Statechart 명세의 등가 관계 검사
(Equivalence Checking for Statechart Specification)

박 명 환 † 방 기 석 † 최 진 영 †† 이 정 아 ††† 한 상 용 ††††
(Myung-Hwan Park)(Ki-Seok Bang)(Jin-Young Choi)(Jeong-A Lee)(Sang-Yoong Han)

요 약 본 논문에서는 가상 프로토타입핑의 주요 명세 언어인 Statechart 명세를 프로세스 알제브라의 일종인 ACSR(Algebra of Communicating Shared Resources)로 변환하는 규칙을 제안한다. Statechart는 사용하기 편리하고 이해하기 쉬운 명세 언어이지만 수학적인 semantics의 정의가 되어 있지 않아 명세의 정확성을 검증하기가 매우 어렵다. Statechart 명세를 ACSR로 바꾸게 되면 Statechart에 수학적인 semantics를 주게 되고 VERSA를 이용해서 Statechart 명세를 수학적으로 검증할 수 있게 된다. 따라서, 두 언어의 장점, 즉 Statechart의 편리함과 ACSR의 정확성을 모두 얻을 수 있다.

Abstract In this paper, we give a formal semantics for Statechart via a translation into Algebra of Communicating Shared Resources(ACSR). Statechart is a very rich graphical specification language, which is suitable to specify complicated reactive systems. However, the incorporation of graph into specification and rich syntax makes Statechart semantics very complicated and ambiguous. Thus, it is very difficult to verify the correctness of Statechart specifications. Also, we propose the formal verification method for Statechart specifications by showing equivalence relation between two Statechart specifications. This makes it possible to combine the advantages of a graphical language with the rigor of process algebra.

1. 서 론

가상 프로토타입핑[1]은 기존의 실물 프로토타입핑보다 개발 시간이 단축되고 실제품의 외형 및 기능 변환을 손쉽게 테스트 해 볼 수 있게 하므로 제품의 생명주기를 상당히 줄 일수 있다. 또한, 가상 프로토타입이 정형 명세로 이루어져 있으므로 설계자나 개발자, 그리고 사용자사이의 의사전달에 애매모호함을 완전히 제거해 준다. 정형 명세언어로는 주로 Statechart[2] 사용되고 있다. Statechart는 그래픽 기반의 명세언어로서 사용하기 편리하고 표현력이 풍부해서 복잡한 시스템을 효율적으로 명세 할 수 있다[2]. 그러나, 명세에 그래프의 도입과 풍부한 문법(syntax)의 정의는 오히려 의미(semantics)의 복잡함과 모호함이라는 부작용을 초래했다. 따라서, Statechart 명세의 정확성을 검증하기는 매우 어려운 작업이 되었다. Statechart에 정형검증을 접목하려는 노력은 크게 두 가지 방법으로 시도되고 있다. 첫 번째는 Statechart명세에 모델 체킹 기법을 도입하려는 노력으로서 Statechart 명세가 safety나 deadlock 등의 특성을 만족하는지를 검증할 수 있는 장점이 있다[3,4,5]. 두번째는 Statechart 명세를 프로세스 알제브라로 변환하여 수학적인 기반의 의미를 주려는 노력으로서 이를 통해 두 Statechart 명세간의 등가 관계(equivalence relation)가 성립하는지 검증할 수 있다[6,7,8]. 본 논문에서는 Statechart 명세를 프로세스 알제브라의 일종인 ACSR[9,10]로 변환하여 Statechart 명세간에 등가 관계를 검증할 수 있는 방법을 제시한다. 논문의 구성은 2장과 3장에서 Statechart와 ACSR에 대해 각각 소개하고 4장에서는 Statechart 명세를

· 본 연구는 1998년도 과학기술기초 중점연구지원사업(1998-016-E00060)의 지원을 받았습니다.
† 학생회원 : 고려대학교 컴퓨터학과
mpark@formal.korea.ac.kr
kbang@formal.korea.ac.kr
†† 종신회원 : 고려대학교 컴퓨터학과 교수
choi@formal.korea.ac.kr
††† 종신회원 : 조선대학교 컴퓨터공학부 교수
jeong@eunhasu.kjist.ac.kr
†††† 정 회 원 : 중앙대학교 컴퓨터공학과 교수
han@arch.csc.cau.ac.kr
논문접수 : 2000년 1월 17일
심사완료 : 2000년 9월 18일

ACSR로 변환하는 규칙을 설명한다. 5장에서는 신호등 제어기 시스템을 ACSR로 바꾸어서 등가 검사(equivalence checking)를 실시하는 예를 소개하고 6장에서 결론을 맺는다.

2. Statechart

Statechart는 reactive 시스템의 명세에 적합한 그래픽 언어이다.[11] 그 근본구조는 상태 전이도(state transition diagram)에 높이(hierarchy), 동시성(concurrency), 통신(communication)의 개념을 도입한 것이다. 높이는 트리에서 서브트리가 파생되듯이, 하나의 스테이트가 서브 스테이트를 가질 수 있게 하는 것으로 계층적 구조를 표현한다. 동시성은 스테이트들 간에 병렬성을 제공하는 것인데, 두개 또는 그 이상의 병렬 컴포넌트가 모여서 하나의 스테이트를 구성하게 된다. 통신은 시스템의 컴포넌트들간에 방송(broadcasting)을 통해 정보를 주고받으면서 시스템이 진행하는 것을 의미한다.

2.1 스테이트

Statechart에는 3종류의 스테이트가 있다. OR-스테이트와 AND-스테이트, 그리고 basic-스테이트이다. OR-스테이트와 AND-스테이트는 서브 스테이트를 가질 수 있다. OR-스테이트가 가지는 서브 스테이트들간의 의미는 'exclusive-OR' 이고, 이것은 서브 스테이트들 중 단 하나만 제어를 가질 수 있다는 의미이다. AND-스테이트가 가지는 서브 스테이트들간의 의미는 'AND'인데 이것은 서브 스테이트들 모두가 제어를 가져야 한다는 의미이다. Basic 스테이트는 스테이트 높이의 가장 밑에 있는 스테이트로서 어떠한 서브 스테이트도 가지고 있지 않다.

2.2 이벤트와 조건

Statechart에서 스테이트는 이벤트에 의해 동기화 된다. 그리고 이벤트는 전체 시스템에 방송된다. 외부 이벤트는 시스템 외부에서 발생하여 시스템 내부로 방송되는 이벤트이고 내부 이벤트는 시스템 내에서 발생하는 이벤트이다. 본 논문에서는 내부 이벤트의 발생을 다음 세 가지 요소로 제한한다. 첫째, 전이의 액션부분에서 발생하는 이벤트이다. 이 이벤트는 전이의 trigger가 참이 되어야 발생한다. 둘째, 어떤 스테이트에 제어가 들어가는 순간에 발생하는 *en*(*S*)와 스테이트에서 빠져나갈 때 발생하는 *ex*(*S*) 이벤트이다. 셋째, 시간 관련 이벤트로서 *timeout* 이벤트가 있다. 또한, 조건도 *in*(*S*) 문법만 사용한다.

2.3 전이

전이는 출발(source) 스테이트에서 출발해서 도착(destination) 스테이트로 가는 화살표로 표시된다. 전이의 레이블은 다음의 형식을 가지고 있다.

trigger[*condition*]/*action*

*trigger*는 이벤트들로서 조건의 값이 참일 경우 전이를 발생시킨다. *Trigger*에 사용된 이벤트들은 *or*, *and*, *not*의 부울 연산에 닫혀있다. *condition*은 *trigger* 이벤트가 발생할 때 현재 값이 참이 아니면 전이의 발생을 억제하는 역할을 한다. *action*은 이벤트의 집합으로서 전이가 진행될 때 발생한다. 위의 세 요소들은 모두 선택적이다.

2.4 Basic System Reaction

시스템은 다음과 같이 작동한다. 먼저 환경이 외부 이벤트를 생성하면 이것은 시스템 내부로 방송되고 전이들이 trigger된다. 서로 충돌이 발생하지 않는 한 trigger가 참이 되는 모든 전이가 발생한다. 그리고 이 전이는 다시 새로운 이벤트를 발생시킨다. 이런 일련의 과정들을 스텝(step)이라고 한다. 모든 이벤트들은 한 스텝이 지나면 소멸된다. 한 스텝이 진행할 동안은 시간이 흐르지 않는다고 가정함으로써 스텝 진행 도중 발생한 외부 이벤트들은 스텝의 진행에 영향을 주지 못한다. 내부에서 발생한 이벤트들에 의하여 시스템은 계속 진행하다가 더 이상 진행을 하지 못하면 이 상태를 안정(stable) 상태라고 말한다. 외부 이벤트가 발생하여 안정 상태가 깨진 후 다음 안정 상태로 가기 위하여 거친 모든 스텝을 *superstep*이라 한다. 안정 상태에서는 외부 이벤트가 들어오지 않는 한 시스템은 어떠한 변화도 발생하지 않는다.

3. ACSR(Algebra of Communicating Shared Resources)

ACSR은 CCS[12]기반의 프로세스 알제브라로서 실시간 시스템을 효과적으로 명세할 수 있는 여러 특성들을 가지고 있다. 실시간 시스템의 가장 두드러진 특징은 프로세스의 동기화를 위한 지연과 공유자원의 사용가능 여부이다. 기존의 실시간 프로세스 알제브라는 동기화를 위한 지연은 처리할 수 있으나 공유 자원에 관해서는 무한한 자원이 있는 이상적인 환경을 가정함으로써 해결하였다. 그러나 ACSR에서는 시스템은 제한된 공유자원을 가지고 있고 또한 그 자원은 특정 순간에 하나의 액션만 수행할 수 있다고 가정함으로써 좀더 사실적으로 실시간을 명세할 수 있다. 제한된 자원을 가정함으로써 공유자원간의 scheduling이 중요한 관점으로 떠오른다. ACSR은 스케줄링 문제를 자원사용에 있어 우선 순

위를 줌으로써 해결하였다. 또한, 실시간 시스템에 필수적인 deadlines, 통신, 동기화, 그리고 자원에 대한 개념이 모두 포함되어 있다. ACSR은 다른 프로세스 알제브라에서 제공되는 많은 공통적인 연산자들을 제공한다. *prefix*는 액션이나 이벤트들을 순서화(sequencing) 시켜준다. *choice*는 두개의 프로세스들 중에서 한 개를 선택한다. *parallel*은 두개의 프로세스가 병렬적으로 동작하는 것을 나타낸다. *restriction*은 통신의 세부사항을 추상화시켜주는 역할을 한다. *recursion*은 무한히 돌아가는 프로세스를 나타낸다. 또한, ACSR은 다양한 종류의 시간 관련 연산자들을 제공한다. 특히, *timeout*과 *interrupt*는 다른 프로세스 알제브라에서는 없는 대표적인 시간관련 연산자들로써 실시간 시스템을 명세할 때 매우 효과적으로 사용될 수 있다. 본 논문에서 등장하는 ACSR 문법의 부분집합은 다음과 같다.

$P ::= NIL \mid A{:}P \mid (a,n).P \mid P+Q \mid P\|Q \mid P\backslash F \mid P\Delta_t(R,S) \mid recX.P$

*NIL*은 아무 행동도 하지 않는 프로세스, 즉 deadlock된 프로세스이다. 만약 *NIL*과 다른 프로세스가 동시에 실행된다면 그 프로세스를 포함한 전체 프로세스 또한 *NIL*이 될 것이다. 일반적으로 *NIL*은 모든 프로세스의 종료, 또는 오류를 나타낸다. 실시간 프로세스는 종료시 완전히 멈추어지는 것보다 아무것도 하지 않고 그냥 시간만 보내는 idling으로 표현된다. 프로세스 $A{:}P$는 한 단위시간 동안 집합 A에 포함된 자원들을 각각의 우선 순위로 사용하고 프로세스 P로 전이한다. $(a,n).P$는 이벤트 (a,n)을 수행하고 시간 소모없이 P 프로세스로 진행한다. 선택 연산자 $P+Q$는 비결정성을 나타내며, 자원의 제한과 환경에 따라 두 프로세스 중 어느 한 쪽이라도 선택되어 실행될 수 있음을 의미한다. 병렬 연산자 $P\|Q$는 동시에 실행되는 두개의 프로세스를 나타낸다. 제한 연산자 $P\backslash F$는 P의 행위를 제한한다. 즉 프로세스 P는 집합 F안에 정의된 이벤트는 실행할 수 없음을 표현한다. Scope 연산자 $P\Delta_t(R,S)$는 프로세스 P에 timeout과 interrupt의 개념을 부여한다. 위의 Scope의 의미는 프로세스 P가 실행되다가 t time이 되면 timeout이 발생하여 프로세스 R이 실행된다. 그리고 프로세스 P가 실행되고 있는 동안 interrupt가 발생하면 interrupt handler인 프로세스 S가 실행된다. 회귀 연산자 $recX.P$는 무한히 반복적인 행위를 명세한다.

3.1 등가관계

ACSR 프로세스들 사이의 등가는 바이시뮬레이션(bisimulation)이라는 개념에 기초하고 있다. 이 개념은 만약 프로세스 P(또는 Q)가 액션 α를 수행하여 한 단계 진행하면 Q(또는 P) 또한 액션 α를 수행하여 한 단계 진행되어야 하며, 다음의 프로세스 역시 바이시밀러(bisimilar)하다는 뜻이다[13]. 약한 바이시뮬레이션(weak bisimulation)은 프로세스들의 관찰가능한 외부 동작들을 비교하는 등가관계로서 어떤 두 프로세스의 외부 동작이 같을 때 약한 바이시뮬레이션 관계에 있다고 한다. 약한 바이시뮬레이션은 "$\approx$"와 같이 표기한다.

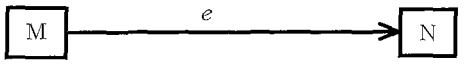
4. 변환 규칙

이 장에서는 Statechart 명세를 ACSR로 바꾸는 일반적인 규칙을 설명한다. 본 논문에 적용된 Statechart는 D.Harel에 의해 제안되고 Statemate[14]에 적용된 Statechart를 사용한다. 변환의 간편함을 위하여 activity관련 이벤트나 value passing, 조건 변수, history 스테이트, static reaction등은 고려하지 않는다.

4.1 의미론적 차이

4.1.1 Non-delayed VS Delayed 이벤트

Statechart에서는 전이의 triggering 조건이 참이 되지 않는 한 제어가 그 스테이트에 계속해서 머무를 수 있다. 즉, 다음 그림에서 이벤트 e가 발생하지 않는 한 제어는 M 스테이트에 머물러 있다.



반면에, ACSR에서는 제어가 특정 프로세스에 머무를 수 없다. 예를 들면, $Q = e.P$의 표현에서 Q 프로세스에 제어가 온다면 시간 소모 없이 바로 이벤트 e를 받고 제어가 P로 넘어가게 된다. 따라서, ACSR 프로세스에 제어를 유지하기 위한 mechanism이 필요하다. 아래 식은 위의 Statechart 명세와 동일한 의미를 가지는 ACSR 명세이다.

$$Q \stackrel{def}{=} recX.(e.P + \varnothing : X)$$

Q 프로세스는 이벤트 e를 받고 P 프로세스로 진행하거나 아니면 idling하면서 제어를 Q 프로세스에 유지한다.

4.1.2 방송 VS 동기화

ACSR은 동기적인 통신 mechanism을 가지고 있는 반면에 Statechart는 방송 mechanism을 가지고 있다. 다시 말하면, Statechart에서는 이벤트가 발생하면 그 이벤트를 필요로 하는 모든 곳에서 사용할 수 있지만 ACSR에서는 우선 순위가 가장 높은 프로세스만이 그것을 사용할 수 있다. 따라서, ACSR에서 이벤트들을 방송 시켜주는 mechanism이 필요한데 이것은 다음의 새로운 프로세스를 정의함으로써 해결하였다.

$$a??.P \overset{def}{=} recX.(a?.P + \varnothing : X)$$

$$a!!.Q \overset{def}{=} recX.(a!.X + Q)$$

$a??.P$ 프로세스는 sender가 이벤트 a를 보내줄 때까지 시간에 무관하게 기다리는 프로세스이다. $a!!.Q$ 프로세스는 시간 진행없이 a 이벤트를 방송 시켜준다. 즉, 시간을 소모하는 구조가 존재하지 않고 반복적으로 a 이벤트를 보내게 된다.

4.1.3 스테이트 변수

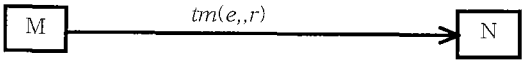
Statechart에서는 스테이트와 관련된 이벤트와 조건이 있다. 어떤 특정 스테이트에 제어가 들어가면 그 순간에 *en*이라는 이벤트가 발생되고 *in* 조건은 참이 된다. 그리고 그 스테이트에서 제어가 빠져나오면 *ex*라는 이벤트가 발생하고 *in* 조건은 거짓이 된다. ACSR에서는 프로세스와 관련된 이런 이벤트나 조건은 존재하지 않는다. 따라서, 프로세스에 제어가 들어갈 때나 나올 때, 그리고 제어가 현재 프로세스에 있는지 없는지를 알려주는 2가지 mechanism이 필요하다. 전자는, 프로세스에 제어가 들어갈 때나 나올 때 이벤트를 방송시켜주는 프로세스를 추가하면 해결할 수 있다. 이 부분은 다음절에서 다룬다. 후자는, Statechart에서 *in* 조건에 해당하는 부분인데 ACSR에서는 조건의 개념이 없으므로 이벤트로 변환시켜야 한다. 아래의 ACSR 표현은 ACSR의 *in* 조건을 이벤트로 변환시킨 것이다.

1. $P\_Mtest \overset{def}{=} P\_Mf$
2. $P\_Mf \overset{def}{=} f\_M!.P\_Mf + enter\_M?.P\_Mt + \varnothing : P\_Mf$
3. $P\_Mt \overset{def}{=} t\_M!.P\_Mt + exit\_M?.P\_Mf + \varnothing : P\_Mt$

위의 ACSR 프로세스는 Statechart의 스테이트마다 한 개씩 존재하여야 하며 이런 상태 표현 프로세스는 실제 시스템의 행위를 표현하는 프로세스와 병렬적으로 돌아가야 한다. 위의 상태 표현 프로세스의 행위를 세부적으로 살펴보자. 먼저 1의 표현처럼 프로세스에 제어가 없는 상태에서 출발한다. 2의 표현에서 프로세스에 제어가 들어오지 않으면 제어가 없다는 이벤트(*f_M*)를 계속 보내면서 시간을 idling 한다. 그러나, 제어가 들어왔다는 이벤트(*enter_M*)가 들어오면 3의 표현으로 넘어간다. 3에서는 제어가 나가지 않는 이상 현재 제어를 가지고 있다는 이벤트(*t_M*)를 계속 보내면서 idling하다가 제어가 나갔다는 신호(*exit_M*)가 오면 다시 1번 표현으로 넘어간다.

4.1.4 Timing construct

Statechart에서 timing construct는 timeout이 있다. timeout의 문법은 $tm(e,r)$인데 이것은 이벤트 e가 발생한 후 r time 단위 후에 $tm(e,r)$ 이벤트가 발생한다는 것을 의미한다.



위 그림은 Statechart에서 M 스테이트와 N 스테이트간의 전이에 timeout 이벤트가 적용된 경우이다. M 스테이트에 제어가 있고 이벤트 e가 발생하면 r time이 지나면 $tm(e,r)$ 이벤트가 발생하고 이것은 다시 전이를 trigger한다. 위 표현을 ACSR로 고치면 다음과 같다.

$$M \overset{def}{=} e??.IDLE\Delta_r(N, NIL)$$

위 ACSR 식은 Scope 연산자를 사용한 것으로서 이벤트 e가 발생하길 기다리고 있다가 이벤트 e가 발생되면 이 프로세스는 Idling하면서 시간을 소모한다. r time unit이 지나면 제어는 Scope를 벗어나서 timeout handler인 N 프로세스를 시작한다. 즉, Statechart의 의미와 동일하게 된다.

4.1.5 이벤트 방송 및 동기화

Statechart에서는 이벤트가 발생하면 그것을 감지하기 위해서는 한 time이 소모된다. 그러나 ACSR에서는 이벤트의 발생과 그것의 감지는 시간 소모없이 동시에 진행된다. 따라서 Statechart의 명세를 시간 고려없이 그대로 ACSR로 바꾸게되면 두 명세간에 시간의 의미가 틀려지고 또한 Statechart의 Synchrony Hypothesis [14]가 깨지게 된다. 이것을 해결하기 위하여 ACSR의 이벤트 발생과 그것을 감지하는데 한 time 단위가 소모되게 하는 시간 지연이 필요하다. 시간 지연은 외부 이벤트를 포함해서 Statechart 명세에서 발생하는 모든 이벤트를 가장 먼저 동기화해서 한 time을 idling한 후에 방송 시켜주는 프로세스이다. time delay는 각 이벤트마다 하나의 프로세스가 필요하고 그 프로세스들이 서로 병렬적으로 수행되게 구현한다.

$$TD \overset{def}{=} ((1) \| (2) \| \ldots \| (n))$$

그리고 이벤트(1)이 t_n이라고 하고 j가 가장 높은 우선 순위이고 k가 그 다음 높은 우선 순위라고하면 이벤트는 다음과 같이 구현된다.

$$(1) \overset{def}{=} (t\_n?,k).\varnothing : recX.((t\_n!,j).X + (1)) + \varnothing : (1)$$

위의 식을 간략히 설명하면 이벤트(1)은 t_n이 발생하면 이것을 바로 잡아서 한 time을 쉰 후 전체 시스템으로 보내준다. t_n이 들어오지 않으면 idling하면서 시간을 소모한다.

4.2 변환 방법

이 절에서는 Statechart 명세를 ACSR 명세로 바꾸는 방법을 소개한다.

4.2.1 스테이트

각 스테이트는 두개의 ACSR 프로세스를 가지고 있는데 하나는 스테이트의 행위를 기술하는 프로세스이고 다른 하나는 스테이트의 상태를 나타내는 프로세스이다. 이 두 프로세스는 항상 동시에 돌아가야 한다. Statechart의 스테이트가 ACSR의 프로세스로 바뀌는 과정은 다음과 같다. 먼저, 시스템의 가장 상위 스테이트 P_M에 제어가 들어가면서 $enter_M!$을 발생시킨다. 이것을 ACSR로 표현하면 다음과 같다.

$$P\_M \overset{def}{=} (enter\_M!, p).P\_M'$$

P_M' 프로세스는 스테이트의 행위를 나타내는 프로세스로서 다음 3가지 프로세스로 분류된다.

1. Basic 스테이트 M,

$$P\_M' \overset{def}{=} IDLE$$

2. 서브 스테이트 M_1, M_2, ... ,M_n을 갖고 있는 동시 수행 스테이트 M,

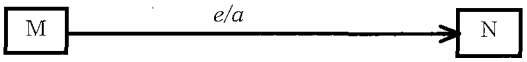
$$P\_M' \overset{def}{=} P\_M_1 \| ... \| P\_M_n$$

3. 초기 상태 M_1을 갖고 있는 OR 스테이트 M,

$$P\_M' \overset{def}{=} P\_M_1$$

4.2.2 전이

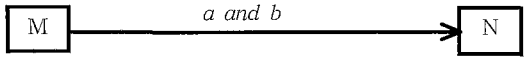
스테이트들간의 전이는 trigger 조건에 따라 여러가지로 변환될 수 있다. trigger가 이벤트이고 액션을 가진 경우의 Statechart의 명세는 다음과 같다.



위의 표현은 제어가 스테이트 M에 있을 때 이벤트 e가 발생하면 전이가 발생하고 이벤트 a를 발생한다는 의미이다. 이것을 ACSR 표현으로 바꾸면 다음과 같다.

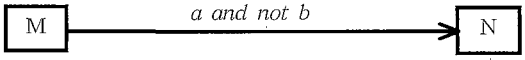
$$M \overset{def}{=} (e?,1).(exit\_M!,1).(e!,1).N + \varnothing : M$$

이 표현의 의미는 M 프로세스에서 이벤트 e를 기다리다가 e가 들어오면 M 프로세스를 나간다는 신호($exit\_M!$)와 이벤트 a를 발생시킨 후에 프로세스 N으로 제어가 넘어간다. trigger가 이벤트의 논리곱으로 되어있을 경우에는 두개의 이벤트가 동시에, 즉 같은 스텝에 발생하여야 전이가 발생한다. 이것의 Statechart 명세는 다음과 같다.



위 식과 동일한 의미를 가지는 ACSR 표현은 다음과 같다.

$$M \overset{def}{=} (a?,1).((b?,2).(exit\_M!,1).N + M) + (b?,1).((a?,2).(exit\_M!,1).N + M) + \varnothing : M$$

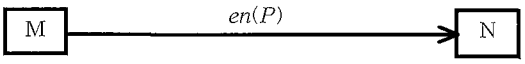


프로세스 M은 이벤트 a나 b가 발생하길 기다리다가 둘 중 어느 한 개가 발생하면 다른 쪽 이벤트도 발생했는지 확인한다. 나중 확인하는 이벤트의 우선순위가 높은 이유는 Choice의 다른 쪽 프로세스와 동기화하는 것을 방지하기 위해서이다. 두 이벤트 중 한개만 발생하면 전이가 발생하지 않는다. Trigger가 a *and not* b의 형태로 되어 있는 경우는 이벤트 a가 발생한 순간에 이벤트 b가 존재하지 않아야 전이가 발생한다. 이것의 Statechart 표현은 다음과 같다.

위 식의 ACSR 변환은 다음과 같다.

$$M \overset{def}{=} (a?,1).((b?,1).M + (exit\_M!,1).N) + \varnothing : M$$

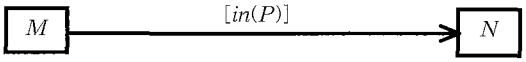
trigger가 en일 경우는 병렬적으로 돌아가는 다른 스테이트의 제어 이동을 감지해서 전이의 trigger로 하는 경우이다.



위 식의 ACSR 변환은 아래와 같다.

$$M \overset{def}{=} (enter\_P?,1).(exit\_M!,1).N + \varnothing : M$$

trigger가 ex일 경우는 입력되는 이벤트가 $exit_P?$로 다를 뿐 나머지 부분은 위의 식과 동일하다. trigger가 in 조건일 경우는 다음과 같이 표현된다.



ACSR에서는 조건이 존재하지 않으므로 이전 장에서 설명한 것처럼 조건을 이벤트로 바꾸어 준다. 따라서, 일반적인 이벤트 trigger와 다를 바 없다. 아래는 위 그림의 ACSR표현이다.

$$M \overset{def}{=} (t\_P?,1).(exit\_M!,1).N + \varnothing : M$$

trigger가 [*not in*(P)]일 경우의 ACSR표현은 아래와 같다.

$$M \overset{def}{=} (f\_P?,1).(exit\_M!,1).N + \varnothing : M$$

4.2.3 우선 순위 규칙

ACSR에서는 이벤트에 우선 순위가 부여되는데 이

우선 순위의 역할은 선택 상황에서 2개 이상의 이벤트가 동기화 되어야 할 때 그들 중 우선 순위가 가장 높은 이벤트가 선택된다. 우선 순위가 모두 동일하다면 선택될 가능성도 모두 동일한 것이다. Statechart 명세를 ACSR로 바꾸는 과정에서 다른 이벤트보다 높은 우선 순위를 가져야 하는 이벤트들이 있다. 명세에서 가장 높은 우선 순위를 가져야 하는 이벤트는 시간 지연 부분에 있는 이벤트들이다. 또한 시간 지연에서 이벤트를 받아들이는 쪽보다 이벤트를 방송하는 쪽이 더 높은 우선 순위를 가져야 한다. 두번째로 높은 우선 순위는 스테이트에 제어가 들어갈 때 발생하는 *enter_M*!이다. 선택 상황에서 이 이벤트가 선택되지 않고 다른 이벤트가 선택되면 제어가 지금 어느 프로세스에 있는지에 대한 정보가 상실된다. 세번째는 앞절에서 설명한 전이 변환 부분에서 *a and b*가 trigger일 경우에 앞의 이벤트보다 뒤의 이벤트가 높은 우선 순위를 가져야 한다.

4.2.4 시스템

ACSR로 전체 시스템을 명세하기 위해서는 다음 3가지 구성 요소가 서로 병렬적으로 돌아가야 한다. 첫번째는 시스템의 행위를 나타내는 프로세스로서 Statechart의 모든 스테이트와 전이를 표현한다. 두 번째는 스테이트들의 상태를 표현하는 프로세스로서 이것은 다음과 같이 정의된다.

$$P\_additional \overset{def}{=} (P\_M_1 test \,||\, \llcorner \,||\, P\_M_n test)$$

세번째는 시간 지연이다. 따라서 전체 시스템을 나타내는 프로세스는 다음과 같이 표현된다.

$$P\_M \overset{def}{=} (P\_M' \,||\, P\_additional \,||\, TD) \setminus F$$

위의 식에서 F는 동기화 되어야 하는 모든 이벤트들의 집합을 나타낸다.

4.3 최적화

앞에 절에서 Statechart 명세를 그것과 동일한 의미를 가지는 ACSR 프로세스로 변환하는 방법을 설명하였다. 그러나, 위의 방식을 그대로 적용하여 변환을 하면 변환 전과 변환 후의 시스템은 동일한 의미를 가지지만 ACSR 명세의 검증에는 전혀 사용되지 않는 많은 프로세스들이 생성된다. 이런 프로세스들은 단지 두 언어간에 의미만 동일하게 만들 뿐 명세를 검증할 때에는 오히려 상태 폭발 현상의 원인이 된다. 따라서, ACSR 명세에서 이런 사용되지 않는 프로세스들을 제거시키는 것이 명세의 검증에는 도움이 된다. 불필요한 프로세스는 주로 다음 3부분을 통해 제거할 수 있다. 첫째, 모든 프로세스에 제어가 들어가고 나올 때 *enter*!와 *exit*! 이벤트가 발생하게 되는데 이들 이벤트 중 상당 부분은 필요로 하는 프로세스가 없기 때문에 버려지게 된다. 따라서, 동기화 되지 않는 위의 2 종류의 이벤트는 프로세스에서 제거될 수 있다. 둘째, 스테이트의 상태를 알려주는 프로세스 들 중 상당 부분은 상태 정보가 불필요하기 때문에 제거될 수 있다. 즉, Statechart 명세에서 전이의 trigger에 [*in*(*S*)] 조건이 나올 수 있는데 이 표현에서 *S*에 포함되지 않는 스테이트는 프로세스로 전환될 때 상태를 알려주는 프로세스는 필요하지 않다. 세째, 두 프로세스의 등가를 검사할 때 두 프로세스에 공통적으로 속해있는 동일한 프로세스는 제거해도 등가 관계에 영향을 미치치 않으므로 제거될 수 있다.

5. 적용 예

5.1 신호등 시스템(Traffic Light Controller)

본 논문에서 소개하는 신호등 시스템은 Mead와 Conway에 의해 소개된 신호등 제어기[15]인데 그 구조는 다음과 같다. 차가 거의 다니지 않는 농장 도로와 고속화 도로가 교차하고 있다. 신호등이 교차로에서 교통을 통제하고 있으며 이 신호등 제어기는 고속화 도로 신호등이 녹색으로 남아 있는 시간을 최대화 할 수 있도록 구현한다. 제어기는 타이머와 센서, 농장 도로 신호 제어기와 고속화 도로 신호 제어기로 이루어져 있다. 신호등은 양방향에 있고 고속화 도로에는 계속적으로 차가 있으므로 고속화 도로 신호등에 우선 순위를 준다. 따라서 고속화 신호등은 농장 도로에 차가 다니지 않는다면 계속 녹색 등을 유지한다. 이 신호등에는 녹색, 황색, 적색이 있으며 녹색과 적색 등이 켜져 있는 시간이 황색 등이 켜져 있는 시간보다 길다. 또한 불이 켜지는 순서는 녹색, 황색, 적색의 순서이며 이때 황색 등이 켜져 있는 동안은 상대방 신호등의 변화가 없다. 여기서 이 신호등의 시간제어는 타이머가 한다. 타이머는 양쪽 방향의 신호등 제어기가 일정 시간이 지나면 녹색, 황색, 적색으로 변할 수 있도록 시간을 제어한다. 농장 길에 차가 나타났는지의 여부는 농장 길에 설치된 센서가 감지하게 된다. 차가 나타나면 센서는 차가 있다는 신호를 제어기들에 알려 준다. 이때 제어기들이 서로 작동을 하여 신호가 바뀌게 된다. 그러나 이때 조심할 것은 농장 신호등 제어기는 농장 길에 차가 다 지나가고 없다면 일정 시간과 관계없이 고속화 도로에 차가 지나갈 수 있도록 신호가 바뀌어야 한다. 또한 농장 길에 녹색 등이 켜져 있는 동안 더 차가 있더라도 일정시간 녹색을 유지했다면 고속화 도로에 우선권이 있으므로 고속화 도로에 차가 지나갈 수 있도록 신호가 바뀌어야 한다.

5.2 신호등 제어기의 명세 및 ACSR로의 변환

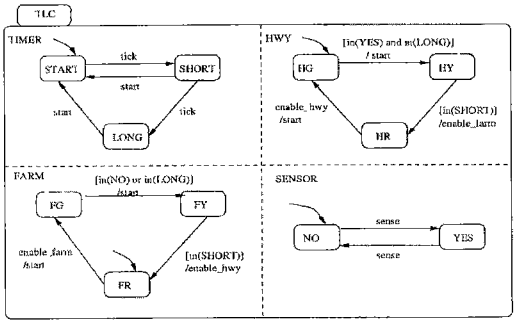


그림 1 신호등 제어기 명세

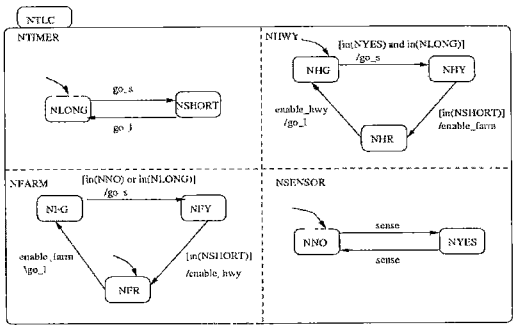


그림 3 수정된 신호등 제어기 명세

```
TIMER = START;
START = T_START;
T_START = (tick,1).SHORT + {}:T_START;
SHORT = ('enter_SHORT,1).T_SHORT;
T_SHORT = (tick,1).('exit_SHORT,1).LONG +
          (start,1).('exit_SHORT,1).START + {}:T_SHORT;
LONG = ('enter_LONG,1).T_LONG;
T_LONG = (start,1).('exit_LONG,1).START + {}:T_LONG;
```

그림 2 신호등 제어기의 ACSR 명세(타이머 부분)

```
NTIMER = NLONG;
NSHORT = ('enter_SHORT,1).NT_SHORT;
NT_SHORT = (go_L,1).('exit_SHORT,1).NLONG +
           {}:NT_SHORT;
NLONG = ('enter_LONG,1).NT_LONG;
NT_LONG = (go_S,1).('exit_LONG,1).NSHORT +
          {}:NT_LONG;
```

그림 4 수정된 신호등 제어기의 ACSR 명세(타이머 부분)

그림 1은 지금까지 설명한 신호등 제어기를 Statechart로 명세한 것이다. 4개의 스테이트가 동시에 돌아가고 있으며 sensor에서 감지한 *sense* 이벤트와 랜덤하게 생성되는 *tick* 이벤트를 외부 이벤트로 받아들여서 시스템이 반응한다. 고속화 도로 제어에 우선권이 있으며 농장도로 제어는 농장도로에 차가 있고 고속화 도로에 배당된 시간이 지나야 녹색 신호를 가질 수 있다. Timer는 3개의 스테이트를 가지고 있는데 START 스테이트는 timer의 초기 스테이트이고 SHORT와 LONG 스테이트는 황색불과 적색 불이 켜져있는 시간을 나타낸다.

그림 1의 명세를 앞에 절에서 소개한 규칙을 적용하여 ACSR로 표현하면 그림 2와 같이 된다.

5.3 수정된 신호등 제어기의 명세 및 ACSR로의 변환

신호등 제어기 명세에서 timer 부분을 유의해서 보아야 하는데 START 스테이트는 단지 timer를 초기화시켜 주는 역할만 수행하며 *tick* 이벤트는 시간의 진행을 나타내기 위해 필요한 이벤트이다. 이런 스테이트나 이벤트는 실제 신호등 시스템에서 반드시 필요한 요소들이 아니다. 따라서 그림 1의 timer 명세에서 START 스테이트와 *tick* 이벤트를 제거하고 timer를 다시 명세하면 그림 3과 같이 좀 더 간결한 신호등 제어기가 된다. 수정된 Statechart 명세를 다시 ACSR 명세로 고치면 그림 4와 같이 된다. Timer 부분에서 초기화를 위한 스테이트가 없어졌고 고속화 도로 제어와 농장 도로 제어에서 액션 부분이 수정되었다.

5.4 두 명세의 등가 검사

앞 절의 명세 중 첫번째 신호등을 지금까지 사용하여 왔고 이제 Timer 부분이 수정된 명세의 신호등으로 교체를 검토하고 있다고 가정한다면 수정된 명세가 과연 지금까지 사용해왔던 신호등과 동일하게 행동할 것인가가 중요한 관심사가 된다. Statechart 명세만을 가지고는 두 시스템이 동일하게 행동하는지 검증하기는 매우 어렵다. 그러나 ACSR 명세는 VERSA[16]라는 tool을 사용해서 두 시스템의 행동이 동일한지를 검증할 수 있다. 즉, 두 명세간에 바이 시뮬레이션(bisimulation) 관

```
:-) NTLC == TLC?
    ufi failed--following pair could not be matched:

<(NTLC')\{enable_farm,enable_hwy,go_S,go_L},(TLC')
                \{start,enable_farm,enable_hwy,tick}>
                --following pair was matched:
      <NTLC,TLC>
    false (by prioritized strong equivalence)
    true (by prioritized weak equivalence)
:-)
```

그림 5 검증 결과

계가 성립하는지 확인함으로써 등가 검사를 실시할 수 있다. 그림 5는 실제 위의 두 ACSR 명세를 VERSA를 사용하여 등가 검사를 실시한 결과이다.

그림 5의 결과에서 나타난 것처럼 두 시스템은 약한 바이시뮬레이션(weak equivalence)이 성립함을 알 수 있다. 다시 말하면, 간한 등가(strong equivalence)가 성립하지 않는다는 것은 내부적으로는 다르게 작동한다는 것을 의미하고 약한 등가(weak equivalence)가 성립한다는 것은 외부적으로 나타나는 행위는 동일하다는 것을 의미한다. 따라서 위의 두 시스템은 내부적으로는 다르지만 겉으로 보기에는 동일한 행위를 한다. 즉, 수정된 명세의 신호등으로 교체를 해도 전혀 문제가 없다는 것을 확인할 수 있다.

6. 결 론

Statechart는 사용이 편리하고 표현력이 강한 명세 언어이지만 명세의 정확성을 검증하는 능력에는 많은 한계를 가져왔다. 따라서, Statechart 명세의 정형검증에 대하여 많은 연구가 이루어지고 있다. 대표적인 예로 STATEMATE를 이용해서 설계한 Statechart명세를 정형검증 도구인 SPIN[17]이나 SMV[18]의 입력 언어 형태로 변환시켜 모델 체킹을 수행하는 방법이다. 이 방법은 Statechart의 명세를 상태 기계(state machine)로 변환시켜 그 동작을 검사하는 방법이다. 이 방법은 모델 체킹의 방법을 따르기 때문에 사용자가 이해하기 쉽고 검증이 간단하지만, Statechart 명세의 모든 부분을 검증할 수 없다. 때문에 Statechart의 많은 부분에 제약을 가하고 EHA(Extended Hierarchical Automata)[19]의 형태로 한번 더 변환시키고 그 오토마타를 분석하여 모델 체킹 도구의 입력으로 변환시켜야 한다. 그리고 SPIN[17]과 SMV[18]는 시간개념이 명확하지 않으므로 Statechart에서의 timeout과 같은 condition과 서로 다른 레벨간의 전이인 interlevel transition, 그리고 action의 assignment와 같은 많은 부분을 없애야 하는 단점이 있다. 그리고 또 다른 분야가 프로세스 알제브라로 바꾸어서 등가 검사를 실시하는 것이다. 본 논문에서는 Statechart 명세를 프로세스 알제브라의 일종인 ACSR로 바꾸어서 명세들간의 등가 관계를 검증할 수 있는 방법을 제시하였다. 그 방법은 다음 3가지 단계로 구분할 수 있는데 먼저 Statechart 명세를 동일한 의미를 가진 ACSR 명세로 바꾸고, 이 ACSR 명세를 논문에서 설명한 3가지 방법을 사용하여 최적화한 다음 VERSA를 통해 등가관계를 검사하면 된다. 이렇게 함으로써 두개의 Statechart명세가 서로 등가인지 간접적으로 확인할 수 있다. 하지만 본 논문에 설명한 ACSR로의 변환규칙은 전체 Statechart 문법에 적용되지 못하고 있다. 그 중 대표적인 것으로는 history 스테이트, 조건 변수, value passing등을 들 수 있는데 이 부분의 변환에 대해 더 많은 연구가 필요할 것이다.

참 고 문 헌

[1] 이희웅, 최진영, Virtual Prototyping이란?, 전자공학회지 25권 2호, 1998.

[2] David Harel, Statechart: A Visual Formalism For Complex Systems, Science of Computer Programming, 1987.

[3] Erich Mikk, Yassine Lakhnech and Michael Siegel, Hierarchical automata as model for statechart.

[4] Nancy Day, A Model Checker for Statecharts, 1993.

[5] W.Chan, et. al., Model Checking Large Software Specifications, IEEE Transaction on Software Engineering, vol. 24, no.7, pp.498-519, July 1998.

[6] Andrew C. Uselton and Scott A. smolka, A Process Algebraic Semantics for Statecharts via State Refinement, 1994.

[7] Andrew C. Uselton and Scott A. Smolka, A Compositional Semanitcs for Statecharts using Labeled Transition Systems, LNCS 836, Springer, 1994.

[8] Andrea Maggiolo-Schettini, A. Peron and S. Tini, Equivalences of Statecharts, In Proc. of CONCUR 96. LNCS 1119, 1996.

[9] Jin-Young Choi and Insup Lee, A Process Algebraic Method for the Specification and Analysis of Real-Time Systems, Formal Methods for Real-Time Computing, 1996.

[10] Jin-Young Choi and Inhye Kang, Translation of Modechart Specification to Algebra of Communicating Shared Resources, Proceedings of the first International Workshop on Real-Time Computing Systems and Applications, 1994.

[11] David Harel and Michal Politi, Modeling Reactive Systems with Statecharts, McGraw-Hill, 1998.

[12] R. Milner, Communication and Concurrency, Prentice-Hall, 1989.

[13] I. Lee, H. Ben-Abdallah, J. Y. Choi, "A Process Algebraic Method for the Specification and Analysis of Real-Time Systems," Formal Methods for Real-Time Computing, edited by C. Heitmeyer and D. Mandrioli, WILEY, 1996.

[14] David Harel and Amnon Naamad, The Statemate Semantics of Statecharts, ACM Trans. Soft. Method, 1996.

[15] C. Mead and L. Conway, Introduction to VLSI Systems, Addison-Wesley

[16] Duncan Clarke, VERSA: Verification, Execution and Rewrite System for ACSR, Real-Time Group Report, 1998.
[17] Gerald J. Holzmann, "The Model Checker SPIN?," IEEE Transactions on Software Engineering, VOL. 23, NO 5, pp279 - 295, MAY 1997.
[18] Kenneth L. McMillan, SYMBOLIC MODEL CHECKING, Kluwer Academic Publisher 1993.
[19] E. Mikk, Y. Lakhnech and M. Siegel, Hierarchical automata as model for statechart.

부록 A. 신호등 제어기의 ACSR 명세

```
TLC = (TLC' || P_additional || TD)\{start, enable_
      farm, enable_hwy, f_YES, t_YES,
      f_NO, t_NO, f_LONG, t_LONG, f_SHORT, t_
      SHORT, enter_NO, exit_NO,
      enter_YES, exit_YES, enter_LONG, exit_LONG,
      enter_SHORT,
      exit_SHORT,enter_START, exit_START, tick};
TD = (TSTART || ENABLE_FARM || ENABLE_
      HWY || F_YES || T_YES || F_NO || T_NO ||
      F_LONG || T_LONG || F_SHORT || T_SHORT
      || ENTER_NO || EXIT_NO ||
      ENTER_YES || EXIT_YES || ENTER_LONG ||
      EXIT_LONG ||NENTER_SHORT
      || EXIT_SHORT || TICK);
TSTART = (start,4).{}:rec X.(('start,5).X + TSTART)+
      {}:TSTART;
ENABLE_FARM = (enable_farm,4).{}:rec
      X.(('enable_farm,5).X +
                  ENABLE_FARM) +
      {}:ENABLE_FARM;
ENABLE_HWY = (enable_hwy,4).{}:rec
      X.(('enable_hwy,5).X +
            ENABLE_HWY) + {}:ENABLE_HWY;
F_YES = (f_YES,4).{}:rec X.(('f_YES,5).X + F_YES)
      + {}:F_YES;
T_YES = (t_YES,4).{}:rec X.(('t_YES,5).X + T_YES)
      + {}:T_YES;
F_NO = (f_NO,4).{}:rec X.(('f_NO,5).X + F_NO) +
      {}:F_NO;
T_NO = (t_NO,4).{}:rec X.(('t_NO,5).X + T_NO) +
      {}:T_NO;
F_LONG = (f_LONG,4).{}:rec X.(('f_LONG,5).X +
      F_LONG) + {}:F_LONG;
T_LONG = (t_LONG,4).{}:rec X.(('t_LONG,5).X +
      T_LONG) + {}:T_LONG;
F_SHORT = (f_SHORT,4).{}:rec X.(('f_SHORT,5).X +
      F_SHORT) + {}:F_SHORT;
T_SHORT = (t_SHORT,4).{}:rec X.(('t_SHORT,5).X
      + T_SHORT) + {}:T_SHORT;
ENTER_NO = (enter_NO,4).{}:rec X.(('enter_NO,5).X
      + ENTER_NO) +
            {}:ENTER_NO;
EXIT_NO = (exit_NO,4).{}:rec X.(('exit_NO,5).X +
      EXIT_NO) + {}:EXIT_NO;
ENTER_YES = (enter_YES,4).{}:rec
      X.(('enter_YES,5).X + ENTER_YES) +
            {}:ENTER_YES;
EXIT_YES = (exit_YES,4).{}:rec X.(('exit_YES,5).X +
      EXIT_YES) +
            {}:EXIT_YES;
ENTER_LONG = (enter_LONG,4).{}:rec
      X.(('enter_LONG,5).X + ENTER_LONG) +
            {}:ENTER_LONG;
EXIT_LONG = (exit_LONG,4).{}:rec
      X.(('exit_LONG,5).X + EXIT_LONG) +
            {}:EXIT_LONG;
ENTER_SHORT = (enter_SHORT,4).{}:rec
      X.(('enter_SHORT,5).X + ENTER_SHORT) +
            {}:ENTER_YES;
EXIT_SHORT = (exit_SHORT,4).{}:rec
      X.(('exit_SHORT,5).X + EXIT_SHORT) +
            {}:EXIT_SHORT;
TICK = (tick,4).{}:rec X.(('tick,5).X + TICK) +
      {}:TICK;

P_additional = ( P_NO || P_YES || P_LONG ||
      P_SHORT);

P_NO = P_NOf;
P_NOf = ('f_NO,1).P_NOf + ( enter_NO,1).P_NOt +
      {}:P_NOf;
P_NOt = ('t_NO,1).P_NOt + ( exit_NO,1).P_NOf +
      {}:P_NOt;
P_YES = P_YESf;
P_YESf = ('f_YES,1).P_YESf + (
      enter_YES,1).P_YESt + {}:P_YESf;
P_YESt = ('t_YES,1).P_YESt + ( exit_YES,1).P_YESf
```

```
    + {}:P_YESt;
P_LONG = P_LONGf;
P_LONGf = ('f_LONG,1).P_LONGf + (
    enter_LONG,1).P_LONGt + {}:P_LONGf;
P_LONGt = ('t_LONG,1).P_LONGt + (
    exit_LONG,1).P_LONGf + {}:P_LONGt;
P_SHORT = P_SHORTf;
P_SHORTf = ('f_SHORT,1).P_SHORTf + (
    enter_SHORT,1).P_SHORTt + {}:P_SHORTf;
P_SHORTt = ('t_SHORT,1).P_SHORTt + (
    exit_SHORT,1).P_SHORTf + {}:P_SHORTt;

TLC' = ( TIMER || SENSOR || HWY || FARM ||
    T_G) ;
T_G = ('tick,1).T_G + ('tick,1).{}:T_G + {}:T_G;
TIMER = START;
START = ('enter_START,3).T_START;
T_START = (tick,1).('exit_START,1).SHORT +
    {}:T_START;
SHORT = ('enter_SHORT,3).T_SHORT;
T_SHORT = (tick,1).('exit_SHORT,1).LONG +
        (start,1).('exit_SHORT,1).START +
    {}:T_SHORT;
LONG = ('enter_LONG,3).T_LONG;
T_LONG = (start,1).('exit_LONG,1).START +
    {}:T_LONG;
SENSOR = NO;
NO = ('enter_NO,3).T_NO;
T_NO = (sense,1).('exit_NO,1).YES + {}:T_NO;
YES = ('enter_YES,3).T_YES;
T_YES = (sense,1).('exit_YES,1).NO + {}:T_YES;
HWY = HG;
HG = (t_YES,1).((t_LONG,2).('start,1).HY + HG)
    + (t_LONG,1).((t_YES,2).('start,1).HY + HG) +
    {}:HG;
HY = (t_SHORT,1).('enable_farm,1).HR + {}:HY;
HR = (enable_hwy,1).('start,1).HG + {}:HR;
FARM = FR;
FG = (t_NO,1).('start,1).FY + (t_LONG,1).('start,1).FY
    + {}:FG;
FY = (t_SHORT,1).('enable_hwy,1).FR + {}:FY;
FR = (enable_farm,1).('start,1).FG + {}:FR;
```

부록 B. 수정된 신호등 제어기의 ACSR 명세

```
NTLC = (NTLC' || NP_additional|| NTD)\{
    enable_farm, enable_hwy, f_YES, t_YES,
    f_NO, t_NO, f_LONG, t_LONG, f_SHORT,
    t_SHORT, enter_NO, exit_NO,
    enter_YES, exit_YES, enter_LONG, exit_LONG,
    enter_SHORT, exit_SHORT,go_S,go_L};
NTD = (NENABLE_FARM || NENABLE_HWY ||
    NF_YES || NT_YES || NF_NO || NT_NO ||
    NF_LONG || NT_LONG || NF_SHORT ||
    NT_SHORT || NENTER_NO || NEXIT_NO ||
    NENTER_YES || NEXIT_YES ||
    NENTER_LONG || NEXIT_LONG ||
    NENTER_SHORT
    || NEXIT_SHORT || NGO_S || NGO_L);
NENABLE_FARM = (enable_farm,4).{}:rec
    X.(('enable_farm,5).X +
                NENABLE_FARM) +
    {}:NENABLE_FARM;
NENABLE_HWY = (enable_hwy,4).{}:rec
    X.(('enable_hwy,5).X +
                NENABLE_HWY) +
    {}:NENABLE_HWY;
NF_YES = (f_YES,4).{}:rec X.(('f_YES,5).X +
    NF_YES) + {}:NF_YES;
NT_YES = (t_YES,4).{}:rec X.(('t_YES,5).X +
    NT_YES) + {}:NT_YES;
NF_NO = (f_NO,4).{}:rec X.(('f_NO,5).X + NF_NO) +
    {}:NF_NO;
NT_NO = (t_NO,4).{}:rec X.(('t_NO,5).X + NT_NO) +
    {}:NT_NO;
NF_LONG = (f_LONG,4).{}:rec X.(('f_LONG,5).X +
    NF_LONG) + {}:NF_LONG;
NT_LONG = (t_LONG,4).{}:rec X.(('t_LONG,5).X +
    NT_LONG) + {}:NT_LONG;
NF_SHORT = (f_SHORT,4).{}:rec X.(('f_SHORT,5).X
    + NF_SHORT) + {}:NF_SHORT;
NT_SHORT = (t_SHORT,4).{}:rec X.(('t_SHORT,5).X
    + NT_SHORT) + {}:NT_SHORT;
NENTER_NO = (enter_NO,4).{}:rec
    X.(('enter_NO,5).X + NENTER_NO) +
    {}:NENTER_NO;
NEXIT_NO = (exit_NO,4).{}:rec X.(('exit_NO,5).X+
    NEXIT_NO) + {}:NEXIT_NO;
NENTER_YES = (enter_YES,4).{}:rec X.(('enter_YES,
```

```
5).X + NENTER_YES) + {}:NENTER_YES;
NEXIT_YES = (exit_YES,4).{}:rec X.(('exit_YES,5).X
    + NEXIT_YES) + {}:NEXIT_YES;
NENTER_LONG = (enter_LONG,4).{}:rec X.(('enter_
    LONG,5).X + NENTER_LONG) + {}:NENTER_
    LONG;
NEXIT_LONG = (exit_LONG,4).{}:rec X.(('exit_
    LONG,5).X + NEXIT_LONG) + {}:NEXIT_
    LONG;
NENTER_SHORT=(enter_SHORT,4).{}:rec
    X.(('enter_SHORT,5).X + NENTER_SHORT) +
    {}:NENTER_YES;
NEXIT_SHORT=(exit_SHORT,4).{}:rec
    X.(('exit_SHORT,5).X + NEXIT_SHORT) +
    {}:NEXIT_SHORT;
NGO_S = (go_S,4).{}:rec X.(('go_S,5).X + NGO_S) +
    {}:NGO_S;
NGO_L = (go_L,4).{}:rec X.(('go_L,5).X + NGO_L) +
    {}:NGO_L;
NP_additional = ( NP_NO || NP_YES || NP_LONG ||
    NP_SHORT);
NP_NO = NP_NOf;
NP_NOf = ('f_NO,1).NP_NOf + ( enter_NO,1).NP_NOt
    + {}:NP_NOf;
NP_NOt = ('t_NO,1).NP_NOt + ( exit_NO,1).NP_NOf
    + {}:NP_NOt;
NP_YES =NP_YESf;
NP_YESf = ('f_YES,1).NP_YESf + ( enter_YES,
    1).NP_YESt + {}:NP_YESf;
NP_YESt = ('t_YES,1).NP_YESt + ( exit_YES,1).
    NP_YESf + {}:NP_YESt;
NP_LONG = NP_LONGf;
NP_LONGf = ('f_LONG,1).NP_LONGf + ( enter_
    LONG,1).NP_LONGt + {}:NP_LONGf;
NP_LONGt = ('t_LONG,1).NP_LONGt + ( exit_
    LONG,1).NP_LONGf + {}:NP_LONGt;
NP_SHORT = NP_SHORTf;
NP_SHORTf = ('f_SHORT,1).NP_SHORTf + ( enter_
    SHORT,1).NP_SHORTt + {}:NP_SHORTf;
NP_SHORTt = ('t_SHORT,1).NP_SHORTt + ( exit_
    SHORT,1).NP_SHORTf + {}:NP_SHORTt;
NTLC' = ( NTIMER || NSENSOR || NHWY ||
    NFARM ) ;
NTIMER = NLONG;
NSHORT = ('enter_SHORT,3).NT_SHORT;
NT_SHORT = (go_L,1).('exit_SHORT,1).NLONG +
    {}:NT_SHORT;
NLONG = ('enter_LONG,3).NT_LONG;
NT_LONG=(go_S,1).('exit_LONG,1).NSHORT
    +{}:NT_LONG;
NSENSOR = NNO;
NNO = ('enter_NO,3).NT_NO;
NT_NO = (sense,1).('exit_NO,1).NYES + {}:NT_NO;
NYES = ('enter_YES,3).NT_YES;
NT_YES = (sense,1).('exit_YES,1).NNO + {}:NT_YES;
NHWY = NHG;
NHG = (t_YES,1).((t_LONG,2).('go_S,1).NHY + NHG)
    + (t_LONG,1).((t_YES,2).('go_S,1).NHY + NHG)
    + {}:NHG;
NHY = (t_SHORT,1).('enable_farm,1).NHR + {}:NHY;
NHR = (enable_hwy,1).('go_L,1).NHG + {}:NHR;
NFARM = NFR;
NFG = (t_NO,1).('go_S,1).NFY + (t_LONG,1).('go_S,
    1).NFY + {}:NFG;
NFY = (t_SHORT,1).('enable_hwy,1).NFR + {}:NFY;
NFR = (enable_farm,1).('go_L,1).NFG + {}:NFR;
```


박 명 환

1994년 공군사관학교 전산과 졸업. 2000년 고려대학교 컴퓨터학과 대학원(석사) 졸업. 관심분야는 Real time system, Formal Specification, Formal verification.

방 기 석

1997년 고려대학교 정보공학과 졸업. 2000년 고려대학교 컴퓨터학과 석사. 2000년 ~ 현재 고려대학교 컴퓨터학과 박사과정. 관심분야는 정형기법, 실시간 시스템, 운영체제, 소프트웨어 공학, 네트워크 프로토콜, 보안 프로토콜 등.

최 진 영

1982년 서울대학교 컴퓨터공학과 졸업. 1986년 Drexel University Dept. of Mathematics and Computer Science석사. 1993년 University of Pennsylvania Dept. of Computer and Information Science 박사. 1993년 ~ 1996년 Research Associate, University of Pennsylvania. 1994년 ~ 현재 고려대학교 컴퓨터학과 부교수. 관심분야는 컴퓨터 이론, 정형기법(정형 명세, Formal verification), 실시간 시스템, 분산 프로그래밍 언어, 소프트웨어 공학.

이 정 아

1982년 서울대학교 컴퓨터공학과(학사). 1985년 미국 인디에나주립대학 컴퓨터학과(석사). 1990년 미국UCLA 컴퓨터공학과(박사). 1990년 ~ 1995년 미국 휴스턴 주립대학 전기전산공학과(조교수). 1993년 ~ 1994년 미국 국립초전도가속기연구소(객원연구원). 1995년 ~ 현재 조선대학교 컴퓨터공학부(부교수). 관심분야는 Computer Arithmetic, Application-Specific Processor Design, (Re)Configurable Computing.

한 상 용

1975년 서울대학교 공과대학 졸업(학사). 1984년 University of Minnesota 전산학과 졸업(박사). 1984년 6월 ~ 1995년 2월 IBM 연구소 책임 연구원. 1996년 1월 ~ 1996년 8월 미국 AMD사 consultant. 1995년 3월 ~ 현재 중앙대학교 컴퓨터공학과 교수. 관심분야는 VLSI CAD, Virtual Engineering, 컴퓨터 구조 등.