as a graph rewriting and transformation engine [18], and are applied to the tool-native models to obtain the eFSM-conformant models. When the transformation encounters elements for which a mapping rule is not defined, it flags a violation of the elements, thereby testing conformance of the eFSM. Rules for mapping from the "canonical form to the tool are part of future work.

3.5 Execution Semantics

Given the basic modeling elements, composition rules, constrained relationship and ontology, and parafunctional [1] specification support, the eFSM is further constrained to achieve the following execution semantics. The FSM processes only one event in each computation cycle. A computation cycle, modeled using a parafunctional element, starts from an arrival of an triggering event and ends with the production of the output. The next computation cycle begins only after the completion of the preceding computation cycle. All parts of the input are available before the computation cycle starts and are stable during the cycle. Discretized continuous control behavior is modeled as an action, triggered by a periodic event occurring at the fixed time period, required by the executing control algorithm. A simple unit of continuous control behavior can then be modeled as a function, $f(x)$. Its domain and co-domain are limited to a specified topological vector space. Multiple functions, f_1, f_2, and f_3, which are limited to the same topological vector space, may be composed sequentially as $f_1(f_2(f_3(x)))$ to create a complex behavior. These semantics assume the completion of the action before the start of next computation cycle.

The assumption on the action completion is specified as a rule and must be verified in order to ensure correct operation. To allow the verification, the eFSM is extended to associate some normalized equivalent of its worst case execution time (WCET).

4 Proof of Concept Exercises

Our proof-of-concept exercises is to examine the mapping from the semantic domains of two selected commercial tools, Rhapsody [19] from I-Logix and Stateflow [20] from Mathworks, to the eFSM semantic domain for unambiguous interpretation and processing. The exercises follow the transformation principles in Section 3.4. Since neither Stateflow nor Rhapsody provides API-s for specifying and checking user-specified constraints, the constraints and transformation rules were checked during and after transformation.

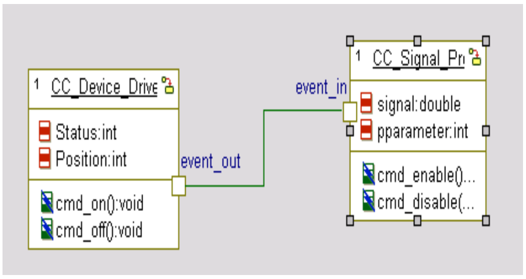
[1] Parafunctional refers to properties of software that are considered beyond the functional requirements, and is equivalent to Quality of Service (QoS) properties.

Table 1. Transformation rules from Rhapsody Statechart to eFSM ($r2e()$ is a injective mapping from a Rhapsody element to an eFSM element)

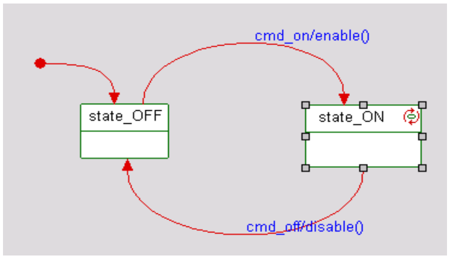
Rhapsody element	eFSM element	Mapping rules
basic states S_R	states S_F	$s_r \in S_R \Rightarrow (s_f = r2e(s_r))$ $\wedge(S_F = S_F \cup s_f)$
termination states S_R^t	final states S_F^t	$s_r \in S_R^t \Rightarrow (s_f = r2e(s_r))$ $\wedge(S_F^t = S_F^t \cup s_f)$
actions A_R	actions A_F	$a_r \in A_R \Rightarrow (a_f = r2e(a_r))$ $\wedge(A_F = A_F \cup a_f)$
triggers E_R	events E_F	$e_r \in E_R \Rightarrow (e_f = r2e(e_r))$ $\wedge(E_F = E_F \cup e_f)$
guards G_R	guards G_F	$g_r \in G_R \Rightarrow (g_f = r2e(g_r))$ $\wedge(G_F = G_F \cup g_f)$
transitions T_R	transitions T_F	$t_r \in T_R \Rightarrow (t_f.event = r2e(t_r.trigger))$ $\wedge(t_f.guard = r2e(t_r.guard))$ $\wedge(t_f.action = r2e(t_r.action))$ $\wedge(T_F = T_F \cup t_f)$
termination connectors C_R^t	final states S_F^t	$c_r \in C_R^t \Rightarrow (s_f = r2e(c_r))$ $\wedge(S_F^t = S_F^t \cup s_f)$
junction connectors C_R^j	transitions T_F	$(j_r \in C_R^j) \wedge (t_r^1 \ldots t_r^i, t_r^j \in T_R)$ $\wedge(\{t_r^1, ..., t_r^i\} \rightarrow j \rightarrow t_r^j)$ $\Rightarrow (t_f^1.event = r2e(t_r^1.trigger))$ $\wedge(t_f^1.guard = r2e(t_r^1.guard))$ $\wedge(t_f^1.action = r2e(t_r^1.action + t_r^j.action))$ $\ldots$ $\wedge(t_f^i.event = r2e(t_r^i.trigger))$ $\wedge(t_f^i.guard = r2e(t_r^i.guard))$ $\wedge(t_f^i.action = r2e(t_r^i.action + t_r^j.action))$ $\wedge(T_F = T_F \cup \{t_f^1, ..., t_f^i\})$
condition connectors C_R^c	states S	$(c_r \in C_R^c) \wedge (t_r^c, t_r^1 \ldots t_r^i \in T_R)$ $\wedge(t_r^c \rightarrow c_r \rightarrow \{t_r^1, ..., t_r^i\})$ $\Rightarrow (t_f^1.event = r2e(t_r^c.trigger))$ $\wedge(t_f^1.guard = r2e(t_r^c.guard + t_r^1.guard))$ $\wedge(t_f^1.action = r2e(t_r^c.action + t_r^1.action))$ $\ldots$ $\wedge(t_f^i.event = r2e(t_r^c.trigger))$ $\wedge(t_f^i.guard = r2e(t_r^c.guard + t_r^1.guard))$ $\wedge(t_f^i.action = r2e(t_r^c.action + t_r^j.action))$ $\wedge(T_F = T_F \cup \{t_f^1, ..., t_f^i\})$
action_on_entry $a_r^i \in s_r$	action a_f	$a_r^i \in s_r \Rightarrow (t_f.tostate = r2e(s_r))$ $\wedge(a_f = r2e(a_r^i))$ $\wedge(t_f.action = t_f.action + a_f)$ $\wedge(T_F = T_F \cup t_f)$
action_on_exit $a_r^o \in s_r$	action a_f	$a_r^o \in s_r \Rightarrow (t_f \in T_F) \wedge (t_f.fromstate = r2e(s_r))$ $\wedge(a_f = r2e(a_r^o))$ $\wedge(t_f.action = a_f + t_f.action)$ $\wedge(T_F = T_F \cup t_f)$
reaction_in_state $a_r^r \in s_r$	transition t_f	$a_r^r \in s_r \Rightarrow (t_f.fromstate = s_r)$ $\wedge(t_f.tostate = r2e(s_r))$ $\wedge(t_f.event = r2e(a_r^r.trigger))$ $\wedge(t_f.guard = r2e(a_r^r.guard))$ $\wedge(t_f.action = r2e(a_r^r.action))$ $\wedge(T_F = T_F \cup t_f)$

4.1 Rhapsody

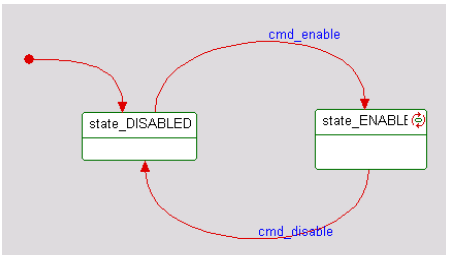
Rhapsody 6.0 employs Statecharts for state-based behavior modeling. In Rhapsody Statecharts, a *State* can be a basic state, or a termination state, or an or-state, or an and-state. Each state can have *action_on_entry*, *action_on_exit*, and *reaction_in_state* defined. A connector can be *Condition*, or *History*, or *Termination*, or *Junction*, or *Diagram*, or *Sync-Join*, or *Sync-Fork*. A transition has *Trigger*, *Guard*, and *Actions*. To support unambiguous model exchange, we mapped a defined subset of the Statechart to our eFSM with the rules, as shown in Table 1, according to the behavioral semantics of the modeling constructs. The transformation rules shown in Table 1 show the mathematical mapping of syntactic constructs of Rhapsody into eFSM, and are operationalize with a rule-based transformation engine [21] that matches the appropriate syntactic construct in Rhapsody and performs the mapping actions as shown in the table. For example, a junction connector j_r with a set of transitions $\{t_r^1 \ldots t_r^i, t_r^j\}$ in Rhapsody indicates that $t_r^1 \ldots t_r^i$ sharing some common actions in t_r^j. Consequently, the mapping rule for j_r transforms the junction connector with its transitions $\{t_r^1 \ldots t_r^i, t_r^j\}$ into a set of eFSM transitions $\{t_f^1 \ldots t_f^i\}$ with the event and guard of $t_f^k (1 \leq k \leq i)$ being the trigger and guard of t_r^k and the actions of t_f^k being the actions of t_r^k followed by the actions of t_r^j. Since the eFSM does not yet incorporate State hierarchies, composite states and state hierarchy were outside the defined subset. Similarly, since the eFSM execution semantics allows only one event to be processed each computation cycle, History is outside the defined subset.



(a) Sensor interface and signal processor



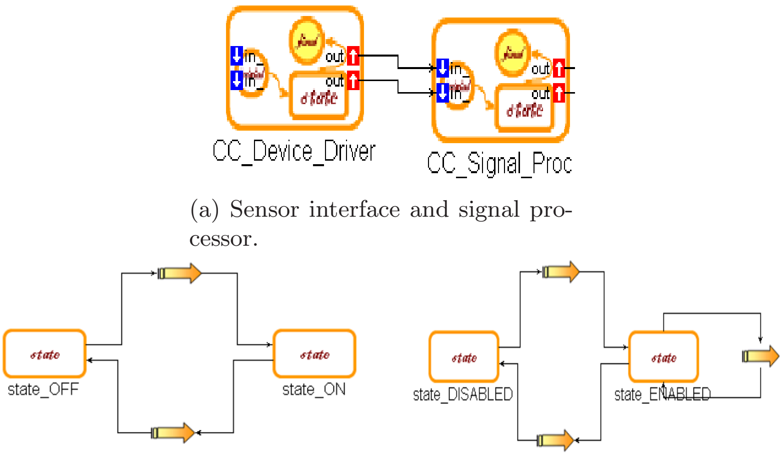
(b) Sensor interface



(c) Signal processor

Fig. 4. Behavioral models in Rhapsody

With the above defined transformation rules, we translated a part of a behavioral model for a cruise control defined using elements within the defined subset of Rhapsody Statecharts, to eFSM realized in the Generic Modeling Environment (GME) [22]. Fig. 4 and 5 show the sensor interface and signal processing portion of the whole model in Rhapsody and in eFSM in GME, respectively. After the transformation, we performed checks of the rules and constraints defined in the eFSM, and detected errors such as signal mismatches in value ranges or units, which were not detectable in the Rhapsody model.



(a) Sensor interface and signal processor.

(b) Sensor interface

(c) Signal processor

Fig. 5. Behavioral models in GME conforming to our eFSM

As can be seen, the states in Rhapsody models were transformed to states in eFSM. The sensor interface Statechart model in Rhapsody contained entry and exit actions, which were combined with actions of corresponding transitions as composite actions in eFSM-compliant sensor interface transitions. The composition followed the function composition rules and was implemented as mapping rules in Table 1. Similarly, the *reaction_in_state* action in *state_Enable* state in the Rhapsody signal processor model was transformed to a separate, self-loop transition in eFSM-compliant signal processor model. In addition to the mapping rules in Table 1, other constraints and rules implemented in eFSM, such as the constraint for signal type match and the rule of generalization-specialization relationship, were also enforced during the transformation.

4.2 StateFlow

Stateflow models behaviors of dynamical systems based on finite state machines, and uses a different Statechart formalism with additional semantic elements, notably junction structures, and flow charts. Also, the Stateflow action language

Table 2. Transformation rules from Stateflow to eFSM ($s2e()$ is a injective mapping from a Stateflow element to an eFSM element)

Stateflow element	eFSM element	Mapping rules
leaf states S_S	states S_F	$s_s \in S_S \Rightarrow (name(s_f) = s2e(s_s))$ $\wedge(S_F = S_F \cup s_f)$
actions A_S	actions A_F	$a_s \in A_S \Rightarrow (a_f = s2e(a_s))$ $\wedge(A_F = A_F \cup a_f)$
condition actions C_{A_S}	actions A_F	$c_{a_s} \in A_S \Rightarrow (a_f = s2e(c_{a_s}))$ $\wedge(g_f = s2e(c))$ $\wedge(A_F = A_F \cup a_f)$ $\wedge(G_F = G_F \cup g_f)$
triggers E_S	events E_F	$e_s \in E_S \Rightarrow (e_f = s2e(e_s))$ $\wedge(E_F = E_F \cup e_f)$
guards G_S	guards G_F	$g_s \in G_S \Rightarrow (g_f = s2e(g_s))$ $\wedge(G_F = G_F \cup g_f)$
transitions T_S	transitions T_F	$t_s \in T_S \Rightarrow (t_f.event = s2e(t_s.trigger))$ $\wedge(t_f.guard = s2e(t_s.guard))$ $\wedge(t_f.action = s2e(t_s.conditionaction))$ $\cup(s2e(t_s.action))$ $\wedge(T_F = T_F \cup t_f)$
entry_action $a_s^i \in s_s$	action a_f	$a_s^i \in s_s \Rightarrow (t_f.tostate = s2e(s_s))$ $\wedge(a_f = s2e(a_s^i)$ $\wedge t_f.action = t_f.action \cup a_f)$ $\wedge T_F = T_F \cup t_f$
exit_action $a_s^o \in s_s$	action a_f	$a_s^o \in s_s \Rightarrow (t_f.fromstate = s2e(s_s))$ $\Rightarrow (a_f = s2e(a_s^o))$ $\wedge(t_f.action = a_f \cup t_f.action)$ $\wedge(T_F = T_F \cup t_f)$
during_action $a_s^d \in s_s$	transition t_f	$a_s^d \in s_s \Rightarrow (t_f.fromstate = s2e(s_s))$ $\wedge(t_f.tostate = s2e(s_s))$ $\wedge(t_f.event = s2e(E_S - \{e_s \mid t_s.fromstate = s_s\})$ $\wedge(t_f.action = s2e(a_s^d))$ $\wedge(T_F = T_F \cup t_f)$

differs from Statecharts, and has been extended to reference Matlab functions, and Matlab workspace variables.

As in the Rhapsody case, we restricted Stateflow modeling elements to a defined subset, disallowing Junctions, History, State hierarchies, and Function States [20] to avoid ambiguity. Table 2 summarizes the transformation rules for mapping the defined Stateflow subset into eFSM. Similarly, the mapping rules are based on the behavioral semantics of the modeling constructs in Stateflow and eFSM. For example, the entry actions a_s of a state s in Stateflow is transformed into the last actions of each incoming transition t_f of the s_s in the eFSM model. According to the rule, $t_f.tostate = s2e(s_s)$ identifies all transitions t_f in the eFMS, whose destination state is s_s. The Stateflow action a_s is then transformed into the eFSM action a_f and is added to the transition's action set with $t_f.action = t_f.action \cup a_f$. Due to different graphical representations, the transformed eFSM model in Fig. 5 has the trigger events and the transition actions as embedded elements in the blockarrows instead of representing as textual labels as in the Rhapsody model. One major advatage of using embedded type elements over textual labels is that the type elements support better type checking.

In addition to the above rules, while parsing and mapping actions, we checked that (i) the arguments of the actions are from the set of inputs and/or outputs to the Stateflow model, similar to what the Stateflow compiler does, and (ii) the functions are contained in a pre-defined set of mathematically well-formed functions. According to the Stateflow semantics, *Data* variables are persistent and retain their their values over multiple computation cycles, while *Event*s are transient and consumed in a single computation cycle. These implicit semantics results in creation of unintentional state variables. In the transformation rules specified here these are transformed to explicit definition in eFSM as parameterized *Event*s carrying *Data* of sampled signals. This results in elimination of the implicitly defined state variables in the original Stateflow model.

As in the Rhapsody case, we have been able to map Stateflow models created within the eFSM-imposed constraints, into eFSM models in GME. We are currently investigating techniques for natively enforcing the eFSM restrictions within the Stateflow environment using the Stateflow provided API-s.

4.3 Lessons Learned

Through the proof-of-concept exercises, we gathered some valuable lessons on modeling language support for high integrity VMC control software. Modeling language support for strong type and modeling constraints are essential for unambiguous model exchange. For example, while the signals in both Rhapsody and Stateflow can be strongly typed with a specified unit, these tools do not incorporate any automated check for type compatibility prior to simulation or code generation. Some of the typing errors related to programming data-types are occasionally caught by a compiler; however, the more serious ones related to units and dimensionality are never caught since programming languages do not offer any abstractions for capturing physical quantities. In such a modeling environment, integration of discrete and continuous control behaviors in a unified, unambiguous model is not possible. Large amounts of effort have been spent on reducing model ambiguity through restrictions using some ad hoc approach such as a style guide for Stateflow. However, these approaches have not enabled unambiguous exchange of models across different tools.

As the behavioral models for VMC have complex interactions, ad hoc transformation is infeasible. The mathematical foundation of the eFSM allows creation of rules and constraints formally so that they can be interpreted and executed by machines automatically. Our experiences indicate that domain-specific rules and constraints are the key to support correct-by-construction modeling and unambiguous model exchange. Such domain-specific rules allow unambiguous semantic model transformation, thus preventing errors in the model and enabling a correct-by-construction process. For example, a constraint that any transition involving physical devices must be explicit and deterministic with all exceptions and interruptions captured as events will be implemented as a rule in eFSM.

Many semantic mistakes made by the designer during modeling, which cannot be captured otherwise, can be captured with such domain-specific rules.

5 Conclusions and Future Work

Semantics-preserving cross-tool model exchange is a key requirement to support a correct and efficient systems engineering process. In this paper, we have presented a math-based eFSM to enable software engineering of high integrity systems, e.g. drive-by-wire vehicles, with higher confidence and lower effort than current techniques. The eFSM contains modeling elements with explicitly-defined generalization-specialization relationship, mathematical composition rules and constraints, and domain-specific ontology. It also enables mathematical reasoning, transformation, and checking for the satisfaction of system requirements from early stages of the engineering life cycle. Models conforming to the eFSM can be unambiguously exchanged across different tools. The developed eFSM enables better "process efficacy" in the systems engineering processes, mostly in the requirements specification and verification and validation aspects.

It should be noted that our approach involves overlaying a restricted semantic domain (eFSM semantics) on the wider semantic domains of COTS tools such as Stateflow and Rhapsody. Enforcing this common semantic domain across multiple tools enables semantic preserving transformation and reduces the problem of verifying transformation correctness from establishing behavioral equivalence to structural equivalence.

One of the consequential challenge of this approach lies in imposing restrictions on the engineers using the Stateflow and Rhapsody tools. This is both a technological and an educational challenge. The availability of certain features in a tool, despite their semantic ambiguity, makes it attractive for the users. Automatic overlays, and online constraint checkers embedded within tools may make it technologically feasible to impose the restrictions, by automatically ensuring conformance to the high confidence high integrity design subset. Educational aids must also be created to assist the developers in understanding the cause and impact of the ambiguity in the use of certain abstractions. Some degree of restrictions are already in use by way of adherence to "best practice" and "safety guidelines" developed by Automotive Manufacturers, and other bodies such as Mathworks Automotive Advisory Board (MAAB).

The restrictions also impose challenge on the overall scalability of our approach, since the restrictions while improve the semantic preciseness of the models, also increase the effort in representing some behaviors. For example history driven behaviors can be conveniently represented with history junctions in Stateflow, and have to be otherwise represented with larger number of states. The transformations as described earlier are polynomial complexity and are not subject to scalability concern.

The eFSM represents early work on building the foundations for correct-by-construction process for VMC domain. However, much work remains to be done in the area of transformation and verification. We will continue this research to extend and evaluate the eFSM through constructing challenge problem model sets of representative VMC applications and platforms, and exercising the full systems engineering process with tool-assisted modeling and transformation. The extensions will include additional domain-specific rules for VMC modeling and transformation. The evaluations will include examining the breadth of the eFSM applicability, i.e., the scope of the domains over which various rule-sets hold, proving semantic mapability between tool native modeling elements and the eFSM modeling elements in both directions, and studying the effect on software correctness, process efficacy, system complexity reduction, and overall scalability of eFSM.

References

1. Birla, S.: Challenge problems for model-based integration of embedded systems (MoBIES. DTIC AFRL-IF-WP-TR-2004-1523 (2004)
2. Torngren, M., Larses, O.: Characterization of model based development of embedded control systems from a mechatronic perspective: drivers, processes, technology and their maturity. Technical Report TRITA-MMK 2004:23, Mechatronics Lab, Department of Machine Design, Royal Institute of Technoglogy, KTH, Stockholm, Sweden (2004)
3. Mellor, S.J., Clark, A.N., Futagami, T.: Model-driven development. IEEE Software 20(5), 14–18 (2003)
4. Price, D.: Ap233 state machine support (2005), `http://www.ap233.org/ModuleSets/Behavior/AP233_State_Machine_2005-09-15_update.zip/view`
5. Object Management Group: Uml for systems engineering, request for proposal (2003), `http://www.sysml.org/artifacts/refs/UML-for-SE-RFP.pdf`
6. Object Management Group: Unified modeling language: Superstructure, version 2.0 (2005), `http://www.omg.org/docs/formal/05-07-04.pdf`
7. International Society for Automotive Engineers (SAE) AADL Team: Architecture analysis & design language (AADL) standard (2004), `http://www.aadl.info`
8. EAST-EEA Partners: East-eea architecture description language, version 1.0.2 (2004), `http://www.east-eea.net/start.asp`
9. Hamon, G., Rushby, J.: An Operational Semantics for Stateflow. In: Wermelinger, M., Margaria-Steffen, T. (eds.) FASE 2004. LNCS, vol. 2984, pp. 229–243. Springer, Heidelberg (2004)
10. Anton, J., da Costa, P., Errington, L.: Formal synthesis of generators for embedded systems. Technical report, Kestrel Technology, Palo Alto, CA (2005)
11. Arango, G.: Domain Analysis. In: Marciniak, J. (ed.) Encyclopedia of Software Engineering, vol. 1, pp. 424–434. Wiley, Chichester (1994)
12. The Software Engineering Institute (SEI): A framework for software product line practice version 4.2 (2005), `http://www.sei.cmu.edu/productlines/framework.html`
13. Chen, K., Sztipanovits, J., Neema, S.: A case study on semantic unit composition. In: MISE 2007: Proceedings of the International Workshop on Modeling in Software Engineering, Washington, DC, USA, p. 3. IEEE Computer Society Press, Los Alamitos (2007)
14. Villa, T., Kam, T., Brayton, R.K., Sangiovanni-Vincentelli, A.L.: Synthesis of Finite State Machines: logic Optimization. Kluwer Academic Publishers, Dordrecht (1997)

15. Girault, A., Lee, B., Lee, E.A.: Hierarchical finite state machines with multiple concurrency models. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 18(6), 742–760 (1999)
16. Lee, E.A., Varaiya, P.: Structure and Interpretation of Signals and Systems. Addison-Wesley, Reading (2003)
17. Broy, M.: Algebraic specification of reactive systems. In: Nivat, M., Wirsing, M. (eds.) Proceedings of the 5th International Conference on Algebraic Methodology and Software Technology, Lecture Notes in Computer Science, p. 487. Springer, Heidelberg (1996)
18. Agrawal, A., Karsai, G., Shi, F.: Graph transformations on domain-specific models. Technical Report ISIS-03-403, Institute for Software Integrated Systems, Vanderbilt University, Nashville, TN (2003)
19. I-Logix Inc.: Rhapsody user guide (2005)
20. The MathWorks: Stateflow and stateflow coder, user's guide version 6 (2005), http://www.mathworks.com/access/helpdesk/help/pdf_doc/stateflow/sf_ug.pdf
21. Karsai, G., Agarwal, A., Shi, F., Sprinkle, J.: On the use of graph transformation in the formal specification of model interpreters. Journal of Universal Computer Science 9(11), 19–27 (2003)
22. Institute for Software Integrated Systems: The generic modeling environment (2005), http://www.isis.vanderbilt.edu/Projects/gme