효과적인 임베디드 소프트웨어 설계를 위한 제어흐름 모델의 자동 검증

박 사 천[†] · 권 기 현[††] · 하 순 회[†††]

요 약

하드웨어와 소프트웨어를 통합 설계하는 프레임워크인 PeaCE(Ptolemy extension as a Codesign Environment)에서는 데이터 흐름과 제어 흐름을 모두 표현할 수 있다. PeaCE에서 제어 흐름을 표현하는 fFSM 명세를 정형 검증하기 위해 fFSM의 단계 의미를 정의하였다. 본 논문에서는 이전 연구에서 정의된 정형 의미를 바탕으로 개발한 자동 검증 도구를 소개한다. 이 도구는 내부 모델체커로 SMV를 사용하며 사용자는 직접 논리식을 기술하지 않고도 레이스 조건, 애매한 전이, 순환 전이 등의 주요한 버그들을 검증할 수 있다.

키워드 : 상태 기계, 모델체킹, 자동 검증, 통합 설계, 단계 의미

Automatic Verification of the Control Flow Model for Effective Embedded Software Design

Sachoun Park[†] · Gihwon Kwon[††] · Soonhoi Ha[†††]

ABSTRACT

Hardware and software codesign framework called PeaCE(Ptolemy extension as a Codesign Environment) allows to express both data flow and control flow. To formally verify an fFSM specification which expresses control flow in PeaCE, the step semantics of the model was defined. In this paper, we introduce the automatic verification tool developed by formal semantics of previous work. This tool uses the SMV as inner model checker and, through our tool, users can formally verify some important bugs such as race condition, ambiguous transition, and circulartransition without directly writing logical formulae.

Key Words : State Machine, Model Checking, Automatic Verification, Codesign, Step Semantics

1. 소 개

PeaCE(Ptolemy extension as a Codesign Environment)는 임베디드 시스템 개발을 돕기 위한 하드웨어/소프트웨어 통합 설계 환경이다[1]. 이 환경에서는 시스템의 데이터 흐름과 제어 흐름을 모두 표현할 수 있는데, 시스템의 제어 흐름은 *f*FSM(*flexible* Finite State Machine) 모델로 나타낸다. 이 모델은 표현력을 높이기 위해서 유한 상태 기계를 확장했으며, 하렐의 스테이트챠트[2]와 같이 병행성 및 계층성을 표현할 수 있는 구문과 여러 기계간의 동기화를 위해 이벤트 및 상태 변수 등의 구문을 제공한다.

이전 연구에서는 *f*FSM 모델의 정형 분석(예를 들어 레이스 조건, 애매한 전이, 순환 전이 등이 모델에 없는지를 검증)을 돕기 위해서 *f*FSM 모델의 의미를 정형적으로 정의했다[3]. 이를 위해서 모델에 있는 계층 구조를 평탄화한 후에 단계 의미를 정의했는데, 유한 상태 기계 모델의 의미를 정의하는 데는 단계 의미(step semantics)가 전통적으로 사용된다[4]. 단계 의미란 모델이 이벤트를 받아서 한 형상(모델의 스냅샷을 일컫는 용어로서 형상은 상태 집합으로 표현된다)에서 다른 형상으로 변하는 과정을 정의하는 동시에 출력될 이벤트 집합을 정의한다.

우리는 이렇게 정의된 정형 의미를 바탕으로 자동 검증 도구인 Stepper를 개발했다. 이 도구는 내부 검증기로 SMV (Symbolic Model Verifier, [5])를 사용하는데, 본 논문을 통해서 *f*FSM 모델의 변환 규칙 및 정의된 속성들이 형식화되는 과정을 설명한다. 그 결과 어려운 논리식을 사용하지 않고도 레이스 조건, 애매한 전이, 순환 전이와 같은 버그를 자동으로 검출할 수 있었다.

2. 제어흐름 모델의 구문과 의미

이 장에서는 이전 연구에서 정의된 *f*FSM 모델의 구문과

※ 본 과제는 정보통신부 선도기반기술개발사업의 지원으로 수행되었습니다.
† 준 회 원 : 경기대학교 전자계산학과 박사과정
†† 종신회원 : 경기대학교 정보과학부 교수
††† 정 회 원 : 서울대학교 컴퓨터공학과 교수
논문접수 : 2005년 9월 21일, 심사완료 : 2005년 12월 12일

의미를 설명한다(본 논문에서는 평탄화된 *f*FSM 모델의 구문과 의미에 대해서만 다룬다. 상세한 설명은 [3]과 [6]을 참조하기 바람).

- **정의 1(평탄화된 모델)** : 평탄화 된 모델은 6-튜플(I, O, IT, M, γ, V)로 구성된다. 여기서 I, O, IT는 전과 같이 이벤트들의 집합이다. V 전역 변수의 집합이고, $M=\{m_1, ..., m_n\}$는 단순 FSM의 집합이다. $\Sigma = \bigcup_{i=1}^{n} S_i$은 M에 속한 모든 상태들의 집합이다. 계층 관계 γ: $\Sigma \rightarrow 2^M$는 상태를 그 상태가 포함하는 상태 기계들로 사상하는 함수이다.

계층함수 γ는 다음의 세 가지 속성을 갖는다. 첫째, 유일한 루트 상태 기계를 갖는다. 둘째, 루트를 제외한 모든 상태 기계들은 정확하게 하나의 조상 상태를 갖는다. 셋째, 계층수는 싸이클을 포함하지 않는다. $sub: \Sigma \rightarrow 2^{\Sigma}$일 때, $sub(s) = \{ s' \mid M_i \in \gamma(s) \wedge s' \in S_i \}$는 어떤 상태에 포함된 하위 상태들을 돌려주는 함수이다. sub^+은 sub의 추이 클로우져(transitive closure)이고 sub^*는 sub의 반사적 추이 클로우져(reflexive transitive closure)이다.

- **정의 2(단순 기계)** : 각각의 단순 기계 m_i는 4-튜플(S_i, s_i^0, T_i, scr_i)이다. 여기서, $S_i = (s_i^0, s_i^1, ..., s_i^n)$는 m_i의 유한 상태들의 집합이며, s_i^0는 초기상태, T_i는 m_i의 전이들의 집합이고 T_i에 속하는 전이 $t = (s, g, A, s')$는 출발 상태와 도착 상태 s, s' 및 부울식으로 표현되는 가드 조건 g 그리고 액션 집합 A로 구성된다. $scr_i: S_i \rightarrow 2^{Script}$는 상태에 스크립트를 사상하는 함수이다.

변수와 이벤트를 포함하는 가드는 아래와 같은 구문으로 정의된다:

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

여기서 n은 정수를 대표하고, $v \in V$는 전역변수이며, $\cdot \in \{+, -, \times, /\}$는 이진 연산자를 대표한다. 전이 $t = (s, g, A, s')$의 한 요소를 선택하기 위해서 추출 함수가 다음과 같이 사용된다: $source(t) = s$, $target(t) = s'$, $guard(t) = g$, $action(t) = A$. 또한, 액션 집합 A의 각 원소 $a \in A$들은 아래와 같이 변수의 값 배정이나 출력 이벤트의 방출로 구성된다:

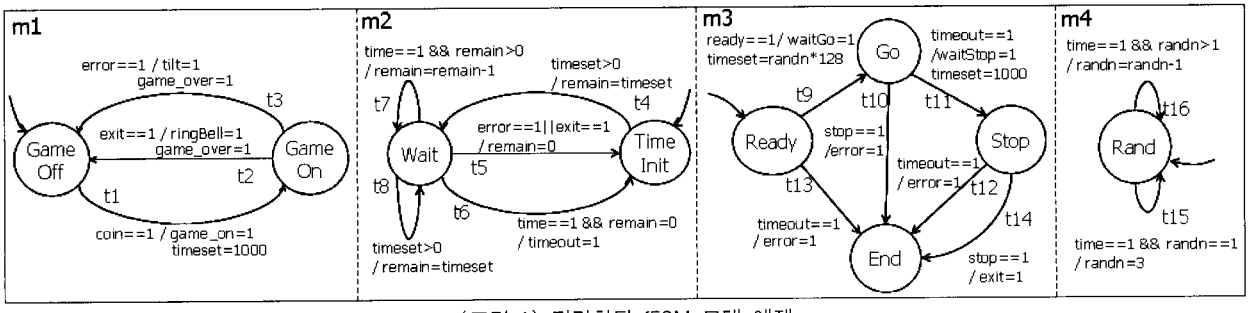
$$a ::= v := Exp \mid e := Exp.$$

액션에 대해서도 다음과 같은 세 개의 추출 함수가 쓰인다, 이들은 각각변수의 갱신, 출력 이벤트의 방출, 내부 이벤트의 생성에 관한 부분을 추출할 때 사용된다.

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

(그림 1)은 평탄화된 *f*FSM 모델이다. 이 모델의 이벤트로는 *coin*, *ready*, *stop*, *time*이 있다. 마지막 이벤트는 게임에서 잔여 시간을 줄이는데 사용되고 나머지 이벤트는 사용자에 의해 발생된다. 이 게임의 시나리오는 다음과 같다. 먼저 동전을 넣으면 *coin* 이벤트가 발생하고 시스템은 대기한다. 임의의 시간 후에 *ready* 이벤트가 발생하면 게이머는 재빨리 *stop* 버튼을 누른다. 이때 게임은 *stop* 이벤트를 발생시키고 *ready* 이벤트와 *stop* 이벤트가 발생한 시간 차이를 계산한다. 이 계산을 위해서 변수 상태 *remain*과 *randn*이 사용된다. 그리고 평탄화된 모델을 정형적으로 나타내면 다음과 같다:

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}, $\gamma(m_2) = \gamma(m_3) = \gamma(m_4) = \phi$,
$m_1 = (S_1, s_1^0, T_1, scr_1)$, s_1 = {GameOff, GameOn}, s_1^0 = GameOff,
T_1 = {(GameOff, coin = 1, {game_on = 1, timeset =



(그림 1) 평탄화된 fFSM 모델 예제

1000}, GameOn), (GameOn, exit = 1, {game_over = 1, ringBell = 1}, GameOff), (GameOn, error = 1, {game_over = 1, tilt = 1}, GameOff}, scr1 = ϕ

*f*FSM의 단계 의미는 입력 이벤트나 내부 이벤트에 의해서 모델의 형상이 변해가는 과정과 동시에 액션의 수행을 묘사하는 것인데, 여기서 액션은 출력 이벤트의 방출과 전역변수의 갱신 그리고 또 다른 내부 이벤트의 생성으로 구성된다.

- **정의 3(형상)** : Δ는 평탄화된 모델의 전체 형상을 나타낸다. 각 m_i에 대한 형상 집합의 정의는 $\Delta = \{\{s_1, \ldots, s_n\} \mid \exists s_i \in S_i, 0 < i \le n\}$이다. 특히 $\delta_0 \in \Delta$는 초기 형상으로서 $\delta_0 = \{s_1^0, \ldots, s_n^0\}$이다.

- **정의 4(활성화된 상태)** : 상태 s가 형상에서 활성화되었다는 것은 다음과 같이 정의될 수 있다:

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

- **정의 5(만족성)** : 어떤 전이가 활성화된 전이(enabled transition)인지를 결정하기 위해, 이벤트의 집합 $E \subseteq 2^{I} \cup 2^{IT}$과 현재의 형상 δ가 주어졌을 때, 형상 δ와 이벤트 E는 가드 g를 만족한다는 것, $\langle \delta, E \rangle \models guard(t)$을 아래와 같이 귀납적으로 정의한다.

$\langle \delta, E \rangle \models true$	*iff* $true$
$\langle \delta, E \rangle \models \neg G$	*iff* $\langle \delta, E \rangle \models G$
$\langle \delta, E \rangle \models G_1 \wedge G_2$	*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* $val(v) = val(Exp)$

여기서 $val(n) = n$이고, $val(e)$은 이벤트 e의 현재 값을 나타내며, $val(v)$는 변수 v의 현재 값을 나타낸다.

$$val(Exp_1 \cdot Exp_2) = val(Exp_1) \cdot val(Exp_2).$$

- **정의 6(활성화된 전이)** : 활성화된 상태와 만족성 관계의 정의에 의해서, 각 활성화된 상태에 대해서 활성화된 전이의 집합을 다음과 같이 정의한다.

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

- **정의 7(실행 전이)** : 실행 전이(executable transition)들의 집합은 충돌이 없는 가장 큰 전이들의 집합이고 반드시 모든 단순 FSM에서 최대 하나의 전이가 포함된다. *f*FSM 모델은 계층간 전이를 허용하지 않기 때문에, 충돌은 단지 우선순위를 비교할 수 있는 두 전이 사이에서만 발생한다.

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

- **정의 8(LKS)** : 단계 의미를 정의하는 프레임워크로서 LKS(Labeled Kripke Structure)가 사용된다. LKS는 4-튜플 (Q, q_0, R, L)로 구성된다. 여기서, $Q = \{q_0, \ldots, q_n\}$는 LKS의 유한 상태 집합, $q_0 = (\delta_0, \emptyset, c_0)$는 초기상태, $R \subset Q \times 2^{Act} \times Q$는 액션으로 레이블 된 전이관계, $L: Q \rightarrow 2^{Script}$는 레이블 함수로써 $L(q_i) = \bigcup_{\delta_i \models s} scr(s)$이다.

- **정의 9(마이크로 스텝)** : 현재 형상 δ와 이벤트의 집합 E가 주어졌을 때, 마이크로 스텝 $(\delta, E, c) \xrightarrow{Act} (\delta', E', c')$은 LKS 상에서 다음과 같이 정의된다:

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

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

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

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

- **정의 10(매크로 스텝)** : 단계 의미는 실행(execution) $q \xrightarrow{Exe} q'$로 정의된다. 하나의 입력 이벤트 집합과 연속해서 발생되는 내부 이벤트들의 집합은 더 이상 내부 이벤트가 없을 때까지 전이를 발생시킨다. 매 마이크로 스텝 후에, 델타-지연에 의해서 이전의 이벤트들은 모두 소멸된다. 아래의 정의에서, $k > 1$는 $E_{i,k}$가 $\emptyset$이 되게 하는 첫 번째 k를 의미한다. 그리고 $Exe_i = \bigcup_{\forall 0 < j < k} Act$는 매크로 스텝 동안에 이뤄지는 액션들의 집합을 의미한다.

$$(\delta_i, E_i, c) \xrightarrow{Exe_i} (\delta'_{i+1}, E'_{i+1}, c'_{i+1})$$

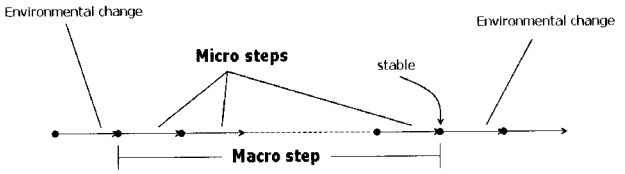
iff

$$(\delta_{i,1}, E_{i,1}, c_{i,1}) \xrightarrow{Act_1} \ldots \xrightarrow{Act_{k-1}} (\delta'_{i,k}, \emptyset, c'_{i,k}).$$

3. 제어흐름 모델의 CTL 모델체킹

이 장에서는 앞에서 정의한 정형 의미를 바탕으로 모델체킹 하는 과정을 다룬다. 먼저 만족해야 하는 속성을 CTL (computation tree logic)식으로 나타내는 방법을 살펴본 후에 모델을 대표적인 CTL 모델체커인 SMV의 입력언어로 변화하는 방법에 대해서 설명한다. CTL 논리의 구문과 의미 및 SMV 입력언어에 대해서는 [5]를 따른다.

속성의 정의와 정형화를 쉽게 하기 위해서 먼저 매크로 스텝과 마이크로 스텝, 그리고 동기화 가정(synchrony hypothesis)에 대해서 살펴보자. (그림 2)는 환경에 변화에 따라 동작하는 시스템의 행위를 스텝을 기준으로 나타낸 것이다. 환경의 변화는 시스템에 대한 외부의 자극이라고도 생각해 볼 수 있다. 즉 외부의 자극과 자극 사이에 시스템의 동작을 매크로 스텝이라고 하고, 한 매크로 스텝 동안에 시스템 내부 동작을 마이크로 스텝이라고 한다. 환경 변화 후 시스템의 내부 동작이 모두 완료되는 시점을 안정한(stable) 상태라고 한다. 여기서 동기화 가정이란 매크로 스텝은 외부의 변화보다 훨씬 빠르다는 가정이다. 즉 환경 변화의 속도보다 시스템의 처리 속도가 비교할 수 없을 만큼 빠르다는 것이다.



(그림 2) 동기화 가정

3.1 속성 표현

레이스 조건(Race condition)은 한 매크로 스텝 안에서 변수 상태 값이 두 번 이상 변경되거나, 출력 이벤트가 두 번 이상 방출될 때로 정의된다. 즉 한 매크로 동안에는 그 값이 일정해야 한다는 것이다. 따라서 "*시스템이 안정하지 않은 상태에서 어떤 변수나 이벤트가 한번 변경되거나 방출되었다면 시스템이 다시 안정화 될 때까지 해당 변수나 이벤트의 변경과 방출은 허용되지 않는다.*"라고 표현할 수 있다. $change(v)$를 시스템 내부의 변수 v의 변경 여부를 식별하는 함수라고 하고, $emit(o)$는 출력 이벤트 o의 방출이라고 정의하자. CTL은 명제논리에 기반 하기 때문에 변수 $v \in V$에 대해서 각각 검사해야 하고 출력 이벤트 o에 대해서도 같은 방식이 적용된다. 이제 레이스 조건을 CTL식으로 나타내면 다음과 같다. 여기서 안정화된 상태를 의미하는 $stable$은 내부 이벤트가 발생되지 않는 상태로 정의할 수 있으며, 다음 절에서 설명될 것이다.

$$AG((change(v) \wedge \neg stable) \Rightarrow AX\, A[\neg change(v) \cup stable])$$

$$AG((emit(o) \wedge \neg stable) \Rightarrow AX\, A[\neg emit(o) \cup stable])$$

애매한 전이(Ambiguous transition)는 하나의 상태에서 두 개 이상의 전이가 동시에 발생하는 상황을 의미한다. 이것은 전이의 쌍을 검사함으로써 알 수 있다. 만일 상태 s에서 발생되는 전이 집합이 $\{t_1, t_2, t_3\}$라고 한다면 상태 s에 대한 애매한 전이의 존재 여부는 아래와 같이 CTL 식으로 나타낼 수 있다.

$$AG\neg((t_1 \wedge t_2) \vee (t_2 \wedge t_3) \vee (t_1 \wedge t_3))$$

순환 전이(Circular transitions)는 마이크로 스텝이 계속되는 것을 의미한다. 즉, 외부 이벤트에 의해서 전이가 발생한 후 계속되는 내부 이벤트에 의해서 시스템이 안정화되지 않는 경우이다. 이를 CTL 식으로 표현하면 아래와 같다.

$$AG(\neg stable \Rightarrow A[\neg stable \cup stable])$$

위의 세 종류 외에도 데드락(Deadlock)과, 도달 불가능한 상태 분석과 발생되지 않는 전이, 사용되지 않는 컴포넌트에 대한 조사를 할 수 있다. 먼저 사용되지 않는 컴포넌트는 쉽게 표현된다. 여기서 컴포넌트란 변수가 될 수도 있고 이벤트가 될 수도 있다. 그러나 변수 v와 내부 이벤트 e, 출력 이벤트 o에 대해서 $EF\ omit(o)$, $EF\ occur(o)$ 또는 $EF\ change(v)$ 형태의 식으로 간단히 검사할 수 있다. 여기서 $occur(o)$와 $omit(o)$는 각각 내부 이벤트와 출력 이벤트의 발생을 의미하고 $change(v)$는 변수의 갱신이다. 전이 발생 여부는 전이 t의 출발상태 $source$와 도착상태 $target$이 있을 때, $EF(\wedge EX\ target)$로 표현될 수 있다. 데드락은 모든 전이가 더 이상 활성화 되지 않는 상태가 영원히 지속됨으로 표현할 수 있다. 즉, fFSM의 모든 전이들의 집합을 $fT = \{t_1, \cdots, t_n\}$라고 할 때, 다음과 같이 CTL 식으로 표현할 수 있다.

$$\neg EF\ AG \neg (t_1 \ldots t_n)$$

$Deadlock(fT) = \neg \bigcup_{\forall t \in fT} t$라고 정의하면 위 식은 $EF\ AG\ Deadlock(fT)$로 대신 쓸 수 있다. 데드락은 전체 시스템에 해당하는 것과 일부 시스템에 해당하는 것으로 구분할 수 있다. 전자를 전역(Global) 데드락이라고 하고 후자를 지역(Local) 데드락이라고 한다. 지역 데드락은 마치 라이브락(Livelock)과 같아서 전체 시스템 관점으로 보면 동작하는 것처럼 보이지만 시스템의 내부 모듈이 블록 되어서 향후 전혀 동작하지 않는 경우이다. 단순 FSM의 집합을 $\{M_1, \cdots, M_n\}$이라고 하고 각 모듈에 대한 전이 집합을 $\{T_1, \cdots, T_n\}$라고 한다면, 지역 데드락에 대한 식은 $\neg EF\ AG\ Deadlock(T_1) \vee \cdots \vee Deadlock(T_n)$으로 표현될 수 있다. 이때 $Deadlock(T_i)$는 단순 FSM의 집합에 속하는 원소 M_i의 데드락을 나타내며 전역 데드락을 다시 정의하면 $\neg$

$EF\,AG\ Deadlock(T_1) \wedge \cdots \wedge Deadlock(T_n)$가 된다.

마지막으로 unreachable guards의 존재 여부를 분석할 때, 식 $EF\ t$를 적용할 수 있다. 여기서 t는 모델의 전이이다. 즉 "*가드 조건에 도달하는가?*" 하는 문제는 그 가드가 만족될 때, 발생되는 전이를 찾는 문제로 생각할 수 있고, 전이는 상태의 쌍으로 표현될 수 있다.

3.2 SMV 입력언어로 변환

모델 체킹을 하기 위해서 상태도를 모델 체커의 입력언어로 모델을 변환하는 연구는 지금껏 많이 수행되어왔다. 우리는 *f*FSM의 모델 체킹을 위해서 Chan[7]과 Clarke[8]의 방식에서 많은 힌트를 얻었다. 2장에서 설명한 것과 같이 *f*FSM의 의미를 다루기 위해 우리는 N개의 머신으로 계층을 평탄화한 모델을 정의했다. SMV의 변수들은 모두 병렬적으로 동작하므로 평탄화된 머신과 상태들은 SMV의 변수와 변수 값으로 자연스럽게 변환할 수 있다.

- **변환규칙 1(머신과 상태)** : 평탄화된 모델이 (I, O, IT, M, γ, V)일 때, M에 속한 머신 $m_i = (S_i, s_i^0, T, scr_i)$에 대해서 해당 상태들은 $VAR\ m_i : S_i$로 변환된다.

```
VAR
Machine1 : {GameOff, GameOn};
Machine2 : {Timeinit, Wait};
```

- **변환규칙 2(전이)** : 전이는 정의 6의 활성화된 전이 조건 $\langle \delta, E \rangle \vDash guard(t) \wedge \delta \vDash source(t)$에 의거하여 SMV로 변환하다. 즉, $m_i = (S_i, s_i^0, T, scr_i)$와 $t \in T_i$에 대해서 어떤 상태를 포함하는 상위 상태들을 돌려주는 함수가 $up(s) = \{s' \mid M_i \in \gamma(s') \wedge s \in S_i\}$는 이고, up^*는 up의 반사적 추이 클로우져일 때, 각각의 전이들은 $DEFINE\ \ m_i_t := up^*(source(t)) \& guard(t))$로 변환된다.

```
DEFINE
Machine1_t1 := Machine1 = GameOn & error=1;
Machine1_t2 := Machine1 = GameOff & coin=1;
Machine1_t3 := Machine1 = GameOn & exit=1;
                    ⋮
ASSIGN
init(Machine1) := GameOff;
next(Machine1) :=
    case
    Machine1_t1 : GameOff;
    Machine1_t2 : GameOn;
    Machine1_t3 : GameOff;
    1 : Machine1;
    esac;
```

- **변환규칙 3(동기화 가정)** : 동기화 가정을 SMV에서 표현하기 위해, (그림 2)에서 설명한 안정한 상태를 의미하는 특별한 이진 변수 *stable*을 추가한다. 안정한 상태는 내부 이벤트 $\{internal_1, \cdots, internal_l\}$가 발생하지 않는 상태로 간단히 정의할 수 있지만, 우리는 생성된 반례의 가독성을 높이기 위해서 입력 이벤트 $\{input_1, \cdots, input_m\}$와 다중 값을 갖는 내부이벤트 $\{valued_1, \cdots, valued_n\}$를 함께 사용해서 변환한다.

```
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
```

- **변환규칙 4(입력 이벤트)** : 초기값이 d인 입력 이벤트 e의 도메인 D이고 도메인의 최소값과 최대값이 각각 $\min_e$, $\max_e$일 때, 다음과 같이 변환된다.

```
ASSIGN
init(e) := d
next(e) :=
    case
    stable : min_e .. max_e
    1 : 0;
    esac;
```

- **변환규칙 5(출력 이벤트)** : 어떤 전이 $t = (s, g, A, s')$이고, $action(t) = A$일 때, 초기값 d를 갖는 출력 이벤트 e에 대해서 $output(A) = \{e := Exp \mid \exists e \in O.\ (e := Exp) \in A\}$을 만족하는 모든 전이 집합 $\{t_1, \cdots, t_n\}$과 각 식의 집합 $\{Exp_1, \cdots, Exp_n\}$ 대해서 다음과 같이 변환한다.

```
ASSIGN
init(e) := d
next(e) :=
    case
    t_1 : Exp_1
    ⋮
    t_n : Exp_n
    1 : 0;
    esac;
```

• **변환규칙 6(내부 이벤트)** : 어떤 전이 $t = (s, g, A, s')$이고, $action(t) = A$일 때, 초기값 d를 갖는 내부 이벤트 e에 대해서 $signal(A) = \{e := Exp \mid \exists e \in IT \cdot (e := Exp) \in A\}$을 만족하는 모든 전이 집합 $\{t_1, \cdots, t_n\}$과 각 식의 집합 $\{Exp_1, \cdots, Exp_n\}$ 대해서 다음과 같이 변환한다.

ASSIGN
init(e) := d
next(e) :=
case
t_1 : Exp_1
⋮
t_n : Exp_n
1 : 0
esac;

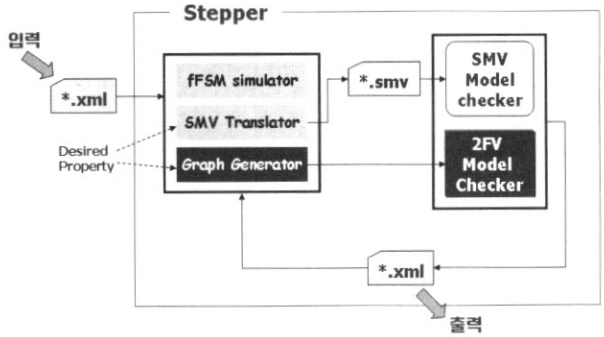
• **변환규칙 7(변수)** : 어떤 전이 $t = (s, g, A, s')$이고, $action(t) = A$일 때, 초기값 d를 갖는 변수 v에 대해서 $update(A) = \{v := Exp \mid \exists v \in V \cdot (v := Exp) \in A\}$을 만족하는 모든 전이 집합 $\{t_1, \cdots, t_n\}$과 각 식의 집합 $\{Exp_1, \cdots, Exp_n\}$ 대해서 다음과 같이 변환한다.

ASSIGN
init(v) := d
next(v) :=
case
t_1 : Exp_1
⋮
t_n : Exp_n
1 : v
esac;

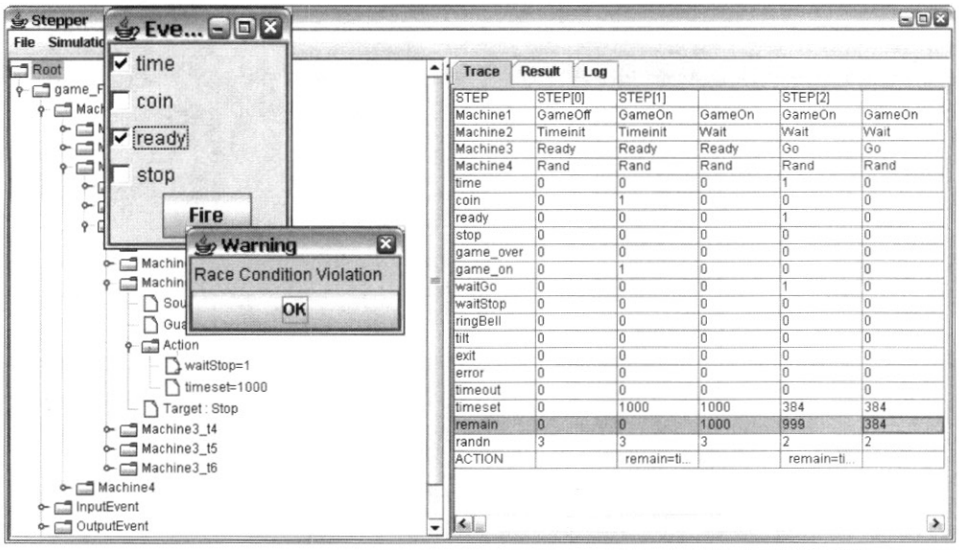
4. 제어흐름 모델 자동 검증

PeaCE 도구에서 정형 모델 분석을 위해서 (그림 3)과 같이 속성을 자동으로 검사할 수 있는 검증 도구를 개발했다. 사용자는 6개의 주요 속성(레이스 조건, 애매한 전이, 순환하는 전이, 비사용 컴포넌트, 도달되지 않는 가드, 데드락)에 대해서 자동으로 검사할 수 있다. 이 도구는 XML 형태의 입력을 받아서 검증결과를 XML 파일로 돌려준다. Stepper는 내부적으로 두개의 모델체커를 가지고 있다. 따라서 암시적 모델체커인 SMV 변환 모듈과 명시적 모델체커를 위한 그래프 생성 알고리즘 모듈 및 검증 모듈로 구성되어 있다.

버그 검출의 예로써 레이스 조건 위반의 예를 생각해 보자. 만일 현재의 형상이 {GameOn, Wait, Ready, Rand} 이고, 발생된 이벤트가 각각 ready 와 time일 때, 그리고 변수 상태 remain 의 값이 0일 때, 드문 경우이기는 하지만, 출력 이벤트와 변수 상태의 값이 한 실행 동안 지속되어야 한다는 제약사항을 위반하게 된다. 우리는 *f*FSM 모델에 대한 버그 검출을 위해 이 논문에서 제안한 단계의미를 적용해서 우선 시뮬레이터를 구현하였다. (그림 4)는 시뮬레이터를 이용해서 레이스 조건 위반을 검출하는 과정을 보여준다.

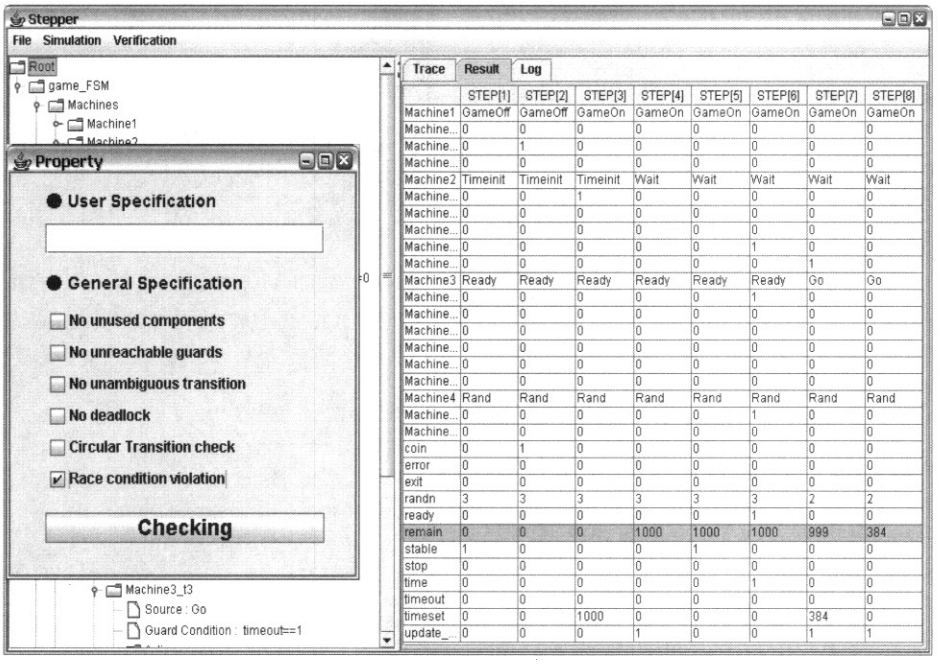


(그림 3) Stepper의 구조



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
waitStop	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	999	384
randn	3	3	3	2	2
ACTION		remain=ti..		remain=ti..	

(그림 4) 시뮬레이션으로 레이스 조건 위반 검출



	STEP[1]	STEP[2]	STEP[3]	STEP[4]	STEP[5]	STEP[6]	STEP[7]	STEP[8]
Machine1	GameOff	GameOff	GameOn	GameOn	GameOn	GameOn	GameOn	GameOn
Machine...	0	0	0	0	0	0	0	0
Machine...	0	1	0	0	0	0	0	0
Machine...	0	0	0	0	0	0	0	0
Machine2	Timeinit	Timeinit	Timeinit	Wait	Wait	Wait	Wait	Wait
Machine...	0	0	1	0	0	0	0	0
Machine...	0	0	0	0	0	0	0	0
Machine...	0	0	0	0	0	0	0	0
Machine...	0	0	0	0	0	1	0	0
Machine...	0	0	0	0	0	0	1	0
Machine3	Ready	Ready	Ready	Ready	Ready	Ready	Go	Go
Machine...	0	0	0	0	0	1	0	0
Machine...	0	0	0	0	0	0	0	0
Machine...	0	0	0	0	0	0	0	0
Machine...	0	0	0	0	0	0	0	0
Machine...	0	0	0	0	0	0	0	0
Machine...	0	0	0	0	0	0	0	0
Machine4	Rand	Rand	Rand	Rand	Rand	Rand	Rand	Rand
Machine...	0	0	0	0	0	1	0	0
Machine...	0	0	0	0	0	0	0	0
coin	0	1	0	0	0	0	0	0
error	0	0	0	0	0	0	0	0
exit	0	0	0	0	0	0	0	0
randn	3	3	3	3	3	3	2	2
ready	0	0	0	0	0	1	0	0
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_	0	0	0	1	0	0	1	1

(그림 5) 모델체킹을 이용한 레이스 조건 위반 검출

STEP[0]은 시스템이 초기 형상으로 초기화 되어 있음을 나타낸다. STEP[1]은 coin 이벤트가 발생된 후, 내부 이벤트 timeset이 연속해서 발생된 것을 나타낸다. 이때의 형상이 {GameOn, Wait, Ready, Rand}이고 ready 와 time 이벤트가 동시에 발생하면 한 매크로 스텝 내에서 remain 값이 두 번 이상 변경되므로 시뮬레이터는 레이스 조건 위반을 검출하게 된다.

사용자가 손쉽게 버그를 잡도록 돕는데 시뮬레이션이 빈번히 사용되지만, 시뮬레이션만으로는 중요한 버그를 철저한 정형 분석은 하기 어렵다. 따라서 모델체킹 기법으로 시스템의 행위를 철저하게 검사하는 검증 활동이 필요하다. 우리는 시뮬레이션 툴에 모델체킹이 가능하도록 하기 위해, 3장에서 설명한 것과 같이 *f*FSM 모델을 SMV 입력 언어로 바꾸고, 사용자에 의해 검증할 속성을 선택하도록 한 후, 속성을 CTL 논리로 변환하여 검증을 수행하도록 했다. 내부 검증기로는 Cadence SMV[9]를 사용했다. (그림 5)는 시뮬레이션을 이용해서 검출했던 레이스 조건 위반 사례를 모델체킹을 이용해서 검출한 것이다. 이 버그를 검출하기 위해 모델체킹 시간은 6초, BDD 노드는 456022개, 메모리는 15Mb를 사용했다. 비록 작은 모델이기는 하지만, 비교적 빠른 시간 안에 전체 상태공간을 탐색하게 된다. 특히 실제 시뮬레이션으로 같은 버그를 찾아내는 과정은 이보다 훨씬 더 많은 노력과 시간이 들어간다. 우리는 6가지 built-in 속성뿐 아니라 사용자에 의해서 직접 CTL 논리로 속성을 기술할 수 있도록 하였다.

5. 결론과 향후 연구

복잡한 임베디드 시스템을 설계하는 방법으로, 하드웨어/소프트웨어 통합설계가 제품의 생산성을 높일 새로운 방법론으로 주목 받고 있다. PeaCE[1]는 하드웨어/소프트웨어 통합 설계 프레임워크로써 데이터 흐름과 제어 흐름을 표현하는 모델을 가지고 있다. *f*FSM는 PeaCE의 제어 흐름을 표현하는 모델이다. 이전 연구를 통해서 *f*FSM 모델의 단계의미를 정의하였다.

본 연구에서는, 그 단계의미를 기반 하여 시뮬레이션과 모델체킹이 가능한 검증 도구를 개발했다. 그 결과 해당 버그를 정형적으로 분석할 수 있게 되었고, 사용자들은 푸쉬 버튼 방식으로 속성들을 선택할 수 있기 때문에 보다 쉽게 제어흐름 모델의 에러를 검사할 수 있도록 하였다. 현재 우리는 본 논문에서 정의된 *f*FSM의 단계 의미를 기반으로 *f*FSM 전용의 모델체커를 개발하고 있다. 이를 위해 효율적인 추상화 방법을 연구하고 있다. 뿐만 아니라 앞으로는 제어흐름 모델에 대한 정형 의미와 두 모델간의 의미 통합에 관한 연구를 진행할 것이다.

참 고 문 헌

[1] D. Kim, S. Ha, "Static Analysis and Automatic Code Synthesis of flexible FSM Model," in the Proceedings of ASP-DAC, pp.18-21, 2005.

[2] D. Harel, A. Naamad, "The STATEMATE semantics of statecharts," ACM Transactions on Software Engineering Methodology, Vol.5, No.4, 1996.

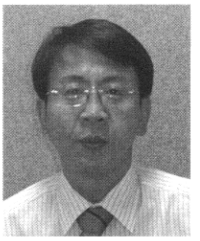
[3] S. Park, G. Kwon, and S. Ha, "Formalization of fFSM Model and Its Verification," in the Proceedings of the ICESS, LNCS 3820, Springer, pp.361-372, 2005.

[4] A. Pnueli and M. Shalev. "What is in a step: On the

semantics of Statecharts," in the Proceedings of the TACS, LNCS 526, Springer, pp.244-264, 1991.
[5] E. M. Clarke, O. Grumberg and D. Peled, Model Checking, MIT Press, 1999.
[6] D. Kim, "System-Level Specification and Cosimulation for Multimedia Embedded Systems," Ph.D. Dissertation, Computer Science Department, Seoul National University, 2004.
[7] W. Chan, "Symbolic Model checking for Large software Specification," Ph.D. Dissertation, Computer Science and Engineering, University of Washington, 1999.
[8] E. M. Clarke and W. Heinle, "Modular translation of Statecharts to SMV," Technical Report CMU-CS-00-XXX, Carnegie Mellon University, 2000.
[9] http://embedded.eecs.berkeley.edu/Alumni/kenmcmil/smv/
[10] J. B. Lind-Nielsen, "Verification of Large State/Event Systems," Ph.D. Dissertation, Department of Information Technology, Technical University of Denmark, 2000.

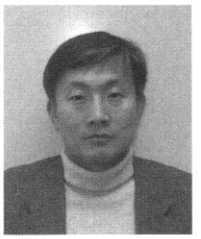
박 사 천

e-mail : sachem@kyonggi.ac.kr
2001년 경기대학교 전자계산학과(학사)
2003년 경기대학교 전자계산학과(이학석사)
2004년~현재 경기대학교 전자계산학과 박사과정
관심분야 : 모델 체킹, 정형기법, 소프트웨어 공학 등



권 기 현

e-mail : khkwon@kyonggi.ac.kr
1985년 경기대학교 전자계산학과(학사)
1987년 중앙대학교 전자계산학과(이학석사)
1991년 중앙대학교 전자계산학과(공학박사)
1998년~1999년 독일드레스덴 대학 전자계산학과 방문교수
1999년~2000년 미국 카네기 멜론 대학 전자계산학과 방문교수
1991년~현재 경기대학교 정보과학부 교수
관심분야 : 소프트웨어 모델링, 소프트웨어 분석, 정형 기법 등



하 순 회

e-mail : sha@iris.snu.ac.kr
1985년 서울대학교 컴퓨터공학과(학사)
1987년 서울대학교 컴퓨터공학과 (공학석사)
1992년 University of California, Berkeley (PhD in Electrical Engineering and Computer Science)
1993년~1994년 현대전자근무
1994년~현재 서울대학교 컴퓨터공학과 교수
관심분야 : Hardware-software codesign, design methodology for embedded systems , PC clusters