Available online at www.sciencedirect.com
SCIENCE DIRECT
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 109 (2004) 43–56
www.elsevier.com/locate/entcs

Semantic Translation of Simulink/Stateflow Models to Hybrid Automata Using Graph Transformations

Aditya Agrawal[1] Gyula Simon[2] Gabor Karsai[3]

Institute for Software Integrated Systems (ISIS)
Vanderbilt University
Nashville, TN 37235, USA

Abstract

Embedded systems are often modeled using Matlab's Simulink and Stateflow (MSS), to simulate plant and controller behavior but these models lack support for formal verification. On the other hand verification techniques and tools do exist for models based on the notion of Hybrid Automata (HA) but there are no tools that can convert Simulink/Stateflow models into their semantically equivalent Hybrid Automata models. This paper describes a translation algorithm that converts a well-defined subset of the MSS modeling language into an equivalent hybrid automata. The translation has been specified and implemented using a metamodel-based graph transformation tool. The translation process allows semantic interoperability between the industry-standard MSS tools and the new verification tools developed in the research community.

Keywords: Graph Transformations, Embedded Systems, Semantics, Hybrid Systems.

1 Introduction

Model-based development of embedded systems is a process that uses explicit domain-specific constructs with well-defined semantics to represent, analyze, and synthesize systems [1]. A model should be a faithful and formal description of a system, which can be used in analysis (to verify the various

[1] aditya@isis.vanderbilt.edu
[2] simon@isis.vanderbilt.edu
[3] gabor.karsai@vanderbilt.edu

doi:10.1016/j.entcs.2004.02.055

properties of a system), and in synthesis (to actually construct the real system). In model-based development often many design tools are used for different needs. These tools need to be integrated in a coherent framework that ensures *semantic interoperability*. The various design tools must share semantics: that is the meaning of a model must be the same across multiple tools. One such need comes up in the embedded systems community where Matlab Simulink/Stateflow (MSS) is used for simulation while hybrid automata based tools (like, for instance, Charon [2]) are used for verification.

This paper describes the "semantic translator" that transforms models expressed in the MSS language into Hybrid System Interchange Format (HSIF). HSIF is an XML based standard developed by a community of researchers to represent dynamic networks of hybrid automata. The goal of the translator is to allow MSS models to be verified by HSIF based verification tools. In order to make the verification results meaningful the translation must preserve the semantics of the MSS models.

The problem of semantic translation problem between MSS and HSIF can be posed as follows: Given the model of a dynamic system in MSS, compute an equivalent dynamic system model in HSIF, which produces the same execution traces when executed, given the operational semantics of HSIF. For pragmatic reasons, we had to relax this requirement. First, MSS includes procedural components which are impossible to express in HSIF; we had to impose restrictions on MSS and allow only a subset of the MSS modeling language. Second, HSIF was defined using mathematical definitions in English, and not operationally (i.e. not via a simulation algorithm). Therefore, we had to come up with a mapping between constructs available in HSIF (e.g. discrete locations, differential equations, transition guards, etc.) and similar constructs in MSS such that the two models describe the same dynamic system.

A graph transformation language called Graph Rewriting and Transformation (GReAT) has been used to describe (and simultaneously to implement) the translator from MSS to HSIF. In the subsequent sections we describe the inputs and the outputs of the tool, specify the translation strategy, describe how we specified the transformations, and give an illustrative example for the use of the translator. We have verified the translation using test examples, as the complexity of the translator precludes the use of currently available formal techniques.

2 The inputs and outputs of the semantic translator

2.1 The output: HSIF

HSIF is an interchange format that allows representation of hybrid systems using dynamic networks of hybrid automata. The detailed specification is available in [3]. The automata in HSIF follow the definition of hybrid automata [4] with a finite number of locations (or discrete states), where each location has a number of differential and algebraic equations associated with it. Differential equations capture continuous time dynamics in that location, while algebraic equations describe dependencies among variables. HSIF is capable of expressing networks of hybrid automata, where the automata can interact with each other using signals and shared variables. Signals are single writer-multiple reader variables that follow synchronous semantics, while shared variables can have multiple writers and multiple readers.

2.2 The input: A subset of the MSS language

Simulink has a rich set of model elements (Simulink blocks) covering various areas of signal processing, and continuous dynamics and discrete behavior can be mixed in arbitrarily. On the other hand, HSIF has a clean separation between continuous and discrete behavior. Mapping arbitrary MSS models that have complex interactions between continuous and dynamic behavior are very difficult to transform into a HA. The pragmatic solution was to choose a subset of Simulink/Stateflow that maintains a clean separation between the continuous and discrete behavior. We have also restricted the supported primitive blocks from MSS to a carefully chosen set that provides a useful coverage. The supported Simulink blocks are as follows:

- Continuous time blocks: Integrator, State-space, Transfer Function, Zero-Pole
- Mathematical operators: Product, Sum, Gain, Min/Max, and any single-input/single-output function (Abs, Trigonometric, etc.) No logical blocks are allowed in the current implementation.
- Sources: Constant, In, and Sinks: Out
- Nonlinear elements: Switch
- Stateflow diagrams

The input models must comply with the following restrictions: (1) Stateflow diagrams can receive and provide continuous signals from and to Simulink. (2) Stateflow can also provide *switching signals*, that are always connected to the control input of a Switch block. (3) Switches can be controlled *only* by these

switching signals. These restrictions result in a clear separation of discrete and continuous behavior where all structural changes on the system are made through switches. Intuitively, each combination of these switches corresponds to a discrete location of the HA.

3 Example: Tank Level Control

To illustrate what steps a translation algorithm has to take, an example is provided in this section. As shown in Figure 1, there is a tank containing fluid, with an inlet pipe and two outlet pipes. Each pipe has a valve, named V1, V2 and V3 that can be in either open or closed state. A valve is modelled as a switch in MSS. Sensors can sense the height of fluid in the tank (h) and the flow through valve V3 (em_flow). A controller regulates the system using the state machine shown on the figure. In the initial state of the system V1 is closed and V2 is open. When the height of the tank goes above 10 units then outlet values V1 and V3 are opened. When the flow through V3 becomes greater than 5 units the inlet value V2 is closed. The inlet V2 is opened and outlet V1 is closed when the fluid level drops below 8 units.

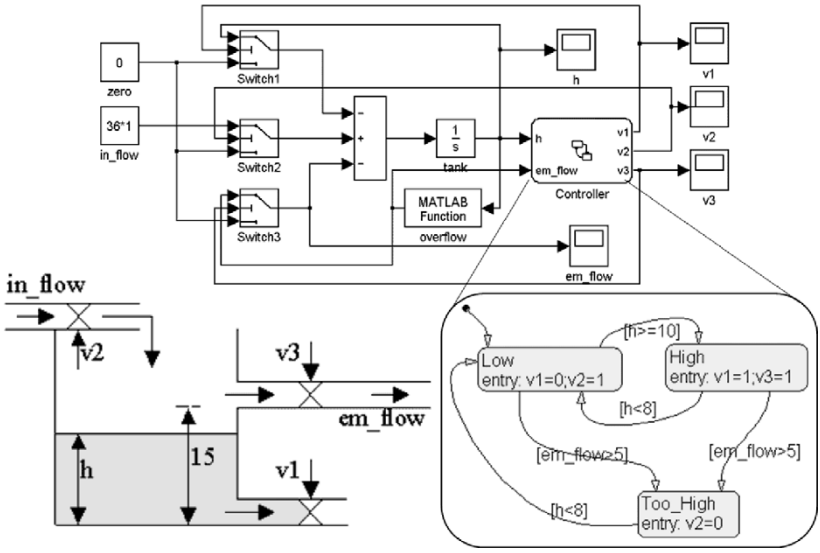

Fig. 1. A tank with three valves

Looking at the models, the number of locations in the final hybrid automata is not apparent. On closer inspection we see that the in the initial state *Low*, valve V1 is closed and V2 is open however the value of value V3 is unspecified, thus the initial state has discrete behavior, represented by the

opening or closing of V3. Thus state *Low* needs to be split into two states such that one of the states is active when V3 is open, while the other one is active when the V3 is closed, connected via a state transition. Having inspected the entire system and the controller's state machine, the resulting state machine diagram can be drawn up as shown in Figure 2.

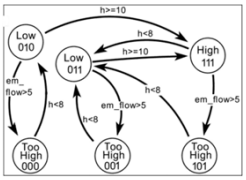

Fig. 2. The "true" (hybrid automata) state machine for the tank example

After all the discrete states are identified, the next step is to find the differential equations for each state. Since the value of the switches are all defined for a given state, the Simulink diagram is now purely continuous and a variable substitution can be used to find the differential equation. Differential equations are calculated from the output of the integrator block (see block with 1/S in Figure 2). For example, for location *High111* in Figure 2 the differential equation for the tank (block 1/S in Figure 2) block can be found as follows. Let the output of each block have the same name as the block. Then, $\frac{d}{dt}(tank) = Sum$, where Sum is the output of the summation block that can be substituted with the sum of its inputs: $\frac{d}{dt}(tank) = (-Switch1 + Switch2 - Switch3)$

Since the settings of the switches for this location are known, those paths will be chosen. Value 1 indicates that the top most input of the switch is passed through. Thus, Switch1 will be replaced by the tank variable. Switch2 is replaced by 36*1 and Switch3 is replaced by the output of the MATLAB function which is 3*max(0,tank-15). Finally the differential equation of the tank level is:

$$\frac{d}{dt}(tank) = -tank + 36 - 3 * max(0, tank - 15)$$

4 The translation algorithm

This section gives a formal definition for the transformation algorithm.

Definition 4.1 The flat Stateflow state machine contains the set of states $S = \{s_1, s_2, s_3, ...s_N\}, s_1$ being the initial state. The set of transitions is $T \subseteq S \times S$ where $t_{i,j} \in T$ is a transition from s_i to s_j. The corresponding transition condition is denoted by $w_{i,j}$.

Definition 4.2 An output variable in the Stateflow diagram is called a switching signal if it is connected to a Control Input of a Switch block in the

Simulink diagram. The set of switching signals in the state machine is $Q = \{q_1, q_2, q_3, ...q_M\}$. The value of the switching signal q in state s is $value(q, s)$.

Definition 4.3 The *switch value* of a switching signal q in state s is the following:

$$switchvalue(q, s) = \begin{cases} 1 \text{ if } value(q, s) \geq threshold(b) \\ 0 \text{ otherwise} \end{cases}$$

where b is the unique Switch block connected to q.

Definition 4.4 For a switching signal q and state s_i, $defined(q, s_i) = true$ if either of the following conditions hold:

- q is explicitly set in s_i, or
- there exist a switch value u, such that for all j for which $t_{j,i} \in T$ it is true that $defined(q, s_j)$ and $switchvalue(q, s_j) = u$.

Definition 4.5 The *rank* of state s is the number of switching signals that are defined in s. The *defect* of s is defined as $defect(s) = M - rank(s)$.

Definition 4.6 The sequence of undefined switching signals in s_i is defined as $U_i = \langle q_{k_1}, q_{k_2}, q_{k_3}, ..., q_{k_{defect(s_i)}} \rangle$, where $defined(q_{k_l}, s_i) = false$ for all $l = 1, 2, ...defect(s_i)$, and $k_1 < k_2 < k_3 < ... < k_{defect(s_i)}$.

The algorithm consists of the following **steps**.

Step 1. Each state s_i is split into $D = 2^{defect(S_i)}$ locations. The set of locations generated from s_i is $\sum_i = \{\sigma_{i,1}, \sigma_{i,2}, ..., \sigma_{i,D}\}$.

Definition 4.7 The switch code of location $\sigma_{i,j}$ is a binary sequence of length M, denoted by $C_{i,j} = \langle b_{i,j,1}, b_{i,j,2}, ..., b_{i,j,M} \rangle$. The binary values are defined as follows:

$$b_{i,j,k} = \begin{cases} switchvalue(q_k, s_i) \text{ if } q_k \notin U_i \\ bit(j-1, n) \qquad\quad \text{if } q_k = q_{k_n}, \text{ where } U_i = \langle q_{k_1}, ..., q_{k_{defect(s_i)}} \rangle \end{cases}$$

The function $bit(x, y)$ defines the y^{th} bit of the binary representation of x, the 1^{st} bit being the least significant bit.

Definition 4.8 The coloring is defined on the elements of the switch code. The binary values of the code are either *black* or *red*, as follows:

$$color(b_{i,j,k}) = \begin{cases} red \quad \text{if } q_k \in U_i \\ black \text{ if } q_k \notin U_i \end{cases}$$

Step 2. The locations are coded and colored according to Definition 4.7 and Definition 4.8.

Step 3. Create a transition $\tau_{i,j,n,m}$ between $\sigma_{i,n}$ and $\sigma_{j,m}$ if $t_{i,j} \in T$, and there is no k such that $b_{i,n,k} \neq b_{j,m,k}$ and $color(b_{j,m,k}) = red$. The transition guard for this transition is the predicate $w_{i,j}$.

Definition 4.9 The set of all transitions in the HSIF description is denoted by Φ.

Definition 4.10 The Simulink diagram containing M Switch blocks describes the reconfigurable dynamic system χ. The dynamic system with a particular setting of the switches with switch values $x_1, x_2, ..., x_M$ is denoted by $\chi(x_1, x_2, ..., x_M)$.

Step 4. For each state s_i copy the algebraic equations defined in the state to locations $\sigma_{i,j}$, for all $j = 1, 2, 3, ..., 2^{defect(s_i)}$. For each location $\sigma_{i,j}$ generate the additional algebraic and differential equations of the system $\chi(C_{i,j})$.

Step 5. Choose $\sigma_{1,1}$ to be the initial location.

Step 6. Add the following invariants to location $\sigma_{i,j}$:

- switching signal values from the entry action of s_i, and
- $\neg(\bigvee_m W_{i,m})$ for all indices m for which there exist n such that $\tau_{i,m,j,n} \in \Phi$. The operations $\neg$ and $\vee$ are the logical not and or operations, respectively.

Definition 4.11 The *location dependency graph* is a directed graph on the set $\Sigma_1 \cup \Sigma_2 \cup ... \cup \Sigma_N$ with edges Φ. A location σ is *unreachable* if there is no directed path in the location dependency graph from $\sigma_{1,1}$ to σ.

Step 7. Prune all unreachable locations from the HSIF description. Also delete the transitions connected to unreachable locations.

5 GReAT: The transformation language

The translation algorithm described in the previous section has been implemented in the Graph Rewriting and Transformation (GReAT) language. GReAT is a tool that allows users to specify graph transformations in a graphical form with precise formal and executable semantics. In this paper only the necessary language constructs are explained, [5] describes the full approach and support tools, and the operational semantics of GReAT is formally defined in [10]. GReAT is based on the theoretical work of graph grammars and transformations [6][8][9] and belongs to the set of practical graph transformations systems, like AGG and PROGRES.

GReAT has two parts: (1) graph transformation language, and (2) control flow language. The graph transformation language is used to specify transformations on localized subgraphs and follows the Single Pushout (SPO)

algebraic approach [6]. A production (also referred to as rule) is the basic unit of transformation and it contains a pattern graph that consists of pattern vertices and edges. Each pattern element has an attribute called *role* that specifies what happens during the transformation step. A pattern element can play one of three roles: *Bind*, *Delete*, *New*. The execution of a rule involves matching every pattern object marked either *Bind* or *Delete*. If the match is successful and an (optional) guard condition is true, then for each match the pattern objects marked *Delete* are deleted from the match and objects marked *New* are created.

Traditionally, in graph grammars and transformations there is no ordering imposed on the productions, but practical model-to-model transformations often require strict control over the execution sequence. GReAT has a high-level control flow language built on top of the graph transformation language with the following constructs: (1) sequencing, (2) non-Determinism, (3) hierarchy, (4) recursion and (5) branching.

Sequencing is used to specify an order of execution for a set of transformation rules. For example, Figure 3 shows a sequence of rules, *CreateHierarchicalStateChart*, *HSM2FSM*, *CreateVarAs*, *StateSplitting* and *Reachability* which are executed sequentially. Hierarchy is also shown: the sequence is contained in a compound rule called the *StateflowPart* rule.

A "Test/Case" construct is used to choose between different execution paths, similarly to the 'if' statement in programming languages. In Figure 5, the compound rule *SetImplicitValues* contains a test called *TestImplicit* that contains two cases. The test will first try *Case?*, if *Case?* succeeds then the outputs will be passed to the respective output ports and similarly for *CaseDifferent*. Once all inputs have been evaluated the next rules in the sequence will be executed.

6 Implementing the algorithm in GReAT

The translation algorithm mentioned in Section 4 has been implemented using GReAT. It contains 131 rules, 40 compound rules and 22 test/cases. The implementation is divided into two parts, the first deals with finding all the discrete locations in the Simulink/Stateflow diagram and the second deals with inferring the continuous dynamics for each location.

6.1 Translating Stateflow

In the Stateflow part of the algorithm (see Figure 3), first the Stateflow models are converted into an internal representation in *CreateHierarchicalStateChart*. Next, the hierarchical concurrent state machine is converted to its equivalent,

"flat" finite state machine in *HSM2FSM*. Then in *CreateVarAs*, associations of Simulink switches with the states are transferred to the flat machine. At this stage *StateSplitting*, the splitting algorithm is performed (explained in detail in next paragraph). After all the required discrete states/locations have been found, *Reachability* is executed that performs reachability analysis on the models to eliminate all unreachable states. At this state we know the number of discrete states in the system and create the corresponding locations in HSIF.

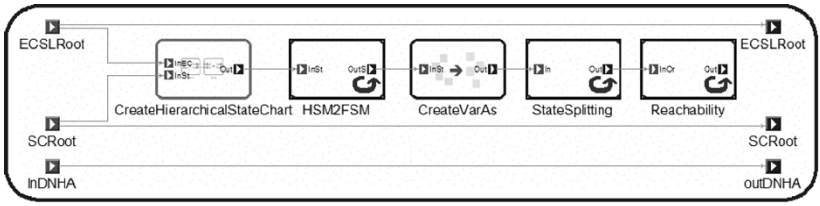

Fig. 3. The StateflowPart Rule

StateSplitting (see Figure 4) is one of the most complex parts of the mapping and it is done in stages. The first stage is *Infer_Implicit_Signals* and it implements **Step 2**. This is followed by *NewMachine* which creates an empty state machine. The *Create_State_Tribes* performs state splitting based on **Step 1**. The next step is *Transfer_Transitions* which implements **Step 3** by appropriately mapped transitions to the new machine. If the initial state was split, an initial state is selected according to **Step 5** in *CreateInit*. *CarryBlockRef* and *In2Out* perform housekeeping operations at the end.

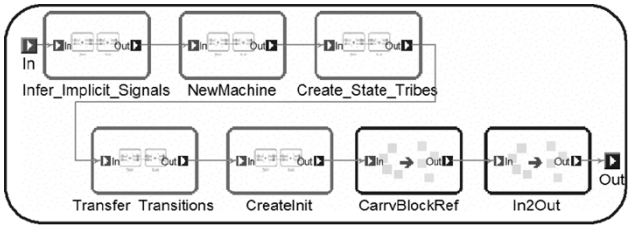

Fig. 4. The StateSplitting rule

The *Infer_Implicit_Signals* block in Figure 4 is performed repeatedly. In every iteration step, for every state the *SetImplicitValue* rule (see Figure 5) is called. In the *SetImplicitValue* block all switching signals with color red are chosen. If there is an incoming transition, which alters the state of the signal, then the transition is used to infer the new state of the signal. The translator will iterate until none of the signals changes during a run, i.e. the iteration reaches a fixpoint.

There are two main cases that can change the default interpretation of switching signal values. The first case is shown in Figure 5. For a given *State* and switch variable (called *Data* in the diagram), if there exists another state (*OtherState*) with a transition to *State*, *OtherState* may influence the value of *Data*. Each state has a relation with *Data*, and the relation has two attributes: *color* and *value*. *Color* can be either black or red, black implying that the state is set to the *value*, while red implying that the value was inferred. Value can be 0, 1, ?, X, where '?' specifies that the state doesn't influence data, while 'X' specifies that the state can set the data to either '0' and '1'.

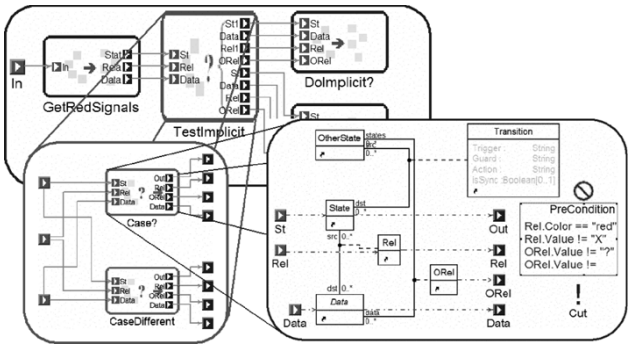

Fig. 5. The SetImplicitValues Rule

In *Case?* if *State's* relation with *Data* is 'red' and value is not 'X' and *OtherState's* relation with the *Data* is '?' then we can infer the value of the current state's relation with data is also '?'. In *CaseDifferent* if *OtherState's* relation with *Data* is not '?' and is not the same as *State's* relation with *Data*. In this case the *State's* relation with *Data* is altered according to the following rules. If *State's* relation was '?' then it will take *OtherState's* relation. If *State's* relation is not the same as *OtherState's* then it will take the value of 'X'.

6.2 Translating Simulink

After all the states of the hybrid automata have been created, the next step is to identify the algebraic and differential equations for each location (**Step 4**). The various steps in this translation are (1) identification of state variables, (2) identification of input and output variables (3) discovery of algebraic equations for dependent variables and (4) discovery of the differential equations for the state variables.

Each integrator block in Simulink is assigned a state variable. Each input port to the entire system becomes an input variable. Each source block of

Simulink also becomes an input variable. Sink blocks and output ports become output variables. Some intermediate variables are created for interfacing with Stateflow. These variables depend on other independent variables in the system.

After all the variables have been identified, the next step is to determine algebraic equations of dependent variables and differential equations for state variables. These equations are location dependent, thus for each location the differential and algebraic equations are inferred using a backward trace algorithm. Starting from a Simulink block/port the variable is associated with a backward trace is used to determine the blocks that provide input to the block. For each such block the block's type determines the kind of sub expression the block will add to the equation (see Table 1). The back trace yields a tree with the termination points being state variables, input variables and constants.

Table 1
Mapping Simulink blocks to sub expressions

Simulink Block Type	#in	#out	Corresponding Sub expression
Sum	2..*	1	$(\pm S_1 \pm S_2 \pm S_3 \pm \ldots \pm S_n)$
Mult	2..*	1	$(S_1 * S_2 * S_3 * \ldots * S_n)$
Switch	2+1	1	Chose path based on switch position in given location
Gain	1	1	(G*S)
Min/Max	2..*	1	$\text{Min}(S_1,S_2,S_3,\ldots,S_n)$/ $\text{Max}(S_1,S_2,S_3,\ldots,S_n)$
Single Input Function (Abs, Signum/ Saturate)	1	1	<function name>(S)
Integrator	1	1	(<integrator variable name>)
Constant	0	1	(<constant value>)

7 Translating the Tank Level Control example

This section shows how the algorithm described in Section 4 and implemented using GReAT in Section 6 can be used to translate the Simulink/Stateflow example described in Section 3 and Figure 1.

Initially, in state *Low*, the value of V3 is undefined while the value of V2 is undefined in state *High*. In state *Too_High* the value of V1 and V3 is undefined. After running the *Infer_Implicit_Signals* block there are some implicit values for undefined variables (see Figure 6(b)). For example, in state *Low*, the value of V3 can be both 0 and 1, while in state *High* the value of V2 was set to 1. After we determined the value of the switches in each state we can split the states that have switches with undefined values. In this example the state *Low* will be split into two while the state *Too_High* will be split into four new states (see Figure 6(c)).

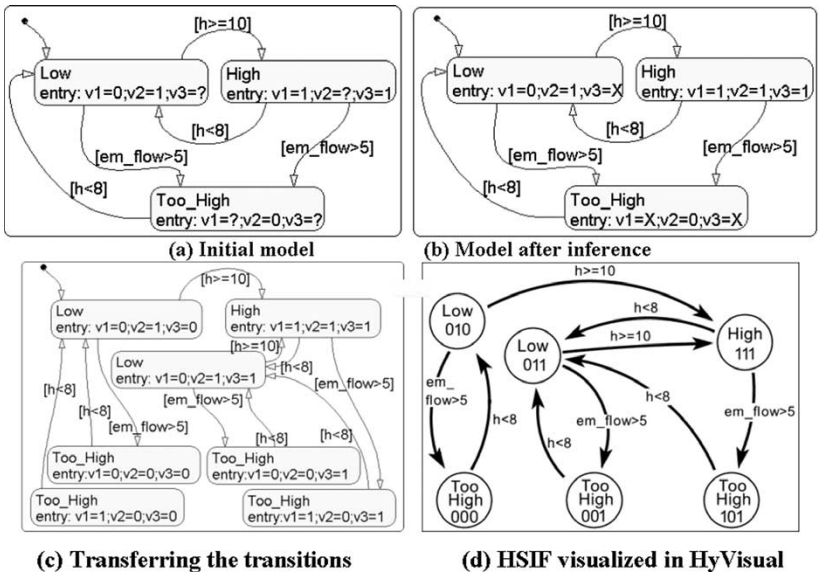

Fig. 6. Stages of Stateflow splitting

After the states are split, transitions from the original machine need to be transferred to the new larger machine. The algorithm takes care of mapping the transitions correctly. After the equivalent machine is created, reachability analysis is performed. The analysis will reveal that state *Too_High* with value of V1 = 1, V2 = 0 and V3 = 0 will never occur and it can thus be eliminated. Figure 6(d) shows the locations in HSIF. The visualization is provided by HyVisual [12]. After all the discrete locations have been identified, the continuous time dynamics for each location will be found using the backward trace algorithm.

8 Related Work

Semantic mapping between different design tools is a common problem one often encounters in practice. Frequently the mapping is implemented in code, although automated mappings have been discussed in literature and a subset of these have been implemented. In [16] graph transformations have been used to specify program transformations. Semantics of a hierarchical state machine have been defined by specification of a transformation to FSM in [17]. [18] describes the support of design patterns, while tool integration is via transformations in described in [19]. [13] describes the algorithm for mapping discrete-time Simulink blocks to Lustre. Verification of Simulink/Stateflow models has been performed in [14] using a model checker. The mapping how-

ever was performed by hand. Semantics of Stateflow have been described by defining a mapping to pushdown automata in [15].

9 Summary and future work

We have described a method for converting MSS models into HSIF models. The MSS models may contain continuous time blocks, Stateflow blocks, and switches, while the resulting HSIF model consists of a hybrid automaton that exhibits the same dynamic behavior as the original MSS model. The transformation has been specified using a formal technique based on graph transformations.

A natural next step for extending this work is the formal verification of the transformation itself. For practical applications, more features from the MSS blocks could be implemented, provided they are expressible in HSIF. Yet another potential work could be to extend HSIF with the capability of representing sampled-data systems, and extend the translator to map the "discrete time" blocks in MSS into the corresponding HSIF constructs. The latter one requires further research on the verification of hybrid automata that also have discrete-time dynamics.

10 Acknowledgement

The DARPA/IXO MOBIES program (F30602-00-1-0580) and the NSF ITR: "Foundations for Embedded and Hybrid Systems" has supported, in part, the activities described in this paper.

References

[1] J. Sztipanovits, and G. Karsai, "Model-Integrated Computing", IEEE Computer, Apr. 1997, pp. 110-112.

[2] R. Alur, T. Dang, J. Esposito, R. Fierro, Y. Hur, F. Ivancic, V. Kumar, I. Lee, P. Mishra, G. Pappas, and O. Sokolsky, "Hierarchical Hybrid Modeling of Embedded Systems." Proceedings of EMSOFT'01: First Workshop on Embedded Software, October 8-10, 2001.

[3] The Hybrid System Interchange Format, for details see http://www.isis.vanderbilt.edu/Projects/mobies/downloads.asp.

[4] T. A. Henzinger. "The Theory of Hybrid Automata", In Proc. of IEEE Symposium on Logic in Computer Science (LICS'96), IEEE Press, pp 278–292, 1996.

[5] Agrawal A., Karsai G., Ledeczi A., "An End-to-End Domain-Driven Software Development Framework", 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Anaheim, California, October 26, 2003.

[6] Rozenberg G. (ed.), "Handbook on Graph Grammars and Computing by Graph Transformation: Foundations"; Vol.1-2. World Scientific, Singapore, 1997.

[8] D. Blostein, H. Fahmy, and A. Grbavec: "Practical Use of Graph Rewriting"; 5th Workshop on Graph Grammars and Their Application To Computer Science, Lecture Notes in Computer Science, Heidelberg, 1995.

[9] Andries, M. et al., "Graph Transformation for Specification and Programming", Sci. Comput. Program., Vol. 34, No. 1, pp. 1-54, 1999.

[10] Karsai G., Agrawal A., Shi F., Sprinkle J., "On the Use of Graph Transformations for the Formal Specification of Model Interpreters", JUCS, November 2003.

[12] Hylands, C., Lee, E., Liu, J., Liu, X., Neuendorffer, S., Zheng, H.,"HyVisual: A Hybrid System Visual Modeler," Technical Memorandum UCB/ERL M03/1, University of California, Berkeley, CA 94720, January 28, 2003.

[13] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, "Translating Discrete-Time Simulink to Lustre", pp 84-99, Proc. of EMSOFT'03, Philadelphia, USA, 13-15 Oct., 2003.

[14] S. Sims, K. Butts, R. Cleaveland and S. Ranville, "Automated Validation Of Software Models", 16th International Conference on Automated Software Engineering, pages 91-96, Coronado Island, California, November 2001. IEEE Computer Society Press.

[15] A. Tiwari , "Formal Semantics and Analysis methods for Simulink Stateflow Models", Technical report, SRI International, 2002.

[16] U. Assmann, "How to Uniformly specify Program Analysis and Transformation", Proceedings of the 6 International Conference on Compiler Construction (CC) '96, LNCS 1060, Springer, 1996.

[17] A. Maggiolo-Schettini, A. Peron, "A Graph Rewriting Framework for Statecharts Semantics", Proc. 5th Int. Workshop on Graph Grammars and their Application to Computer Science, 1996.

[18] A. Radermacher, "Support for Design Patterns through Graph Transformation Tools", Applications of Graph Transformation with Industrial Relevance, Monastery Rolduc, Kerkrade, The Netherlands, Sep. 1999.

[19] A. Bredenfeld, R. Camposano, "Tool integration and construction using generated graph-based design representations", Proceedings of the 32nd ACM/IEEE conference on Design automation conference, p.94-99, June 12-16, 1995, San Francisco, CA.