Static Analysis and Automatic Code Synthesis of flexible FSM Model

Dohyung Kim
Researcher
ISRC, Seoul National Univ., Seoul, Korea
dhkim@iris.snu.ac.kr

Soonhoi Ha
Associate Professor
School of CSE, Seoul National Univ., Seoul, Korea
sha@iris.snu.ac.kr

Abstract – To describe complex control modules, the following four features are requested for extended FSM models: concurrency, compositionality, static analyzability, and automatic code synthesis capability. In our codesign environment we use a new FSM extension called flexible FSM model. It extends the expression capabilities by concurrency, hierarchy, and state variable while it maintains formal property. Because of formality and the structured nature of fFSM model, we can apply a static analysis method to find ambiguous behavior and synthesize software/hardware automatically, which is the main focus of this paper. We expect that the proposed technique can be applied to other compositional FSM extensions.

I. Introduction

Among diverse models of computation, finite state machine (FSM) is the most popular model to describe the control module of a system. Even though FSM is simple to use, its unstructuredness and state explosion problem due to system concurrency and memory prohibits FSM model from practical representation. Instead, many extensions have been proposed to overcome these problems.

To describe complex control modules, the extensions should support various kinds of concurrency. Another desired feature is compositionality whether the complex module can be represented as a composition of simpler modules. Then, modules can be easily reused to construct a large system. Moreover, because of their complexity, subtle design errors are difficult to find and the equivalence check of an implementation is not easy to achieve. Therefore, it is desired to have a static analysis method to check ambiguous behavior. Finally for fast prototyping, automatic hardware (or software) synthesis is needed from the extended model. However most existent FSM models are not successful to meet all those requirements.

In this paper, we present how those requirements can be satisfied with a proposed FSM model. In particular, static analysis and automatic code synthesis techniques are our main focus. The proposed FSM model is called *flexible* FSM model meaning that there are more than one way of expressing concurrency. fFSM model is devised as a part of system-level specification to specify control activity of the system in our codesign environment [1]. In our codesign environment, computation tasks are represented by dataflow model while at top-level a task model specifies the system behavior as a composition of control and computation tasks. Therefore, the fFSM model provides a way of sending control commands and signals to computation tasks. However, we will not handle this issue in this paper.

In section II, we review well-known FSM extensions and how they are related with the fFSM model. Section III will present the fFSM model in a formal way. The proposed static analysis technique to detect ambiguity in section IV. Automatic code synthesis from the fFSM model is explained in section V. Finally we conclude the paper at section VI.

II. Related Works

Statechart [2] introduces the AND composition of FSM subgraphs to represent the concurrency, and the OR composition for hierarchical and structural representation. He invented the notion of internal event by which the communication between concurrent FSMs and hierarchical FSMs performs. Many Statechart variants have been proposed to overcome the ambiguity problem of the original Statechart model [3] and to define the formal semantics of Statechart [4]. However, most of them are not compositional to keep the expressive power of the Statechart. But our fFSM model restricts some semantics of the Statechart, which makes the model compositional. For example, cross layer communication between FSMs is not allowed.

Codesign FSM [5] describes the system as the loosely coupled FSM networks. Unlike the basic FSM model that assumes synchrony hypothesis, CFSM takes into account the execution delay during a state transition. Even though it is a formal model and implementation independent [6], the network representation was admitted to be improper to model a complex control system. We borrow their formality to express the fFSM in a formal way in the next section.

Hierarchical FSM [7] specifies concurrency using the combination of heterogeneous models of computation. Instead of defining the AND composition of FSMs, to express concurrency, HFSM uses the outer model of computation within which FSMs are placed. For example, two FSMs in the outer dataflow model have data-driven interaction. In spite of theoretic interest, HFSM model seems difficult to understand and unclear how to synthesize from the specification.

Thus, the fFSM model takes the benefits of previous approaches: expressive power from Statechart and formal properties from CFSM. In addition, fFSM model has a special syntax to express memory in a compact form. The next section defines the fFSM model in a formal way.

III. Definition of fFSM Model

First, we define an event observed at a time instance as a tuple of an event name and a value as follows.

Definition 1 An event is defined as (e_n, e_v) where

- e_n is the name or symbol of the event,
- $e_v \in e_V$ is the value of the event and e_V is the set of allowed values.

We also define two special values ε and ϕ. The value ε specifies an occurrence of an event without any actual value. The value ϕ is a default value of all events, which indicates that the event is not valid at that time instance.

In fFSM model, we have three different types of events: input event, output event and internal event. We can read a value from an input event, write a value to an output event and read (or write) a value from(or to) an internal event. Because an internal event has both properties of an input event and an output event, internal event sets can be defined as an intersection of input event sets and output event sets. Definition 2 shows the definition of event sets in fFSM model.

Definition 2 Definition of event sets

- $I = \{(i_{n_1}, i_{V_1}), (i_{n_2}, i_{V_2}), \ldots\}$ is a finite set of input event names and of the corresponding finite sets of allowed values.
- $O = \{(o_{n_1}, o_{V_1}), (o_{n_2}, o_{V_2}), \ldots\}$ is a finite set of output event names and of the corresponding finite sets of allowed values.
- $IT = I \cap O$ is a finite set of internal event names and of the corresponding finite sets of allowed values.

For example, Fig.1 shows a fFSM graph that represents a concurrent FSM with an AND composition of two fFSM subgraphs. This example defines three input events but does not have any output event or internal event.

I = { (a, {ε, ϕ}), (b, {ε, ϕ}), (c, {ε, ϕ}) }
O = {}, IT = {}

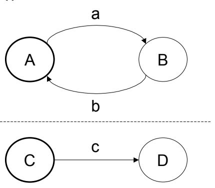

Fig. 1. A concurrent fFSM graph

fFSM model supports three different types of composition: AND composition, OR composition, and a variable state. A variable state is introduced to represent a memory in a compact form. It can be regarded as a separate concurrent FSM graph in which each state is mapped to a value that the variable state can have. Therefore, the number of values that can be assigned to the variable state should be finite. Although the variable state is defined as a state-set, it can be handled as a special event of which the value is preserved across time. Therefore, we can read (or write) a value from (or to) a variable state similar to an internal event.

To define the compositions in a formal way, we define each constituent FSM as a state-set. There are two state-sets in the fFSM graph of Fig.1. A variable state is also distinguished by a separate state-set. Then, we define the set of state-sets that compose the fFSM model, and their initial values (or initial states) as follows.

Definition 3 Definitions of state sets and related initial values (or states)

- $X = \{(x_{n_1}, x_{V_1}), (x_{n_2}, x_{V_2}), \ldots\}$ is a finite set of state-set names and of the corresponding finite sets of allowed states in the state-set.
- $V \subset X$ is a finite set of variable state names and of the corresponding finite sets of allowed states (or values).
- $R \subseteq \{(x_n, x_v) \mid (x_n, x_V) \in X, x_v \in x_V\}$ is a set of initial states. An initial state should exist for each state-set.

In a hierarchical fFSM graph, a special value ϕ should be an element of x_V to indicate an inactive state. As for the fFSM graph of Fig.1, we obtain the following.

X = {(s_1, {A, B}), (s_2, {C, D})}
V = {}
R = { (s_1, A), (s_2, C) }

Next, we need to define the transition between states. A transition connects two different states: one is a source state and the other is a destination state. It is also associated with a condition and actions. The condition is a Boolean expression composed of input events and variable states. An action assigns an output event with a function of input events and variable states. When the Boolean expression (the condition) of the transition meets, the expression of each action is evaluated and the result value is assigned to an output event or a variable state (Fig.2).

Definition 4 Transition set $F \subseteq f^{XI} \times F^G \times f^{XO} \times f^A$

- $f^{XI}, f^{XO} \subseteq \{(x_n, x_v) \mid (x_n, x_V) \in X, x_v \in x_V\}$ are a set of source states and destination states of a transition
- $f^G = f(e_{n_1}, e_{n_1}, \ldots), (e_{n_k}, e_{V_k}) \in I \cup V$ is a Boolean expression composed of input events and variable states
- $f^A \subseteq \{(r_n, f^R) \mid (r_n, r_V) \in O \cup V, f^R = f(e_{n_1}, e_{n_2}, ..) \subseteq r_V, (e_{n_k}, e_{V_k}) \in I \cup V\}$ is a set of actions which consist of a destination r_n and a function f^R composed of input events and variable states

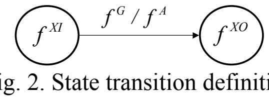

Fig. 2. State transition definition

For the fFSM graph of Fig.1, the transition set becomes

F = { ({(s_1, A)}, (a=ε), {(s_1, B)}, {}),
({(s_1, B)}, (b=ε), {(s_1, A)}, {}),
({(s_2, C)}, (c=ε), {(s_2, D)}, {}).

From the definitions 2 to 4, we can define flexible FSM fF. An fFSM graph consists of events (input events, output events and internal events), state-sets (states and variable states), initial states, and transitions.

Definition 5 flexible FSM $fF = (I, O, IT, X, V, F)$.

In the concurrent composition, constituent fFSM graphs become active simultaneously. In the hierarchical composition, when the parent state becomes active, all state-sets in the child fFSM graph become active. Transitions in the child fFSM graph are only performed when the parent state is active. Fig.3 illustrates a hierarchical fFSM graph and its definition is shown below.

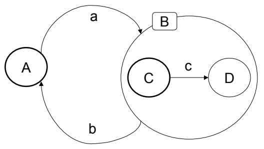

Fig. 3. An hiearchical fFSM graph

I = { (a, {ε, ϕ}), (b, {ε, ϕ}), (c, {ε, ϕ}) }
O = {}, IT = {}
X = { (s_1, {A, B}), (s_2, {ϕ, C, D}) }
V = {}
R = { (s_1, A), (s_2,ϕ) }
F = { ({(s_1, A)}, (a=ε), {(s_1, B), (s_2, C)}, {}),
({(s_1, B)}, (b=ε), {(s_1, A), (s_2, ϕ)}, {}),
({(s_1, B), (s_2, C)}, (c=ε), {(s_2, D)}, {}) }

State-set s_2 has initial state ϕ because its parent state, B, is not the initial state of the parent fFSM. When state B becomes active, the constituent fFSM becomes active and has the initial state C. Once state B becomes inactive, it is changed to invalid state ϕ.

Another fFSM graph with a variable state is shown in Fig.4. An input event "time" is an external clock and an output signal "timeout" is the result output which indicates that timeout occurs. A variable state is defined to keep track of the remaining time. The value of the variable state can be read in conditions and updated by actions.

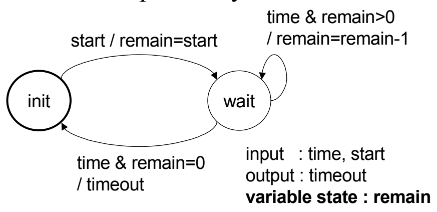

Fig. 4. Timeout fFSM graph with a variable state

I = { (start, {1..10, ϕ}), (time, {ε, ϕ}) }
O = { (timeout, {ε, ϕ}) }
IT = {}
X = { (s_1, {init, wait}), (remain, {0..10}) }
V = { (remain, {0..10}) }
R = { (s_1, init), (remain, 0) }
F ={({(s_1,init)},(start≠ ϕ), {(s_1,wait)}, {(remain,start)}), ({(s_1,wait)}, (time=ε&&remain>0), {(s_1,wait)}, {(remain, remain-1)}), ({(s_1, wait)}, (time=ε&&remain=0), {(s_1, init)}, {(timeout, ε)}) }

IV. Static Analysis of fFSM Model

While the pure FSM model can be analyzed by several static analysis methods to verify the correctness, extended FSM models with currency are not easy to analyze statically. One way of doing it is to flatten the model to the pure FSM model and apply the static analysis techniques for the FSM model. But it is not a feasible approach as the complexity of the system grows because of the state explosion problem.

In this section, we propose a static analysis technique to validate the deterministic behavior of the fFSM model utilizing the compositional structure of the model. To make fFSM model fully compositional, we make the following restrictions. First, a variable state is referenced and updated by only an atomic fFSM graph. Second, no inter-transition across the hierarchy is allowed. Third, communication between concurrent fFSMs is achieved only by internal events.

An execution (macro-step) of fFSM model is composed of several delta-delays (micro-steps) similarly to [4]. Initially, the fFSM graph is triggered by input events and makes a transition when its condition is satisfied. If any transition produces internal events, transitions triggered by the internal events are made subsequently until there is no more internal event. After every delta-delay, it clears all existing events and sets newly produced internal events. We call each delta-delay period as a phase of execution. Remind that the variable states and the output events keep their values to make them persistent.

Such execution rule of fFSM model may produce non-deterministic behavior as following. First, multiple transitions from one state can be enabled simultaneously but one transition should be chosen non-deterministically. Second, there can be multiple simultaneous writers for an event during processing of internal events. Then, the final value becomes non-deterministic. Lastly, there may exist circular transitions by cascaded internal transitions.

In the proposed technique, we construct a causality graph to detect such non-deterministic behavior by analyzing transitions of concurrent fFSM graphs. It shows triggering sequence of transitions by internal events. In a causality graph, each node indicates a transition in an fFSM graph. A node is associated with the variable states and output events which are updated by the transition. If a transition invokes another transition, we draw a directional line between them. For each transition, if the transition produces an internal event, we draw a solid line to each transition that will be activated by the internal event. If the transition can activate only one transition among many possible transitions, they are connected by dashed lines.

Because a causality graph shows at which phase each transition is executed, multiple transitions from one state can be analyzed statically. Second, once we figure out the cascaded transitions from a causality graph by all possible sets of input events, we can analyze which events (internal event, output event, variable state) are overwritten during the cascaded transitions invoked. Finally, if there exists a circular transition sequence, there will be a loop in the causality graph.

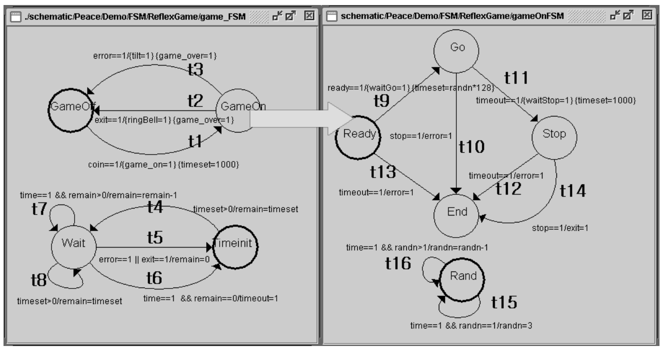
./schematic/Peace/Demo/FSM/ReflexGame/game_FSM
schematic/Peace/Demo/FSM/ReflexGame/gameOnFSM
error==1/{tilt=1} {game_over=1}
t3
GameOff
t2
GameOn
exit==1/{ringBell=1} {game_over=1}
t1
coin==1/{game_on=1} {timeset=1000}
time==1 && remain>0/remain=remain-1
t7
t4
timeset>0/remain=timeset
Wait
t5
Timeinit
error==1 || exit==1/remain=0
t8
t6
timeset>0/remain=timeset
time==1 && remain==0/timeout=1
Go
ready==1/{waitGo=1} {timeset=randn*128}
t9
t11
timeout==1/{waitStop=1} {timeset=1000}
Ready
stop==1/error=1
t10
Stop
t13
timeout==1/error=1
t12
t14
timeout==1/error=1
End
stop==1/exit=1
time==1 && randn>1/randn=randn-1
t16
Rand
t15
time==1 && randn==1/randn=3

Fig. 5. fFSM example of reflex game

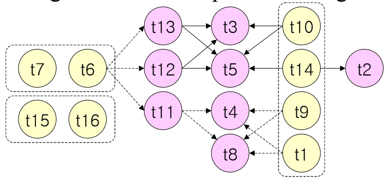

Fig. 6. Causality graph of Fig.5

Table II Cascaded transitions for one input event

	Phase 1	Phase 2	Phase 3	Phase 4
Coin event	**t1**	→ **t4**		
current states	GameOffTimeInit	Ready,TimeInit	Ready,Wait	
Event values	game_on=1, remain=1000			
ready event	**t9**	→ **t8**		
current states	Ready,Wait	Go,Wait	Go,Wait	
Event values	waitGo=1, remain=randn*128			
Stop event	**t10**	→ **t3, t5**		
current states	Go,Wait	End,Wait	GameOff,TimeInit	
Event values	tilt=1, game_over=1, remain=0			
Stop event	**t14**	→ **t2, t5**		
current states	Stop,Wait	End,Wait	GameOff,Timeinit	
Event values	RingBell=1, game_over=1, remain=0			
Time event	**t6**	→ **t13**	→ **t3, t5**	
current states	Wait,Ready	TimeInit,Ready	TimeInit,End	TimeInit, GameOff
Event values	tilt=1, game_over=1, remain=0			
time event	**t6**	→ **t12**	→ **t3, t5**	
current states	Wait,Stop	TimeInit,Stop	TimeInit,End	TimeInit, GameOff
event values	tilt=1, game_over=1, remain=0			
time event	**t6**	→ **t11**	→ **t4**	
current states	Wait,Go	TimeInit,Go	TimeInit,Stop	Wait,Stop
event values	WaitStop=1, remain=1000			

The reflex game example (Fig. 5) shows how we can detect non-determinism using the causality graph. Input events are *time*, *coin*, *ready* and *stop*, among which the last three input events cannot occur simultaneously. Output events are *game_on*, *waitGo*, *waitStop*, *ringBell*, *tilt,* and *game_over*. Internal events are *timeset*, *timeout*, *error* and *exit*. And, variable states are *remain* and *randn*. The game scenario is as follows: after a coin is inserted, ready signal becomes on after a randomly distributed latency. When the ready signal is one, the player should put down the stop button as quickly as possible. Then the output is produced to indicate the time duration between the ready signal and the stopping action.

Associated with the fFSM graph of Fig. 5, we construct a causality graph as shown in Fig.6. Transitions triggered by internal events have dark gray color.

Since there is no loop in the causality graph there will be no circular transitions in the graph. Now we analyze the graph by traversing the causality graph at each input event set. Table II shows seven sets of cascaded transitions in case one input event arrives. Each set consists of three rows; First two rows explain the transition and the current state at each phase. Last *event values* row shows the status of output events and variable states. If there are multiple assignments to the same output event or variable state, non-deterministic behavior is detected in the fFSM graph. In Table II, we cannot find any non-deterministic transition.

Table III shows selected three sets of cascaded transitions when two simultaneous input events occur. First two sets show the cascaded transitions when *stop* and *time* input events occur simultaneously, which has no problem. The third set, however, may cause a serious problem. It occurs when the fFSM graph gets input events *ready* and *time*, and the variable state *remain* is zero. The variable state *remain* has multiple assignments and the final state would be unexpected state, {Stop, Wait}, instead of {GameOff, Timeinit} or {Go, Wait}. We can avoid the case by changing the condition of t6 to "time==1&&ready!=1&&remain==0". This example shows that such static analysis with the causality graph can detect a subtle non-deterministic error.

Fig. 7 shows a pseudo code of the proposed static analysis algorithm. Since the algorithm traverses possible transitions for all reachable states, the complexity of the algorithm becomes $O(\sum_{\forall\ reachable\ states} \sum_{k=1..n} {}_{n}C_{k})$ where n is the number of possible initial transitions in each reachable state. Even though the complexity of the algorithm is not polynomial in theory, the actual complexity in real examples is pseudo-polynomial because each reachable state has usually a small number of possible initial transitions.

Table III Cascaded transitions for two simultaneous input events

	phase 1	phase 2	phase 3	phase 4
stop event time event	**t10** **t6**	→ **t3**		
current states	Go, Wait	End, Timeinit	GameOff, Timeinit	
event values	tilt=1, game_over=1, remain=0			
stop event time event	**t14** **t6**	→ **t2**		
Current states	Stop, Wait	End, Timeinit	GameOff, Timeinit	
event values	ringBell=1, game_over=1, remain=0			
ready event time event	**t9** **t6**	→ **t4** → **t11**	 → **t8**	
current states	Ready, Wait	Go, Timeinit	Stop, Wait	Stop, Wait
event values	**waitGo=1, remain=0→randn*128→1000, waitStop=1**			

```
push a set of the initial states to the stack
put a set of the initial states to the state list
while (!stack.empty()) {
  pop a set of states from the stack
  acquire possible initial transitions
  for (all possible sets of possible transitions) {
    performs cascaded transitions
    if (there is a problem) report error
    if (arrived set of states are not in the state list) {
      push the arrived set of states to the stack
      put the arrived set of states to the state list
} } }
```

Fig. 7. Proposed static analysis algorithm

V. Automatic Code Synthesis from fFSM model

In this section, we explain how to synthesize the software or hardware exploiting the compositional structure of the fFSM model.

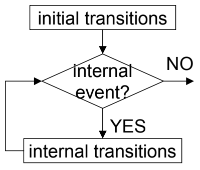

Fig. 8. Flow chart of fFSM C code structure

Fig.8 shows a flow chart of fFSM C code structure. We distinguish initial transitions and internal transitions for efficient implemetation. Because of delta-delay execution, input events are only valid during the initial phase. So, we check initial transitions first, and then if there exists any internal event produced, we iterate to make internal transitions until there is no more internal event as shown in Fig.9 a). Concurrent fFSM graphs are generated as separated ‘switch-case’ codes and the ‘switch-case’ code of hierarchical fFSM is generated inside the parent state as shown in Fig.9 b).

Fig.10 shows the structure of a top fFSM block diagram of HW code generation. The control FSM block separates processing between initial transitions and internal transitions, and the main FSM block as shown in Fig.11. When the control FSM block gets a request signal, it sends an enable signal and a process signal outside. Then, the main FSM block starts to process input events. After one clock, the control FSM deactivates the enable signal and the main FSM block processes internal events until there is no more internal event. Finally, the main FSM block finishes an execution and sends a done signal to the control FSM, which deactivates the process signal.

```
read input events;
process initial transitions;
while(internal events?) {
  process internal transitions;
}
send output events;
```

(a)

```
switch (each concurrent FSM) {
case each state:
  if (each condition) {
    do_actions;
  } else if (another condition) {
    ...
  } else {
    if (hierarchical FSM?)
      execute hierarchical fFSM;
  }
case ...
}
```

(b)

Fig. 9. a) Simplified code sequence for fFSM model, b) Generated code for each fFSM graph

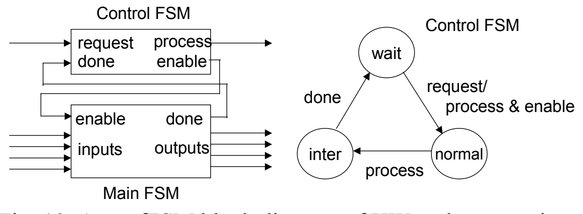

Fig. 10. A top fFSM block diagram of HW code generation

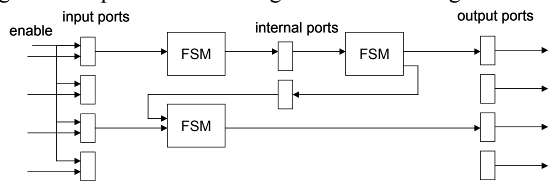

Fig. 11. Block diagram of the main FSM in HW generation

Fig.11 shows a block diagram of the main FSM block in HW code generation. The block diagram consists of input port blocks, internal port blocks, output port blocks and concurrent FSM blocks. Each port block stores an event and sends the event according to the execution semantic of fFSM model. A constituent fFSM graph is generated as an FSM block in the figure, which includes sequential logic to store states and combinational logic to process transitions. If an FSM graph is a child graph in hierarchy, it checks the current state of the parent fFSM before making a state transition. Combination logic is also activated only when the next state of the parent fFSM is valid.

VI. Conclusion

In this paper we introduced a novel extension of FSM model, fFSM model, which is successfully used to describe complex control behavior in our codesign environment. And we proposed a static analysis technique to detect non-deterministic behavior of the specification if any. We also proposed a method to synthesize the software of hardware preserving the composition structure of the fFSM model. We expect that the proposed techniques can be applied to other compositional FSM extensions.

Acknowledgements

This work was supported by National Research Laboratory Program (number M1-0104-00-0015) and Brain Korea 21 Project. The RIACT and ISRC at Seoul National University provided research facilities for this study.

References

[1] D. Kim, M. Kim and S. Ha, "A Case Study of System Level Specification and Software Synthesis of Multi-mode Multimedia Terminal", *ESTIMedia*, Newport Beach, USA Oct 2003

[2] D. Harel, “Statecharts: a visual formalism for complex systems”, *Sci. Comput. Program*, Vol. 8, pp. 231-274, 1987.

[3] M. von der Beeck, “ A comparison of statecharts variants”, *Proceedings of Formal Techniques in Real Time and Fault Tolerant Systems*, LNCS 863, pp. 128-148, Springer-Verlag, Berlin, 1994.

[4] A. Pnueli and M. Shalev. “What is in a step: On the semantics of Statecharts” In *TACS ’91*. vol. 526 of LNCS, page 244-264. Springer-Verlag, 1991

[5] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, and A. Sangiovanni-Vincentelli, “A Formal Specification Model for Hardware/Software Codesign”, In *Proceeding of Int. Workshop on Hardware-Software Codesign*, Oct.1993.

[6] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, A. Sangiovanni-Vincentelli, E. Sentovich, and K. Suzuki, “Synthesis of Software Programs for Embedded Control Applications”, In *Proceeding of DAC*, June 1995.

[7] A. Girault, B. Lee, and E. Lee, “Hierarchical finite state machines with multiple concurrency models”*, IEEE Transactions on CAD*, Vol. 18, No. 6, pp. 742-760, June 1999.