2002년 한국정보처리학회 춘계학술발표논문집 제 9 권 제 1 호

병행성을 이용한 상태도의 검증

구자철, 권기현
경기대학교 전자계산학과
e-mail : {jachul, khkwon}@kyonggi.ac.kr

Verification of Statechart using Parallel Machines

Jachul Koo, Gihyun Kwon
Software Engineering Lab., Dept. of Computer Science, Kyonggi University

요 약

상태도의 계층구조를 이용하여 모형 검사를 수행하는 연구들이 있었다. 기존 연구들은 상태도를 SMV 입력언어로 변환할 때, 계층 구조를 유지하였다. 비록 이러한 연구가 계층형 모형검사의 초기 시도에 기여를 했지만, SMV 는 평탄화 된 구조를 이용한다는 점에서 실효성은 없다. 본 연구에서는, 계층구조를 평탄화 시킨 후 상태도를 병행적으로 실행되는 여러 개의 독립된 기계로 분리한다. 따라서, 상태도의 구조가 단순해지고 SMV 언어로 쉽게 상태도를 변환할 수 있다. 본 연구의 목적은 SMV 를 이용하여 상태도를 모형 검사 할 때 SMV 의 능력을 최대한 활용하고자 한다.

1. 서론

시각 명세 언어로 널리 사용되는 상태도는 유한 상태 기계를 확장하여 표현력을 높이고 의미를 잘 전달할 수 있다 [1]. 이러한 개념은 상태들간의 계층성과 병행성, 공중방송(broadcasting)등과 같은 표현의 추가를 통해 이루어 졌다. 이러한 장점으로 많은 상용도구에서 상태도를 지원한다. 그렇지만, 모형 작성시 디자이너가 원하지 않는 오류를 포함할 수 있다. 이를 해결할 수 있는 방법이 모형 검사라고 믿고 있으며, 모형 검사 수행을 위한 연구가 이루어 졌다[2, 3, 4].

SMV(Symbolic Model Verifier)에서 모형 검사의 수행은 시스템 모형을 작성하고, 해당 모형 검사기의 입력 언어로 변환한 뒤 원하는 속성을 CTL(Computation Tree Logic)로 표현하여 모형 검사를 실시 하게 된다. SMV 는 유한 상태기계와 상태도 등의 모형을 검사할 수 있다[5].

디자이너는 상태도로 모형을 작성할 때 상태간의 병행성과 계층성, 공중방송을 통하여 시각적으로 디자인 한다. 그러나, SMV 의 내부에서 모든 모형들은 평탄화 된다는 특성을 가지고 있다. 상태도가 계층적 구조를 이용하는 반면에 SMV 는 평탄화 된 구조를 이용하는 것은 앞으로 모형 검사 부분에서 극복해야 할 사항이다. 그렇지만, 현재의 모형 검사기가 계층적 구조를 이용하지 않는다면, 입력언어로 변환된 상태도 역시 병행적 구조를 이용하여 보다 직관적이고 편리한 방법으로써 모형검사를 할 수 있다. 이전 연구[2]에서 SMV 의 입력 언어로 변환된 상태도의 계층성을 표현하기 위해 노력했다. 계층성을 표현하는 입력 언어 변환은 많은 의미를 부여할 수 있지만, SMV 내부로 들어 갔을 경우 평탄화 되기는 마찬가지라는 문제점이 있다. 결국, 기존의 연구들이 상태도 위주의 모형 검사를 위하여 입력언어로 변환을 했다면, 지금 시도 되고 있는 우리의 연구는 SMV 를 중심으로 상태도를 입력언어로 변환 시킨다. 이러한 방법으로 모형 검사를 수행 할 때 직관적 방법으로써 이용될 수 있다. 또한 모형 검사기의 입력 언어로써의 상태도는 그 구조가 매우 단순해 질 수 있다.

2 장에서는 계층형 상태도의 구조를 유지하여 모형 검사 언어로 변환할 때의 문제점과 병행적 구조의 상태도 구성 방법에 대하여 기술하고, 3 장에서는 ETL 에 대한 구문과 의미를 정의하고, 4 장에서는 평탄화 된 상태도를 SMV 의 입력언어로 변화는 방법에 대하여 설명한다. 마지막으로 5 장에서는 결론을 내린다.

2. 병행적 구조의 상태도

2.1 계층형 상태도의 문제점

그림 1 은 상태도의 계층적 구조를 잘 보이고 있으며, 각 상태들 간의 관계에서 Working 은 And 상태이며 두 개의 자식 상태를 가지고 있으며 각 상태는 Image 와 Sound 이다. 이처럼 상태 간의 관계는 중첩된 구조로써 나타난다. 상태에서 상태로의 이동은 전이를 통하여 이루어 진다. 만약 Standby 에서 전이 t4 가 발생하면 Disconnected 로 이동하게 된다.

상태도는 시각적으로 시스템을 명세할 때 편리하게 사용될 수 있을 뿐만 아니라 모형 검사를 수행할 때 모형 검사기의 입력 모형으로 사용이 된다. 우리가 관심을 가지고 있는 부분은 시스템의 모형 부분이다. 모형 검사를 수행하기 전에 모형은 모형 검사기의 입력 언어로 변환이 되어 진다.

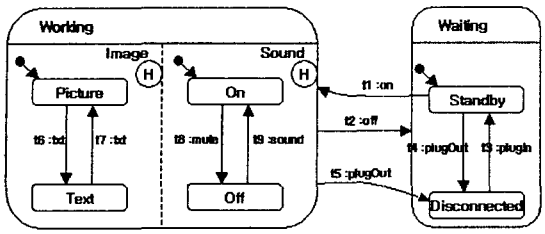

그림 1 계층형 상태도의 예

상태도의 경우 계층적 구조를 이용 하여 SMV 의 입력 언어로 변환이 되어 진다. SMV 는 입력 언어 방식에 상관없이 내부적으로 모두 평탄화 되어 진다. 기존의 많은 연구들이 상태도를 모형 검사하는데 있어서 계층성을 유지하는 입력언어로 변환하는데 관심을 기울였다. 이러한 계층성의 유지는 상태도의 구조를 모형 검사기의 입력언어로 바꾸는데 이질 적인 부분을 감소시키는 방법을 제시했다. 그렇지만, 이러한 연구들이 근본적인 문제를 해결하지는 못했다. 즉, SMV 내부에서는 모두 평탄화 되고 계층화된 구조가 소용이 없어진다는 것이 문제점으로 남는다.

2.2 병행적 구조의 상태도 변환

기본 적인 개념은 2.1 절에서 설명한 계층형 상태도의 문제점을 극복하기 위하여 상태간의 관계를 모두 독립적인 구조로 가져간다. 최상위 상태는 And 상태로써 구성이 되고, 그 하위 상태들은 Or 로써 구성을 한다. 각 상태간의 계층 관계가 소멸되고, 최상위 상태를 제외한 모든 상태들은 병행적 구조로 구성이 되어 진다.

1. 최상위 상태가 And 이면 그대로 유지되고, Or 혹은 History 이면 And 타입의 새로운 최상위 상태 Root 를 생성한다.
2. 상태 s 가 And 타입 일 때 자식 상태들 중에서 타입이 Or 또는 History 인 자식상태에 대하여 3 을 적용 하고 타입이 And 인 자식은 2 를 다시 적용한다.
3. 만약 상태 s 가 Or 또는 History 일 때 s 는 Root 의 자식 상태로 복사되고, s 의 모든 자식 상태들은 복사되어 유지된다.
4. 만약 자신의 상태 s 가 Basic 이라면 s 에 대한 수행은 종료된다.

최상위 상태 Root 의 자식 상태들은 cWorking, cImage, cSound, cWatiting 이 된다. 각 상태들은 평탄화가 되고, 전이의 구조가 바뀌게 된다. 전이의 변화는 각 상태들의 내포 관계와 상태들간의 레벨에 따라 전이의 구조에 영향을 주게 된다. 다시 말해, t1 은 On 에서 Working 으로 가는 전이 였으나 평탄화가 이루어 지면서 cTV 의 Waiting 에서 Working 으로 향하는 전이가 되었다. 레벨이 한단계 높아지면서 t1 에서의 전이에 In(Standby)라는 상태 참조가 이루어 졌다. 이는 원래의 계층 구조에서 병행적 구조로 변환하는데 매우 중요하다. 즉 cSound 의 상태가 On 에 머무르고 있어야 cTV 의 Waiting 상태에는 on 이벤트를 실행 시킬 수 있다. 평탄화 된 상태도에서 전이에 따른 상태들간의 관계는 레벨이 같아져서 표현이 된다. 또한, 상위 상태에서의 탈출 전이는 모든 하위 상태들에게 영향을 준다. Working 의 하위 상태들의 전이에 Working 의 탈출 전이를 '¬' 붙여서 연접 시킨다. 이는 전이의 갈등 해결 표현에 중요한 역할을 한다.

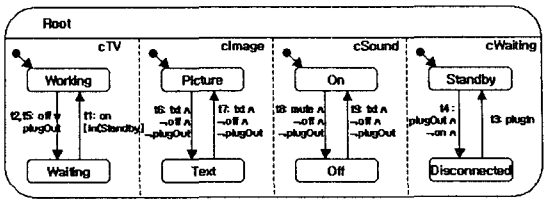

그림 2 병행적 구조의 상태도

그림 2 에서 나타나듯이 상태간의 계층 구조를 무시하고 단지 상태 참조만으로 같은 행위를 하는 평탄화 된 상태도를 구성할 수 있는 것은 모형 검사를 수행할 때 커다란 장점으로 적용된다. 그리고, History 의 표현은 병행적 구조를 이용할 때 부가적인 표현이 필요가 없다. 기본 상태로의 전이 역시 쉽게 표현할 수 있다. 병행적 구조의 상태도를 이용하여 직관적이고, 단순한 상태도의 구성을 자연스럽게 유도 할 수 있다.

3. ETL 의 정의

3.1 기본개념

상태도에 기술된 내용을 SMV 로 변환하기 위해서 몇 가지 고려해야 할 사항이 있다. 상태도의 경우 과거 시제에서 현재 시제로써의 전이를 표현하고, SMV 의 경우 현재 시제에서 미래 시제의 전이를 표현한다. 결국, 상태도와 SMV 는 하나의 시제 차이를 갖게 된다. 이러한 문제점을 해결하기 위해 시제논리의 확장인 ETL(extended temporal logic)[2]을 이용한다.

3.2 구문과 의미

기본적인 기호들의 집합들은 아래에 기술된 내용을 기반으로 한다.

- 변수 $v_1, v_2, \ldots$ 그리고, 상수 $c_1, c_2, \ldots$
- 초기 만족조건 *init* 와 이항 동치 연산자(=)를 포함하는 적정 차수의 술어 P
- 적정 차수의 함수 f
- 명제 접속사 $\wedge, \vee, \neg, \ldots,$
- 단항 시제 접속사 ■, ★

또한 일반적 텀의 정의는 다음과 같다.

- 변수와 상수는 텀이다.
- 함수 f 가 n 개의 텀을 갖고, $t_1, \ldots, t_2$ 가 텀이라면, $f(t_1, \ldots, t_n)$은 텀이다.

정의 1 : ETL 식은 귀납적으로 다음과 같다.

- 술어 P 와 텀 $t_1, \ldots, t_n$ 의 $P(t_1, \ldots, t_n)$
- 식 A 의 $\neg A$, ■A, ★A
- 식 A 와 B 의 불린 조합 $A \wedge B, A \vee B, A \rightarrow B$
- 텀 t_1 과 t_2 의 ■$t_1 = t_2$, ★$t_1 = t_2$

SMV 에 의해 생성되는 모델들 일반적으로 run 이라고 한다. 이러한 run 들의 특성은 무한하고, 이산적이며, 초기 지점으로부터 후-단계(next step) 관계에 의해 생성되는 선형적 시퀀스이다. 이러한 분리된 점들은 각 지점의 변수의 값으로 인해 특성화 된다. 이러한 분리된 점들을 '단계(step)'라 한다. 단계들은 ETL 모형을 생성하는데 중요한 정보를 가지고 있다.

정의 2 : ETL 프레임은 시퀀스 I 에서 해석 I_j 와 각 도매인 D 의 범위로써 표현한다. 이때, $j = 0, 1, \ldots$이고, $I_j = (P_j, f_j, C_j)$ 이다.

- $\mathcal{P}_j(P)$는 적합한 n의 D^n에서의 술어이다. 이때 n은 술어 P의 차수이다.
- $\mathcal{P}_j(=)$은 모든 j에 대하여 확장된 동치이다.
- $\mathcal{F}_j(f)$는 적합한 n의 함수 $S^n \rightarrow S$이다. 이때 n은 함수 f의 차수이다.
- $\mathcal{C}_j(c)$는 S의 객체에 상수를 할당하는 함수이다.

ETL 모델은 ETL 프레임과 S의 객체에 모든 변수를 할당하는 평가 α_j의 시퀀스이다. ETL모델은 흔히 run이라 불리우고, 각 요소 (l_j, α_j)는 단계라 한다. (l_0, α_0)은 초기 단계라고 한다.

다음은 ETL 식을 평가하는 정의이다.

정의 3 : 모델 M과 정수 j에 대하여 j 번째의 단계에서 텀의 평가를 귀납적으로 정의한다.

- 각 변수의 해석 $v^{\gamma} = \alpha_j(v)$
- 각 상수의 해석 $c^{\gamma} = \mathcal{C}_j(v)$
- 함수 f와 텀 $t_1, \ldots, t_n$의 해석 $f(t_1, \ldots, t_n)^{\gamma} = \mathcal{F}_j(f)(t_1^{\gamma}, \ldots, t_n^{\gamma})$

또한, 아래의 조건이 만족하면 $M,j \vDash A$ 이다. 즉, M이 j 단계에서 식 A를 만족한다고 말할 수 있다.

- $M,0 \vDash init$ and $M,j+1 \nvDash$ init for all j
- $M,0 \vDash P(t_1, .., t_n)$ iff $\mathcal{P}_j(f)(t_1^{\gamma}, \ldots, t_n^{\gamma}) = T$
- $M,j \vDash \neg A$ iff $M,j \nvDash A$
- $M,j \vDash A \wedge B$ iff $M,j \vDash A$ and $M,j \vDash B$
- $M,j \vDash \blacksquare A$ iff $M,j+1 \vDash A$
- $M,0 \nvDash \bigstar A$ for all formula A
- $M,j+1 \vDash \bigstar A$ iff $M,j \vDash A$
- $M,j \vDash t_1 = t_2$ iff $t_1^{\gamma+1} = t_2^{\gamma}$
- $M,j \vDash \blacksquare t_1 = t_2$ iff $t_1^{\gamma} = t_2^{\gamma}$
- $M,0 \vDash \bigstar t_1 = t_2$ for all terms $t1$ and $t2$
- $M,j+1 \vDash \bigstar t_1 = t_2$ iff $t_1^{\gamma} = t_2^{*j+1}$

ETL 식 A가 모든 j에서 $M,j \vDash A$를 만족한다면 ***정당한 모델***$(M \vDash A)$이라고 하며, 프레임을 기반으로 한 모든 모델이 정당하다면 ***정당한 프레임***이라고 한다. 마지막으로, 모든 모델이 정당하다면 ***정당***(valid)하다고 말할 수 있다.

보조 정리 1 : ETL 식 A와 B는 다음의 내용을 만족한다.

- $\vDash \blacksquare(A \wedge B) \leftrightarrow \blacksquare A \wedge \blacksquare B$
- $\vDash \bigstar(A \wedge B) \leftrightarrow \bigstar A \wedge \bigstar B$
- $\vDash \blacksquare \bigstar A \leftrightarrow A$
- if $\vDash A$ then $\vDash \blacksquare A$
- $\vDash \blacksquare(\bigstar t_1 = t_2) \leftrightarrow \blacksquare t_2 = t_1$
- $\vDash \bigstar(\blacksquare t_1 = t_2) \leftrightarrow \bigstar t_2 = t_1$

보조 정리 1의 증명은 위의 정의를 이용하면 쉽게 증명할 수 있고, 5번째와 6번째의 내용은 시제의 이동을 보인다.

4. SMV 로 변환

계층적 구조를 이용하지 않고 병행적 구조의 상태도를 SMV 코드로 변환하는데 초점을 맞춘다. 최상위 상태의 타입이 And 라는 점은 SMV 코드로 변환할 때 장점으로 이용할 수 있다. SMV 의 실행 메커니즘은 병행적 구조이다. 따라서 선언된 모든 변수는 병행적으로 처리가 되고, 병행적 상태도 역시 SMV 의 실행 메커니즘의 구조를 잘 반영할 수 있다. 또한 main 모듈 이외의 다른 모듈은 사용을 하지 않는다. 상태도의 모든 정보가 main 안에서 표현이 될 수 있다는 것은 코드의 간결성을 유지시키고, 이해성을 높이는데 많은 도움을 줄 수 있다. 상태도에 관한 내용을 정형적으로 표현하기 위하여 ETL 술어 *region, active, state, default* 를 이용한다.

- 상태 s가 영역 r에 포함이 된다면 *region*(s) = r 라 한다.
- *region*(s) = r 이고 영역의 제어가 s에 머무를 때 *state*(r) = s 이다.
- 영역 r에 제어가 머무를 때 *active*(r)은 참이다. 또한 상태 s에 제어가 머무를 때 *active*(s)는 참이다.
- 기본 전이에 의해 s가 활성화 되었을 때 *default*(s)는 참이다.

4.1 상태 변환

상태 : 상태도를 SMV 로 변환 할 때 계층적 구조에 관한 표현의 중심은 상태간의 관계를 기술하는 부분이었다. 본 연구는 상태들의 관계를 좀더 단순하게 처리하기 위하여 병행적 구조의 상태도를 이용한다. 병행적 구조의 상태도에서는 Root 의 자식 상태만을 고려하면 된다. 자식 상태들은 Or 타입이기 때문에 Or 의 자식 상태는 변수로 간주한다. 각 Or 상태들은 Basic 상태만을 갖는다. 따라서, 상태들은 다음과 같은 SMV 코드로 변환된다.

```
VAR state : {s1, ..., sn};
```

기본 상태의 표현역시 고려해야 한다. 상태도에서 각 Or 영역 r 의 초기 상태가 s 임을 나타낸다. 따라서 *default*(r) = s 이 참이 된다. 이것을 SMV 로 나타내면 다음과 같다.

```
init(state):= s;
```

활성 상태 : 상태들간의 관계는 Root 를 기준으로써 명시적으로 기술 할 수 있다. Root 는 시스템이 종료될 때 까지 제어가 머무르는 것은 *active*(Root)이다. 영역 r 이 활성화 된 것은 직관적으로 표현이 된다.

$$active(r) \leftrightarrow active(\text{Root})$$

상태 s 가 활성화 된다는 것은 영역 r 이 활성화 되고, 차트의 상태가 s 일 때 만족된다.

$$active(s) \leftrightarrow state(r) = s \wedge active(r)$$

상태도의 상태들을 SMV 로 표현하기 위하여 모든 상태들의 간접 상태를 이용한다. 이는 제어의 흐름을 명시적으로 표현하기 위한 수단이 된다. 활성화 된 상태들은 SMV 의 DEFINE 문을 이용하여 정의한다.

```
DEFINE
   in_Root := 1;
   in_region := in_Root;
   in_state := in_region & state = s;
```

4.2 전이

실행 전이 : 술어 *trigger*(S, T)는 상태 S 에서 T 로 전이되는 평가를 나타낸다. 상태 S 를 출발 상태라고 하고, T 를 목적 상태라고 한다. 상태의 전이가 발생되기 위해서는 *trigger* 가 참으로 평가 되어야 한다. 상태도의 전이는 하나의 영역에 오직 하나의 전이만이 실행된다. 즉, 영역의 현재 상태 S 가 만족되고, *trigger* 가 만족되어 T로의 전이가 실행될 수 있다. 이는 모든 영역에서 동시적으로 적용된다. 이때, *trigger* 가 만족 되지 않는다면 상태는 S를 유지한다.

$$active(r) \rightarrow (state(r) = S \wedge ((trigger(S, T) \wedge \blacksquare(state(r) = T)) \vee (\neg trigger(S, T) \wedge \blacksquare(state(r) = S)))$$

이 조건을 SMV 로 나타내면 다음과 같다.

```
DEFINE
 t := in_state & trigger
```

```
ASSIGN
    next(state) = case
        state = S & t : T;
            ...
        1 : S;
    esac;
```

4.3 이벤트, 조건과 액션

상태도에 전이의 구성 요소는 이벤트와 조건 변수 그리고 액션이다. 이벤트는 오직 한 단계동안만 유효하고, 조건 변수는 값이 변경 될 때까지 유효하게 지속된다. 그리고, 이벤트와 조건 변수는 상태도에서 참과 거짓을 나타내는 불린 값으로 표현된다. 다음은 이벤트와 조건 변수에 대한 표현이다.

<event>:= e |e1 **and** e2|e1 **or** e2|**not** e|**en**(s)|**ex**(s)|
<condition>|**tr**(<condition>)| **fs**(<condition>)
| **ch**(<condition>)
<condition>:= c|c1 **and** c2|c1 **or** c2|**not** c| **in**(s)

4.3.1 이벤트와 조건

상태도에서의 이벤트와 조건 변수는 전역적으로 처리된다. 이벤트와 조건 변수는 어떠한 상태에서도 참조 될 수 있지만, 같은 차트에서는 오직 하나만이 참조 된다. 따라서, 이벤트와 조건 변수는 상태들의 표현과 마찬가지로 SMV 내에서 main 모듈 안에서 처리된다. 이는 SMV 코드의 단순성과 이해성을 높일 수 있다. 이벤트의 경우 발생 유무에 따라 참과 거짓의 값만을 갖는다. 또한, 이벤트들은 모두 공중 전파 되기 때문에 각 전이에서 영향을 준다. 따라서, 이벤트를 SMV 로 변환을 할 때 boolean 타입의 변수로써 지정된다.

```
VAR event : boolean;
```

4.3.2 복합 이벤트와 조건

다른 변수들의 참조를 표현하기 위하여 복합 이벤트와 조건을 정의한다. 이러한 표현은 전이의 라벨에 명시적으로 표현이 되며, 상태도와 SMV 간의 시제 차이를 일치 시키기 위하여 복합적 구성 요소들의 재귀적 표현을 위하여 시제 논리 Δ의 의미를 정의한다. SMV 로의 변환은 다음의 정의에 의해 쉽게 변환될 수 있다.

정의 4 : V 는 이벤트 변수이고 e, e_1, e_2 는 이벤트 표현일 때, 다음과 같다.

- $\Delta(V) = V$
- $\Delta(\mathbf{not}\ e) = \neg\Delta(e)$
- $\Delta(e_1\ \mathbf{and}\ e_2) = \Delta(e_1) \wedge \Delta(e_2)$
- $\Delta(e_1\ \mathbf{or}\ e_2) = \Delta(e_1) \vee \Delta(e_2)$

조건의 표현도 이벤트와 유사하게 정의 된다.

정의 5 : c 는 조건 표현일 때 다음과 같다.

- $\Delta(\mathbf{tr}(c)) = \bigstar\neg\Delta(c) \wedge \Delta(c)$
- $\Delta(\mathbf{fs}(c)) = \bigstar\Delta(c) \wedge \neg(\Delta(c))$
- $\Delta(\mathbf{ch}(c)) = \neg(\bigstar\Delta(c) \leftrightarrow \Delta(c))$
- $\Delta([c]) = \Delta(c)$

다음으로 상태와 관련된 이벤트와 조건에 관한 의미이다.

정의 6 : 위에서 정의된 내용을 기반으로 아래와 같이 정의된다.

- $\Delta(\mathbf{en}(s)) = active(r) \wedge state(r) = s \wedge$
  $((\bigstar active(r) \wedge \bigstar state(r) = s) \vee$
  $(\bigstar\neg active(r) \wedge (\bigstar default(s) \vee \bigstar state(r) = s)$
- $\Delta(\mathbf{ex}(s)) = \bigstar active(r) \wedge \bigstar state(r) = s \wedge$
  $(state(r) = s' \rightarrow active(r))$
- $\Delta(\mathbf{in}(s)) = active(r) \wedge state(r) = s$

4.3.3 액션

액션을 표현하는데 필요한 정보는 단지 액션의 발생유무 이다. 다음은 액션 변수에 대한 표현이다.

<act>:= nil|e|**tr!**(c)|**fs!**(c)|c:=<condition>

따라서 boolean 변수로 쉽게 나타낼 수 있다. 초기 상태에서 액션 e 는 발생하지 않는다.

$\neg init(e)$

액션은 전이 부분 중 $/e$ 에 의해서 발생할 수 있다. 이러한 액션의 발생은 이벤트 변수를 오직 한단계에서만 참이 되게 한다. 예를 들어 전이 t 에 의해서 S 에서 T 로의 이동이 발생된다면 액션의 표현은 다음과 같다.

$(t \wedge \bigstar state(r)=S \wedge state(r) = T) \rightarrow e$

만약 e 를 생성하는 전이 행동이 아무도 없는 경우
$\neg t \rightarrow \neg e$ 이다. 위 식을 종합해서 SMV 코드로 나타내면 다음과 같다.

```
VAR
   e: boolean;
ASSIGN
   init(e):= 0;
   next(e):= case
      t & state=S & next(state)=T: 1;
      1:0;
   esac;
```

5. 결론

평탄화 된 상태도에 대하여 설명하고, SMV 코드로 변환하는 방법에 대하여 기술하였다. 모형 검사에 있어서 SMV 코드 표현은 계층성을 유지하는데 중요한 비중을 차지했다. 그러나, SMV 내부에서의 처리는 평탄화 된다. 이러한 방법은 입력 언어로의 변환이 매우 복잡하고 번거로운 작업이었다. 계층화된 입력언어를 받아들이는 모형 검사기의 경우 유용하게 사용이 될 수 있다. 그러나, 아직까지 현실적으로는 이를 지원하지 못한다. 본 연구는 이러한 계층적 표현을 제거하고, SMV 내부의 방식에 적합하게 변환하는 과정에 대하여 설명하였다. 평탄화 된 방식으로써 SMV 의 입력 언어 변환 과정은 단순하고, 의미를 이해하기 쉽게 유도한다. 따라서, 보다 SMV 다운 입력언어를 생성 할 수 있다.

참고문헌

[1] D. Harel, A. Naamad, "The STATEMATE semantics of Statecharts," ACM Transactions on Software Engineering and Methodology, Vol.5, No.4, pp.293-333, 1996

[2] E. M. Clarke, W. Heinle, "Modular Translation of Statecharts to SMV", Carnegie Mellon University, August 3, 2000

[3] E. Mikk, Y. Lakhnech, and M. Siegel. "Hierarchical automata as model for statecharts", In Asian Computing Science Conference (ASIAN'97), volume 1345 of LNCS. Springer Verlag, December 97

[4] William Chan, et. Al., IEEE Transactions on Software Engineering 24(7), pages 498-520, July 1998

[5]K.L. McMillan, "Symbolic Model Checking: An approach to the state explosion problem, PhD thesis, Carnegie Mellon University, 1992