# Formalization of *f*FSM Model and Its Verification

Sachoun Park[1], Gihwon Kwon[1], and Soonhoi Ha[2]

[1] Department of Computer Science, Kyonggi University,
San 94-6, Yiui-Dong, Youngtong-Gu, Suwon-Si, Kyonggi-Do, Korea
{sachem, khkwon}@kyonggi.ac.kr
[2] Department of Computer Engineering, Seoul National University,
Seoul, Korea 151-742
sha@iris.snu.ac.kr

**Abstract.** PeaCE(Ptolemy extension as a Codesign Environment) was developed for the hardware and software codesign framework which allows us to express both data flow and control flow. The *f*FSM is a model for describing the control flow aspects in PeaCE, but it has difficulties in verifying their specifications due to lack of their formality. Thus we propose the formal semantics of the model based on its execution steps. To verify an *f*FSM model, it is translated into SMV input language with properties to be checked, automatically. As a result, some important bugs such as race condition, ambiguous transition, and circular transition can be formally detected in the model.

**Keywords:** Finite state machine, Step semantics, Formal verification, Model checking.

## 1 Introduction

To make narrow the gap between design complexity and productivity of embedded systems, hardware/software codesign has been focused as a new design methodology. Various codesign procedures have been proposed, and formal models of computation for system specification by using "correct by construction" principle make ease design validation. The PeaCE[1] is the codesign environment to support complex embedded systems. The specification uses synchronous dataflow (SDF) model for computation tasks, extended finite state machine (FSM) model for control tasks and task-level specification model for system level coordination of internal models (SDF and FSM). It gives automatic synthesis framework from the proposed specification with good results compared with hand-optimized code, and the automatic SW/HW synthesis from extended FSM model, called *f*FSM(flexible FSM), and automatic SW synthesis from task-model is developed. The synthesis framework generates architecture independent code which can be used for functional simulation, design space exploration, synthesis and verification steps by varying the definitions of APIs.

The *f*FSM is another variant of Harel's Statecharts, which supports concurrency, hierarchy and internal event as Statecharts does. Also it includes global variables as

* This work was supported in part by IT Leading R&D Support Project funded by Ministry of Information and Communication, Republic of Korea.

memories in a system. This model is influenced from STATEMATE of i-Logix inc.[2] and the Ptolemy[3] approaches. But the formal semantics for internal models is not defined explicitly. Especially, in the case of *f*FSM(flexible FSM), the absence of formal semantics causes problems such as confidence for simulation, correctness of code generation, and validation of a system specification. Since no formal semantics exit, unexpected behavior may occur after system built and also it dilute original purpose of codesign to produce complex embedded system cost-effectively

In this paper, we define the step semantics for *f*FSM model, which becomes foundation about reliable code generation and formal verification. Step semantics or operational semantics of an *f*FSM defines how the model changes state from one configuration to another on the reception of some events, while it at the same time executes actions consisting of emitting output and internal events and updating of global variables. In this field, many works have proposed, but among these formal semantics, we turned our attention to Erich Mikk's hierarchical automata[4] and Lind-Nielsen's hierarchical state/event model[6].

Hierarchical automata semantics was defined to formally express the STATEMATE semantics of Statecharts described by Harel and Naamad in 1996[5]. After he defined pure hierarchal automata which have no inter-level transition, he described EHA (extended hierarchical automata) to handle the inter-level transition. As the semantics of EHA was presented in the Kripke structure, three rules at EHA were applied to: progress rule, stuttering rule, and composition rule. If any enabled transition is activated, sequential automaton takes progress rule. If an active sequential automaton does not have an enabled transition and the active state is a basic state then the automaton stutters and consumes events. And each automaton delegates its step to its sub-automata with respect to the composition rule. But it wasn't dealt with the delta-delay and variables.

HSEM(Hierarchical State/Event Model), the variant of Statecharts in IAR visualSTATE[7], is based on the Unified Modeling Language(UML) state diagram, where again the UML is based on Harel's Statecharts. Although HSEM has its origin in Statecharts, its semantics is distinguishable. The behavior of the model described *N* flat parallel machines, where the *N* is the number of Or-states: serial and history states. Thus a configuration of HSEM consists of exactly one state per each Or-sate, so it may include inactive states. This method is able to perform compositional model checking which one of solution for state explosion problem. However, in the HSEM semantics, there is only use of state reference to express guarding condition, without event occurring.

Firstly, we define the step semantics with concept of the delta-delay, variables and event. And then verifying some system properties, we automatically translate the model to SMV input language with these properties. This translation is based on the proposed step semantics and synchrony hypothesis.

In this paper, the semantics of the *f*FSM model in PeaCE approach is defined by borrowing from EHA and HSEM semantics. In the next section, *N* flat parallel machine of *f*FSM, *p*FSM, is defined with its example. The definition of step semantics of *p*FMS is presented in section 3, our efforts for debugging a model is described at section 4 and then we conclude the paper in section 5.

## 2 Formal Definition of *p*FSM

### 2.1 Reflex Game: The Example of *f*FSM Model

This version of reflex game is used for describing formal model of *f*FSM. The set of input events to the system are *coin*, *ready*, *stop*, and *time*. All but the last are user inputs, while the last generated by system simply counts off time. 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. To compute the time duration, we use "remain" and "randn" as variable states. The resultant *f*FSM graph is concurrent and hierarchical.

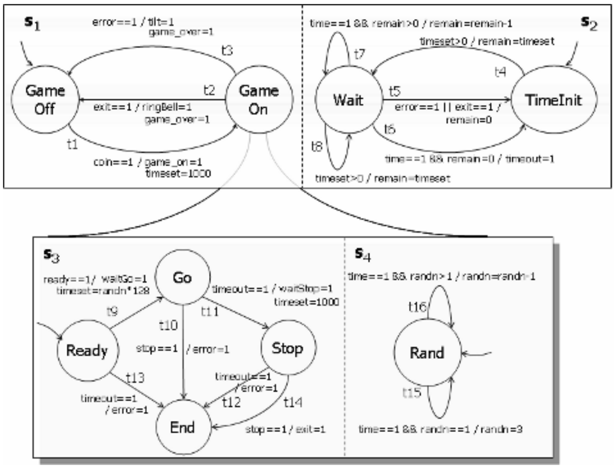

s1
error==1 / tilt=1 game_over=1
t3
Game Off
t2
Game On
exit==1 / ringBell=1 game_over=1
t1
coin==1 / game_on=1 timeset=1000
time==1 && remain>0 / remain=remain-1
s2
timeset>0 / remain=timeset
t7
t4
Wait
t5
error==1 || exit==1 / remain=0
TimeInit
t6
t8
time==1 && remain=0 / timeout=1
timeset>0 / remain=timeset
s3
ready==1/ waitGo=1 timeset=randn*128
Go
timeout==1 / waitStop=1 timeset=1000
t9
t11
t10
Ready
stop==1
/ error=1
Stop
t13
timeout==1 / error=1
t12
t14
timeout==1 / error=1
End
stop==1 / exit=1
s4
time==1 && randn>1 / randn=randn-1
t16
Rand
t15
time==1 && randn==1 / randn=3

**Fig. 1.** *f*FSM example of reflex game

Initially, each atomic *f*FSM is triggered by input events and makes a transition when its guard condition is satisfied. If the transition produces internal events, transitions triggered by the internal events are made iteratively until there is no more internal event. After every delta-delay, it clears all existing events and sets newly produced internal events at the previous delta-delay. Briefly, an execution of an atomic *f*FSM consists of a transition triggered by an input event and subsequent transitions triggered by internal events produced by the previous transition. And variable state and output events keep their values to make them persistent.

### 2.2 Syntax of *p*FSM

In this section, we introduce definitions about *p*FSM which is based on the thesis[8]. To define the step semantics of *f*FSM, we propose *N* flat parallel machine of *f*FSM, *p*FSM. Like an *f*FSM, there exit events, global variables, states, and transition.

*I, O,* and *IT* are sets of input events, output events, and internal events, respectively. Unlike previous definition of the event of *f*FSM, these sets disjoint each other. Each event $e_i$ in $I \cup O \cup IT = \{e_1, ..., e_n\}$ is composed of its domain $D_i$ and initial value $d_i$, and $val(e_i)$ denotes current value of the event. *Simple* FSM is defined by 4-tuples $(S, s^0, T, scr)$, where *S* is a set of states, $s^0$ is the initial state, and *T* is set of transition relations. In the PeaCE approach, dataflow models are controlled by external signals generated by *f*FSM, which can be labeled at a proper state as a set of atomic proposition. We call the labels *scripts*, and *Script* represents the set of all scripts occurring in the *f*FSM. Thus, $scr : S \rightarrow 2^{Script}$ is the label function to map a set of scripts into a state.

*f*FSM has two types of composition like other variants of Harel's Statecharts: concurrent and hierarchical compositions. HSEM used *N* flat parallel machines for describing its operational semantics, because BDD(Binary Decision Diagram) could be easily represented and the compositional model checking applied. In this paper, we refer to HSEM semantics to define the *f*FSM semantics.

**Definition 1 (*p*FSM).** Now, formal semantics of *f*FSM is defined as *N* flat parallel machines *p*FSM. $pFSM = (I, O, IT, M, \gamma, V)$, where *I, O, IT* are set of events above, *V* is a set of global variables, $v_j \in (D_j,\ d_j)$ , and $M = \{m_1, ..., m_n\}$ is the set of simple FSM. Let $\Sigma = \bigcup_{i=1}^{n} S_i$ be the set of all states in *M*, hierarchical relation γ maps a state to the set of machines which belong to the state: $\gamma : \Sigma \rightarrow 2^M$.

The hierarchical function γ has three properties: there exist a unique root machine, every non-root machine has exactly one ancestor state, and the composition function contains no cycles. Let $sub : \Sigma \rightarrow 2^\Sigma$, and $sub(s) = \{s' \mid M_i \in \gamma(s) \wedge s' \in S_i\}$ is another function to relate between a super state and its sub states. $sub^+$ denotes the transitive closure of *sub* and *sub** denotes the reflexive transitive closure of *sub*.

**Definition 2 (*Simple* FSM).** $m_i = (S_i, s_i^0, T_i, scr_i)$

i. $S_i = \{s_i^0, s_i^1, ..., s_i^n\}$ is the finite set of states of $m_i$,
ii. $s_i^0$ is a initial state,
iii. $T_i$ is the set of transition of $m_i$, and a transition $t \in T_i = (s, g, A, s')$ is composed of source and target states $s$, $s' \in S_i$, guarding condition *g* which is Boolean expression, and set of actions *A*,
iv. $scr_i : S_i \rightarrow 2^{Script}$ is a function to map a set of script into a state.

Guards that include variables and events have the following minimal grammar.

$$G ::= true \mid \neg G \mid G_1 \wedge G_2 \mid e < Exp \mid e = Exp \mid v < Exp \mid v = Exp$$
$$Exp ::= n \mid v \mid Exp_1 \bullet Exp_2 ,$$

where *n* is an integer constant, $v \in V$ is a global variable, $\bullet \in \{+, -, \times, /\ \}$ represents a set of binary operators. To select some facts of a transition $t = (s, g, A, s')$, following projection functions are useful: *source(t)* = *s*, *target(t)* = *s′*, *guard(t)* = *g*, *action(t)* = *A*.

Also, in a set of actions $A$, each action element $a \in A$ consists of variable assignment or event emission:

$$a ::= v{:=}Exp \mid e{:=}Exp.$$

For an action element, following three projection functions are used, because an action set is composed of updating variables, emitting some output events, and producing internal events.

$$update(A) = \{v := Exp \mid \exists v \in V.(v := Exp) \in A\}$$
$$output(A) = \{e := Exp \mid \exists e \in O.(e := Exp) \in A\}$$
$$signal(A) = \{e := Exp \mid \exists e \in IT.(e := Exp) \in A\}$$

Figure 2 shows *p*FSM corresponding to figure 1, and the below example presents the definition of figure 2.

$I = \{coin, ready, stop, time\}$, $O = \{game\_on, waitGo, waitStop, ringBell, tilt, game\_over\}$
$IT = \{timeset, timeout, error, exit\}$, $V = \{randn, remain\}$
$M = \{m_1, m_2, m_3, m_4\}$, $\gamma(m_1) = \{m_3, m_4\}$, $\gamma(m_2) = \gamma(m_3) = \gamma(m_4) = \varnothing$,
$m_1 = (S_1, s_1^0, T_1, scr_1)$, $S_1 = \{GameOff, GameOn\}$, $s_1^0 = GameOff$,
$T_1 = \{(GameOff, coin = 1, \{game\_on = 1, timeset = 1000\}, GameOn),$
$(GameOn, exit = 1, \{game\_over = 1, ringBell = 1\}, GameOff),$
$(GameOn, error = 1, \{game\_over = 1, tilt = 1\}, GameOff)\}$, $scr_1 = \varnothing$
…

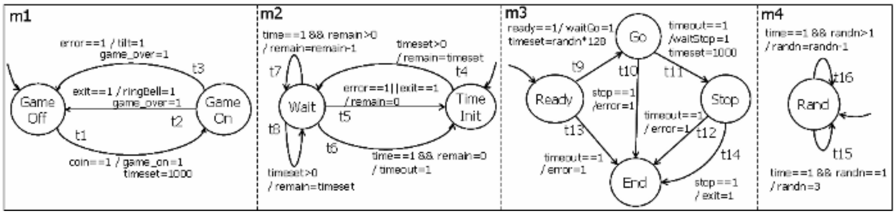


**Fig. 2.** Flatten machine *p*FSM

## 3 Semantics of *p*FSM

Step semantics or operational semantics of an *f*FSM defines how the model changes state from one configuration to another on the reception of some events, while it at the same time executes actions consisting of emitting output and internal events and updating of global variables.

**Definition 3 (Configuration).** Δ represents the whole configurations of *p*FSM model, for each $m_i$ of *p*FSM, its formal definition is $\Delta = \{\{s_1, ..., s_n\} \mid \exists s_i \in S_i, 0 < i \le n\}$. $\delta_0 \in \Delta$, $\delta_0 = \{s_1^0, \ldots, s_n^0\}$ is an initial configuration.

Erich's definition of a configuration is a set of active states, so the size of the set is not fixed. But the size of a configuration δ is the same of the set of machines $M$, so we need to define which state is active.

**Definition 4 (Active state).** It can be defined that a state *s* is active in configuration δ as follow:

$$\delta \models s \;\; iff \;\; \forall s' \in \Sigma . s \in sub^{*}(s') \Rightarrow s' \in \delta .$$

**Definition 5 (Satisfiability).** To decide which transition is enabled, given the set of events $E \subseteq 2^{I} \cup 2^{IT}$ and the current configuration δ, configuration δ and events *E* satisfies a guard *g*, $\langle \delta, E \rangle \models guard(t)$, is defined inductively.

$$\begin{array}{lll}
\langle \delta, E \rangle \models true & iff & true \\
\langle \delta, E \rangle \models \neg G & iff & not \langle \delta, E \rangle \models G \\
\langle \delta, E \rangle \models G_1 \wedge G_2 & iff & \langle \delta, E \rangle \models G_1 \; and \; \langle \delta, E \rangle \models G_2 \\
\langle \delta, E \rangle \models e < Exp & iff & e \in E \; and \; val(e) < val(Exp) \\
\langle \delta, E \rangle \models e = Exp & iff & e \in E \; and \; val(e) = val(Exp) \\
\langle \delta, E \rangle \models v < Exp & iff & val(v) < val(Exp) \\
\langle \delta, E \rangle \models v = Exp & iff & val(v) = val(Exp),
\end{array}$$

where $val(n) = n$, $val(e)$ denotes the current value of event *e,* and $val(v)$ denotes the current value of variable *v*. $val(Exp_1 \bullet Exp_2) = val(Exp_1) \bullet val(Exp_2)$.

**Definition 6 (Enabled transition).** Through the definitions of active states and satisfiability relation, we define a set of enable transitions for each active state.

$$ET = \{ t \mid \forall i \in \{1, ..., n\} . t \in T_i \wedge \langle \delta, E \rangle \models guard(t) \wedge \delta \models source(t) \}$$

**Definition 7 (Executable transition).** The set of executable transitions are non-conflicting set of transitions and every simple FSM must contribute at most one transition. As *f*FSM model has no inter-level transition, the conflict only occurs between two transitions which have different and comparable priorities.

$$XT = \{ t \in ET \mid \neg \exists t' \in ET . source(t) \in sub^{+}(source(t')) \},$$
$$\forall m_i \in M . \mid XT \cap T_i \mid \leq 1$$

**Definition 8 (LKS).** Step semantics of *p*FSM is defined by *LKS*(Labeled Kripke Structure). $LKS = (Q, q_0, R, L)$ is defined:

$Q = \{q_0, \dots, q_n\}$ is the finite set of states in LKS,

$q_0 = (\delta_0, \varnothing, c_0)$ is an initial state,

$R \subseteq Q \times 2^{Act} \times Q$ is the set of transitions with label as the set of actions,

$L : Q \to 2^{Script}$ is label function such that $L(q_i) = \bigcup_{\forall s \dashv \delta} scr(s)$.

The step could be explained both micro step and macro step. The micro step stands for one step triggered by input or internal events, and macro step is a finite sequence of micro steps each of which is triggered by one input event or consequent internal events until the produced internal event won’t exit any more.

**Definition 9 (Micro step).** $(\delta, E, c) \xrightarrow{Act} (\delta', E', c')$

Given the current configuration δ and the set of events *E*, the next configuration δ′ is defined as follow:

$$\begin{aligned}\delta' = \{s' \mid \forall s \in \delta . \exists t \in XT.(s = source(t) \Rightarrow s' = target(t))\\ \vee (s \in sub^{+}(source(t)) \Rightarrow s' = reset(s))\\ \vee (s \notin sub^{*}(source(t)) \Rightarrow s' = s)\}\end{aligned}$$

where $reset(s) = s_i^0,\ s \in sub(s'') \wedge m_i \in \gamma(s'') \wedge s \in S_i$

$$E' = \bigcup_{\forall t \in XT} signal(action(t)),$$

and $c' = L(q')$,

$$Act = \bigcup_{\forall t \in XT} output(action(t)) \cup \bigcup_{\forall t \in XT} update(action(t))$$

**Definition 10 (Macro step).** Step semantics of *p*FSM is represented by $q \xrightarrow{Exe} q'$, called *an execution*, which is triggered by input event and produces cascaded input events. Thus one input event and consequent internal events make transitions until any internal event cannot be produced. After each micro step, all previous events are consumed by delta-delay. In the following definition, $k > 1$ is the first *k* which makes $E_{i,k}$ to ∅, and $Exe_i = \bigcup_{\forall 0<j<k} Act_j$ is a set of all actions during macro step.

$$(\delta_i, E_i, c_i) \xrightarrow{Exe_i} (\delta_{i+1}, \varnothing, c_{i+1}) \text{ iff } (\delta_{i,1}, E_{i,1}, c_{i,1}) \xrightarrow{Act_1} \cdots \xrightarrow{Act_{k-1}} (\delta_{i,k}, \varnothing, c_{i,k}).$$

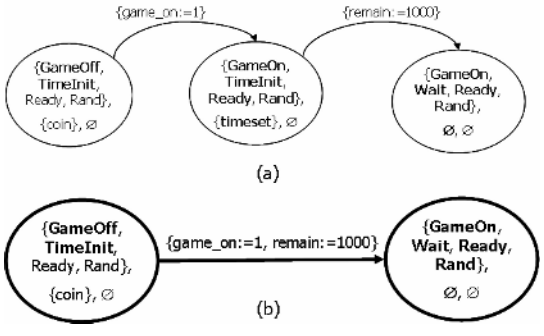

{game_on:=1}
{remain:=1000}
{GameOff, TimeInit, Ready, Rand}, {coin}, ∅
{GameOn, TimeInit, Ready, Rand}, {timeset}, ∅
{GameOn, Wait, Ready, Rand}, ∅, ∅
(a)
{GameOff, TimeInit, Ready, Rand}, {coin}, ∅
{game_on:=1, remain:=1000}
{GameOn, Wait, Ready, Rand}, ∅, ∅
(b)

**Fig. 3.** Example of micro step(a) and macro step(b)

When input event "*coin*" occurs, control of the system is moved from the initial configuration {GameOff, TimeInit, Ready, Rand} to {GameOn, wait, Ready, Rand}. For error detection, we define three kinds of error: Race Condition(There can be multiple writers for an output event or a variable state during an execution of *f*FSM model), Ambiguous transition(Multiple transitions from one state can be enabled simultaneously), and Circular transition(There can be exist circular transitions by cascaded transitions).

For example, if a snapshot of the system contains a configuration {GameOn, Wait, Ready, Rand}, occurring event set is consist of user event "*ready*" and system event "*time*", and variable *remain* is zero, although it is a rare case, it breaks the output constraint that output value must be persistent during one execution, since *remain* have multiple assignments. Table 1 presents it.

**Table 1.** Violation of Race Condition for a variable "remain"

| events | Configuration | Actions |
|---|---|---|
| {time, ready} | {GameOn, Wait, Ready, Rand} | {waitGo:=1, remain:=0} |
| {timeset, timeout} | {GameOn, TimeInit, Go, Rand} | {waitStop:=1, remain:=randn*128} |
| {timeset} | {GameOn, Wait, Stop, Rand} | { remain:=1000} |
| ∅ | {GameOn, Wait, Stop, Rand} | – |

## 4 Debugging *f*FSM Model

### 4.1 Stepper: Simulation Tool for *f*FSM Model

To debug a model, simulating a system model is widely used. In the PeaCE approach, integrated simulation is provided. But it could not simulate control model only. Thus, we develop a simulation tool for *f*FSM model with respect to our step semantics. Figure 4 shows the framework of the Stepper.

Stepper receives textual description of *f*FSM model written by design tool in the PeaCE framework. Then, it presents a model like a tree structure and input event generator. Input events generated by a user execute one macro step. The Stepper, then, shows all micro steps during one execution. Also it provides translation from *f*FSM to SMV, with some important properties to be checked automatically. In figure 5, as you can see, after "*time*" and "*ready*" events occur, the variable *remain* is updated in twice 999, 384 at the macro step, STEP[2].

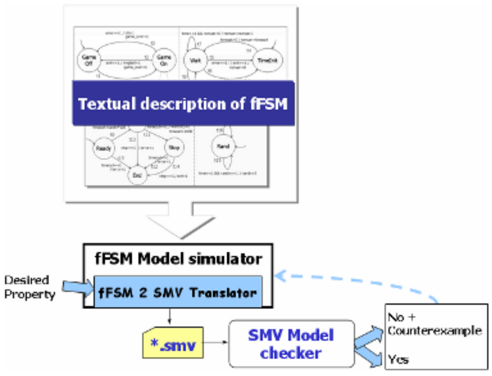
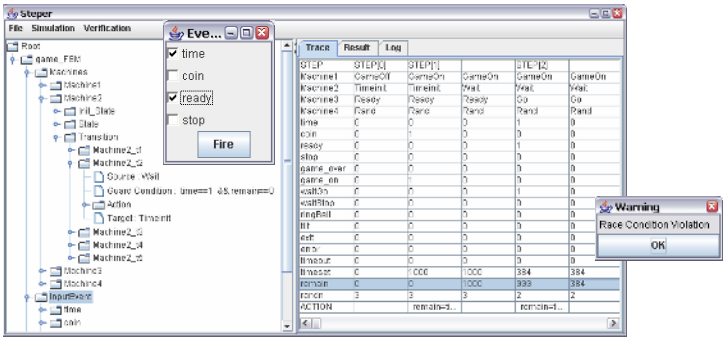


**Fig. 4.** Framework of the Stepper



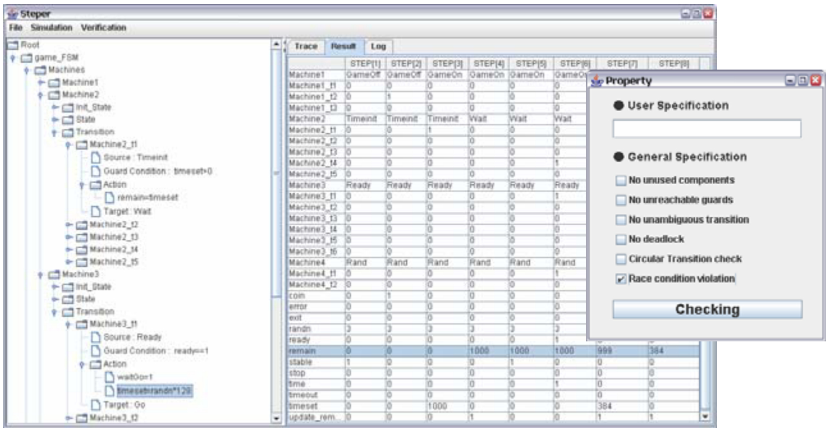
Race Condition Violation

| STEP | STEP[0] | STEP[1] | | STEP[2] | |
|---|---|---|---|---|---|
| Machine1 | GameOff | GameOn | GameOn | GameOn | GameOn |
| Machine2 | TimeInit | TimeInit | Wait | Wait | Wait |
| Machine3 | Ready | Ready | Ready | Go | Go |
| Machine4 | Rand | Rand | Rand | Rand | Rand |
| time | 0 | 0 | 0 | 1 | 0 |
| coin | 0 | 1 | 0 | 0 | 0 |
| ready | 0 | 0 | 0 | 1 | 0 |
| stop | 0 | 0 | 0 | 0 | 0 |
| game_over | 0 | 0 | 0 | 0 | 0 |
| game_on | 0 | 1 | 0 | 0 | 0 |
| waitGo | 0 | 0 | 0 | 1 | 0 |
| waitRing | 0 | 0 | 0 | 0 | 0 |
| ringBell | 0 | 0 | 0 | 0 | 0 |
| tilt | 0 | 0 | 0 | 0 | 0 |
| exit | 0 | 0 | 0 | 0 | 0 |
| error | 0 | 0 | 0 | 0 | 0 |
| timeout | 0 | 0 | 0 | 0 | 0 |
| timeset | 0 | 1000 | 1000 | 384 | 384 |
| remain | 0 | 0 | 1000 | 383 | 384 |
| randn | 3 | 3 | 3 | 2 | 2 |
| ACTION | | remain=1... | | remain=1... | |

**Fig. 5.** Detecting a race condition violation via simulation in Stepper

## 4.2 Model Checking *f*FSM Model

Simulation is very useful tool to present an error in a model, tracing an execution path step by step. But by simulation, it is apt to spend much time or in some cases may be impossible to detect an error. So more efficient debugging, automatic and formal technique is required. Our tool provides formal verification which is implemented by translating *f*FSM into SMV. Figure 6 shows the result of detecting a race condition violation via model checking and the result is the same of simulator's.



No unused components [ ]
No unreachable guards [ ]
No unambiguous transition [ ]
No deadlock [ ]
Circular Transition check [ ]
[✓ Race condition violation]

| | STEP[1] | STEP[2] | STEP[3] | STEP[4] | STEP[5] | STEP[6] | STEP[7] | STEP[8] |
|---|---|---|---|---|---|---|---|---|
| Machine1 | GameOff | GameOff | GameOn | GameOn | GameOn | GameOn | | |
| Machine1_t1 | 0 | 0 | 0 | 0 | 0 | 0 | | |
| Machine1_t2 | 0 | 1 | 0 | 0 | 0 | 0 | | |
| Machine1_t3 | 0 | 0 | 0 | 0 | 0 | 0 | | |
| Machine2 | TimeInit | TimeInit | TimeInit | Wait | Wait | Wait | | |
| Machine2_t1 | 0 | 0 | 1 | 0 | 0 | 0 | | |
| Machine2_t2 | 0 | 0 | 0 | 0 | 0 | 0 | | |
| Machine2_t3 | 0 | 0 | 0 | 0 | 0 | 0 | | |
| Machine2_t4 | 0 | 0 | 0 | 0 | 0 | 1 | | |
| Machine2_t5 | 0 | 0 | 0 | 0 | 0 | 0 | | |
| Machine3 | Ready | Ready | Ready | Ready | Ready | Ready | | |
| Machine3_t1 | 0 | 0 | 0 | 0 | 0 | 1 | | |
| Machine3_t2 | 0 | 0 | 0 | 0 | 0 | 0 | | |
| Machine3_t3 | 0 | 0 | 0 | 0 | 0 | 0 | | |
| Machine3_t4 | 0 | 0 | 0 | 0 | 0 | 0 | | |
| Machine3_t5 | 0 | 0 | 0 | 0 | 0 | 0 | | |
| Machine3_t6 | 0 | 0 | 0 | 0 | 0 | 0 | | |
| Machine4 | Rand | Rand | Rand | Rand | Rand | Rand | | |
| Machine4_t1 | 0 | 0 | 0 | 0 | 0 | 1 | | |
| Machine4_t2 | 0 | 0 | 0 | 0 | 0 | 0 | | |
| coin | 0 | 1 | 0 | 0 | 0 | 0 | | |
| error | 0 | 0 | 0 | 0 | 0 | 0 | | |
| exit | 0 | 0 | 0 | 0 | 0 | 0 | | |
| randn | 3 | 3 | 3 | 3 | 3 | 3 | | |
| ready | 0 | 0 | 0 | 0 | 0 | 1 | | |
| remain | 0 | 0 | 0 | 1000 | 1000 | 1000 | 999 | 384 |
| stable | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| stop | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| time | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| timeout | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| timeset | 0 | 0 | 1000 | 0 | 0 | 0 | 384 | 0 |
| update_rem... | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 |

**Fig. 6.** Detecting a race condition violation via model checking

## 4.3 Translation *f*FSM Model to SMV

Our translation rules are based on Chan[10] and Clarke[11]. Following translation rules are based on the step semantics defined in the previous section.

**Rule 1 (Machine and states).** For each machine, $m_i = (S_i, s_i^0, T_i, scr_i)$, machine and its states are encoded as $VAR\ m_i : S_i; \quad ASSIGN\ init(m_i) = s_i^0;$.

**Rule 2 (Transitions).** Transition relations can be expressed in SMV through the definition 6. For $m_i = (S_i, s_i^0, T_i, scr_i)$ and $t \in T_i$, each transition $t$ in $m_i$ is encoded as $DEFINE\ m_i\_t := up^*(source(t))\ \&\ guard(t);$, where $up(s) = \{s' | M_i \in \gamma(s') \wedge s \in S_i\}$ is the function to return a super-state of $s$ and $up^*$ is transitive reflexive closure of $up$.

```
DEFINE
  Machine1_t1 := Machine1 = GameOn & error=1;
  ⋮
VAR
  Machine1 : {GameOff, GameOn};
ASSIGN
  init(Machine1) := GameOff;
  next(Machine1) :=
       case
         Machine1_t1 : GameOff;
         ⋮
         1 : Machine1;
       esac;
```

**Rule 3 (Synchrony hypothesis).** To express synchrony hypothesis in SMV, we define a particular variable *stable*, which means that a system stays in stable state where any events does not occur. *stable* is also used in formulating circular transition and race condition. When sets of internal events, input events and variables are respectively $\{internal_1, \ldots, internal_l\}$, $\{input_1, \ldots, input_m\}$ and $\{variable_1, \ldots, variable_n\}$, the translation rule is:

```
VAR
  stable : boolean;
ASSIGN
  init(stable) := 1;
  next(stable) :=
       case
         !(valued_1 = next(valued_1)) : 0;
               ⋮
         !(valued_n = next(valued_n)) : 0;
         next(input_1) | … | next(input_m) : 0;
         (internal_1=0) & … & (internal_l=0) : 1;
         1 : 0;
       esac;
```

**Rule 4 (Input event).** Every input event *e* whose initial value is *d* and domain *D* is [*min*, …, *max*] can be translated as follow:

```
 VAR
   e : min .. max;
 ASSIGN
   init(e) := d;
   next(e) :=
       case
         stable : min .. max;
         1 : 0;
       esac;
```

**Rule 5 (Output event and internal event).** For a transition $t = (s, g, A, s')$ and each output or internal event *e* whose initial value is *d*, through a set of transitions $\{t_1, \ldots, t_n\}$, a set of expressions $\{exp_1, \ldots, exp_n\}$ and $\{e := Exp \mid \exists e \in O \cup IT.(e := Exp) \in A\}$, the translation rule can be expressed as follow:

```
VAR
  e : min .. max;
ASSIGN
  init(e) := d;
  next(e) :=
     case
       t1 : Exp1;
            :
       tn : Expn;
       1 : 0;
     esac;
```

**Rule 6 (Variable).** While default value of event *e* is 0 because of delta-delay, each variable *v* stores its previous value. This rule of variable is similar with Rule 5.

```
VAR
  v : min .. max;
ASSIGN
  init(v) := d;
  next(v) :=
     case
       t1 : Exp1;
            :
       tn : Expn;
       1 : v;
     esac;
```

Table 2 shows CTL templates for some important properties which can be automatically generated for user's convenience. The first three are trivial, but the rest are more complex.

**Table 2.** Built-in properties and its CTL formulae

| Properties | CTL formulae |
|---|---|
| No unused components | $\mathbf{EF}\ component_1 \wedge \ldots \wedge \mathbf{EF}\ component_n$ |
| No unreachable guard | $\mathbf{EF}\ (s_i \wedge \mathbf{EX}\ s_j)$, $source(t) = s_i$ and $target(t) = s_j$ |
| No unambiguous transitions | $\mathbf{AG}\ \neg((t_1 \wedge t_2) \vee (t_2 \wedge t_3) \vee (t_1 \wedge t_3))$, where $\{t_1, t_2, t_3\}$ is a set of outgoing transition from the same state. |
| No deadlocks | $\neg\mathbf{EF}\ \mathbf{AG}\ Deadlock(fT)$, where $Deadlock(fT) = \neg\bigcup_{t \in fT} t_n$ |
| No divergent behavior | $\mathbf{AG}(\neg stable \Rightarrow \mathbf{A}[\neg\ stable\ \mathbf{U}\ stable])$ |
| Race condition violation | $\mathbf{AG}\ ((update(v) \wedge \neg stable) \Rightarrow \mathbf{AX}\ \mathbf{A}[\neg update(v)\ \mathbf{U}\ stable])$ $\mathbf{AG}\ ((emit(o) \wedge \neg stable) \Rightarrow \mathbf{AX}\ \mathbf{A}[\neg emit(o)\ \mathbf{U}\ stable])$ |

In below table, components might be all states, events and variables. Checking about a guard is replaced with whether the transition labeled by the guard is enabled or not. Ambiguous transitions must be checked in all states with their possible transitions. In the formula encoding the deadlock, *f*T denotes a set of all transitions of a model. The formula to detect circular transition, "$\mathbf{AG}(\neg \mathit{stable} \Rightarrow \mathbf{A}[\neg\ \mathit{stable}\ \mathbf{U}\ \mathit{stable}])$", means "whenever the system is in an unstable state, eventually it must reach a stable state." To formulate the race condition, the additional functions *update* and *emit* are introduced. Encoding *update* or *emit* for each output event or variable could be implemented by a new Boolean variable. Thus, user can select some properties or type in CTL properties.

## 5 Conclusions

*f*FSM is a model for describing the control flow aspects in PeaCE, but due to lack of their formality, it has difficulties in verifying the specification. In this paper, for lifting the reliability for code generation in the codesign framework and enabling formal verification of the control model, we defined the step semantics for the *f*FSM model. And we implemented tool to simulate and verify an *f*FSM model. As a result, some important bugs such as race condition, ambiguous transition, and circular transition can be formally detected in the model. Especially, to obtain the convenience of user to check properties, we constructed some templates for automatic generation of specifications. Now we are developing a specific model checker for *f*FSM and researching an effective abstraction technique to be applied in the new model checker.

## References

1. D. Kim, S. Ha, "Static Analysis and Automatic Code Synthesis of flexible FSM Model", ASP-DAC 2005 Jan 18-21, 2005
2. iLogix: http://www.ilogix.com/
3. http://ptolemy.eecs.berkeley.edu/
4. E. Mikk, Y. Lakhnech, M. Siegel, "Hierarchical automata as model for statecharts", LNCS Vol. 1345, Proceedings of the 3rd ACSC, pp. 181-196, 1997.
5. D. Harel, A. Naamad, "The STATEMATE semantics of statecharts", ACM Transactions on Software Engineering Methodology, 5(4), October 1996.
6. J. Lind-Nielsen, H. R. Andersen, H. Hulgaard, G. Behrmann, K. J. Kristoffersen, K. G. Larsen, "Verification of Large State/Event Systems Using Compositionality and Dependency Analysis", FMSD, pp. 5-23, 2001.
7. IAR: http://www.iar.com/products/vs/
8. D. Kim, "System-Level Specification and Cosimulation for Multimedia Embedded Systems," Ph.D. Dissertation, Computer Science Department, Seoul National University, 2004.
9. J. B. Lind-Nielsen, "Verification of Large State/Event Systems," Ph.D. Dissertation, Department of Information Technology, Technical University of Denmark, 2000.
10. W. Chan, "Symbolic Model checking for Large software Specification," Dissertation, Computer Science and Engineering at University of Washington, pp. 13-32, 1999.
11. E. M. Clarke, W. Heinle, "Modular translation of Statecharts to SMV," Technical Report CMU-CS-00-XXX, CMU School of Computer Science, August 2000.