태스크 상호작용 테스팅을 위한 MSC 명세로부터의 전체 유한 상태 기계 생성

(Construction of Global Finite State Machine from Message Sequence Charts for Testing Task Interactions)

이 남 희 † 김 태 효 † 차 성 덕 ††
(Nam Hee Lee) (Tai Hyo Kim) (Sung Deok Cha)
신 석 종 ††† 홍 인 표 ††† 박 기 웅 †††
(Seog Jong Shin) (In Pyo Hong) (Ki Wung Park)

요 약 MSC는 통신 소프트웨어에서의 병행 수행 태스크들 사이의 상호작용을 기술하기 위해서 많이 사용되어져 왔다. 요구사항 단계에서 검증된 MSC 명세는 상태 기반의 설계 모델을 합성하는데 사용될 수 있을 뿐만 아니라, 테스트 시퀀스 생성에 사용될 수도 있다. 지금까지는 MSC에 기술된 각 이벤트의 위치 정보 만을 이용하여 전체 상태 그래프를 생성함으로써 검증을 수행하였다. 본 논문에서는 MSC의 조건문을 시나리오 활성화 조건과 상태 변경을 기술할 수 있도록 확장하고, 이를 이용하여 전체 상태 그래프를 생성함으로써 테스트 시퀀스 생성에 이용한다. 생성된 전체 상태 그래프인 GFSM은 시스템의 의미있는 상태 정보와 가능한 메시지 시퀀스만을 가지게 된다. 생성된 GFSM에 기존의 FSM 기반의 테스팅 기법을 적용하여 테스트 시퀀스를 추출할 수 있다.

Abstract Message Sequence Charts (MSC) has been used to describe the interactions of numerous concurrent tasks in telecommunication software. After the MSC specification is verified in requirement analysis phase, it can be used not only to synthesize state-based design models, but also to generate test sequences. Until now, the verification is accomplished by generating global state transition graph using the location information only. In this paper, we extend the condition statement of MSC to describe the activation condition of scenarios and the change of state variables, and propose an approach to construct global finite state machine (GFSM) using this information. The GFSM only includes feasible states and transitions of the system. We can generate the test sequences using the existing FSM-based test sequence generation technology.

1. 서 론

근래에 들어 프로세서나 메모리 등의 기술이 급속히 발전하면서 휴대폰이나 디지털 TV 등 소형 가전기기의 다양한 서비스 제공에서부터 자동차나 항공기 등의 제어에 이르기까지 대규모의 복잡한 내장 소프트웨어가 구현되어 사용되고 있다. 소형 가전기기와 같은 제품의 경우에는 생명 주기가 매우 짧으면서도 높은 수준의 품질을 요구하고 있다. 하지만, 이러한 제품을 구성하는 내장 소프트웨어에 대한 적절한 테스팅 방법과 자동화된 지원 도구의 부족으로 제품 개발과 품질 보증에 많은 어려움이 있다. 특히, 다수의 태스크들로 구현된 내장 소프트웨어의 경우 태스크들 사이의 상호작용에 대한 적절성을 테스팅할 수 있는 기법의 개발이 필요하다.

태스크들 사이의 상호작용에 관한 기존의 테스팅 방법에서는 상태 순회(state exploration)를 이용하는 모델

· 이 연구는 첨단정보기술연구센터(AITrc)의 지원을 부분적으로 받았음.
† 비 회 원 : 한국과학기술원 전산학과
nhlee@salmosa.kaist.ac.kr
taihyo@salmosa.kaist.ac.kr
†† 종신회원 : 한국과학기술원 전산학과 교수
cha@salmosa.kaist.ac.kr
††† 비 회 원 : (주) 삼성전자 CTO전략실 소프트웨어 센타 연구원
sjshin@swc.sec.samsung.co.kr
iphong@swc.sec.samsung.co.kr
kwpark@swc.sec.samsung.co.kr
논문접수 : 2000년 12월 21일
심사완료 : 2001년 7월 23일

검사에서와 같이 각 태스크들의 상호 작용 행위를 유한 상태 기계(Finite State Machine)로 작성하고, 시스템의 전체 행위를 구하기 위하여 FSM들을 합성하였다[1,2]. 이때, 상태 폭발(state explosion) 문제가 발생할 수 있을 뿐만 아니라, FSM 기반의 명세는 새로운 상호작용의 추가에 대해서 유연하게 대처할 수 없다는 단점이 있다. 즉, 새로운 상호작용이 추가되면 관련된 FSM들의 상태와 전이를 모두 파악하여 수정해야 되고, 때로는 모든 FSM 명세를 재작성 해야 될 수도 있다.

본 논문에서는 내장 소프트웨어의 각 외부 입력 이벤트에 대하여 태스크들 사이의 상호작용 시나리오를 명세하고, 이를 합성하여 전체 시스템의 행위를 구한다. 시나리오 기반의 명세를 사용함으로써 보다 읽기 쉽고, 이해하기 쉬우면서 점증적인 명세가 가능하다는 장점을 갖는다. 본 논문에서는 시나리오 명세 언어로 Message Sequence Charts(MSC)를 사용한다. 일반적인 소프트웨어에서의 통합 테스팅 단계에서는 단위 테스팅이 끝난 모듈들 사이의 인터페이스가 적절하게 구현되어 사용되고 있는 지에 대한 테스팅을 수행하게 된다. 하지만, 내장 소프트웨어의 통합 테스팅 단계에서는 인터페이스 적절성 이외에 일련의 상호작용에 의한 각 태스크의 상태 변화의 적절성을 추가적으로 테스팅하게 된다. 따라서, 본 논문에서는 MSC를 이용한 시나리오 명세에 상호작용을 수행 가능하게 하는 활성화 조건에 대한 기술과 상호작용의 수행에 의한 시스템의 상태 변화를 기술하도록 한다.

Message Sequence Charts(MSC)는 병행 시스템을 구성하는 태스크들 사이의 상호작용을 기술하는 정형 언어이다[3]. MSC의 구문은 크게 basic MSC와 high-level MSC로 나눌 수 있는데, basic MSC(bMSC)는 하나의 시나리오를 표현하기 위해 사용되고, high-level MSC(hMSC)는 bMSC들 사이의 관계를 표현하기 위해 사용된다. MSC의 정형적 시맨틱스는 프로세스 대수를 이용하는 방법[4], Message Flow Graph에 바탕을 둔 방법[5,6], Petri Net을 이용한 방법[7] 등이 있다. 이러한 시맨틱스들을 이용하면 MSC 명세에 대한 전체 상태 그래프를 구할 수 있다. 하지만, MSC로부터 전체 상태 그래프를 생성하는 기존의 연구들에서는 메시지나 선택적 수행과 같은 일부 구문만을 이용하여 기술된 MSC를 대상으로하고 있다.

검증을 위한 MSC 명세에서의 전체 상태 그래프 생성에 관한 기존의 연구에서는 하나의 메시지 전송에 의한 상호작용을 전체 시스템의 상태 변화로 본다[5,6]. 따라서, 메시지의 송신 이벤트와 수신 이벤트를 다른 것으로 해석하는 비동기적인 방법을 사용하고 있기 때문에, 상태 폭발 문제를 발생할 수 있다. 하지만, 본 논문에서와 같이 각 태스크의 상태 정보가 기술된 MSC 명세에서는 전체 상태 그래프의 상태를 각 태스크들의 상태의 조합으로 구할 수 있고, 그들 사이의 전이는 일련의 상호작용이 된다. 따라서, 메시지 전송과 수신이 동시에 발생한다고 해석하는 동기적 시맨틱스를 적용해도 결과적인 시스템의 상태에는 영향을 미치지 않게 되어, 비동기적 시맨틱스를 적용하였을 경우보다 상태 폭발 문제를 감소할 수 있다.

본 논문에서는 MSC를 이용하여 태스크 상호작용을 명세하고, 이로부터 Global FSM(GFSM)을 구한다. 이때, 각 태스크의 상태정보를 함께 기술하도록 함으로써, GFSM 생성 시 발생할 수 있는 상태의 수를 줄이고자 한다. MSC'96에서는 condition을 단순히 조건 이름으로만 사용하였다. 하지만, 본 논문에서는 MSC 2000[8]에 제안된 condition에 상태 변수와 값을 가지도록 확장하여 시나리오의 활성화 조건과 상태 값 변경에 사용한다. GFSM의 상태는 사용된 상태 변수 값의 조합이 되고, 전이는 상태들 사이에 기술된 하나 이상의 메시지 시퀀스가 된다.

본 논문에서는 제안된 방법을 삼성전자(SEC)가 개발한 Digital TV (DTV) 소프트웨어에 적용하여 태스크들 사이의 상호 작용을 명세하고, GFSM을 생성한다. SEC DTV 소프트웨어는 병행적으로 동작하는 여러 태스크들과 그들 사이의 상호작용이 매우 복잡하게 이루어지는 시스템이다. 생성된 GFSM에 기존의 FSM 기반의 테스팅 방법들을 적용하여 태스크들 사이의 상호작용에 관한 테스트 시퀀스를 생성할 수 있다.

본 논문의 구성은 다음과 같다. 2장에서는 태스크들 사이의 상호작용을 기술하기 위한 MSC를 정의하고, 3장에서 MSC에서 GFSM을 생성하는 알고리즘을 기술한다. 4장에서는 SEC DTV 소프트웨어를 대상으로 하여 명세하고 GFSM을 생성한 결과를 설명한다. 그리고, 5장에서는 MSC로 부터 전체 상태 그래프를 구하는 관련 연구들에 대해서 살펴보고, 제안한 방법과 비교한다. 마지막으로 5장에서 결론 및 향후 연구 방향에 대해서 기술한다.

2. MSC 명세언어

본 논문에서는 ITU-T Z.120에 정의된 MSC 표준에서 제공하는 구문 중에서 'Instance', 'Message', 'Reference', 'Alt Inline', 'Opt Inline', 'Loop Inline', 'Condition'만을 사용한다. 'Instance'는 명세 대상 태스

크를, 'Message'는 태스크들 사이의 메시지 송수신을 기술한다. 'Reference'는 다른 bMSC에 대한 참조를, 'Alt Inline'과 'Opt Inline'은 선택적으로 수행 가능한 시나리오를, 그리고 'Loop Inline'은 반복적으로 수행 가능한 시나리오를 기술한다.

'Condition'은 시나리오의 수행 여부을 결정하는 활성화 조건을 기술하거나, 메시지의 전송에 의한 시스템의 상태 변화를 기술한다. 이것은 GFSM 생성시 상태 정보를 구할 수 있도록 하기 위한 추가적인 정보로 사용된다. MSC'96의 확장된 버전인 MSC 2000에서는 활성화 조건을 'when'을 사용하여 표현하도록 하고, 기존 'Condition'은 상태 변수 변경의 의미로 사용한다[8]. 즉, 'Condition'이 'when variable = value'의 형태로 기술된 것은 시나리오 활성화 조건을 의미하는 것으로, 시나리오의 수행 중 'Condition'을 만났을 당시의 상태 변수 'variable'의 값과 'value'를 비교하여 같으면 계속 진행하고, 그렇지 않으면 해당 시나리오를 수행하지 않는다. 'Condition'이 'variable = value'의 형태로 기술된 것은 상태 변수의 변경을 의미한다. 즉, 시나리오의 수행 중에 이와 같은 'Condition'을 만나면 'variable'의 값을 'value'로 바꾸고, 계속해서 시나리오를 진행한다.

본 논문에서 사용되는 MSC 명세의 의미 해석을 위해서 몇 가지 가정을 한다. 첫째, 가장 상위에 기술된 bMSC가 반복적으로 수행됨으로써 시스템의 무한한 동작을 기술한다고 가정한다. MSC에서 시스템의 전체 행위는 hMSC를 이용하여 기술하지만, hMSC의 loop이나 alternative, parallel 모두 bMSC의 inline으로 표현 가능하므로 위와 같은 가정에 무리가 없다. 둘째, 'Alt Inline'의 각 아이템들에 대한 세부 사항은 'Reference'를 이용하여 기술하도록 하고, 이때 참조되는 bMSC는 활성화 조건으로 시작하도록 한다. 그리고, 'Alt Inline'을 구성하는 아이템은 완전(complete)하게 작성하도록 한다. 즉, 활성화 조건에 사용된 상태 변수의 모든 가능한 경우에 대하여 'Alt Inline'의 아이템을 기술하도록 한다. 셋째, 참조되는 bMSC는 상위 bMSC와 synchronous concatenation[9] 된다고 가정한다. 즉, 상위 bMSC의 모든 메시지 전송이 이루어진 후에만 하위 bMSC의 메시지 전송이 이루어질 수 있다. 이것은 참조되는 bMSC를 상위 bMSC에 합칠 때 발생할 수 있는 경우의 수를 제한한다. 마지막으로, 메시지 송수신은 동기적으로 이루어진다고 가정한다. 즉, 메시지의 송신과 수신은 항상 함께 이루어진다.

그림 1은 4개의 태스크 ('Instance')가 메시지를 주고받는 상호작용을 기술한 것으로, 'a' 메시지나 'b' 메시지는 서로 순서 관계를 정할 수 없다. 하지만, 'c' 메시지는 이들 보다 나중에 발생하여야 한다. 그것은 본 논문에서 동기적 메시지 전송을 가정하고 있기 때문이다. 즉, 'IUT1'에서 'a'의 수신이 'c'의 수신보다 먼저 일어나야하고, 메시지의 송수신은 동시에 일어나기 때문에 'a'의 송신이 'c'의 송신보다 먼저 발생하게 된다. 하지만, 만약에 비동기적 메시지 전송이라면, 'a'의 수신이 'c'의 수신보다 앞서지만, 'a'의 송신과 'c'의 송신사이에는 순서관계를 정할 수가 없다. 마지막으로, 'c' 메시지가 발생한 후에는 'ref1' 혹은 'ref2' bMSC가 수행 가능하다는 것을 'Alt Inline'으로 나타내고 있다.

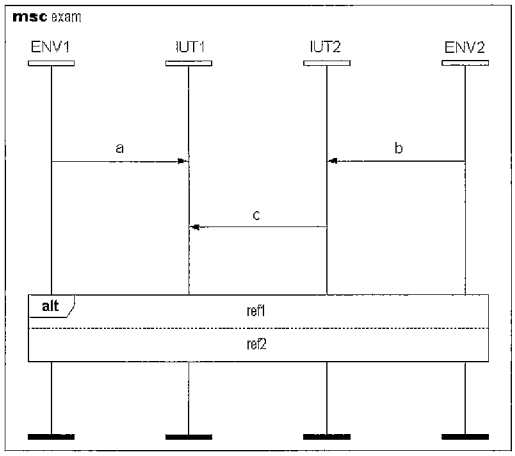


그림 1 MSC exam

그림 2와 3에는 그림 1에서 참조하는 두 개의 bMSCs를 기술한 것이다. 두 bMSCs는 'cond' 상태 변수의 값에 따라 선택적으로 수행 가능하다. 즉, 'cond'의 값이 '1'일 때에는 'ref1' bMSC가 수행되어 'd' 메시지가 전송되고, 'cond'의 값이 '2'일 때에는 'ref2' bMSC

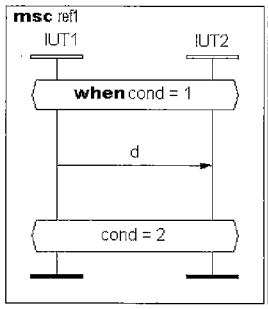


그림 2 ref1 시나리오

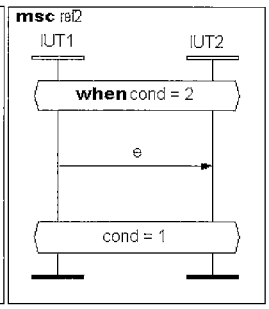


그림 3 ref2 시나리오

가 수행되어 'e' 메시지가 전송된다. 각 bMSCs의 수행 후에는 'cond' 상태 변수의 값을 '2' 또는 '1'로 변경한다. 이때, 본 논문에서는 'Alt Inline'의 활성화 조건은 모든 경우를 기술하도록 가정하고 있다. 따라서, 이 예제에서는 'cond'의 값은 '1'과 '2'만 가능한 것으로 가정하여야 한다.

시스템을 구성하는 전체 'Instance'의 집합을 P로, 이벤트의 집합을 E, 이름의 집합을 Σ로 표현한다. 여기서, 이벤트의 집합 $E = E^M \cup E^C \cup E^R \cup E^A \cup E^O \cup E^L$ 이다. 'Message' 이벤트의 집합을 나타내는 $E^M = E^{M_S} \cup E^{M_R}$로 구성되는데, E^{M_S}는 송신 메시지를, E^{M_R}은 수신 메시지를 나타낸다. 각 송신 메시지에 함수 $f: E^{M_S} \mapsto E^{M_R}$을 적용하면, 대응하는 수신 메시지를 구할 수 있다. 'Condition' 이벤트의 집합을 나타내는 $E^C = E^{C_A} \cup E^{C_S}$ 로 구성되는데, E^{C_A}는 시나리오 활성화 조건을 나타내는 이벤트이고, E^{C_S}는 상태 변수 변경을 나타내는 이벤트이다. 마지막으로, E^R과 E^A, E^O, E^L은 각각 'Reference'와 'Alt Inline', 'Opt Inline', 'Loop Inline'을 나타내는 이벤트이다.

그림 1, 2, 3의 bMSCs를 구성하는 'Instance'는 네개가 있고, 이벤트는 다섯개의 'Message' 이벤트와 네개의 'Condition' 이벤트, 하나의 'Alt Inline' 이벤트, 그리고 두개의 'Reference' 이벤트로 구성된다. 이때, 'a' 메시지 이벤트를 e_1이라 하면, 이는 다시 'a' 메시지를 송신하는 이벤트 e_1^s와 수신하는 이벤트 e_1^r로 나눌 수 있는데, $f(e_1^s) = e_1^r$ 의 관계가 있다. 나머지 메시지 이벤트의 경우에도 마찬가지로 적용된다.

이름의 집합 $\Sigma = \Sigma^R \cup \Sigma^P \cup \Sigma^M \cup \Sigma^C$로 구성된다. Σ^R은 bMSCs의 이름이고, Σ^P는 'Instance'의 이름을 나타낸다. Σ^M은 'Message'의 이름을 나타내고, Σ^C는 'Condition'의 이름을 나타낸다. 상태 변수의 집합을 V, 그리고 각 상태 변수 V의 영역을 D_V, 상태 값 할당 함수를 $I: V \mapsto D_V$, 상태 변수와 값 사이의 관계를 rel_{op}로 하면, 'Condition' 문은 $\{(v, rel_{op}, I(v)) \mid v \in V\}$로 기술된다. 이때, 각 상태 변수에 대한 조건은 곱 (and)으로 해석하고, 사용가능한 rel_{op}은 '='와 '!='로 한정한다. 함수 $g_v: \Sigma^C \mapsto V$는 'Condition' 이름 Σ^C로부터 상태 변수 V를, 함수 $g_a: \Sigma^C \mapsto D_V$는 값을 구한다. 마지막으로, $l: P \cup E \mapsto \Sigma$은 인스턴스와 이벤트의 이름을 구하는 함수이다.

그림 1, 2, 3에서 Σ^R = {'exam', 'ref1', 'ref2'}, Σ^P = {'ENV1', 'IUT1', 'IUT2', 'ENV2'}, Σ^M = {'a', 'b', 'c', 'd', 'e'}, 그리고 Σ^C = {'when cond = 1', 'cond = 2', 'when cond = 2', 'cond = 1'}로 구성된다. 여기서 상태 변수 V = {'cond'}이고, 'cond'이 가질 수 있는 값 D_{cond}는 1과 2이다. 'when cond = 1'에 상태 변수를 구하는 함수 g_v를 적용하면 'cond'을 구할 수 있고, 값을 구하는 함수 g_a를 적용하면 '1'을 구할 수 있다. 마지막으로, 각 이벤트나 인스턴스에 해당하는 이름을 구하기 위해서는 l을 적용하면 되는데, 예를들어 $l(e_1^s)$ = 'a'이다.

정의 1 (MSC 구조)

전체 MSC 명세의 구조를 나타내는 $MSC = (S, s_0, \Sigma^R, l_r, S_r)$로 정의할 수 있고, 각각의 의미는 다음과 같다.

1) S는 bMSC (시나리오)들의 집합이다.

2) $s_0 \in S$는 최상위 시나리오를 나타내는 bMSC이다.

3) Σ^R은 bMSC에 관한 이름의 집합이다.

4) $l_r: S \mapsto \Sigma_r$은 각 bMSC에 이름을 할당하는 함수이다.

5) $S_r = \{(s_1, s_2) \mid s_1, s_2 \in S\}$는 bMSC들 간의 참조 관계를 나타낸다.

그림 1,2,3에는 세 개의 시나리오 s_0, s_1, s_2가 존재하고, $l_r(s_0)$ = 'exam', $l_r(s_1)$ = 'ref1', $l_r(s_2)$ = 'ref2'이다. 그리고, 시나리오의 참조 관계 S_r = $\{(s_0, s_1), (s_0, s_2)\}$으로 구성된다. 가정에 의해서 s_0가 반복적으로 수행되면서 상호작용이 일어난다. 예를들어, s_0의 시나리오에 의해서 'a';'b';'c' 또는 'b';'a';'c'의 상호작용이 일어나고, 'cond'의 초기 값을 '1'로 가정하면 s_1 시나리오가 수행 가능하다. 본 논문에서는 synchronous concatenation 을 가정하고 있으므로, 'ref1' 시나리오의 수행은 항상 'exam' 시나리오에서 참조하는 시점에만 수행된다. 따라서, 'c' 이후에 'd'가 발생하고, 'cond'의 값은 '2'로 변경한다. 그리고 나서, 다시 s_0의 'a';'b';'c' 또는 'b';'a';'c'가 수행된다. 이번에는 'cond'의 값이 '2'이므로 s_2의 'e'를 수행하고 'cond'의 값을 '1'로 변경한다. 그림 1,2,3은 이러한 시퀀스가 무한히 반복되는 것을 나타내고 있다. 다음은 MSC를 구성하는 bMSC를 정형화한 것이다.

정의 2 (단위 시나리오)

각 단위 시나리오를 나타내는 $S_p = (P_s, E_s, \Sigma_s, p_s, l_s, <_s)$, $S_p \in S$로 정의할 수 있고, 각각의 의미는 다음과 같다.

1) $P_s \subseteq P$는 S_p를 구성하는 인스턴스의 집합이다.

2) $E_s \subseteq E$는 S_p를 구성하는 이벤트의 집합이다.

3) $\Sigma_s \subseteq \Sigma$는 S_p를 구성하는 이름의 집합이다.

4) $p_s : E_s \mapsto 2^{P_s}$는 각 이벤트를 인스턴스에 할당하는 함수이다. 메시지의 경우에는 하나의 인스턴스에만 할당된다. 하지만, 'Condition', 'Reference', 'Alt Inline', 'Opt Inline', 'Loop Inline'의 경우에는 모든 인스턴스 P_s에 할당되는 것으로 가정한다.

5) $l_s : (P_s \cup E_s) \mapsto \Sigma_s$는 각 인스턴스와 이벤트에 이름을 할당하는 함수이다. 이때, l_s는 해당하는 아이템의 이름에만 할당된다고 가정한다. 즉, 인스턴스에 메시지의 이름을 할당하는 경우는 없다.

6) $<_s : \bigcup <_{P_s} \cup \{(m_s, f(m_s)) \mid m_s \in E_s^{M_s}\}$는 이벤트들 간의 순서를 나타낸다. 이때, $<_{P_s} = \{(e_1, e_2) \mid e_1, e_2 \in E_s\}$로 각 인스턴스 P_s 상의 이벤트, 즉 $p_s(E_s)$의 값이 같은 E_s들 사이의 순서 관계이다.

그림 2의 'ref1' 시나리오에서는 전체 네 개의 인스턴스중 두개의 인스턴스 p_1과 p_2 만이 상호작용을 수행하고, 이때 $l_s(p_1)$ = 'IUT1', $l_s(p_2)$ = 'IUT2'이다. 활성화 조건과 메시지, 상태 변경 이벤트가 각 하나씩 포함되어 있는데, 각각을 e_7, $e_8 = \{e_8^s, e_8^r\}$, e_9라고하면, p_s 함수는 이벤트가 속하는 인스턴스를 구한다. 예를들어, $p_s(e_7) = \{p_1, p_2\}$이고, $p_s(e_8^s) = p_1$이다. p_1 인스턴스에 관한 이벤트들 간의 순서 관계 $<_{P_1} = \{(e_7, e_8^s), (e_8^s, e_9)\}$이다.

3. Global Finite State Machine의 생성

본 논문에서는 bMSC를 'Instance', 'Message', 'Condition', 'Alt Inline', 'Opt Inline', 'Loop Inline', 'Reference'만을 이용하여 구문을 정의하였다. 위와 같은 요소만을 이용한 bMSC의 경우 MSC의 시멘틱스는 행위에 해당하는 이벤트의 연속을 받아 들이는 오토마타로 생각할 수 있다. 본 논문에서는 이러한 이벤트의 연속을 받아들이는 오토마타를 행위 오토마타 (Behavior Automata)로 정의하고, 최종적으로는 이러한 행위 오토마타에서 GFSM을 생성하게 된다.

3.1 행위 오토마타 (BA) 생성

행위 오토마타는 하나의 bMSC에 대하여 해석 가능한 모든 시나리오를 나타내는 오토마타이다. 행위 오토마타를 추출하기 위하여 본 논문에서는 각각의 인스턴스와 이벤트들이 만나는 위치에 대하여 위치 값을 지정하였다. 위치 값은 각 인스턴스의 $<_{P_s}$의 관계에 따라서 차례대로 자연수를 부여한다. 그림 4는 그림 1에 대하여 위치 값 (밑줄 친 자연수)을 부여한 결과를 보여준다.

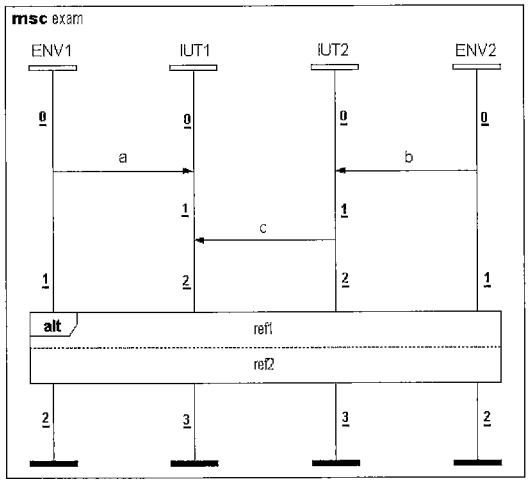


그림 4 MSC exam에 위치 값을 부여한 결과

그림 4에서와 같이 위치 값은 각각의 이벤트가 연관된 인스턴스에서의 상대 위치 정보를 갖는다. 즉, 각각의 인스턴스별로 해당 이벤트가 일어날 수 있는 순서 값이다. 이와 같은 위치 값을 이용하여 해당 이벤트는 어떤 순간에 자신이 일어날 수 있는 지 정의할 수 있다.

정의 3 (위치 정보)

$p_i \in P_s$에 대하여 L_i는 p_i의 현재 위치를 나타낼 때, bMSC의 위치정보 $L = <L_1, L_2, ..., L_n>$과 같다. 단, 각 p_i에 대하여 초기 위치 값은 0으로 한다.

위치정보는 한 bMSC에서 각 인스턴스가 진행된 정도를 위치 값으로 표현하는 것이다. 그림 4의 예제에서 초기 위치정보는 <0,0,0,0>이다. 따라서, bMSC의 의미는 각 위치정보를 오토마타의 상태로 하였을 때, 한 위치정보에서 일어날 수 있는 이벤트들이 일어났을 때의 다음 위치정보를 전이로 한 오토마타로 해석할 수 있다. bMSC $S_p = (P_s, E_s, \Sigma_s, p_s, l_s, <_s)$에 대하여 행위 오토마타는 다음과 같이 정의된다.

정의 4 (행위 오토마타)

행위 오토마타 $BA = (Q, \Sigma_{BA}, \delta, q_0, F)$로 정의할 수 있고, 각각의 의미는 다음과 같다.

1) $Q \subseteq L$은 상태의 집합이다.

2) $\Sigma_{BA} = E_s$는 입력 이벤트의 집합으로 S_p의 입력 이벤트 집합과 같다.

3) $\delta \subseteq Q \times \Sigma_{BA} \times Q$는 전이의 집합이다.

4) $q_0 \in Q$는 초기 상태이다.

5) $F \in Q$는 최종 상태이다.

행위 오토마타를 구하기 위하여, 본 논문에서는 위치 정보와 이벤트 사이에 다음과 같은 함수를 정의한다.

정의 5 (Location Function)

$LF : E_s \mapsto 2^{P_S \times N}$

정의 6 (Firable Function)

$FF : L \times E_s \mapsto B$

정의 7 (Advance Function)

$AF : L \times E_s \mapsto L$

LF는 각 이벤트에 대하여 연관된 인스턴스 상에서의 위치 값의 집합을 구하는 함수이고, FF는 이벤트가 위치정보 상에서 실행 가능한지 판별하는 함수이다. 또한, AF는 $p_s(E_s)$에 속한 인스턴스의 위치 값을 하나씩 증가시킨 위치정보를 구하는 함수이다. 예를들어, 그림 4에서 'a' 이름을 가지는 메시지 이벤트 e_1은 'In1'이 송신하고 'In2'가 수신하므로, $LF(e_1) = \{<p_1,1>, <p_2,1>\}$이다. 또한, 초기 상태 <0,0,0,0>에서 'a' 메시지는 송신 가능하므로, $FF(<0,0,0,0>, e_1) = true$이다. 하지만, 초기 상태에서 그 외의 이벤트에 대해서는 *false* 값을 갖는다. 마지막으로, 'a' 메시지를 송신한 후의 위치 정보는 <1,1,0,0>가 되므로, $AF(<0,0,0,0>, e_1) = <1,1,0,0>$이 된다. 이와 같은 함수를 바탕으로 행위 오토마타를 구하는 알고리즘은 다음과 같다.

알고리즘 1 (행위 오토마타 추출 알고리즘)

```
1: q0 = <0, 0, ...>;
2: push(Stack, q0);
3: δ = ∅;
4: while(Stack is not empty) {
5:    conf = pop(Stack);
6:    Q ← Q ∪ conf;
7:    for each e ∈ Es
8:       if( FF(conf, e)==true)
9:          δ ← δ ∪ (conf, e, AF(conf, e));
10:         if( AF(conf, e)  ∉  Q)
11:            push(Stack, AF(conf, e));
12:      fi
13:   rof
14: }
```

알고리즘은 우선 'Stack'에서 다음 상태와 전이를 만들 위치정보 'conf'를 선택한다(5). 이와 같은 위치정보는 처음 나오는 상태이므로 이 위치정보를 행위 오토마타의 상태 집합에 포함시킨다(6). bMSC에 기술된 이벤트들을 비교하여 'conf'에서 실행 가능한지를 보고(8), 실행 가능하면 실행했을 때의 다음 상태를 구해서 해당 전이를 행위 오토마타의 전이에 추가시킨다(9). 만일 진행된 위치정보가 아직 상태 집합에 포함되어 있지 않으면 'Stack'에 넣고 알고리즘을 반복한다. 이와 같은 알고리즘을 그림 4에 적용하여 구한 행위 오토마타는 그림 5와 같다.

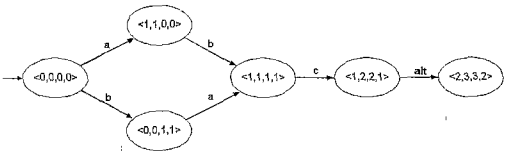


그림 5 MSC exam의 행위 오토마타

3.2 펼친 행위 오토마타 (Unfolded BA) 생성

3.1절에서는 한 bMSC에 대하여 'Reference'와 'Opt Inline', 'Loop Inline', 그리고 'Alt Inline'에 대하여 함축적인 하나의 전이로 나타낸 상태에서 행위 오토마타를 구했다. 하지만, 이와 같은 이벤트는 다른 bMSC를 참조하는 것이므로 참조되는 bMSC의 행위 오토마타를 구하여 합해야 한다. 즉, 그림 4의 예제에서 'ref1'와 'ref2' 등의 bMSC에 대한 행위 오토마타를 그림 5의 행위 오토마타에 추가해야 하는데, 이와 같이 참조된 bMSC의 행위를 모두 구하여 합친 오토마타를 본 연구에서는 펼친 행위 오토마타라 정의한다.

펼친 행위 오토마타를 구하기 위해서는 참조되는 bMSC에 대한 펼친 행위 오토마타를 재귀적으로 구해야 한다. 즉, 참조되는 bMSC내에서도 다른 참조를 가질 수 있기 때문에 더이상의 참조를 가지지 않는 행위 오토마타를 구할 때까지 재귀적으로 펼친 행위 오토마타를 구한다. 'Reference'와 'Opt Inline', 'Loop Inline', 그리고 'Alt Inline'에 대하여 펼친 행위 오토마타를 추출하는 방법은 그림 6,7,8,9와 같다. 참조하는 행위 오토마타에서 'Reference'를 나타내는 전이 'R' 대신에, 참조되는 bMSC에 대한 행위 오토마타의 시작 노드 앞과 종료 노드 뒤에 그림 6과 같이 각각 ε 전이를 추가하여 연결한다. 'Opt Inline'은 조건이 맞을 경우 참조되는 bMSC가 수행되고, 그렇지 않을 경우에는 수행되지 않는다. 이때, 참조되는 bMSC는 'when'을 가지는 조건문으로 시작하므로, 'Reference'와 같은 방법으로 행위 오토마타를 연결하고 조건의 역을 가지는 전이를 그림 7과 같이 추가한다. 'Loop Inline'의 경우에는 참조되는 bMSC의 수행

횟수를 나타내는 loop 변수와 이에 대한 비교, 그리고 loop 변수의 값을 증가하는 연산 등을 그림 8과 같이 연결한다. 마지막으로, 'Alt Inline'은 여러 개의 'Reference'가 있고, 각각의 'Reference'들 중 하나만 선택하여 수행되어야 하므로, 'Alt Inline'의 각각의 참조를 'Reference' 이벤트와 같이 연결시키면 된다(그림 9).

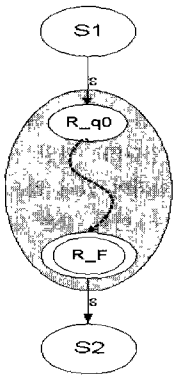


그림 6 Reference

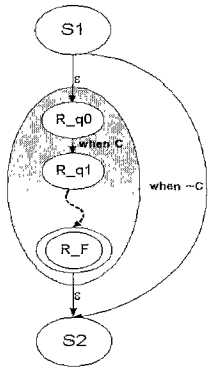


그림 7 Opt Inline

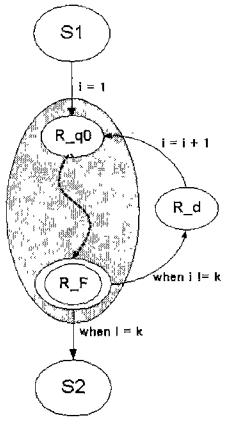


그림 8 Loop Inline

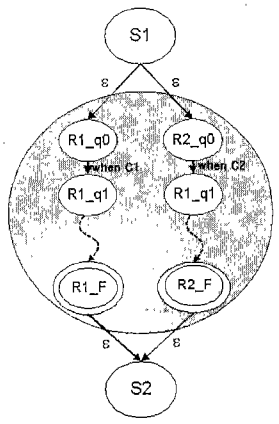


그림 9 Alt Inline

'Reference'와 'Opt Inline', 'Loop Inline', 'Alt Inline' 이벤트 'R'을 이용하여 참조하는 bMSC의 행위 오토마타가 $BA = (Q, \Sigma_{BA}, \delta, q_0, F)$이고, 참조되는 bMSC의 행위 오토마타가 $BA' = (Q', \Sigma_{BA'}, \delta', q_0', F')$ ('Alt Inline'의 경우에는 $BA_i = (Q_i, \Sigma_{BA_i}, \delta_i, q_{0_i}, F_i)$)이고, 'R'의 시작 노드는 'S1', 끝 노드는 'S2', 그리고 참조되는 bMSC의 실행조건은 'C', loop 변수는 'i', 그리고 loop 변수의 값이 'k'이면, 펼친 행위 오토마타 UBA는 다음과 같이 정의한다.

정의 8 (펼친 행위 오토마타)

$UBA = (Q_U, \Sigma_U, \delta_U, q_{0_U}, F_U)$

1) $Q_U = Q \cup Q'$ for 'Reference', 'Opt Inline'
   $Q_U = Q \cup Q' \cup R_d$ for 'Loop Inline'
   $Q_U = Q \bigcup_{\forall i} Q_i$ for 'Alt Inline'
2) $\Sigma_U = \Sigma_{BA} \cup \Sigma_{BA'}$
   for 'Reference', 'Opt Inline', 'Loop Inline'
   $\Sigma_U = \Sigma_{BA} \bigcup_{\forall i} \Sigma_{BA_i}$
   for 'Alt Inline'
3) $\delta_U = \delta \cup \delta' \cup \{(S1, \varepsilon, q_0'), (F', \varepsilon, S2)\}$
   for 'Reference'
   $\delta_U = \delta \cup \delta' \cup \{(S1, \varepsilon, q_0'), (F', \varepsilon, S2),$
   $(S1, \text{'when } \neg C', S2)\}$
   for 'Opt Inline'
   $\delta_U = \delta \cup \delta' \cup \{(S1, \text{'i = 1'}, q_0'),$
   $(F', \text{'when i = k'}, S2), (F', \text{'when i != k'}, R_d),$
   $(R_d, \text{'i = i + 1'}, q_0')\}$
   for 'Loop Inline'
   $\delta_U = \delta \bigcup_{\forall i} (\delta_i \cup \{(S1, \varepsilon, q_{0_i}), (F_i, \varepsilon, S2))\}$
   for 'Alt Inline'
4) $q_{0_U} = q_0$
5) $F_U = F$

그림 2와 3의 'ref1'과 'ref2' 각각의 행위 오토마타를 구하여 그림 5에 연결하면, 최종적으로 그림 10과 같은 펼친 행위 오토마타를 구할 수 있다. 이때, 펼친 행위 오토마타를 구할 때 주의할 것은 한 bMSC가 여러 bMSC에서 참조될 때이다. 즉, 참조되는 bMSC가 어떠한 bMSC에 의하여 참조되었는가의 정보를 가지고 있어야만 올바른 펼친 행위 오토마타를 구할 수 있다. 본 연구에서는 참조되는 bMSC의 행위 오토마타를 생성할 때, 각 상태들에 참조한 bMSC의 이름을 첨가시킴으로써 이를 해결한다. 그림 10을 보면 각 상태에 'exam.ref1'과 같이 참조되는 bMSC의 이름을 '.'을 이용하여 연결한다.

3.3 GFSM (Global Finite State Machine) 생성

지금까지는 하나의 bMSC S_D를 어떻게 오토마타로 해석할 수 있는가에 대하여 기술하였다. 이때 완성된 펼친 행위 오토마타는 위치 정보에 의한 것이므로 매우 많은 상태가 존재한다. 본 논문에서는 'Condition'에 사용된 상태 변수 V를 이용하여 펼친 행위 오토마타를 GFSM으로 변환한다. GFSM은 펼친 행위 오토마타에 나타나는 상태 변수 V의 값의 변화를 구하여 완성한다.

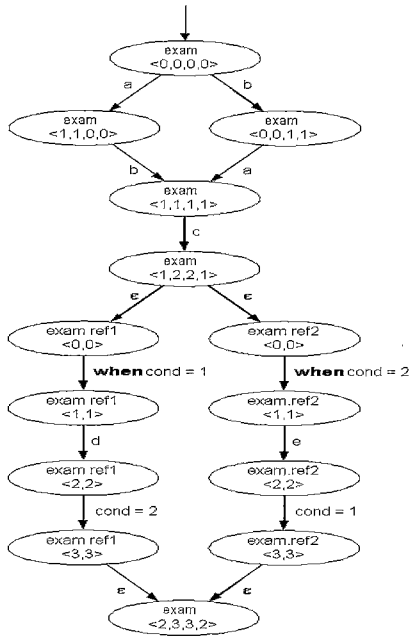


그림 10 MSC exam의 펼친 행위 오토마타

이때, 펼친 행위 오토마타의 각 경로를 따라갈 때, 시나리오 활성화 조건을 만나면 현재 상태 변수의 값과 비교하고, 같지 않으면 현재 경로의 수행을 중지한다. 또한 상태 값 변경을 만나면 해당 상태 변수의 값을 변경하고, 최종 노드까지 수행되면 GFSM의 하나의 전이를 완성한다. 따라서, GFSM의 상태는 각 상태 변수 값의 조합으로 구성되고, 전이는 펼친 행위 오토마타의 하나의 경로를 성공적으로 수행되었을 경우의 메시지 시퀀스가 된다. GFSM을 구하기 위해서 펼친 행위 오토마타에 대하여 다음과 같은 함수을 정의한다.

정의 9 (Valuation Function)

$\sigma : V \mapsto \{I(v) \mid v \in V\}$

정의 10 (Assignment Function)

$ASS : \sigma \times E \mapsto \sigma$

1) $ASS(\sigma, E^{C_i}) = \sigma(g_a(I(E^{C_i})) / g_v(I(E^{C_i})))$

2) $ASS(\sigma,$ other $E) = \sigma$

정의 11 (Satisfy Function)

$SAT : \sigma \times E \mapsto B$

1) $SAT(\sigma, E^{C_a}) = ?(\sigma(g_v(I(E^{C_a}))) == g_a(I(E^{C_a})))$

2) $SAT(\sigma,$ other $E) = true$

Valuation 함수 σ는 V의 현재 값을 돌려주는 함수이다. Assignment 함수 ASS는 상태 값 변경을 나타내는 'Condition' E^{C_i}를 만났을 때, 상태 변수 $g_v(I(E^{C_i}))$의 값을 $g_a(I(E^{C_i}))$으로 변경하여 그 결과를 돌려주는 함수이다. 그리고, Satisfy 함수 SAT는 시나리오 활성화 조건을 나타내는 'Condition' E^{C_a}를 만났을 때, 현재의 상태 값 $(g_v(I(E^{C_a})))$과 'Condition'에 적힌 값 $(g_a(I(E^{C_a})))$이 같은 지를 비교하여 *true*나 *false*를 돌려주는 함수이다. 그림 10에서 'cond'의 초기 값을 '1'로 했을 때, 'exam<0,0,0,0>' 노드에서의 σ('cond') = 1이다. 이 값을 가지고 각 경로를 수행할 때, 'exam.ref1<0,0>' 노드에서 $SAT(\sigma,$ 'when cond = 1') = *true*이지만, 'exam.ref2<0,0>' 노드에서 $SAT(\sigma,$ 'when cond = 2') = *false*가 된다. 마지막으로, 'exam.ref1<2,2>' 노드에서 $ASS(\sigma,$ 'cond = 2')를 수행하면, 'cond'의 값은 '2'가 된다.

3.2절에서 구한 펼친 행위 오토마타 $UBA = (Q_U, \Sigma_U, \delta_U, q_{0U}, F_U)$와 각 상태 변수에 대한 초기 값을 가지고 수행하여, 최종적으로 생성되는 $GFSM$은 다음과 같이 정의된다.

정의 12 (GFSM)

전체 명세 MSC에 관한 $GFSM = (S_{GFSM}, s_{GFSM_0}, \Sigma_{GFSM}, T_{GFSM})$으로 정의할 수 있고, 각각의 의미는 다음과 같다.

1) $S_{GFSM} \subseteq \sigma(V)$는 상태의 집합이다.

2) $s_{GFSM_0} \in S_{GFSM}$은 초기 시스템 상태이다.

3) $\Sigma_{GFSM} = <E_1, E_2, ...>$, $E_i \in E_M$은 전이에 사용되는 이름의 집합으로, 메시지 이벤트의 시퀀스로 이루어진다.

4) $T_{GFSM} \subseteq S_{GFSM} \times \Sigma_{GFSM} \times S_{GFSM}$은 전이의 집합이다.

펼친 행위 오토마타에서 $GFSM$을 생성하는 알고리즘은 다음과 같다.

알고리즘 2 (GFSM 생성 알고리즘)

```
1: Stack = S_GFSM = Σ_GFSM = ∅;
2: find V in UBA;
3: input initial value for V;
4: s_GFSM0 = σ(V);
5: push(Stack, s_GFSM0);
6: while(Stack is empty) {
7:    env = pop(Stack);
8:    if (env ∉ S_GFSM)
9:       S_GFSM ← S_GFSM ∪ σ(env);
```

```
10:      resultPath = generatePath( q0U, σ(env));
11:      for each PATH in resultPath
12:      ΣGFSM ← ΣGFSM ∪( σ(env), PATH, σ(PATH));
13:      rof
14:    fi
15: }
```

GFSM을 생성하기 위해서 먼저 펼친 행위 오토마타에 사용된 상태 변수들을 구하고(2), 그들에 대한 초기 값을 입력 받아야 한다(3). 각 상태 변수의 초기 값을 가지고(7), 펼친 행위 오토마타의 각 경로를 수행한다(10). 성공적으로 수행된 경로 하나 하나는 GFSM의 전이가 된다(12). 여기서 σ(PATH)는 각 경로의 수행후 F_U에 도달했을 때의 상태 변수의 값을 돌려준다고 가정한다. 하나의 경로가 성공적으로 수행된 상태에서의 상태 변수의 값이 새로운 상태일 경우에만 위의 알고리즘을 반복해서 수행한다(8). 펼친 행위 오토마타의 경로 수행은 다음과 같이 재귀적으로 수행되는 'generate Path' 알고리즘을 이용한다.

알고리즘 3 (generatePath)

```
1: generatePath(Node, σ) {
2:  resultPATH = ∅;
3:  outTrs = out transitions of Node;
4:  if(Node ∈ FU)
5:   push (Stack, σ(V));
6:   return {'Final};
7:  fi
8:  for each tr ∈ outTrs
9:   if ( SAT(σ, Etr) == true)
10:      PATH=generatePath(ENODE(Etr), ASS(σ,
     Etr));
11:    for each p∈ PATH
12:    resultPATH=resultPATH∪{Node.Etr.
                    ENODE(Etr),p};
13:    rof
14:  rof
15:  return resultPATH;
16: }
```

'generatePath' 알고리즘은 현재 노드에서 수행 가능한 전이들을 모두 수행한다(8-14). 이때, 현재 노드가 펼친 행위 오토마타의 종료 상태이면 현재 상태 변수의 값을 'Stack'에 넣는다(5). 그렇지 않은 노드들에서는 전이에 의한 새로운 노드가 생성되는데, 이 노드에 대한 경로 수행은 'generatePath' 알고리즘을 재귀적으로 호출함으로써 이루어진다(10). 여기서 ENODE는 주어진

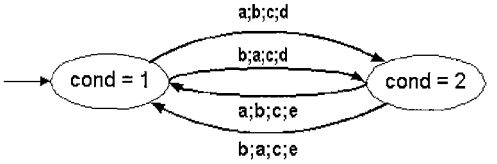


그림 11 MSC exam의 GFSM

이벤트에 대한 *UBA*에서의 다음 노드를 돌려주는 함수이다. 마지막으로, 현재 노드에서 수행가능한 모든 경로에 대한 정보를 담아서 돌려줌으로서 GFSM의 전이에 포함될 메시지의 시퀀스를 구한다(12). GFSM 생성 알고리즘을 그림 10에 적용하면 그림 11과 같은 GFSM이 생성된다. 이때, 'cond'의 초기 값으로는 '1'을 주었다.

4. 적용 예

4.1 SEC DTV 소프트웨어

SEC DTV 소프트웨어는 미국 ATSC의 DASE (Digital television Application Software Environment) 요구사항을 만족하도록 구현한 소프트웨어로서, 크게 DASE API를 제공하기 위해 Java로 구현된 모듈들과 pSOS[1] 및 EsWin(Embedded System Windows)[2] 상에서 동작하는 C 언어로 작성된 native 모듈들로 구성되어 있다[10,11]. 하지만, Java 코드의 느린 실행 속도를 가능한 한 회피하고, 이미 구현되어 사용되고 있는 native 모듈의 활용을 위해서 많은 부분이 C 코드로 작성되었다. 본 논문에서는 SEC DTV 소프트웨어의 일부 기능 만을 대상으로 하여, 제안된 방법을 적용한 예를 설명한다.

DTV 소프트웨어에서 native 응용 프로그램에는 HTML 문서를 보여줄 수 있는 BROWSER 태스크와 방송 프로그램의 정보를 보여주고 채널에 관한 정보를 관리하는 EPG (Electronic Program Guide) 태스크가 있다. 이때 EPG나 BROWSER 응용 프로그램은 동시에 수행될 수 없다. 그리고, 이러한 응용 프로그램들의 실행과 중단을 관리하고, 또한 각 응용 프로그램에 관한 정보 저장과 상태 관리 등의 관리자 기능을 담당하는 AL (Application Launcher) 태스크가 있다. 각 응용 프로그램은 사용자의 리모콘 입력으로 실행 또는 종료될 수 있고, 또한 사용자 메뉴를 보여준 상태에서 해당 아이콘을

1) pSOS는 WindRiver사의 내장 시스템용 운영체제이다.
2) EsWin은 삼성전자가 자체개발한 내장 시스템용 Windows System이다.

마우스로 클릭하여 실행할 수도 있다. 이때 사용자의 리모콘 입력이나 마우스 입력을 받아서 AL에게 전달하는 Remocon이 있다. 본 예제에서 테스팅의 대상(IUT)은 AL, BROWSER, EPG가 되고, 환경(ENV)은 Remocon이 된다. 즉, Remocon에서 AL, BROWSER, EPG로의 메시지 전달은 시스템의 입력이 되고, 그 반대는 출력이 된다. 하지만, 본 예제에서의 출력은 윈도우 생성과 소멸이므로 명시적으로 기술하지는 않는다.

SEC DTV 소프트웨어에서의 입력은 리모콘 입력과 마우스 선택 입력으로 이루어는데, 총 6개가 있다. 먼저 리모콘 입력에는 EPG의 실행을 위한 VK_RC_GUIDE, 사용자 메뉴의 실행을 위한 VK_RC_MENU, 그리고 종료를 위한 VK_RC_EXIT가 있다. 리모콘을 이용한 BROWSER의 실행은 북마크 형태로만 실행가능하고 이때의 입력은 VK_RC_FAV_CH이다. 사용자 메뉴에는 두 개의 아이콘이 있는데, 하나는 일반적인 브라우저 실행을 위한 아이콘이고, 다른 하나는 리모콘의 VK_RC_FAV_CH와 같은 기능을 하는 아이콘이다. 각각은 AL_BROWSER_BUTTON_ID와 AL_FAVORITE_BUTTON_ID의 마우스 선택 입력을 발생한다.

SEC DTV 소프트웨어에서는 메시지 송신을 위해서 SendEventTask() 함수를 이용한다. 송신 태스크에서는 보내고자하는 메시지를 정의한 이벤트 형식으로 구성하여, 대상 태스크의 식별자와 함께 SendEventTask()를 호출하면 대상 태스크의 큐에 이벤트를 넣어 준다. 실제 수행을 위해서 해당하는 파라미터와 값을 명세에 기술하였지만, 본 논문에서는 메시지에 대한 레이블로만 간주한다.

위의 예제에서 전체 시스템 상태를 나타내는 상태 변수로 menuShown과 currentApp가 사용된다. menuShown 상태 변수는 사용자 메뉴가 보여지고 있는지 아닌지를 나타내는 것으로, 0 또는 1의 값을 갖는다. 그리고, currentApp 상태 변수는 현재 수행중인 응용 프로그램의 종류를 나타내는 것으로, AL_APP_NONE, AL_APP_BROWSER, AL_APP_EPG의 값을 갖는데, 각각 어떤 응용 프로그램도 수행하지 않고 있는 상태, BROWSER가 수행 중인 상태, 그리고 EPG가 수행 중인 상태를 나타낸다.

앞에서 설명한 SEC DTV 소프트웨어의 기능을 MSC를 이용하여 기술하면 그림 12와 같이 총 13장으로 구성된다. DTV_native를 최상위로 하여 6가지의 리모콘 또는 마우스 선택 입력을 처리하게 된다. AL은 Remocon으로 부터 송신되는 6가지의 메시지 종류에 따라 각각에 대한 시나리오의 수행을 마친 후 다음 메시지를 처리한다. MENU는 VK_RC_MENU, EXIT는 VK_

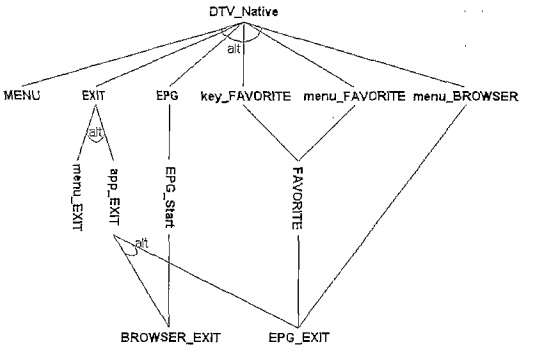


그림 12 명세의 전체 구조

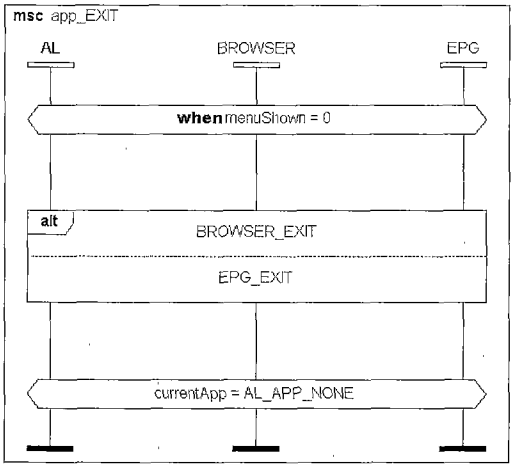


그림 13 응용 프로그램의 종료 시나리오

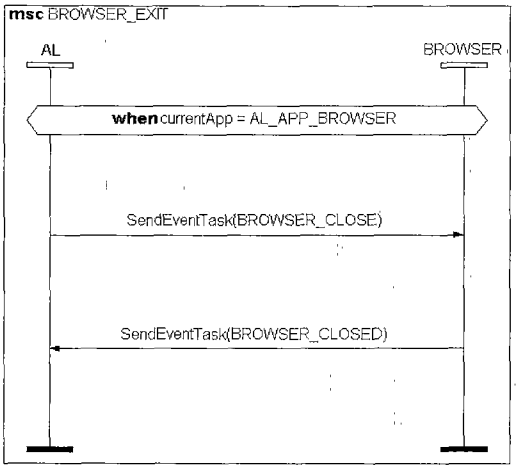


그림 14 currentApp가 AL_APP_BROWSER인 경우

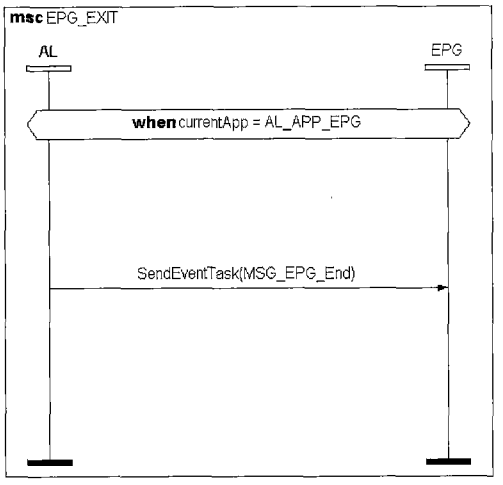


그림 15 currentApp가 AL_APP_EPG인 경우

RC_EXIT, EPG는 VK_RC_GUIDE, key_FAVORITE는 VK_RC_FAV_CH, menu_FAVORITE는 AL_FAVORITE_BUTTON_ID, 그리고 menu_BROWSER는 AL_BROWSER_BUTTON_ID에 대한 시나리오를 기술하고 있다.

그림 13과 14, 15의 응용 프로그램 종료 시나리오는 EXIT에서 참조되는 것으로 VK_RC_EXIT 입력이 들어왔을 경우 menuShown이 0인 상태에서 활성화되고, currentApp의 값에 따라 해당하는 시나리오가 수행된 후 currentApp의 값은 AL_APP_NONE이 됨을 기술하고 있다.

4.2 GFSM 생성

작성된 SEC DTV 소프트웨어 명세에 행위 오토마타 추출 알고리즘을 적용하면 매우 큰 행위 오토마타가 생성되므로, 여기서는 그림 13을 상위 bMSC로 하여 생성된 행위 오토마타를 그림 16에 나타내었다.

그림 16의 각 노드는 reference된 bMSC의 순서와 위치 정보 벡터에 대한 정보로서 이루어 진다. 예를 들어, 'app_EXIT.BROWSER_EXIT<0,0>' 노드는 그림 14의 초기 상태를 나타내는 노드로서, BROWSER_ EXIT 시나리오가 app_EXIT 시나리오에서 참조되었음을 나타낸다. 에지에는 메시지 레이블이 오거나 조건식들이 기술되어진다. 예를 들어, 'when currentApp = AL_APP_BROWSER'와 같이 기술되어 있는 에지를 수행할 경우에는, 현재 상태 값이 이와 같지 않을 경우 경로 수행을 종료한다. 하지만, 'currentApp = AL_APP_NONE'과 같이 기술되어 있는 에지를 수행할 경우에는, 현재 상태

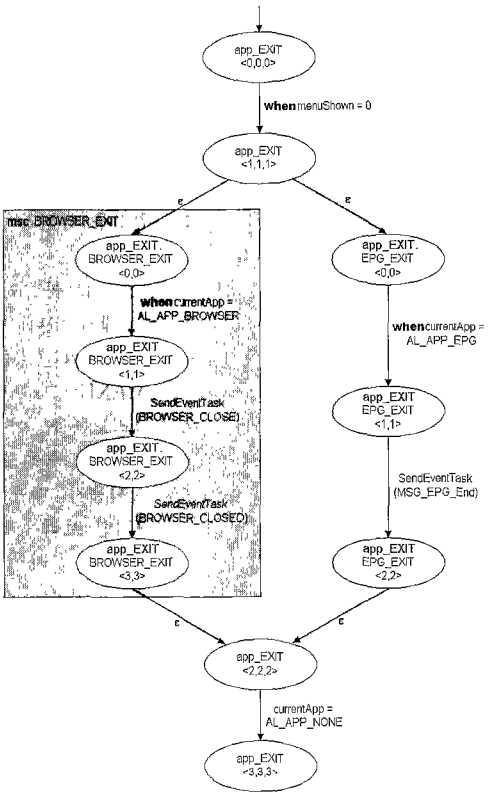


그림 16 app_EXIT에 대한 행위 오토마타

값을 기술되어 있는 값으로 바꾸어 준다.

그림 16과 같이 생성된 전체 행위 오토마타에 GFSM 생성 알고리즘을 수행하면 그림 17과 같은 GFSM이 생성된다. 이때, 각 상태 변수의 초기 값으로 menuShown은 0, currentApp는 AL_APP_NONE을 사용하였다. 상태 s0 - s5가 의미하는 것은 다음과 같다.

s0: currentApp = AL_APP_NONE, menuShown = 0
s1: currentApp = AL_APP_NONE, menuShown = 1
s2: currentApp = AL_APP_EPG, menuShown = 0
s3: currentApp = AL_APP_BROWSER, menuShown = 0
s4: currentApp = AL_APP_BROWSER, menuShown = 1
s5: currentApp = AL_APP_EPG, menuShown = 1

또한, 각 전이에 사용된 레이블 t0 - t19가 의미하는 것은 다음과 같다.

t0:

Remocon AL SendEventTask VK_RC_MENU

...

t19:

Remocon AL SendEventTask AL_BROWSER_BUTTON_ID
AL ALAgent q_send VK_RC_EXIT
AL BROWSER SendEventTask BROWSER_LAUNCH
BROWSER AL SendEventTask BROWSER_LAUNCHED

생성된 GFSM은 총 6개의 상태와 30개의 전이로 구성되어 있다. 두 개의 상태 변수의 조합으로 생길 수 있는 상태의 수와 동일한 수의 상태가 생성되었지만, DTV 전체 시스템에 적용하면 4608개의 가능한 상태 중 152개의 상태 만을 생성하는 결과를 얻을 수 있다. 6개의 상태에서 6개의 가능한 입력이 존재하므로 36개의 전이가 가능하지만, 마우스를 이용한 아이콘 선택 입력에 의해서는 두개의 상태 변수가 모두 변하게 되어 있어서 6개의 전이가 불가능하다. 이와 같이 실제 시스템에서 가능한 상태와 전이가 생성되기 때문에 기존의 방법에서와 같이 불가능한 상태나 전이가 포함되는 경우는 없다.

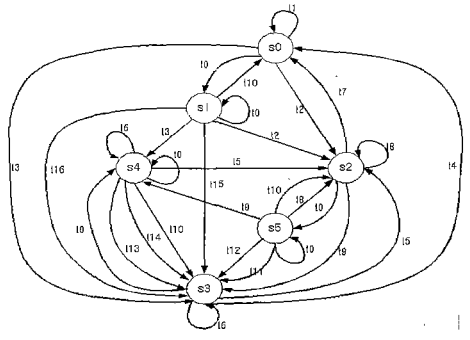


그림 17 MSC DTV에 대한 GFSM

그림 17과 같이 생성된 GFSM은 기존의 테스트 시퀀스 생성 방법을 이용하여 테스트 시퀀스를 생성함으로써, 테스팅에 이용할 수 있다[12,13,14,15]. 예를 들어, 모든 상태를 최소한 한번 이상 방문하고 다시 초기 노드로 돌아오는 state tour coverage를 적용하면 다음과 같은 하나의 테스트 시퀀스가 생성된다.

s0 -(t0)-> s1 -(t16)-> s3 -(t0)-> s4 -(t5)-> s2 -(t0)-> s5 -(t10)-> s2 -(t7)-> s0

이것은 실제로 VK_RC_MENU, AL_FAVORITE_BUTTON_ID, VK_RC_MENU, VK_RC_GUIDE, VK_RC_ MENU, VK_RC_EXIT, VK_RC_EXIT의 이벤트 입력 순서에 관한 테스팅을 수행하는 것이다. s0에서 s5까지는 DFS 방식으로 갈 수 있는 최대한 수행을 하고 나서, s5에서 수행할 수 있는 아직 방문하지 않은 상태가 더 이상 없으므로, s5에서 s0로 돌아가는 최단 경로를 포함하게 된다. 반면에, 모든 전이를 최소한 한번 이상 방문하도록 하는 transition tour coverage를 적용하면 14개의 테스트 시퀀스가 필요하다.

5. 관련 연구

대표적인 전체 상태 그래프 생성 방법인 [5]와 [6]에서는 bMSC의 각 이벤트의 위치정보를 조합한 것으로 상태 그래프의 노드를 정의한다. 생성된 상태 그래프로부터 가능한 모든 이벤트의 시퀀스를 조사함으로써 검증을 수행할 수 있다. 하지만, 실제 시스템에서는 모든 이벤트 시퀀스가 수행 가능한 것이 아니라, 시스템의 상태에 따라 가능한 이벤트 시퀀스가 한정된다. 따라서, 테스트 시퀀스를 생성할 때에는 실제 수행 가능한 이벤트 시퀀스를 생성해야 하지만, 이 방법들에서는 충분한 정보가 없기 때문에 지원되지 않는다. [16]과 [17]에서는 시스템의 상태정보를 추가적으로 기술하고, 이를 상태 그래프의 노드로 사용하였다. 추가적인 상태정보는 매우 적은 노력으로도 충분히 기술 가능하다. 하지만, 이 방법들에서는 전체 상태 그래프가 아니라, 각 태스크에 대한 FSM을 합성하는 방법만을 제공하고 있다. 테스팅을 위해서 전체 상태 그래프를 구하려면 다시 이들을 조합하는 추가적인 노력이 필요하다.

[4]의 ITU-T의 표준에서는 부분 순서관계를 가지는 연결 그래프 (connectivity graph)로서 메시지 상호작용의 의미를 정의하고 있다. 이는 메시지 송신과 수신을 서로 다른 이벤트로 해석하는 비동기적 메시지 전송을 기반으로 하는 것이다. 또한, 시스템의 상태 정보를 표현하기 위해 오토마타 혹은 FSM을 기반으로 MSC의 시멘틱을 정의하려는 노력들이 있었다[5,6]. 그 대표적인 예로 Message Flow Graph (MFG)에 관한 Leue와 Ladkin의 연구를 들 수 있다[5]. MFG의 연구에서 MSC는 최종적으로 Buchi-오토마타 형태를 가지게 된다. MFG에서 MSC는 Buchi-오토마타로의 변환을 위한 중간단계로 Ne/Sig 그래프로 변환된다. Ne/Sig 그래프는 MSC에서 이벤트를 발생하는 위치와 받는 위치를 노드로 표현한다. 또한, 한 프로세스 내에서의 순서 관계에 의한 것은 *next* 이벤트로 프로세스 간의 상호 작용은 *signal* 이벤트로 분리하여, 전자를 점선으로 후자를 실선으로 표현한 그래프이다. Ne/Sig 그래프를 행위 FSM으로 표현하기 위하여 MFG에서는 en(x)와 ta(x)를 정

의하였다. en(x)는 상태 x가 가능하게 되는 상태(enable)를 말하고, ta(x)는 상태 x를 수행한 경우(taken)를 말한다. 그리고, MFG에서는 MSC의 이벤트를 비동기화하여 시멘틱을 정의하였기 때문에 한 이벤트를 주고 받는 과정은 분리해서 일어나게 된다.

하지만, 위와 같은 MFG 방법은 한 장의 basic MSC만을 FSM으로 변환할 수 있고, 더구나 사용 가능한 MSC 표현도 메시지에 한정되어 있다. 이를 해결하기 위해서 [6]에서는 여러 장의 bMSC에 대한 GSTG (Global State Transition Graph)를 구하기 위한 방법을 제시하고 있다. [6]에서는[5]에 동시 영역 (coregion)과 타이머에 대한 구문을 포함하였으며, ADG (Action Dependency Graph)를 이용한 순차적, 선택적, 반복적 연결에 의한 GSTG 생성 방법을 제안하였다. ADG에서는 같은 태스크 내에서의 이벤트 관계는 실선 화살표로 나타내고 *must* 관계에 있는 것으로 정의하였다. 또한, 메시지 송수신은 둥근 화살표로 나타내고 *may* 관계가 있다고 정의하고, 선택적 관계에 있는 이벤트 사이에는 점선 화살표로 나타내고 *complement* 관계가 있다고 정의하였다. 본 연구에서는 [6]과 유사한 방법으로 bMSC들을 연결하는 *unfolding* 방법을 제안하고 있다. 하지만, 조건문을 사용하여 시스템 상태 변화를 구하고 있기 때문에, 최종 생성되는 FSM의 의미가 전혀 다르다.

MSC를 기반으로하는 기존의 테스팅 방법에서는 SDL[18]로 작성된 시스템 모델에서 테스트 시퀀스를 찾아낼 때 MSC가 가능한 시퀀스인지를 판별하는 보조 역할로 사용된다거나[19], 하나의 MSC 안에서 나타날 수 있는 여러 병행적인 행위를 타임 스탬프를 이용하여 추출하여 테스트 시퀀스로 사용하는 방법이 있었다[20]. 하지만, 여러장의 MSC를 이용하여 전체 시스템의 행위를 기술한 명세로부터 테스트 시퀀스를 생성하지는 못하고 있다.

[20]에서는 분산 시스템의 happened-before 관계를 구하기위해 사용되는 타임 스탬프 할당 방법을 MSC에 적용하여, 각 메시지 송수신 이벤트의 타임 스탬프를 구하고, 이들을 비교함으로써 순서관계를 구하는 방법을 사용하고 있다. 이때, 메시지 송신, 수신, 동시 영역, 전역 조건 이벤트 등에 대한 타임 스탬프 할당 규칙을 정의하고 있다. 하지만, 이 방법은 시스템의 전체 행위에 대한 명세는 시나리오를 이용해서 작성하는 것이 불가능하기 때문에 부분적인 명세만을 bMSC를 이용하여 작성하고, 이로부터 FSM을 구하여 테스트 드라이버로 사용하는 것이다. 또한, [20]의 방법에서는 테스트 드라이버 생성시에 ENV 태스크들 사이의 이벤트와 IUT 태스크들 사이의 내부 이벤트를 무시한다. 하지만, 본 연구에서는 이러한 내부 이벤트들도 함께 테스트 시퀀스에 포함하여 모니터링의 대상으로 한다. 즉, [20]에서는 시스템의 입력에 의한 출력 결과만을 비교함는 방법인 반면에, 본 연구에서는 내부 이벤트에 대한 모니터링도 병행함으로써, 어떤 메시지 상호작용에서 오류가 발생하였는 지에 대한 추가적인 정보를 제공할 수 있다는 장점이 있다.

FSM 기반의 테스팅 방법은 명세 기반의 테스팅 방법들 중에서 가장 많이 연구되고 있는 방법이[13,14,15]. 최근에는 여러 FSM간의 통신 방법과 데이타 표현 방법을 확장한 CEFSM (Communicating Extended FSM)을 이용한 명세 방법과 테스팅 방법에 대한 연구가 활발히 진행되고 있다. CEFSM을 기반으로하는 명세 언어로는 ITU-T의 SDL[18]과 ISO의 Estelle[21], 그리고 Statecharts[22]가 있다. SDL이나 Estelle를 이용한 테스트 시퀀스 방법들에서는 CEFSM들의 합성으로[14], Statecharts의 방법에서는 AND나 OR 상태들을 펼쳐서[15] 전체 시스템에 대한 FSM을 만들고, 기존의 FSM 기반의 테스팅 방법들을 이용하여 테스트 시퀀스를 생성한다. 하지만, 이러한 방법에서는 CEFSM의 합성 또는 펼칠 때 상태 폭발 (state explosion) 문제가 발생할 수 있고, 다양한 메시지 전송 방법을 표현할 수 없다.

6. 결론 및 향후연구

본 논문에서는 MSC를 이용하여 태스크들 사이의 상호 작용을 기술하는 방법을 소개하고, 여러 장의 bMSC들로부터 전체 시스템의 동작을 나타내는 하나의 GFSM을 생성하는 방법을 제안하였다. 본 논문에서 제안한 방법에서는 가능한 상태와 전이만을 생성하므로 생성되는 GFSM을 최소화할 수 있다. 또한, 제안된 방법을 이용하여 실제 구현되고 있는 SEC DTV 소프트웨어의 각 태스크들 사이의 상호 작용을 명세하고, GFSM을 생성함으로써 제안된 방법의 효용성을 보였다.

현재 방법에서는 MSC의 많은 기능들을 지원하고 있지 못하다. 즉, 'Alt Inline' 만을 이용하여 명세하기 때문에 실제 시스템의 행위와 작성된 명세에서 도출할 수 있는 행위에 차이가 있다. 따라서, 앞으로 MSC에서 지원되고 있는 병행성, 시간 제약성 등에 대한 GFSM 변환 알고리즘 개발이 필요하다.

태스크들 사이의 상호 작용은 큐를 이용한 비동기적 메시지 전송, 객체 지향 프로그램의 메소드 호출과 같은 동기적 메시지 전송, 전역 변수와 lock을 이용하는 공유 메모리 기법, 그리고 랑데뷰를 이용하는 동기화 기법 등으로 나눌 수 있다. 또한 큐를 이용한 비동기적 메시지 전송에서도 FIFO 큐를 사용할 것인지 아닌지에 따라,

또는 한 태스크가 가지는 큐의 갯수를 하나로 할 것인지, 송신 태스크 각각에 큐를 할 것인지에 따라 여러가지 구현 모델이 있을 수 있다[23]. 이러한 상호 작용 방법들은 하나의 시스템에서 한 가지만이 사용되는 것이 아니라 복합적으로 사용된다. 제안된 방법에서는 FIFO 큐를 이용하는 태스크들 사이의 상호 작용에 관한 명세 및 GFSM 생성을 다루고 있을 뿐이다. 그 외의 상호작용 방법을 기술하고, GFSM을 생성할 수 있도록 확장하는 작업이 필요하다.

참 고 문 헌

[1] T. Katayama, E. Itoh, Z. Furukawa, and K. Ushijima. "Test-case Generation for Concurrent Programs with the Testing Criteria Using Interaction Sequences," In *Asia-Pacific Software Engineering Conference (APSEC'99)*, 1999.

[2] M. Kim, J. Shin, S.T. Chanson, and S. Kang. "An Approach for Testing Asynchronous Communication Systems," *IEICE Transactions on Communications*, pages 81-95, 1999.

[3] Recommendation Z.120. "Message Sequence Chart (MSC'96)". Technical report, ITU-T, 1996.

[4] Recommendation Z.120 Annex B. "Formal Semantics of Message Sequence Charts," Technical report, ITU-T, 1998.

[5] P.B. Ladkin and S. Leue. "Interpreting Message Flow Graphs," *Formal Aspects of Computing*, 7(5):473--509, 1995.

[6] B.M. Kim, H.S. Kim, and W.Y. Kim. "Construction of Global State Transition Graph for Verifying Specification Written in Message Sequence Charts for Telecommunications Software," *IEICE Transactions on Information and Systems*, pages 249-261, 2001.

[7] P. Graubmann, E. Rudolph, and J. Grabowski. "Towards a Petri Net based Semantics Definitions for Message Sequence Charts," In *Proceedings of the Sixth SDL Forum*, 1993.

[8] Prepublished Recommendation Z.120. "Message Sequence Chart (MSC'2000)}. Technical report," ITU-T, 1999.

[9] R. Alur and M. Yannakakis."Model Checking of Message Sequence Charts," In *Proceedings of the Tenth International Conference on Concurrency Theory*, pages 114-129, 1999.

[10] ATSC T3/S13 Doc. 010. "Data Broadcast Specification," Technical report, ATSC, 1999.

[11] ATSC Doc. A/65. "Program and System Information Protocol for Terrestrial Broadcast and Cable," Technical report, ATSC, 1997.

[12] C. Bourhfir, R. Dssouli, and E.M. Aboulhamid. "Automatic Test Generation for EFSM-based Systems," Technical report, University of Montreal, TR-1043, 1996.

[13] T.S. Chow. "Testing software design modeled by finite state machines," *IEEE Transactions on Software Engineering*, 4:178-187, 1978.

[14] O. Henniger and P. Neumann. "Test case generation based on formal specifications in Estelle," In *Proceedings of the 1st IEEE International Workshop on Factory Communication Systems*, 1995.

[15] Y.G. Kim, H.S. Hong, D.H. Bae, and S.D. Cha. "Test Cases Generation from UML State Diagrams," *IEE Proceedings - Software*, 146:187--193, 1999.

[16] I. Kruger, R. Grosu, P. Scholz, and M. Broy. "From MSCs to Statecharts," In *Distributed and Parallel Embedded Systems*, 1999.

[17] J. Whittle and J. Schumann. "Generating Statechart Designs from Scenarios," In *Interanational Conference on Software Engineering (ICSE)*, 2000.

[18] Recommendation Z.100. "SDL Methodology Guidelines," Technical report, ITU-T, Geneva, 1992.

[19] J. Grabowski. "SDL and MSC Based Test Case Generation-An Overall View of the SAMSTAG Method," Technical report, University of Berne, IAM-94-0005, 1994.

[20] I.S. Chung, H.S. Kim, H.S. Bae, and B.S. Lee. "Testing of Concurrent Programs based on Message Sequence Charts," In *International Symposium on Parallel and Distributed Software Engineering (PDSE'99)*, 1999.

[21] T. Budkowski and P. Dembinski. "An Introduction to Estelle: a Specification Language for Distributed Systems," *Computer Networks and ISDN*, 14(1), 1987.

[22] David Harel. "Statecharts: A Visual Formalism for Complex Systems," *Sciences of Computer Programming*, 8:231--274, 1987.

[23] R. Alur, G.J. Holzmann, and D. Peled. "An Analyzer for Message Sequence Charts," *Software Concept and Tools*, 17(2):70--77, 1996.

이 남 희

1991년 2월 한국과학기술원 전산학과 졸업(학사). 1998년 2월 한국과학기술원 전산학과 졸업(석사). 1998년 3월 ~ 현재 한국과학기술원 전산학과 박사과정 재학중. 관심분야는 실시간 시스템 명세 및 검증, 소프트웨어 재사용

김 태 효

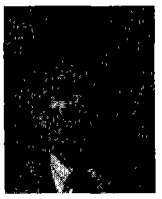
1998년 2월 한국과학기술원 전산학과 졸업(학사). 2000년 2월 한국과학기술원 전산학과 졸업(석사). 2000년 3월 ~ 현재 한국과학기술원 전산학과 박사과정 재학중. 관심분야는 정형 명세 및 검증

차 성 덕

1983년 Univeristy of California at Irvine 전산학 학사. 1986년 University of California at Irvine 전산학 석사. 1991년 University of California at Irvine 전산학 박사. 1990년 ~ 1991년 Hughes Aircraft Co. 연구원. 1991년 ~ 1994년 The Aerospace Corp. 연구원. 1994년 9월 ~ 현재 한국과학기술원 전산학과 부교수

신 석 종

1993년 2월 경원 전문대학 졸업. 1993년 2월 ~ 현재 (주) 삼성전자 CTO 전략실 소프트웨어 센타 선임연구원 재직. 관심분야는 소프트웨어 테스팅 자동화, 테스팅 방법론, Network 등임



홍 인 표

1989년 2월 명지대학교 기계공학과 졸업(학사). 1989년 5월 ~ 현재 (주) 삼성전자 CTO전략실 E-CIM 센터 책임연구원 재직. 관심분야는 소프트웨어 품질보증, 프로세스, 테스팅 자동화 등임

박 기 웅

1983년 2월 원광대학교 전자공학과 졸업(학사). 1985년 2월 경희대학교 전자공학과 졸업(석사). 1985년 4월 ~ 현재 (주) 삼성전자 CTO전략실 수석연구원 재직중. 관심분야는 소프트웨어 품질보증, 테스팅 자동화, 보안, 전자상거래