스테이트차트의 실시간 검증을 위한 모델체커의 확장

(Extending Model Checker for Real-time Verification of Statecharts)

방 호 정 † 홍 형 석 †† 김 태 효 ††† 차 성 덕 ††††
(Hojung Bang) (Hyoungseok Hong) (Taihyo Kim) (Sungdeok Cha)

요 약 본 연구는 스테이트차트의 실시간 검증을 위한 알고리즘을 제안한다. 스테이트차트는 실시간 반응형 시스템의 명세에 많이 사용되고 있으며 동기적과 비동기적인 두개의 시간모델을 지원한다. 그러나 기존의 스테이트차트에 대한 실시간 검증 방법은 비동기적 시간 모델과 호환되지 않거나, 변수를 모델에 추가함으로써 모델의 상태 공간을 증가시키는 단점이 있었다.

우리는 기존의 모델 체킹 알고리즘을 확장하여 이러한 문제점을 해결하였다. 확장된 알고리즘은 시간을 증가시키는 전이만을 고려하기 때문에 스테이트차트의 두 가지 시간 모델에 모두 사용할 수 있으며, 시간의 계산이 알고리즘 내부적으로 이루어지기 때문에 모델에 변수를 추가할 필요가 없어 상태공간을 증가시키지 않는다.

본 연구는 이러한 알고리즘을 토대로 기존의 모델 체커인 NuSMV를 확장하였으며, 사례 연구를 통하여 그 유용성을 확인하였다.

키워드 : 스테이트차트, 실시간 검증, 심볼릭 모델 체킹, 정형 기법

Abstract This paper presents a method for real-time verification of Statecharts. Statecharts has been widely used for real-time reactive systems, and supports two time models: synchronous and asynchronous. However, existing real-time verification methods for them are incompatible with the asynchronous time model or increase state space by introducing new variables to the target models.

We solved these problems by extending existing model checking algorithms. The extended algorithms can be used with both time models of Statecharts because they consider time increasing transitions only. In addition, they do not increase target state space since they count those transitions internally without additional variables.

We extended an existing model checker, NuSMV, based on the proposed algorithms and conducted some experiments to show their advantage.

Key words : Statecharts, real-time verification, symbolic model checking, formal method

1. 서 론

최근 소프트웨어에 대한 의존도가 커지면서 소프트웨어 오류로 인한 피해도 함께 증가하고 있어 소프트웨어의 안정성 확보는 어느 때보다 중요해지고 있다. 이러한 피해는 인공위성과 원자력 등 안전에 민감한 시스템의 경우 더욱 심각해진다. 이와 더불어 소프트웨어의 오류는 소프트웨어 개발에 필요한 시간과 비용을 증가시킨다. 특히, 요구사항 분석이나 설계 단계에서의 오류는 다른 단계의 오류들보다 더 많은 비용을 초래하는 경우가 많은데 그 이유는 초기단계의 오류들은 구현의 많은 부분에 영향을 끼치기 때문이다.

정형검증 기법은 소프트웨어의 오류를 줄이기 위한 시도의 하나이며 자동화가 가능한 경우가 많아 다른 방법으로는 검출하기 힘든 오류를 효과적으로 발견할 수

· 본 연구는 첨단과학기술연구센터(AITrc), 소프트웨어프로세스개선센터(SPIC) 및 인터넷 침입대응기술연구센터(IIRTRC)의 지원을 받음
† 비 회 원 : KAIST 전자전산학과
hjbang@salmosa.kaist.ac.kr
†† 비 회 원 : Systems Design Research Lab, University of Pennsylvania
hshong@saul.cis.upenn.edu
††† 비 회 원 : KAIST 전자전산학과
taihyo@salmosa.kaist.ac.kr
†††† 종신회원 : KAIST 전자전산학과 교수
cha@salmosa.kaist.ac.kr
논문접수 : 2003년 1월 10일
심사완료 : 2004년 3월 26일

있다. 정형검증 기법은 안전에 민감한 시스템의 개발에서 많이 사용되고 있는데, 본 연구에서는 모델 체킹에 초점을 맞춰 논의를 진행한다.

스테이트차트(Statecharts)[1]는 실시간 반응형 시스템(Real-time Reactive System)을 기술하는 데 널리 사용되는 정형 명세 언어이다. 스테이트차트는 의미의 해석을 위해 두가지 시간 모델을 지원하는데, 이중 동기적 (Synchronous) 시간 모델은 모든 전이가 한 단위의 시간을 소요하는 것으로 해석하는 반면, 비동기적 (Asynchronous) 시간 모델은 시스템 내부 반응이 모두 끝나고 시스템이 안정된(stable) 상태에 도달할 때까지는 시간을 증가시키지 않는다. 본 연구에서는 동기적 시간 모델로 해석되는 스테이트차트를 동기적 스테이트차트, 비동기적 시간 모델인 경우는 비동기적 스테이트차트로 지칭한다.

본 연구에서 실시간 검증을 실시간 특성에 대한 검증과 실시간 계산을 포괄하는 의미로 사용한다. 실시간 특성이란 그 만족 여부가 이벤트의 순서 관계와 더불어 발생 시점에 의해 결정되는 특성을 말하며, 실시간 계산은 특정 조건 사이의 최단 또는 최장시간 등 시스템의 시간적 정보를 알아내는 것으로 이를 통해 대상 시스템의 효율성에 대한 정보를 얻을 수 있다.

Campos의 연구[2-4]에서는 실시간 모델 체킹 알고리즘과 실시간 계산 알고리즘을 제시하고 있다. 제안된 실시간 모델 체킹은 Real-time CTL[5]로 기술된 시스템 특성을 검증하는데 시간의 경과를 지나온 전이의 개수로 파악하기 때문에 비동기적 스테이트차트에 적용할 수 없다. 왜냐하면, 비동기적 스테이트차트에는 시간을 증가시키지 않는 전이가 존재하기 때문이다.

Brockmeyer의 연구[6,7]에서는 모델과 특성을 적절히 변환함으로써 이러한 문제를 해결하고자 한다. 이들은 주어진 Timed CTL[8] 특성에 따라 모델에 시간을 계산하는 변수를 추가하고, TCTL 특성을 이 변수를 포함한 CTL 특성으로 변환한다. 변환된 특성과 모델은 기존의 모델 체커로도 검증이 가능하다는 장점이 있는 반면, 추가되는 변수가 모델의 상태 공간을 증가시킨다는 단점이 있다.

본 연구에서는 비동기적 스테이트차트에서 정확한 실시간 모델 체킹 및 실시간 계산을 수행하도록 하기 위해 새로운 알고리즘을 제안하였다. 제안된 알고리즘은 특정 프레디킷을 만족하는 상태에서 출발하는 전이를 거치게 될 경우에만 시간을 한 단위 증가시킨다. 이러한 방법은 시간의 계산을 알고리즘에서 내부적으로 수행하기 때문에 모델에 변수를 추가할 필요가 없으므로 Brockmeyer의 연구와는 달리 모델의 상태공간을 증가시키지 않는다.

이 논문은 다음과 같이 구성되어 있다. 2장은 배경 지식과 관련연구에 대해 설명한다. 3장에서는 본 연구가 제안하는 방법론에 대해 설명하고, 4장에서는 사례연구를 통해 제안된 방법의 유용성을 보인다. 5장에서는 결론과 향후 연구 방향에 대해 설명한다.

2. 배경지식 및 관련 연구

2.1 스테이트차트

스테이트차트(Statecharts)[1]는 유한상태기계(FSM)를 기반으로 병렬성, 계층구조를 추가하고 브로드캐스팅(broadcasting)에 의한 통신을 사용하여 복잡한 시스템을 간결하게 표현할 수 있도록 한 시각적 정형명세 언어이다. 본 연구는 스테이트차트의 의미구조로 Statemate MAGNUM[1]의 의미구조를 기반으로 한다. Statemate MAGNUM은 널리 사용되는 스테이트차트용 도구모음이다. 스테이트차트의 문법과 의미구조의 정의는 Harel의 논문[9]을 참조하였다.

스테이트차트는 동기(Synchronous)와 비동기(Asynchronous)의 두 가지 시간 모델을 지원하며, 동일하게 그려진 스테이트차트라도 어느 시간 모델을 따르느냐에 따라 그 행위가 다르게 결정된다. 시스템이 어느 시간 모델을 따라야 하는지는 대상 시스템 특성에 따라 결정된다.

동기적 시간 모델은 시스템 클럭에 맞춰 동작하는 칩셋 등의 하드웨어를 명세 하는데 사용되며, 비동기적 시간 모델은 반응형 및 내장형 소프트웨어를 기술하는데 사용된다. 반응형 및 내장형 시스템에서 비동기적 시간 모델이 사용되는 이유는 일반적으로 이들 시스템이 입력받는 센서의 검사 주기나 제어하는 작업의 수행시간은 제어를 계산하는데 필요한 시간보다 훨씬 크기 때문이다. 다음에서 시간 모델이 선택에 따른 행위변화를 예시함으로써 두 시간모델의 차이를 설명한다.

2.1.1 동기적 스테이트차트의 행위

동기적 시간 모델에서는 모든 전이가 단위 시간을 소요하며, 외부 이벤트는 항상 받아들여진다. 따라서 이 모델은 특별한 이벤트에 맞춰 단위 작업이 진행되는 시스템을 기술하는데 적합하다.

그림 1은 커피자동판매기를 기술한 스테이트차트이다. 이 시스템에서 동전관리기와 커피관리기는 병렬적으로 동작한다. *coin*은 외부이벤트로 동전 감지 센서의 입력을 나타낸다. 동전이 투입되면 (*coin*) 동전관리기는 동전이 투입되었음을 커피관리기에 알리고 (*inserted*), 커피관리기가 작업이 끝날 때까지 *full* 상태에서 기다린다. 커피관리기는 동전관리기의 신호를 받아 커피를 제공한다.

1) Statemate MAGNUM is a registered trademark of i-Logix Inc.

커피를 제공하는 데는 3단위의 시간이 소요된다 (*tm(begin,3)*). 커피의 제공이 끝나면 작업이 종료되었음을 동전관리기에 알리고 (*done*), 동전관리기는 다시 동전을 받아들일 수 있는 상태가 된다 (*empty*).

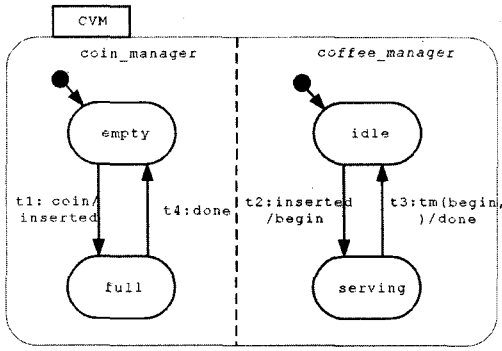


그림 1 간단한 커피자동판매기

그림 2는 이러한 커피자동판매기를 동기적으로 해석하는 경우 가능한 행위이다. 여기서 *EI*는 {*empty, idle*}을 *FS*는 {*full, serving*}을 각각 나타낸다. *coin, inserted* 등은 각각 해당 이벤트가 발생하였음을 나타낸다. 동기적 스테이트차트에서는 모든 전역 전이에 단위 시간이 소요되므로, 동전이 투입된 후 커피가 제공이 완료될 때까지, 즉 *coin*이 발생한 후, *done*이 발생할 때까지 총 6단위의 시간이 소요되고 있다.

2.1.2 비동기적 시간 스테이트차트의 행위

일반적으로 커피자동판매기에서 투입된 동전을 처리하는 시간은 실제로 커피를 준비하는 시간에 비해 훨씬 적게 걸린다. 따라서 커피자동판매기 모델에서는 내부 처리에 소요되는 시간은 무시하고 실제 작업에 걸리는 시간만을 고려하는 것이 타당할 것이다. 따라서 이와 같은 반응형 시스템을 기술하는 데는 비동기적 시간 모델이 적합하다.

비동기적 시간 모델에서는 시스템의 반응은 시스템이 안정된 상태에 도달할 때까지 시간의 소요 없이 계속된다. 안정된(stable) 상태란 외부 이벤트가 개입되지 않고서는 더 이상 내부 반응이 계속되지 않는 상태를 말한다. 안정된 상태가 되면 시간이 증가하고 외부 이벤트가 개입된다. 따라서 전이는 시간을 증가시키는 전이와 그렇지 않은 전이로 구분되는데, 시간을 증가시키는 전이는 안정된 상태에서 출발하는 전이들이다.

그림 3은 비동기적 시간 모델로 해석한 경우에 가능한 경로중의 하나를 예시한 것이다. 여기서 안정된 상태는 이중원으로 그려져 있으며, 점선 사각형으로 둘러싸인 부분은 같은 시간대의 상태들이다.

2.2 관련 연구

2.2.1 Campos 등의 연구

Campos 등은 심볼릭 모델 체킹을 알고리즘을 확장하여 Real-Time CTL로 기술된 실시간 특성에 대한 검증이 가능하도록 하였다.[2,3,4] 또, 실시간 계산을 위해 특정 상태(혹은 조건)사이에 소요되는 최단시간 및 최장시간을 구하는 알고리즘과 특정조건이 만족하는 횟수를 계산하는 알고리즘도 제시하였다. 이러한 연구 결과는 Verus[10]라는 심볼릭 모델 체커로 구현되었으며, 본 연구에서 확장할 모델 체커인 NuSMV[11]에도 채택되었다. 이후 연구[12]에서는 반연속적(semi-continuous) 시간 모델을 제안하였는데, 이 모델은 전이는 단위 시간이 소요되는 전이(unit-time transition)와 시간이 소요되지 않는(zero-time transition)으로 나누어진다는 점에서 본 연구에서 사용하는 실시간 모델과 유사하다.

그러나 Campos의 알고리즘으로는 비동기적 스테이트차트를 검증할 수 없다. 이들의 알고리즘은 시간을 거쳐온 전이의 개수로 파악하고 있는데, 비동기적 스테이트차트에서는 시간이 증가시키지 않는 전이가 존재하기 때문이다. 또한, [12]의 연구는 본 연구와 유사한 모델을 대상으로 하고 있음에도 불구하고, 시각 조건과 종료 조건 사이에 발생하는 특정 이벤트 혹은 상태의 최소 및 최대 횟수를 구하는 알고리즘만 제공할 뿐 실시간 모델

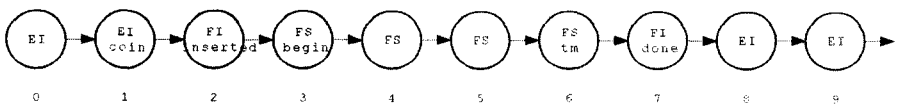


그림 2 동기적 시간 커피자동판매기의 경로

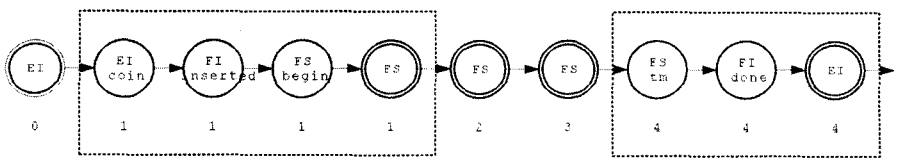


그림 3 비동기적 시간 커피자동판매기의 경로

체킹과 계산 알고리즘은 제공되지 않는다.

2.2.2 Brockmeyer 등의 연구

Brockmeyer 등의 연구[6,7]는 주어진 실시간 특성에 따라 모델과 특성을 CTL 모델 체커로 검증이 가능한 형태로 변환하는 방법을 사용하였다.

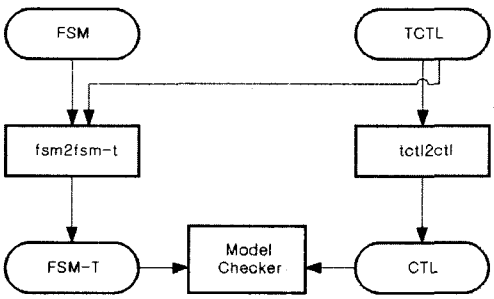


그림 4 Brockmeyer 등의 방법론

Brockmeyer 등의 접근법은 그림 4와 같다. 이들은 tctl2ctl이라는 도구를 사용하여 TCTL 특성을 CTL 특성으로 변환하고, fsm2fsm-t라는 도구를 사용하여 FSM을 FSM-T로 변환한다. FSM-T는 FSM에 실시간 특성의 검증에 필요한 시간계산용 변수를 추가한 것이다.

예를 들어, 주어진 모델에서 시작 상태에서 출발하는 모든 경로에서 5 단위 시간 내에 ϕ가 만족하는 상태가 존재해야 하고, 그때까지의 모든 상태에서는 ϕ가 만족해야 함 ($A[\phi U_{<5}\phi]$)을 검증한다고 하자. 이를 위해서는 먼저 시간의 경과를 기억하기 위한 변수 (s_{clk})를 모델에 추가하는데 크기가 5+1인 이 정수형 변수는 시간의 증가에 따라 그 값이 1씩 증가하게 된다. 다음으로 TCTL 특성을 CTL 특성으로 변환한다. $A[\phi U_{<5}\phi]$은 $\otimes s_{clk}.\, s_{clk}=0 \wedge A[\phi U(\phi \wedge s_{clk}<5)]$으로 변환된다. 여기서, $\otimes$는 존재적 추상화(existential abstraction)를 나타내는 새로운 연산자이다. 즉, 변환된 특성은 $s_{clk}=0$을 만족하면서 동시에 $A[\phi \wedge U(\phi \wedge s_{clk}<5)]$를 만족하는 상태들을 중간 결과로 구한다. 마지막으로 이러한 중간 결과에 $\otimes$ 연산을 적용하여 중간결과의 상태들과 s_{clk}이외에는 동일한 상태들인 최종 결과를 구하게 된다.

이러한 접근법은 변환 과정에서 추가되는 변수가 모델의 상태공간을 증가시키는 단점이 있다. 추가되는 변수의 크기는 검증할 특성에 따라 결정되는데 경우에 따라서는 이러한 변수의 크기가 모델 체킹을 불가능하게 할 가능성이 있다. 예를 들어, 단위시간이 500msec인 어떤 시스템의 특정 작업이 일주일마다 일어나는지를 검사하려면 500msec 단위로 일주일을 세어야 하는데 그러기 위해서는 매우 큰 변수가 필요하게 된다. 이는 경우에 따라서는 상태 폭발을 야기해 검증을 불가능하게 할 수도 있다.

3. 실시간 검증

3.1 실시간 모델 체킹

본 연구에서 실시간 특성은 CTL을 시간의 범위를 추가할 수 있도록 확장한 실시간 CTL(Real Time CTL)[5]로 기술된다. 본 연구에서는 이산적인 시간(discrete time)을 가정하므로 시간의 범위는 자연수의 쌍으로 주어지게 된다. 예를 들어 [3,5]는 3, 4, 5초를 가리키는 것이며, 3.5초 등의 시간은 존재하지 않는다.

3.1.1 실시간 모델

실시간 모델은 확장된 크립케 구조(Kripke Structure)인 $M=\langle Q, Q_0, L, R, time\rangle$이며 각 구성요소는 다음과 같이 정의된다.

- Q는 상태의 유한한 집합이며,
- $Q_0 \subseteq Q$는 초기 상태의 집합이며,
- $L: S \rightarrow 2^{AP}$은 상태의 레이블을 정의하는 함수로, 주어진 상태에서 만족하는 원소 명제(atomic proposition)들의 집합을 반환한다.
- $R \subseteq Q \times Q$은 전이의 집합으로 $\forall q \in Q.\, \exists q' \in Q.\, (q, q') \in R$이 항상 만족해야 한다. R은 시간을 증가시키는 unit-length 전이의 집합 R_{unit}와 증가시키지 않는 zero-length 전이의 집합 R_{zero}로 나뉜다. 즉, $R = R_{unit} \cup R_{zero}$이며,
  - $R_{unit} = \{(q, q') \mid (q, q') \in R \wedge q \models time\}$
  - $R_{zero} = R - R_{unit}$
- $time \in AP$은 시간 프레디킷으로 이것이 만족하는 상태에서만 시스템 시간이 증가한다.

여기서 AP는 모든 원소 명제들의 집합이다.

그림 5는 어떤 실시간 모델의 경로 중 하나를 표시한

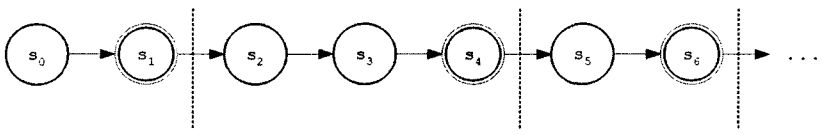


그림 5 실시간 모델의 경로 예시

것이며 이중원으로 표시된 것이 *time*을 만족하는 상태이다. 수직선은 시간을 증가시키는 unit-length 전이를 지나가며, 수직선으로 분할된 부분의 상태들은 동시간대의 상태들이 된다. 예를 들어, s_0와 동일한 시간대의 상태는 s_1이고, 초기상태에서 1단위의 시간이 소요되는 상태들은 $\{s_2, s_3, s_4\}$이며 2 단위의 시간이 소요되는 상태들은 $\{s_5, s_6\}$이다.

실시간 모델이 동기적 스테이트차트를 기반으로 하는 경우 *time*은 항상 만족하게 되며, 따라서 모든 전이는 unit-length 전이가 된다. 반면, 비동기적 스테이트차트의 경우에 *time*은 시스템이 안정적인 상태인 경우에만 만족하게 되며, unit-length 전이와 zero-length 전이가 모두 가능하다.

3.1.2 실시간 특성의 기술

3.1.2.1 실시간 CTL 문법(Syntax)

실시간 CTL(RTCTL)은 CTL을 제한된(Bounded) EU와 제한된 EG를 추가하여 확장한 것으로 그 형식은 다음과 같이 정의된다.

$$\phi ::= p \mid \neg\phi \mid \phi \wedge \psi \mid \mathbf{EX}\phi \mid \mathbf{EG}\phi \mid \mathbf{E}[\phi \mathbf{U} \psi] \mid \mathbf{EG}_{[a,b]}\psi \mid \mathbf{E}[\phi \mathbf{U}_{[a,b]} \psi]$$

여기서 $p \in AP$, AP는 모든 원소 명제들의 집합이며, $a, b \in N$이며 ϕ, ψ는 RTCTL 식이다. $[a, b]$ 시간의 범위로 a, b를 포함한 a, b사이의 자연수로 표현되는 시간을 나타낸다. 그러므로 항상 $b \geq a \geq 0$ 이어야 한다. 나머지 시제 연산자는 다음의 항등관계를 이용하여 간접적으로 정의할 수 있다.

- $\mathbf{EF}\phi \equiv \mathbf{E}[True \mathbf{U} \phi]$
- $\mathbf{AX}\phi \equiv \neg\mathbf{EX}\neg\phi$
- $\mathbf{AF}\phi \equiv \neg\mathbf{EG}\neg\phi$
- $\mathbf{AG}\phi \equiv \neg\mathbf{EF}\neg\phi$
- $\mathbf{A}[\phi \mathbf{U} \psi] \equiv \neg\mathbf{E}[\neg\psi \mathbf{U} \neg\phi \wedge \neg\psi] \wedge \neg\mathbf{EG}\neg\psi$
- $\mathbf{EF}_{[a,b]}\phi \equiv \mathbf{E}[True \mathbf{U}_{[a,b]} \phi]$
- $\mathbf{AF}_{[a,b]}\phi \equiv \neg\mathbf{EG}_{[a,b]}\neg\phi$
- $\mathbf{AG}_{[a,b]}\phi \equiv \neg\mathbf{EF}_{[a,b]}\neg\phi$
- $\mathbf{A}[\phi \mathbf{U}_{[a,b]} \psi] \equiv \neg\mathbf{EF}_{[0,a-1]}\neg\phi \wedge \neg\mathbf{E}[\neg\psi \mathbf{U}_{[a,b]} \neg\phi \wedge \neg\psi] \wedge \neg\mathbf{EG}_{[a,b]}\neg\psi$

3.1.2.2 실시간 CTL 의미구조

RTCTL의 의미구조(semantics)는 실시간 모델을 기준으로 정의된다. 실시간 모델에서 특정 상태의 시간을 구하는 함수 $Time(\rho, i)$은 경로와 해당 상태의 위치를 입력받아 그 상태의 시간을 반환한다. 어떤 상태의 시간은 그 경로의 초기 상태에서 해당 상태까지 오는 도중에 지나온 unit-length 전이의 개수로 계산된다.

RTCTL 연산자들의 의미는 실시간 모델에서 다음과 같이 정의된다.

- $q \models p$ iff $p \in L(q)$
- $q \models \neg\phi$ iff $\neg(q \models \phi)$
- $q \models \phi \wedge \psi$ iff $q \models \phi$ and $q \models \psi$
- $q \models \mathbf{EX}\phi$ iff for some q-path ρ, $\rho(1) \models \phi$
- $q \models \mathbf{EG}\phi$ iff for some q-path ρ, $\rho(i) \models \phi$ for all $i \geq 0$
- $q \models \mathbf{E}[\phi \mathbf{U} \psi]$ iff for some q-path ρ, $\exists i. i \geq 0 \wedge \rho(i) \models \psi$ and $\forall j. 0 \leq j < i \rightarrow \rho(j) \models \phi$
- $q \models \mathbf{EG}_{[a,b]}\phi$ iff for some q-path ρ, $\rho(i) \models \phi$ for all i such that $a \leq Time(\rho, i) \leq b$
- $q \models \mathbf{E}[\phi \mathbf{U}_{[a,b]} \psi]$ iff for some q-path ρ, $\exists i. a \leq Time(\rho, i) \leq b \wedge \rho(i) \models \psi$ and $\forall j. 0 \leq j < i \rightarrow \rho(j) \models \phi$

즉, $q_0 \models EG_{[a,b]}\phi$는 q_0에서 시작하는 경로들 중에 $[a, b]$ 시간범위에 속하는 모든 상태에서 ϕ가 만족하는 경로가 존재한다는 의미이다. 그리고 $q_0 \models E[\phi U_{[a,b]} \psi]$는 q_0에서 시작하는 경로들 중에 $[a, b]$ 시간범위에 도달 가능한 상태에서 ψ가 만족하고 그때까지의 모든 상태에서 ϕ가 만족하는 경로가 존재한다는 의미이다. 여기서, $q_0 \models E[\phi U_{[a,b]} \psi]$는 ψ가 만족하고 그때까지의 모든 상태에서 ϕ가 만족하는 경로가 존재해야 한다는 점에서 $q_0 \models E[\phi U \psi]$와 동일하나 ψ가 만족하는 상태가 반드시 $[a, b]$ 시간 범위 내에 있어야 한다는 점에서 차이가 있다.

3.1.3 실시간 모델체킹 알고리즘

위에서 정의된 의미구조를 기반으로 $E[\phi U_{[a,b]} \psi]$와 $EG_{[a,b]}\phi$를 만족하는 상태를 구하는 알고리즘을 제안한다. 이후로는 $E[\phi BUa..b\psi]$ 또는 EBU는 $E[\phi U_{[a,b]} \psi]$를, $EBGa..b\phi$ 또는 EBG는 $EG_{[a,b]}\phi$를 가리키는 것으로 사용한다.

3.1.3.1 EBG 알고리즘

그림 6은 $EG_{[a,b]}\phi$를 만족하는 상태들을 구하는 고정점 공식(fixpoint equation)이다. $EG_{[a,b]}\phi$는 ϕ, a, b를 받는 함수이며, a, b의 값에 따라 (a)-(c) 중 하나의 수식이 적용된다. 예를 들어, $EG_{[2,4]}f$는 $(a>0, b>0)$이므로 수식 (a)가 적용된다. 그런데 (a)를 계산하기 위해서는 $EG_{[1,3]}f$가 필요하다. 따라서 $EG_{[1,3]}f$의 계산을 위해 동일한 함수를 호출하게 된다. 이러한 과정은 $EG_{[1,3]}f \rightarrow EG_{[0,2]}f \rightarrow EG_{[0,1]}f \rightarrow EG_{[0,0]}f$의 경로를 따라가게 되는데, $EG_{[0,0]}f$은 추가적인 호출이 필요하지 않다. 따라서

$EG_{[2,4]}f$는 그림 6의 (a),(b),(c)를 차례로 적용하며 재귀적으로 구해진다.

이때, 각각의 하위 솔루션들은 모두 고정점으로 해석된다. 예를 들어, $EG_{[1,3]}f$는 $EG_{[0,2]}f$가 미리 정해진 상태에서 $EG_{[1,3]}f$을 받아서 $EG_{[1,3]}f$을 반환하는 함수의 고정점을 구하는 것으로 해석할 수 있다.

if $(a > 0 \wedge b > 0)$:
(a) $\mathbf{EG}_{[a,b]}\phi = (time \wedge \mathbf{EX}\,\mathbf{EG}_{[a-1,b-1]}\phi) \vee (\neg time \wedge \mathbf{EX}\,\mathbf{EG}_{[a,b]}\phi)$

else if $(b > 0)$:
(b) $\mathbf{EG}_{[0,b]}\phi = (time \wedge \phi \wedge \mathbf{EX}\,\mathbf{EG}_{[0,b-1]}\phi) \vee (\neg time \wedge \phi \wedge \mathbf{EX}\,\mathbf{EG}_{[0,b]}\phi)$

else:
(c) $\mathbf{EG}_{[0,0]}\phi = (time \wedge \phi) \vee (\neg time \wedge \phi \wedge \mathbf{EX}\,\mathbf{EG}_{[0,0]}\phi)$

그림 6 $EG_{[a,b]}\phi$ 를 구하는 고정점 공식

우리는 구조적 귀납법과 고정점 귀납법을 사용하여 EBU와 EBG의 알고리즘의 정확성(Correctness)과 완전성(Completeness)을 증명하였다[13]. 증명은 지면의 제약으로 생략한다.

```
01:   proc ebg(g,a,b)
02:   /** (a = b = 0)*/
03:       Y0 := g & time;                          // (c)의 알부분
04:       do
05:           Z := Y;
06:           Y:= Y0 | (! time & g & EX(Y));       // (c)의 뒷부분
07:       while (Z != Y)
08:   /** (b > a = 0)*/
09:       for (i:=0; i<b-a; i++)
10:           Y0:= time & g & EX(Y);               // (b)의 알부분
11:           do
12:               Z := Y;
13:               Y:= Y0 | (! time & g & EX(Y)); // (b)의 뒷부분
14:           while (Z != Y)
15:   /** (b > a > 0)*/
16:       for (i:=0; i<a; i++)
17:           Y0:= time & EX(Y);                   // (a)의 알부분
18:           do
19:               Z := Y;
20:               Y:= Y0 | (! time & EX(Y));       // (a)의 뒷부분
21:           while (Z != Y)
22:
23:       return Y;
```

그림 7 $EG_{[a,b]}\phi$를 계산하는 의사코드

그림 7은 $EG_{[a,b]}\phi$를 만족하는 상태의 집합을 구하는 알고리즘을 의사코드로 기술한 것이다. 대응되는 그림 3.2의 수식은 주석으로 표시하였다. 그림 7에서 time은 *time* 프래디킷이 만족하는 상태의 집합을 가리키며, &, |, ! 는 교집합, 합집합, 여집합을 구하는 연산을 가리킨다. 실제 구현에 있어 상태의 집합을 나타내는 데이타 구조로 BDD[14]를 이용하게 되므로 이들 연산은 BDD 연산이 된다.

3.1.3.2 EBU 알고리즘

그림 8은 $E[\phi U_{[a,b]}\psi]$를 만족하는 상태의 집합을 구하는 공식이며, 크게 세부분으로 이루어져 있다. 첫 번째는 $(0<a<b)$인 경우에 $E[\phi U_{[a,b]}\psi]$를 구하는 고정점 공식이며, 두 번째는 $(a=0, b>0)$인 경우에, 마지막은 $(a=b=0)$인 경우에 $E[\phi U_{[a,b]}\psi]$를 구하는 고정점 공식이다. 그런데 $(0<a<b)$의 경우에 $E[\phi U_{[a,b]}\psi]$를 구하는 과정에서 나머지 두 가지 공식이 사용된다. 예를 들어 $E[\phi U_{[2,3]}\psi]$은 첫 번째 공식으로 $E[\phi U_{[2,3]}\psi]$, $E[\phi U_{[1,2]}\psi]$를 구하고, 두 번째 공식으로 $E[\phi U_{[0,1]}\psi]$을, 마지막 공식으로 $E[\phi U_{[0,0]}\psi]$은 구하게 된다.

if $(a > 0 \wedge b > 0)$:
(a) $\mathbf{E}[\phi\mathbf{U}_{[a,b]}\psi] = (time \wedge \phi \wedge \mathbf{EXE}[\phi\mathbf{U}_{[a-1,b-1]}\psi]) \vee (\neg time \wedge \phi \wedge \mathbf{EXE}[\phi\mathbf{U}_{[a,b]}\psi])$

else if $(b > 0)$:
(b) $\mathbf{E}[\phi\mathbf{U}_{[0,b]}\psi] = \psi \vee (time \wedge \phi \wedge \mathbf{EXE}[\phi\mathbf{U}_{[0,b-1]}\psi]) \vee (\neg time \wedge \phi \wedge \mathbf{EXE}[\phi\mathbf{U}_{[0,b]}\psi])$

else:
(c) $\mathbf{E}[\phi\mathbf{U}_{[0,0]}\psi] = \psi \vee (\neg time \wedge \phi \wedge \mathbf{EXE}[\phi\mathbf{U}_{[0,0]}\psi])$

그림 8 $E[\phi U_{[a,b]}\psi]$를 구하는 고정점 공식

```
01:   proc ebu(f,g,a,b)
02:   /** (a = b = 0)*/
03:       Y0:= g;                                  // (c)의 알부분
04:       do
05:           Z := Y;
06:           Y := Y0 | (!time & f & EX(Y));       // (c)의 뒷부분
07:       while (Z != Y);
08:   /** (b >  a = 0)*/
09:      for (i := 0; i < b-a; i++)
10:           Y0:= g | (time & f & EX(Y));         // (b)의 알부분
11:           do
12:               Z := Y;
13:               Y := Y0 | (!time & f & EX(Y)); // (b)의 뒷부분
14:           while (Z != Y);
15:   /** (b > a > 0)*/
16:       for (i:=0; i<a; i++)
17:           Y0:= time & f & EX(Y);               // (a)의 알부분
18:           do
19:               Z := Y;
20:               Y := Y0 | (!time & f & EX(Y)); // (a)의 뒷부분
21:           while (Z != Y)
22:
23:       return Y;
```

그림 9 $E[\phi U_{[a,b]}\psi]$를 계산하는 의사코드

그림 9는 이러한 알고리즘을 의사코드로 기술한 것이다. 위의 알고리즘은 $E[\phi U_{[0,0]}\psi]$, $E[\phi U_{[0,b]}\psi]$, $E[\phi U_{[a,b]}\psi]$를 각각 구하는 세부분으로 나누어진다. 각 부분에서는 이전 단계의 결과를 토대로 고정점을 구하게 된다.

3.2 실시간 계산

3.2.1 최단시간 계산

최단시간 계산 함수(*MIN*[*start*, *final*])는 시작 조건(*start*)과 종료 조건(*final*)이 주어진 경우, 시작 상태에서 종료 상태에까지 도달하는 최단시간을 계산한다. 이를 위해 시작 상태에서 출발하여 각각의 시간대에 도달 가능한 상태들을 모두 모으고, 그 상태들 중에 종료조건을 만족하는 상태가 있는지 검사한다. 이러한 작업은 그러한 상태가 발견될 때까지 계속된다. 만약 종료조건이 만족되는 상태를 발견하면 그 시간대가 최단 시간이 되지만, 종료상태를 찾지 못한 상태에서 모으고 있는 상태

의 집합이 더 이상 증가하지 않는다면 시작 상태에서 종료 상태로 가는 경로가 없다고 보고 계산을 종료한다. 계산 과정에서 보유하는 상태의 집합은 크기가 점차로 커지나, 유한모델을 대상으로 하므로 상태의 집합이 무한정 크질 수는 없다. 따라서 이 알고리즘은 항상 종료된다.

```
01:   proc min(start, final)
02:       i:=0;                                  // 초기화
03:       RP = start;                            // 시작 조건
04:       do
05:           R0 := RP;
06:           do
07:               R1 := RP;
08:               RP := RP | FW(RP & !time);     // 같은 시간에 도달
09:           while (R1 != RP);                  // 같은 시간 도착 모델체커기
10:           if (RP & final != null) return i; // 도착조건 만족하면 종료
11:           i++;                               // 한단위 시간 증가
12:           RP := RP | FW(RP & time);          // 다음 시간에 도달
13:       while (R0 != RP);                      // 더이상 증가하지 않을때까지
14:       return infinity;                       // 불가능한 경우
```

그림 10 최단시간 계산 알고리즘

최단시간을 계산하는 알고리즘을 의사코드로 나타내면 그림 10과 같다. 여기서 FW(p)는 p가 만족하는 상태의 집합에서 도달 가능한 다음 상태들의 집합을 구하는 함수이며, time은 *time* 프레디킷이 만족하는 상태의 집합을 가리킨다. 위의 연산은 실제로는 BDD간의 연산으로 수행되므로, i를 제외한 start, final, Rp, R0, R1 등은 모두 BDD로 인코딩되는 상태의 집합들이다.

3.2.2 최장시간 계산

최장시간 계산 함수(*MAX*[*start*, *final*])는 시작조건(*start*)과 종료조건(*final*)이 주어진 상태에서 시작상태에서 종료상태에 도달하기위해 소요되는 최장시간을 구한다. 이를 위해 모든 상태들에 도달하는 경로의 길이를 동시에 계산하게 되는데, 시간이 증가시키면서 증가된 시간 또는 그보다 짧은 시간에 도달 가능한 상태들을 제외하므로 계산 과정에서 보유하는 상태의 집합은 계속 줄어들게 된다. 우리는 상태의 집합이 시작 조건을 만족하는 상태들을 하나도 포함하지 않는 상황이 되면 계산을 종료함으로써 시작조건에서 종료조건까지 소요되는 최장시간을 계산할 수 있다. 이 알고리즘은 계산과정에서 보유하는 상태의 집합이 무한정 적어질 수는 없으므로 항상 종료한다.

그림 11에서 R_i는 *final*을 만족하는 상태에 도달하는 데 i단위보다 많은 시간이 소요되는 상태들의 집합이다. 예를 들어, s_1은 R_2에 속해 있으므로 s_1에서 *final*이 만족하는 상태에 도달하는 데는 적어도 3단위의 시간이 소요된다. $R_0=\{s_0, s_1, s_2, s_3, s_4, s_6\}$, $R_1=\{s_0, s_1, s_2, s_3\}$, $R_2=\{s_1, s_2\}$ 그리고 $R_4=\{s_0\}$가 된다.

우리는 R_i를 순서대로 구하면서 *start*를 만족하는 상태가 포함되어 있는지를 조사하고, 하나도 포함되지 않는 R_i가 발견되면 종료한다. 예제에서 *start*가 만족하는 상태는 R_2에는 포함되나 R_3에는 포함되지 않는다. 따라서 $MAX[start, final]=3$이 된다. 만약 *start*를 만족하는 상태를 포함하는 상황에서 더 이상 보유 상태의 집합이 줄어들지 않는 경우는 무한대를 반환하고 계산을 종료한다.

```
01:   proc max (start, final)
02:       i:=0;                                  // 초기화
03:       RP:=True;                              // 시작 상태 집합
04:       do
05:           R0:=RP;
06:           RP:=RP & (EX(RP) & time & ! final); // 이전 시간에 상태들
07:           do
08:               R1:=RP;
09:               RP:=RP | (EX(RP) & ! time & ! final); // 같은 시간에 상태들
10:           while (R1 != RP);                  // 더 도달할때까지
11:           if ((RP & start) = null) return i; // 초기상태 도달안되면 종료
12:           i++;                               // 한단위 더늘은 증가후 반복
13:       while (R0 != RP);                      // 더이상 변화없을때까지
14:       return infinity;                       // 최대값 없거나 무한대
```

그림 12 최장시간 계산 알고리즘

이 알고리즘을 의사코드로 나타내면 그림 12와 같다. 여기서 EX(p)는 p가 만족하는 상태로 도달 가능한 이전 상태들의 집합을 구하는 함수이다.

3.2.3 실시간 계산의 위트니스

실시간 계산의 위트니스는 실시간 계산 결과를 활용하는데 유용하다. 예를 들어, 실시간 계산의 결과가 요구사항을 만족시키지 못하는 경우, 우리는 실시간 계산 결과가 나온 실제 경로를 파악함으로써 문제가 되는 부분을 쉽게 알아낼 수 있고 이를 토대로 필요한 부분을 수정할 수 있을 것이다.

실시간 계산은 계산결과에 대한 위트니스를 제공하지 않지만, 우리는 이러한 경로를 실시간 모델 체킹을 이용하여 구할 수 있다. 예를 들어 $MAX[start, final]=n$ 또는 $MIN[start, final]=n$으로 나온 경우는 이들 계산의

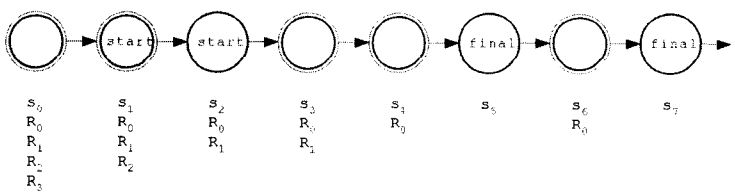


그림 11 최장시간 계산

위트니스 경로는 $AG\ !(start \Rightarrow E[(!final)\,U_{[n,n]}final])$ 특성의 반례로 구해진다.

3.3 알고리즘의 구현

알고리즘의 구현은 NuSMV[11]를 대상으로 한다. NuSMV는 심볼릭 모델체커인 SMV(Symbolic Model Verifier)를 재구현한 것으로 소스코드가 공개되어 있을 뿐 아니라 모듈별로 잘 정리되어 있어 수정이 용이하다. 또한 Campos가 제안한 방법의 동기적인 RTCTL로 기술한 특성에 대한 실시간 모델 체킹과 실시간 계산을 지원한다.

NuSMV는 C 언어로 구현되었으며 소스는 총 길이는 66,239 줄이고 주석을 제외할 경우 27,378줄이다. 이는 CMU SMV의 8,709줄/ 854줄에 비해 크게 늘어난 분량이다. 그러나 이들 분량의 대부분은 부가적인 기능을 지원하기 위한 것으로 모델 체킹이 핵심적인 부분과 관련되는 코드는 크지 않다.

우리는 기존의 NuSMV에 비동기적 스테이트차트를 위한 실시간 모델 체킹과 실시간 계산 기능을 추가로 구현하였다. 또한 *time*을 지정하는 용도로 사용하기 위해 TIME 이라는 예약어를 SMV 언어에 추가하였다.

수정된 NuSMV코드는 정확성이 증명된 알고리즘을 기반으로 하고 있으며, 구현된 코드는 이러한 알고리즘과 일대일로 대응되므로 기본적으로 구현과정에서 오류가 개입될 여지가 크지 않다. 또, 구현된 코드는 Inspection과 예외 상황을 중심으로 한 테스팅을 통해 안정성을 확인하였다.

4. 사례연구

우리의 방법론을 3장에서 설명된 두 가지 접근 방법 중에서 Brockmeyer 등의 연구와 비교하고자 한다. 왜냐하면 Campos 등의 연구의 방법론은 비동기적 스테이트차트를 다루지 못하며 동기적 스테이트차트에 대한 알고리즘은 본 연구와 유사하기 때문이다. 그리고 Brockmeyer 등의 연구와의 비교는 논문의 주제와 부합되는 부분인 실시간 모델 체킹 방법론에만 집중하고자 한다. 따라서 추상화 기법이나 변환과정에서 Statemate의 구성요소에 대한 지원 정도, 변환 도구 등의 비교대상에서 제외하였다. 또한 두 연구에서 사용한 시제 논리의 표현력의 차이로 인해 (본 연구에서는 RTCTL을 Brockmeyer 연구에서는 TCTL을 사용하고 있다) 실험에 사용한 시스템 특성의 기술에는 공통적으로 사용가능한 제한된 AU와 제한된 EU 만을 사용하였다.

본 연구에서 제안하는 방법론은 그림 13과 같다. 우리는 스테이트차트를 SMV 프로그램으로 변환하기 위해 sc2smv를 구현하였다. sc2smv는 Chan 등의 연구[15]

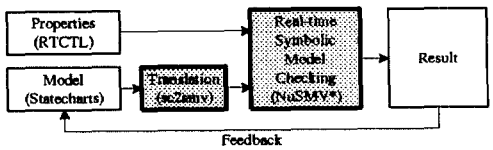


그림 13 본 논문에서 제안한 방법론

의 변환 규칙을 토대로 하여 구현되었다. 다만, RSML [16]을 대상으로 한 규칙을 스테이트차트에 맞게 수정하였다. 예를 들어, [15]의 논문에서 stable의 개념은 어떠한 이벤트도 발생하지 않는 상태로 정의되는데 우리는 이를 어떠한 이벤트도 발생하지 않으면서 어떠한 전이도 활성화되지 않은 상태로 수정하였다. 왜냐하면 스테이트차트의 경우 이벤트가 발생하지 않아도 전이가 활성화되고 선택될 수 있기 때문이다.

비교 실험을 위해, 우리는 [6,7]에서 제시된 방법론을 토대로 NuSMV를 확장하였다. 실험에서 예제로 사용된 모델은 Statemate로 작성하였으며, sc2smv를 사용하여 변환하였다. 변환된 모델은 각각의 모델 체커의 입력에 맞게 수정하여 사용하였다. 모델 체킹에는 Pentium IV 2.0 GHz의 CPU와 1GByte의 메모리를 장착한 PC가 사용되었다.

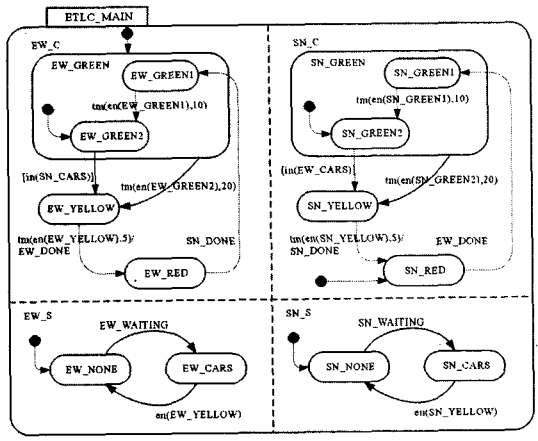


그림 14 교통 신호 제어기

그림 14는 동서방향(*EW*)과 남북방향(*SN*)으로 진행하는 도로의 교차로에서 교통신호를 제어하는 교통신호제어기(Traffic Light Controller)를 기술한 것이다. 그림 14에서 AND-state인 ETLC_MAIN은 EW_C, EW_S, SN_C, SN_S 상태를 하위상태를 가진다. 이중 위의 두 상태는 각각의 제어기를 기술한 것이고, 아래의 두 상태는 각 방향에서 차들이 나타나는 환경을 기술한 것이다.

동서방향의 신호등은 녹색에서 시작하는데, 다른 방향

에서 차가 나타날 때까지 최대 20 단위 시간까지 녹색으로 유지하다가, 차가 나타나거나 시간이 초과되면 노란색으로 바뀐다. 노란색에서 5 단위 시간을 기다린 후, 빨간색으로 바뀌면서 다른 방향의 제어기에 이 사실을 알린다. 다른 신호등은 이러한 연락을 받은 경우에만 녹색으로 바뀌게 된다.

우리는 그림 14의 모델을 비동기적 시간 모델로 해석하였다. 따라서 그림 14에서 안정된(stable) 상태란 어떠한 이벤트도 발생하지 않으면서 어떠한 전이도 활성화되지 않는 상태를 나타내며, 이러한 상태에서만 외부 이벤트가 개입되고 시간이 증가한다. 이 모델에서 우리는 다음의 세 가지 특성을 검사하였다.

1. AG (EW_DONE → AX A[in-SN_GREEN $U_{[10,30]}$ in-SN_YELLOW])
2. AG (SN_WAITING → A[True $U_{[0,20]}$ in-SN_GREEN])
3. AG (SN_WAITING → E[True $U_{[0,20]}$ in-SN_GREEN])

첫 번째 특성은 동서방향의 신호등에서 연락이 오면 남북방향의 신호등이 녹색으로 바뀌어야 하고 최소 10 단위 시간, 최대 30 단위 시간동안 녹색으로 유지되어야 한다는 의미를 가진다. 그리고 두 번째(세 번째) 특성은 남북방향으로 진행하는 차는 20 단위 시간 이내에 녹색 신호를 받아야 (받을 수 있어야) 한다는 것을 나타낸다.

실험 결과는 표 1과 그림 15에 나타나 있다. 여기서 A는 본 연구에서 제안된 방법론을 가리키며, B는 Brockmeyer 등에 의해 제안된 방법론을 가리킨다. A_R은 시작 상태에서 도달 가능한 상태들만을 대상으로 한 결과로서, 도달 가능한 상태를 모으는 시간을 포함하는 결과이다.

실험 결과는 A 방법론이 비교적 적은 시간과 자원을 소모하는 것으로 나타났다. 예를 들어, 첫 번째 특성인 경우에는 B 방법론은 검사에 500 초 이상의 시간이 필요했던 반면, A 방법론은 20초 정도의 시간만이 필요하였으며, A_R 경우 채 2초도 걸리지 않았다. 이는 B 방법론이 첫 번째 특성의 검사를 위해 새로 추가한 변수가 검사할 상태공간을 30배 이상 증가시켰기 때문이다.

표 1 실험결과

No	Our Method (A)	(A_R)	Compared (B)	Ratio (B/A)
1	true	true	true	n/a
	19.8s	1.4s	505.7s	25.5
	$2^{38.7035}$	9,656	$2^{43.7035}$	32.0
2	true	true	true	n/a
	2.9s	2.3s	31.6s	10.9
	$2^{38.7035}$	9,656	$2^{43.1629}$	22.0
3	true	true	true	n/a
	2.7s	2.3s	2.1s	0.8
	$2^{38.7035}$	9,656	$2^{43.1629}$	22.0

그런데 세 번째 특성의 경우에는 오히려 B 방법론이 약간 우세한 것으로 나타나고 있다. 이것은 세 번째 특성이 존재적(existential) 특성을 포함하고 있고, 일반적으로 존재적 특성의 경우 상태 공간의 일부만을 확인하면 되기 때문에 전체 상태공간의 크기에 크게 영향을 받지 않기 때문이다. 반면, 본 연구에서 제안된 알고리즘은 기존의 알고리즘에 *time* 프레디킷이 만족하는지를 검사하는 부분이 추가되었기 때문에 필요한 시간도 소폭 증가하였다. 이러한 이유로 검사하는 상태의 수가 비교적 적은 경우에는 이러한 부담을 가진 A 방법론이 시간을 더 소모하게 되었다. 그러나 그 차이는 크지 않았다.

실험에서 알 수 있듯이 일반적으로 전이의 실행이 시간의 의해 제한되는 실시간 모델의 경우 실제 도달 가능한 상태의 수는 전체 상태의 일부에 불과하기 때문에 미리 도달 가능한 상태들을 구하고 그 상태들만을 대상으로 검사하는 기법이 유용한 경우가 많다. 그러나 B 방법론은 이러한 기법을 사용할 수 없다. 왜냐하면, B 방법론은 먼저 중간결과를 구하고 그 결과에 existential 연산을 취해 최종결과를 구하게 되는데, 중간결과에 포

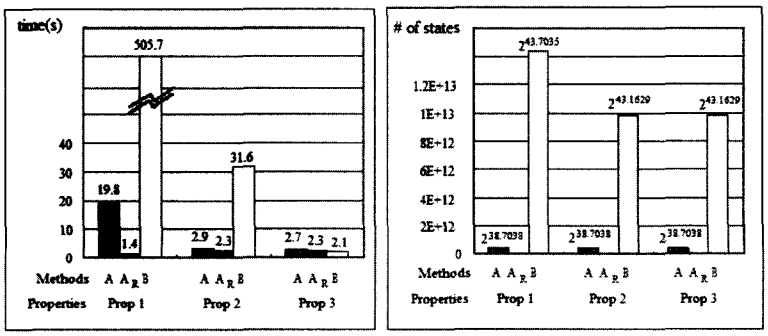


그림 15 사용된 자원의 비교

함된 도달 불가능한 상태들도 최종결과를 계산하는데 중요한 역할을 하기 때문이다. 따라서 도달 가능한 상태들만을 대상으로 B 방법론을 적용하는 경우 이러한 중간결과 상태들이 제외되므로 결국 잘못된 결과를 도출할 수 있다.

5. 결론 및 향후연구방향

스테이트차트는 실시간 반응형 시스템을 기술하는데 널리 사용되는 정형 명세 언어이며 비동기적 스테이트차트는 반응형 소프트웨어 시스템을 기술하는데 많이 사용된다. 그러나 현재의 모델 체커로는 이러한 비동기적 스테이트차트에서 실시간 특성에 대한 검사가 불가능하다. 본 연구는 이러한 문제를 해결하기 위해 모델 체킹 알고리즘을 제안하고 모델 체커를 수정하였다.

본 연구에서 제안하는 실시간 특성에 따라 모델 자체를 수정하는 방법은 기존의 방법에 비해 모델의 상태 공간을 증가시키지 않으며 검증할 특성에 따라 모델을 변경하지 않아도 된다는 장점을 가진다.

본 연구에서 수정한 모델 체커는 동기적 시간 모델을 검증하는데 있어서도 변경전과 동일한 효율성을 가지고 오류 없이 작동함을 확인하였으며, 사례 연구는 이러한 실시간 모델 체킹이 실제 유용하게 사용될 수 있음을 보여 준다.

또한, 실시간 계산과 계산 결과에 따른 위트니스의 생성은 특정 시스템이 어떤 부분에서 요구하는 시간을 초과하는지에 대한 정보를 제공함으로써 모델의 디버깅에 도움을 줄 수 있다.

향후 연구 방향은 다음과 같다. 실시간 계산 알고리즘을 확장하여 최단시간과 최장시간 이외의 다른 시간 정보 및 횟수에 관한 정보를 얻을 수 있도록 할 계획이다. 예를 들어 주어진 시간 범위 내에 특정 조건이 만족하는 상태는 최대 횟수 혹은 최대 지속시간 등에 대한 정보를 얻을 수 있다면 실시간 시스템이 분석에 도움을 줄 수 있을 것으로 생각된다.

다음으로 모델체킹에 의한 테스트 생성을 지원하기 위해 모델 체킹의 위트니스 생성 알고리즘을 강화함으로써 좀 더 많은 정보를 가진 위트니스를 생성할 수 있게 할 계획이다. 예를 들어 하나의 특성에 대해 여러개의 반례를 생성한다든지, 여러개의 특성에 대한 반례를 하나로 포괄하는 등의 작업을 통해 테스트 생성을 효율화할 수 있을 것이다.

참 고 문 헌

[1] D Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231-274, 1987.

[2] S.V. Campos and E. Clarke. Real-Time Symbolic Model Checking for Discrete Time Models. In T. Rus and C. Rattray, editors, *Theories and Experiences for Real-Time System Development.* World Scientific Press, AMAST Series in Computing, 1994.

[3] S. Campos, E. Clarke, W. Marrero, M. Minea, and H. Hiraishi. Computing quantitative characteristics of finite-state real-time systems. In *Proceedings of the Real-Time Systems Symposium.* IEEE Computer Society Press, December 1994.

[4] S. Campos, E. Clarke, and M.Minea. The verus tool: A quantitative approach to the formal verification of real-time systems. In O. Grumberg, editor, *Proc. 9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254, pages 452-455. Springer Verlag, 1997.

[5] E. Allen Emerson, A.K. Mok, A.P. Sistla, and Jai Srinivasan. Quantitative temporal reasoning. In *Lecture Notes in Computer Science*, volume 531, pages 136-145, 1990.

[6] Udo Brockmeyer and Gunnar Wittich. Tamagotchis need not die-verification of STATEMATE designs. *Tools and Algorithms for the Construction and Analysis of Systems(TACAS'98)*, 1998.

[7] Udo Brockmeyer and Gunnar Wittich. Real-time verification of STATEMATE designs. In *Lecture Notes in Computer Science*, pages 537-541, 1998.

[8] Pnueli A. A temporal logic of programs. *Theoretical Computer Science*, 13:45-60, 1981.

[9] David Harel and Amnon Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293-333, 1996.

[10] Sergio Vale Aguiar Campos, Edmund M. Clarke, Wilfredo R. Marrero, and Marius Minea. Verus: A tool for quantitative analysis of finite-state real-time systems. In *Workshop on Languages*, Compilers and Tools for Real-Time Systems, pages 70-78, 1995.

[11] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NUSMV: A new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4): 410-425, 2000.

[12] Sergio Campos, Marcio Teixeira, Marius Minea, Edmund Calrke, and Andreas Kuehlmann. Model checking semi-continuous time models using bdds. In *Proceedings of the International Workshop on Symbolic Model Checking*, 1999.

[13] H.J. Bang. *Extending SMV for Real-time Verification of Statecharts.* Master's thesis, Korea Advanced Institute of Science and Technology, 2003.

[14] K. McMillan. Symbolic Model Checking. Kluwer, 1993.

[15] William Chan, Richard J. Anderson, Paul Beame, Steve Burns, Francesmary Modugno, David Notkin, and Jon Damon Reese. Model checking large software specifications. *Software Engineering*, 24(7):498-520, 1998.

[16] N. Leveson, M. Heimdahl, H.Hildreth, and J.Reese. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, 20(9), 1994.

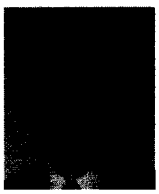
[17] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244-263, 1986.

[18] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. The MIT Press, 1999.

[19] R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. In *Proceedings of the 5th Symposium on Logic in Computer Science*, pages 414-425, Philadelphia, 1990.

[20] G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 35(5):279-295, 1997.

[21] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1-33, Washington, D.C., 1990. IEEE Computer Society Press.



방 호 정

1996년 서울대학교 경제학과 학사. 2003년 KAIST 전자전산학과 전산학전공 석사. 2003년~현재 KAIST 전자전산학과 전산학전공 박사과정. 관심분야는 소프트웨어공학, 정형기법

홍 형 석

1993년 KAIST 전자전산학과 전산학전공 학사. 1995년 KAIST 전자전산학과 전산학전공 석사. 2001년 KAIST 전자전산학과 전산학전공 박사. 2001년~2002년 University of Pennsylvania, Post Doc. 2002년~2003년 KAIST AITrc, 방문 연구원. 2003년~2004년 University of Pennsylvania, Research Associate. 관심분야는 정형기법, 모델체킹, 프로그램 분석

김 태 효

1998년 KAIST 전자전산학과 전산학전공 학사. 2000년 KAIST 전자전산학과 전산학전공 석사. 2000년~현재 KAIST 전자전산학과 전산학전공 박사과정. 관심분야는 정형기법, 모델체킹

차 성 덕

1983년 University of California, Irvine 전산학 학사. 1986년 University of California, Irvine 전산학 석사. 1991년 University of California, Irvine 전산학 박사. 1994년~2001년 KAIST 전자전산학과 전산학전공 조교수. 2001년~현재 KAIST 전자전산학과 전산학전공 부교수. 관심분야는 정형기법 및 명세, 정보보호, 침입탐지