모델 체킹에서 안전성 위반에 대한 효율적인 반례 생성

이 태 훈† · 권 기 현††

요 약

모델 체킹은 주어진 모델과 속성간의 만족성 관계를 검사한다. 만일 모델이 속성을 만족하지 않는 경우, 모델 체킹은 그 이유를 담은 반례를 생성한다. 반례는 모델의 디버깅에 사용되며 모델을 이해하는데 도움을 주기 때문에, 반례 생성은 모델 체킹의 필수 구성 요소 중의 하나이다. 본 논문에서는 모델 체킹에서 안전성 속성이 위반되었을 때 그에 대한 반례를 효율적으로 생성하는 방법을 제시하였고, 푸쉬 푸쉬 게임 풀이에 제안한 방법을 적용했다. 그 결과, 기존 NuSMV로는 전체 50게임 중에서 42개임밖에 풀지 못했으나 본 논문의 방법으로는 50게임을 모두 풀었다. 뿐만 아니라, 반례 생성에 소요된 시간과 메모리 사용량이 기존 NuSMV에 비해서 각각 86%와 62% 개선되었다.

Efficient Counterexample Generation for Safety Violation in Model Checking

Tae-hoon Lee† · Gi-hwon Kwon††

ABSTRACT

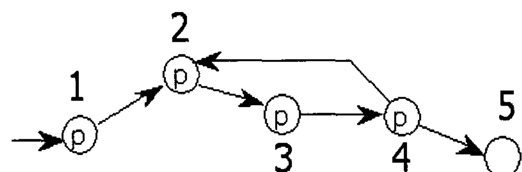
Given a model and a property, model checking determines whether the model satisfies the property. In case the model does not satisfy the property, model checking gives a counterexample which explains where the violation occurs. Since counterexamples are useful for model debugging as well as model understanding, counterexample generation is one of the indispensable components in the model checking tool. This paper presents efficient counterexample generation techniques when a safety property is falsified. These techniques are used to solve Push Push games which consist of 50 games. As a result, all the games are solved with the proposed techniques. However, with the original NuSMV, 42 games are solved but 8 failed. In addition, we obtain 86% time improvement and 62% space improvement compared to the original NuSMV in solving the game.

키워드 : 모델 체킹(model checking), 속성 위반(property violation), 반례 생성(counterexample gneration), 상태 공간 탐색(state space traversal)

1. 서 론

정형 검증 기법중의 하나인 모델 체킹은 1986년 Clarke에 의해 소개되었으며[1], 그 후 약 20년 동안 산업체에서 많은 인기를 누려왔다. 모델 체킹은 하드웨어 검증[2], 소프트웨어 검증[3], 그리고 프로토콜 검증[4] 등에 전통적으로 사용되어 왔을 뿐만 아니라 최근에는 게임에서 풀이 경로 찾기[5], 인공지능에서 계획 수립[6] 등 다양한 분야에 활용되고 있다. 모델 체킹이 이렇게 인기를 누리며 다양한 분야에 활용되는 이유중의 하나로서 반례 생성을 꼽을 수 있다.

모델 체킹은 유한 상태 기계로 표현된 모델과 시제 논리식으로 기술된 속성을 받아서 모델과 속성간의 만족성 관계를 결정한다. 만일 모델이 속성을 만족하지 않는 경우, 모델 체킹은 그 이유를 설명하는 반례를 제공한다. 예를 들어 (그림 1)의 모델에는 5개의 상태가 있는데, 1번 상태는 초기상태이며 5번 상태를 제외한 나머지 상태는 모두 p를 만족한다고 가정하자. 이 모델에 대해서 CTL(Computation Tree Logic, [7]) 논리식으로 표현된 속성 AG p를 모델 체킹해 보자. AG p의 의미가 '도달 가능한 모든 상태에서 항상 p'이기 때문에, 주어진 속성은 만족되지 않는다. 이 경우, 모델 체킹은 경로 1, 2, 3, 4, 5를 반례로 출력한다. 즉 초기상태(1번 상태)로부터 p가 만족되지 않는 최초상태(5번 상태)로 도달되는 경로를 보임으로서, 속성 AG p가 만족되지 않는 이유를 설명한다.



(그림 1) 모델 체킹에서 반례의 예

† 준 회 원 : 경기대학교 정보과학부
†† 종신회원 : 경기대학교 정보과학부 교수
논문접수 : 2004년 10월 22일, 심사완료 : 2004년 12월 14일

살펴본 바와 같이, 주어진 속성이 위반되었을 때 생성되는 반례는 모델 디버깅에 유용한 정보를 제공하기 때문에, 반례 생성은 모델 체킹의 필수 구성 요소 중의 하나이다. 그래서 NuSMV(New Symbolic Model Verifier, [8]) 등 모든 모델 체킹 도구는 반례 생성 기능을 제공한다. 그러나 이들 모델 체킹 도구는 정형 긍정[1]과 정형 부정[2]에 모두 사용되도록 범용적으로 제작되었기 때문에, 특별히 정형 부정 목적으로 모델 체킹 도구를 사용하는 경우 반례 생성이 비효율적인 경우가 종종 있다. 예를 들어, Chan[9]은 항공기 충돌 방지 시스템이 안전성 속성을 위반하고 있음을 입증하기 위해서 NuSMV를 사용하였다. 그 결과, 안전성 속성이 위반되었음을 판정하는 데에는 불과 수초밖에 소요되지 않았지만, 그에 대한 반례를 찾는 데에는 수 시간이 소요되는 것을 경험하였다. 항공기 충돌 방지 시스템의 예와 같이 정형 부정 목적으로 모델 체킹을 사용하려는 의도가 최근 증가하고 있기 때문에, 이에 대한 효율적인 반례 생성 연구가 요구된다.

본 논문에서는 정형 부정을 위한 효율적인 반례 생성 기법을 제시한다. 특히, 본 연구에서 다루는 속성의 범위는 안전성 속성으로 제한한다. 안전성 속성으로 제한한 이유는, Dwyer[10]가 연구 조사에서 밝혔듯이 모델 체킹에서 사용하는 대부분의 속성이 안전성과 관련 있기 때문이다. 안전성 속성이 위반되었을 때 이에 대한 반례를 효율적으로 생성하기 위해 본 논문에서는 다음과 같은 4가지 방법을 제시했다. 첫째, 모델 추상화를 통해서 반례 생성시 고려해야 할 탐색 공간을 축소했다. 그 결과 축소된 탐색 공간에서 반례를 찾아내었다면 원래 탐색 공간에서도 반례가 당연히 존재한다. 둘째, 주어진 속성의 만족성 여부를 판정할 때 사용했던 중간 값을 저장하여 재사용했다. 그 결과 상태 공간의 탐색 횟수를 3회에서 2회로 줄였다. 셋째, 계산된 중간 값을 하드 디스크에 저장했다. 그 결과 상태 폭발 문제를 개선했다. 넷째, 상태 공간 탐색을 위해서 정방향과 양방향 등 다양한 탐색 방법을 사용했다. 그 결과 역방향만 사용하던 기존 모델 체킹 도구들보다 반례를 효율적으로 생성할 수 있었다.

본 논문에서 제안한 반례 생성 기법을 푸쉬 푸쉬 게임 풀이에 적용하였다. 적용 사례로 푸쉬 푸쉬 게임을 선택한 이유는, 이 게임은 정형 부정의 대표적인 사례로서 모델 체킹에서 생성된 반례가 게임의 최단 풀이 경로이기 때문이다[11]. 제안한 기법을 적용하여 푸쉬 푸쉬 게임을 풀 때, 반례 생성이 얼만큼 개선되었는지를 실험하기 위하여 본 논문에서는 NuSMV 소스를 수정하였다. 기존 NuSMV로 게임을 풀었을 때, 전체 50게임 중에서 42게임을 풀었고 8게임은 풀지 못했다. 그러나 제안한 추상화 기법을 적용하여 탐색 공간을 축소한 경우 48게임을 풀었고, 추상화 이전에 비해서 반례 생성에 소요된 시간과 메모리 사용량이 각각 42%와 23% 개선되었다. 또한 중간 결과를 재사용하여 탐색 횟수를 줄인 결과 총 49게임을 풀었고, 기존 NuSMV에 비해서 반례 생성에 소요된 시간과 메모리 사용량이 각각 75%와 40% 개선되었다. 계산 결과를 하드 디스크에 저장하고 다양한 탐색 방법을 사용한 결과 전체 50게임을 모두 풀었고, 반례 생성에 소요된 시간과 메모리 사용량은 기존 NuSMV에 비해서 86%와 62% 개선되었다.

논문의 구성은 다음과 같다. 2장에서는 반례 생성에 대해서 살펴본다. 3장에서는 본 논문에서 제안하는 반례 생성 기법을 설명한다. 그리고 4장에서는 제안한 방법을 적용해서 반례를 생성했을 때 시간과 메모리가 얼만큼 절감되었는지를 실험 결과를 통해서 설명한다. 5장에서 반례 생성과 관련된 연구를 살펴보고, 결론 및 향후 연구 과제를 마지막 6장에서 기술한다.

2. 배경 지식

2.1 모델 체킹

모델 체킹은 유한 상태 기계로 표현된 모델 M과 시제 논리식으로 기술된 속성 ϕ를 받아서 모델과 속성간의 만족성 관계를 결정한다. 모델 체킹에서는 크립키 구조라 불리는 모델 $M=(S, I, R, L)$을 사용한다. 여기서 S는 상태들의 집합, $I \subseteq S$ 는 초기 상태들의 집합, $R \subseteq S \times S$은 상태들 간의 전이 관계, $L: S \rightarrow 2^X$은 각 상태에서 참이 되는 단순 명제들을 해당 상태에 배정하는 함수이다. 여기서 X는 단순 명제들의 집합이다. 모델 체킹은 시스템이 갖는 무한 행위에 대해서 조사를 하기 때문에 상태들 간의 전이를 나타내는 R은 전체 관계이다. 즉 $\forall s \in S \bullet \exists s' \in S \bullet (s,s') \in R$로서, 모든 상태마다 전이할 수 있는 다음 상태가 최소한 하나 이상 존재한다. 경로 $\pi = s_1 s_2 s_3 s_4 \ldots$는 전이 가능한 상태들을 차례대로 나열한 것으로서 $(s_i, s_{i+1}) \in R$, $i \geq 1$이며 그 길이는 무한이다.

모델에 관한 속성은 모델을 트리 관점에서 해석하는 CTL 논리식으로 표현한다. 모델의 초기 상태를 루트로 해서 모델을 풀어헤치면 트리를 얻게 되며, 트리는 모델의 가능한 모든 행위를 표현한다. 모델의 속성을 정형적으로 기술하기 위해서 CTL은 두개의 경로 한정자 A(All), E(Exist)와 네 개의 시제 연산자 X(neXt), F(Future), G(Globally), U(Until)를 갖는다. 경로 한정자와 시제 연산자를 조합하면 8개의 CTL연산자 AX, EX, AF, EF, AG, EG, AU, EU를

1) 주어진 속성이 만족됨을 입증하기 위한 의도로 모델 체킹을 사용하는 경우이다.
2) 주어진 속성이 위반됨을 보이기 위한 의도로 모델 체킹을 사용하는 경우이다.

얻는다. 전 장에서 언급했듯이 안전성 속성에 대한 효율적인 반례 생성이 본 논문의 목적이기 때문에, 본 논문에서는 안전성을 나타내는 논리식 $AG\phi$을 다룬다. 이 논리식의 의미는 '도달 가능한 모든 상태에서 항상 ϕ'이며 이것의 정형적 의미는

$$M, s \models AG\phi \quad \text{iff} \quad \forall \pi = s_1, s_2, \ldots \bullet \forall i \geq 1 \bullet M, s_i \models \phi$$

이다. 여기서 $s = s_1$이다.

M의 모든 초기 상태에서 ϕ가 참인 경우를 $M \models \phi$로 표시하며 'M이 ϕ를 만족한다'라고 읽는다(반대로, 만약 모델 M이 ϕ를 만족하지 않는 경우 $M \nvDash \phi$로 표시한다). M과 ϕ를 받아서 이들 간의 만족성 관계를 결정하기 위해서, 모델 체킹은 ϕ를 만족하는 상태들의 집합을 먼저 구한 후, 이 상태 집합에 초기 상태가 포함되어 있는지를 검사한다. ϕ를 만족하는 상태 집합을 $[\![\phi]\!]$라고 표시하자. 그러면 $AG\phi$를 만족하는 상태 집합은 다음과 같이

$$[\![AG\phi]\!] = \nu Z.([\![\phi]\!] \cap pre_{\forall}(Z))$$

역 방향으로 계산된다. 여기서 함수 $pre_{\forall}(Z) = \{s \in S \mid \forall_{s' \in S} \bullet (s,s') \in R \Rightarrow s' \in Z\}$는 주어진 상태 집합 Z로만 도달되는 이전 상태들의 집합을 출력한다. 즉 역방향으로 상태들의 집합을 구한다. 주어진 CTL 논리식 ϕ에 대한 상태 집합 $[\![\phi]\!]$을 구한 후, 그 다음 작업은 초기 상태 집합 I가 $[\![\phi]\!]$의 부분 집합인지를 검사하는 것이다. 즉

$$M \models \phi \quad \text{iff} \quad I \subseteq [\![\phi]\!]$$

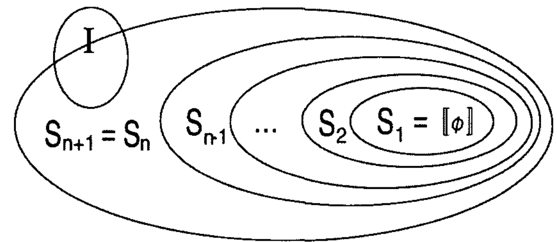
초기 상태가 ϕ를 만족하는 집합에 모두 포함되면 모델은 속성을 만족한다(지면 관계상 논문 전개에 필요한 부분만을 축약해서 모델 체킹을 소개했다. 보다 상세한 사항은 [7]를 참고하기 바란다).

2.2 반례 생성

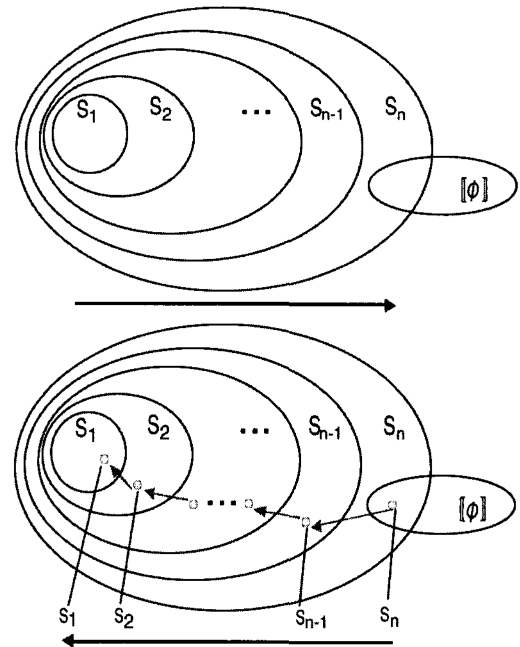
주어진 속성이 만족하지 않는 경우, 모델 체킹은 그 이유를 담은 반례를 생성한다. 여기서는 $M \nvDash AG\neg\phi$의 반례 생성을 설명한다. CTL 논리식에서 $AG\neg\phi$와 $EF\phi$는 쌍대 관계이기 때문에, $AG\neg\phi$의 반례는 $EF\phi$의 증거이다[12]. 그러므로 $EF\phi$의 증거로서 $AG\neg\phi$의 반례를 설명할 수 있다. $EF\phi$를 만족하는 상태 집합은 $[\![EF\phi]\!] = \mu Z.([\![\phi]\!] \cup pre_{\exists}(Z))$이기 때문에 $[\![EF\phi]\!]$는

$$\tau(Z) = ([\![\phi]\!] \cup pre_{\exists}(Z))$$

함수 τ의 최소 고정점이다[7]. 여기서 $pre_{\exists}(Z) = \{s \in S \mid \exists_{s' \in S} \bullet (s,s') \in R \wedge s' \in Z\}$는 Z로 도달되는 이전 상태들을 모두 출력하는 함수이다. τ의 최소 고정점은 공집합으로부터 시작해서 함수 τ를 반복적으로 적용함으로서 구할 수 있다. 즉, 함수를 적용한 순서 $\tau^1(\emptyset) \subseteq \cdots \subseteq \tau^n(\emptyset) \subseteq \cdots$는 언젠가 더 이상 증가되지 않는 상태 $\tau^n(\emptyset) = \tau^{n+1}(\emptyset)$에 이르게 되는데, 이때 $\tau^n(\emptyset)$이 함수 τ의 최소 고정점이 된다. 함수 τ의 중간 계산 값을 $S_1 = \tau^1(\emptyset)$, $S_2 = \tau^2(\emptyset)$, ..., $S_n = \tau^n(\emptyset)$라고 하자. 사실, $S_1 = \tau^1(\emptyset) = [\![\phi]\!]$이다. 왜냐하면 $pre_{\exists}(\emptyset) = \emptyset$이기 때문이다. (그림 2)에서 보듯이 최소 고정점 S_n이 초기 상태와 만나게 되면($S_n \cap I \neq \emptyset$) 논리식 $AG\neg\phi$는 거짓이다.



(그림 2) $S_n \cap I \neq \emptyset \Rightarrow M \nvDash AG\neg\phi$



(그림 3) 정방향과 역방향 탐색을 통해서 반례 생성

거짓인 경우, (그림 3)에서 보는 것처럼 두 단계를 거쳐서 반례를 생성한다. 첫 번째 단계는

S1 = I
Si+1 = post∃(Si) ∪ Si

와 같이 초기 상태에서 ϕ가 참인 상태까지 정방향 탐색을 진행하면서 도달 가능한 상태를 저장한다($S_i \cap [\![\phi]\!] \neq \emptyset$ 일 때 종료한다). 여기서 함수 $post(Q)=\{s' \in S \mid \exists_{s\in S} \bullet (s,s') \in R \wedge s' \in Q\}$는 Q로부터 도달 가능한 다음 상태들의 집합을 리턴하는 함수이다. 두 번째 단계는 ϕ가 참인 상태에서 초기 상태로 역방향으로 오면서 초기 상태에서 ϕ가 참인 상태로 도달 가능한 경로 즉 반례 $<s_1,...,s_n>$를 찾는다.

$s_n \in S_n \cap [\![\varphi]\!]$
$s_{i-1} \in pre_{\exists}(S_i) \cap S_{i-1}$

3. 효율적인 반례 생성 기법

3.1 탐색할 상태 공간 축소

안전성 속성에 대한 반례를 생성하는데 가장 기본적으로 사용되는 것이 추상화이다. 추상화의 목표는 원래 모델 M보다 크기가 더 적으면서도 M의 행위를 보존하는 축소된 모델 M'을 찾는 것이다. 행위 보존을 고려할 때 원래 모델 M과 축소된 모델 M' 사이에는 두 가지 관계가 있다. 하나는 두 모델 사이에서 행위가 강하게 보존되는 바이시뮬레이션 관계($M \equiv M'$)이며, 다른 하나는 행위가 약하게 보존되는 시뮬레이션 관계($M \leq M'$)이다[13].

특히 시뮬레이션 관계인 $M \leq M'$의 경우 모든 ACTL[3] 논리식 ϕ에 대해서 $M' \models \phi \Rightarrow M \models \phi$이 성립한다. 즉 M'에서 ϕ가 만족되면 M에서도 ϕ가 만족한다. 하지만 반대의 경우는 성립하지 않는다. 왜냐하면 M'의 행위가 M보다 더 크기 때문이다. 마찬가지로 $M' \leq M$ 인 경우 ECTL[4] 논리식 ϕ에 대해서 $M' \models \phi \Rightarrow M \models \phi$이 성립한다. 즉 M'에서 ϕ가 만족되면 M에서도 ϕ가 만족한다. 하지만 반대의 경우는 성립하지 않는다. 왜냐하면 M'의 행위가 M보다 적기 때문이다.

원래 모델 M이 주어졌을 때, 추상화의 목표는 사용할 의도에 맞게 모델을 축소하는 것이다. 우리는 상태 투영이라고 불리는 추상화 연구를 통해서 원래 모델 M보다 행위와 크기가 모두 적은 축소된 모델 $M' \leq M$을 얻었다[14]. 아래 정리는 성립한다.

정리: $M' \not\models AG\neg\phi \Rightarrow M \not\models AG\neg\phi$

증명:

1. $M' \leq M$	[14]에서 증명된 정리	
2. $M' \not\models AG\neg\phi$	전제	
3. $M' \models EF\phi$	2번과 동치	
4. $M' \models EF\phi \Rightarrow M \models EF\phi$	1번과 ECTL식의 만족 관계([13] 참조)	
5. $M \models EF\phi$	3번과 4번에 긍정 추론 규칙 적용	
6. $M \not\models AG\neg\phi$	5번과 동치	Q.E.D.

3) CTL논리식의 일부분으로서 모든 경로를 나타내는 A가 시제 연산자 앞에 나오는 논리식이다.
4) CTL논리식의 일부분으로서 어떤 경로를 나타내는 E가 시제 연산자 앞에 나오는 논리식이다.

M'은 M보다 행위가 적기 때문에 M'에서 $AG\neg\phi$가 위반되면 M에서도 당연히 $AG\neg\phi$는 거짓이다. 뿐만 아니라 M'은 M보다 크기가 적기 때문에, 더 적은 노력으로 반례를 생성할 수 있다.

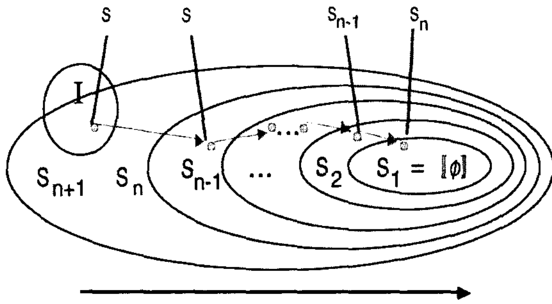
3.2 상태 공간 탐색 횟수 축소

전장에서 살펴본 바와 같이 NuSMV는 정형 긍정과 정형 부정에 모두 사용되는 범용 모델체킹 도구로서, 상태 공간을 총 3번 탐색한 후 반례를 생성한다. 첫 번째 단계는 (그림 2)와 같이 상태 공간을 역방향으로 탐색하면서 모델이 속성을 만족하는지를 검사한다. 만약 만족하면, 참을 출력하고 종료한다. 그러나 만족하지 않는 경우, 어디에서 에러가 발생되었는지를 설명하는 반례를 생성해야 한다. 반례를 생성하기 위해서 (그림 3)과 같이 정방향과 역방향으로 상태 공간을 2번 더 탐색한다. 다시 말해서, 속성을 검사할 때 이미 탐색한 상태 공간을 반례 생성시 다시 탐색하고 있다. 중복 탐색은 메모리 사용량을 증가시킬 뿐만 아니라 반례 생성 시간을 크게 지연시킨다. 특별히, 정형 부정 목적으로 모델 체킹 도구를 사용하는 경우 반례 생성이 비효율적인 경우가 종종 있다. 예를 들어, Chan[9]은 항공기 충돌 방지 시스템이 안전성 속성을 위반하고 있음을 입증하기 위해서 NuSMV를 사용하였다. 그 결과, 안전성 속성이 위반되었음을 판정하는 데에는 불과 수초 밖에 소요되지 않았지만, 그에 대한 반례를 찾는 데에는 수 시간이 소요됨을 경험하였다.

우리는 안전성 속성이 위반된 경우 그에 대한 반례를 효율적으로 생성하도록 NuSMV의 반례 생성 부분을 수정했다[15]. 기존의 NuSMV에서는 S_i를 구하기 위해서 다시 계산을 했지만, 본 논문에서 구현한 방법은 맨 처음 CTL 속성을 검사할 때 방문했던 상태들의 집합을 저장한 후 반례 생성 시 이를 재사용한다. 전장에서 살펴본 바와 같이 첫 번째 역방향 탐색에서 함수 τ의 중간 계산 값을 집합 $S_1=\tau^1(\emptyset)$, $S_2=\tau^2(\emptyset)$, ..., $S_n=\tau^n(\emptyset)$에 저장한다. 저장된 순서 $<S_1, S_2, ..., S_n>$를 이용해서, 두 번째 정방향 탐색에서는 $i<n$일 때까지 다음을 반복해서

$$s_1 \in S_n \cap I$$
$$s_{i+1} \in post_{\exists}(S_{n-(i-1)}) \cap S_{n-i}$$

반례를 생성한다. 정형 부정의 경우 기존 NuSMV는 역방향, 정방향, 그리고 다시 역방향의 순으로 상태 공간을 총 3번 탐색한다. 반면에 수정된 NuSMV는 역방향과 정방향으로 상태 공간을 총 2번 탐색한다. 첫 번째 역방향 탐색에서 저장해 두었던 계산 값을 이용하여, 두 번째 정방향 탐색에서 (그림 4)와 같이 즉시 반례를 구한다.



(그림 4) 수정된 NuSMV는 첫 번째 역방향 탐색시 계산된 중간 값을 이용하여 두 번째 정방향 탐색시 즉시 반례를 생성한다.

3.3 상태 공간 저장 방법을 변경

살펴본 바와 같이 수정된 NuSMV에서는 첫번째 역방향 탐색시 계산된 값을 $<S_1, S_2, ..., S_n>$과 같이 차례대로 보관했다가 두 번째 정방향 탐색에서 이를 이용한다. 여기에서 집합 S_i에 해당되는 상태들은 BDD(Binary Decision Diagram, [16])라는 자료 구조로 변환되어 메인 메모리에 저장된다. 즉 정방향 탐색시 S_i에 해당되는 BDD를 메인 메모리에 저장했다가, 이후 역방향 탐색시 한 번에 하나씩 사용하게 된다. 기존 NuSMV에서는 상태 집합을 나타내는 BDD를 메인 메모리에 저장하기 때문에, 반복 횟수인 n의 값이 클 때 메인 메모리에 BDD를 더 이상 저장하지 못하는 상태 폭발 문제가 발생한다. 이러한 문제를 해결하기 위해서 수정된 NuSMV에서는 BDD 라이브러리 함수를 이용해서 첫 번째 역방향 탐색에서 계산된 값을 메인 메모리 대신에 하드디스크에 저장을 하고, 나중에 반례를 생성하기 위해 역방향 탐색을 수행할 때 S_i에 해당되는 BDD만을 하드 디스크에서 메인 메모리로 불러왔다. 이렇게 함으로서 반복 횟수인 n의 값이 큰 상태 공간을 효율적으로 다룰 수 있다.

3.4 상태 공간 탐색 방향을 변경

전 장에서 살펴본 바와 같이, 모델 체킹은 역방향 탐색을 기본으로 하였다. 그러나 정방향과 양방향 등 다양한 방법으로 상태 공간을 탐색할 수도 있다. 본 논문에서는 상태 공간을 정방향으로 탐색하도록 NuSMV를 수정하였다. 함수 τ를

$$\tau(Z) = post_{\exists}(Z))$$

반복 적용함으로서 고정점을 계산할 수 있으며, 계산된 고정점은 도달 가능한 상태 집합을 나타낸다. 여기서 $Z=I$이다. 고정점과 $[\![\phi]\!]$를 교집합 했을 때, 만일 공통되는 것이 있다면 $AG\neg\phi$는 거짓이다. 이 경우, 수정된 NuSMV는 정방향 탐색을 중단하고 반례를 생성하도록 구현했다.

뿐만 아니라 상태 공간을 역방향과 정방향, 즉 양방향에서 동시에 탐색하도록 NuSMV를 수정하였다. 양방향 탐색 알고리즘은 (그림 5)와 같다. Q는 기본 값으로 초기 상태 I를 가지며, Y는 기본 값으로 $[\![\phi]\!]$를 갖는다. Q는 정방향 탐색에 사용되고, Y는 역방향 탐색에 사용된다. 두 집합을 교집합 했을 때, 만일 공통되는 것이 있다면 $AG\neg\phi$는 거짓이다. 이 경우 양방향 탐색을 중단하고 반례를 생성하도록 구현했다.

```
while{
  if(size(Q)<size(Y)) {
    Q=post∃(Q)
  }else {
    Y=pre∃(Y);
  }
  if(Q∩Y≠Ø) {
    generate counterexample;
    stop;
  }
}
```

(그림 5) 양방향으로 상태 공간을 탐색

4. 실 험

본 논문에서 제시한 반례 생성은 다양한 분야에 적용 가능하다. 반례 생성시 얼만큼의 성능 개선이 있었는지를 실제 확인하기 위해서 푸쉬 푸쉬 게임 풀이에 본 논문에서 제안한 반례 생성 기법을 적용하였다. 푸쉬 푸쉬 게임은 정형 부정의 대표적인 사례로서 모델 체킹에서 생성된 반례가 게임의 최단 풀이 경로이다. 주어진 게임으로부터, 모델 체커에 입력될 유한 상태 모델 M과 CTL 논리식 $AG\neg\phi$를 자동 추출할 수 있다. 여기서 ϕ는 목표 상태를 나타낸다(지면 관계상 푸쉬 푸쉬 게임으로부터 유한 상태

〈표 1〉 상태공간축소, 탐색횟수축소 그리고 하드디스크저장 결과

	기존 NuSMV				수정된 NuSMV			
	상태공간축소 이전		상태공간축소		상태공간축소+탐색횟수축소		상태공간축소+탐색횟수축소+ 하드디스크저장	
레벨	시 간(초)	메모리(MB)	시 간(초)	메모리(MB)	시 간(초)	메모리(MB)	시 간(초)	메모리(MB)
1	0.1	2.3	0.1	2.3	0.1	2.2	0.1	2.2
2	2.4	6.6	2.4	8.7	1.3	5.4	1.6	5.6
3	40.3	16.8	37.3	12.9	24.8	14.1	28.0	12.7
4	1.7	7.8	1.6	6.9	1.4	6.5	1.8	6.5
5	1.4	5.3	0.8	4.1	0.6	3.6	0.8	3.7
6	198.7	14.9	73.6	16.3	96.7	23.2	99.0	13.7
7	56.9	13.7	29.5	13.4	34.5	14.5	35.6	12.8
8	6,365.1	372.3	376.8	29.5	254.1	31.2	229.8	13.4
9	5.9	12.1	3.2	11.0	0.7	4.5	0.9	4.5
10	7.1	12.8	1.3	6.7	0.5	3.7	0.6	3.9
11	5.1	11.3	4.8	12.4	0.3	3.5	0.5	3.5
12	88.7	32.4	39.7	12.9	37.0	12.7	36.4	12.8
13	206.2	44.6	83.8	24.8	0.7	4.5	0.9	4.7
14	143.7	24.1	94.5	21.3	9.5	12.8	11.0	12.6
15	3,250.7	210.0	861.6	34.0	696.3	104.5	695.8	27.6
16	14.4	13.0	13.8	13.0	4.7	12.3	5.2	12.4
17	3,487.7	210.7	984.1	62.2	459.8	99.1	433.8	24.6
18	711.8	59.1	486.1	76.6	64.7	14.7	57.8	14.0
19	2,934.2	139.6	2,491.3	313.0	1,166.5	107.6	2,274.5	105.7
20	7,516.6	342.0	852.3	37.6	591.7	99.9	546.1	22.7
21	3,626.0	132.9	1,610.0	89.2	423.2	106.3	443.2	44.6
22	1,122.8	47.3	354.0	37.2	942.4	166.9	879.8	44.0
23	1,777.0	103.3	145.6	17.6	66.9	14.1	61.6	13.2
24	614.8	60.2	581.6	55.7	253.2	42.9	238.8	15.7
25	410.3	43.1	114.7	22.6	18.7	13.1	21.1	12.4
26	33.3	13.5	31.6	13.1	9.9	13.5	10.9	12.7
27	36.0	13.8	18.9	13.0	17.5	12.8	19.0	13.4
28	356.2	39.5	255.0	40.1	29.9	12.2	32.1	11.6
29	10.8	12.8	8.6	12.3	5.2	12.5	6.2	12.6
30	∞	∞	11,636.0	387.1	4,777.9	333.6	4,786.9	170.
31	662.3	62.7	443.2	70.6	33.3	13.7	34.9	213.3
32	1,523.1	72.9	1,140.6	119.0	34.7	13.5	36.4	13.0
33	∞	∞	4,299.9	374.9	564.7	53.6	537.1	15.1
34	∞	∞	∞	∞	1,387.4	284.5	1,366.1	12.5
35	∞	∞	1,294.7	133.6	218.1	62.9	226.1	12.8
36	∞	∞	13,673.7	301.5	278.0	54.2	283.5	13.2
37	654.1	91.6	500.5	63.1	36.8	13.2	37.8	13.2
38	596.7	34.5	231.5	18.6	38.4	20.4	41.8	92.5
39	191.3	47.3	162.1	34.3	15.7	12.8	17.1	13.2
40	∞	∞	6,930.0	758.9	17.3	13.8	18.3	13.2
41	175.7	82.9	78.3	13.2	42.3	14.6	39.9	92.5
42	1,260.0	67.9	752.2	82.6	175.5	32.2	178.8	13.2
43	1,639.1	129.7	480.0	29.9	583.2	166.9	595.2	92.5
44	1,322.8	131.9	1,116.2	116.1	115.6	16.4	119.2	13.2
45	187.7	32.6	92.3	19.4	9.6	12.3	11.5	12.3
46	237.4	38.9	180.3	32.2	51.8	13.4	50.9	12.8
47	35.2	25.9	19.4	13.3	4.1	10.4	4.9	10.5
48	1,980.6	846.7	736.1	69.1	114.3	20.3	105.8	13.5
49	∞	∞	12,213.8	787.3	2,060.9	238.2	2,066.0	142.2
50	∞	∞	∞	∞	∞	∞	10,449.8	144.2

〈표 2〉 상태 공간의 다양한 탐색 방법

	원래 NuSMV		수정된 NuSMV						역방향, 정방향, 영방향중에서 가장 좋은 결과	
			상태공간축소+탐색횟수축소+하드디스크저장+역방향탐색		상태공간축소+탐색횟수축소+하드디스크저장+정방향탐색		상태공간축소+탐색횟수축소+하드디스크저장+양방향탐색			
레벨	시간(초)	메모리(MB)	시간(초)	메모리(MB)	시간(초)	메모리(MB)	시간(초)	메모리(MB)	시간(초)	메모리(MB)
1	0.1	2.3	0.1	2.2	0.1	2.0	0.1	2.0	0.1	2.0
2	2.4	6.6	1.6	5.6	1.2	6.2	1.8	6.3	1.2	6.2
3	40.3	16.8	28.0	12.7	7.8	11.8	12.0	13.0	7.8	11.8
4	1.7	7.8	1.8	6.5	0.2	2.8	0.3	3.0	0.2	2.8
5	1.4	5.3	0.8	3.7	0.3	3.1	0.5	3.3	0.3	3.1
6	198.7	14.9	99.0	13.7	40.6	16.1	17.4	12.3	17.4	12.3
7	56.9	13.7	35.6	12.8	9.9	12.4	4.1	12.6	4.1	12.6
8	6,365.1	372.3	229.8	13.4	172.9	28.3	190.2	27.8	172.9	28.3
9	5.9	12.1	0.9	4.5	2.0	8.9	0.7	4.4	0.7	4.4
10	7.1	12.8	0.6	3.9	0.9	5.5	0.7	4.3	0.6	3.9
11	5.1	11.3	0.5	3.5	2.2	9.9	0.4	3.4	0.4	3.4
12	88.7	32.4	36.4	12.8	2.5	10.8	3.7	10.8	2.5	10.8
13	206.2	44.6	0.9	4.7	210.6	46.7	1.0	5.0	0.9	4.7
14	143.7	24.1	11.0	12.6	83.0	21.5	16.2	12.7	11.0	12.6
15	3,250.7	210.0	695.8	27.6	172.3	33.4	151.5	18.4	151.5	18.4
16	14.4	13.0	5.2	12.4	6.7	12.6	5.8	12.7	5.2	12.4
17	3,487.7	210.7	433.8	24.6	358.2	67.5	194.3	20.3	194.3	20.3
18	711.8	59.1	57.8	14.0	429.5	76.2	90.0	13.1	57.8	14.0
19	2,934.2	139.6	2,274.5	105.7	527.3	110.4	243.4	44.2	243.4	44.2
20	7,516.6	342.0	546.1	22.7	271.4	37.3	319.4	23.8	546.1	22.7
21	3,626.0	132.9	443.2	44.6	528.3	106.2	57.0	16.2	57.0	16.2
22	1,122.8	47.3	879.8	44.0	236.3	43.4	331.2	22.4	331.2	22.4
23	1,777.0	103.3	61.6	13.2	76.8	20.1	30.8	13.3	30.8	13.3
24	614.8	60.2	238.8	15.7	335.6	61.9	216.5	18.4	238.8	15.7
25	410.3	43.1	21.1	12.4	78.1	22.6	10.1	12.5	10.1	12.5
26	33.3	13.5	10.9	12.7	16.0	13.0	10.1	12.5	10.1	12.5
27	36.0	13.8	19.0	13.4	7.2	12.5	8.0	12.4	8.0	12.4
28	356.2	39.5	32.1	11.6	222.9	40.9	47.4	13.2	32.1	11.6
29	10.8	12.8	6.2	12.6	4.6	12.7	4.1	11.8	4.1	11.8
30	∞	∞	4,786.9	170.2	2,992.9	337.7	126.6	21.2	126.6	21.2
31	662.3	62.7	34.9	13.3	423.0	72.1	48.5	13.2	34.9	13.3
32	1,523.1	72.9	36.4	13.0	1,068.2	125.1	48.2	13.5	36.4	13.0
33	∞	∞	537.1	15.1	3,803.7	378.3	966.6	23.6	537.1	15.1
34	∞	∞	1,366.1	162.3	∞	∞	141.0	27.7	141.0	0.0
35	∞	∞	226.1	31.6	736.6	139.9	46.0	14.0	46.0	0.0
36	∞	∞	283.5	14.2	136.6	24.6	136.6	24.6	136.6	24.6
37	654.1	91.6	37.8	13.1	580.5	81.6	39.4	13.3	37.8	13.1
38	596.7	34.5	41.8	15.1	4.1	12.9	2.6	10.4	2.6	10.4
39	191.3	47.3	17.1	12.5	164.4	39.1	6.4	12.3	6.4	12.3
40	∞	∞	18.3	12.8	∞	∞	21.2	13.6	18.3	12.8
41	175.7	82.9	39.9	13.2	36.4	13.4	54.2	12.8	54.2	12.8
42	1,260.0	67.9	178.8	13.2	583.4	86.5	245.2	15.3	178.8	13.2
43	1,639.1	129.7	595.2	92.5	208.0	63.5	24.0	13.1	24.0	13.1
44	1,322.8	131.9	119.2	13.2	∞	∞	298.1	14.6	119.2	13.2
45	187.7	32.6	11.5	12.3	170.8	32.2	13.6	13.2	11.5	12.3
46	237.4	38.9	50.9	12.8	182.3	39.2	36.1	12.9	36.1	12.9
47	35.2	25.9	4.9	10.5	13.5	13.1	4.5	11.8	4.9	10.5
48	1,980.6	846.7	105.8	13.5	616.4	70.3	164.8	14.4	105.8	13.5
49	∞	∞	2,066.0	142.2	∞	∞	171.2	29.3	171.2	0.0
50	∞	∞	10,449.8	144.2	∞	∞	17,948.5	293.3	10,449.8	144.2

모델을 생성하는 과정, 목표 상태를 CTL논리식으로 표현하는 방법, 추상화된 모델을 생성하는 과정 등에 관한 설명은 생략한다. 이들에 대한 상세한 내용은 관련 연구 [5, 11, 14]을 참조하기 바람). 만약 초기 상태에서 목표 상태로 가는 경로가 존재한다면, 모델 체킹은 게임을 풀 수 있는 최단 경로를 반례로 출력한다.

실험을 위해서 기존 NuSMV와 본 논문에서 수정한 NuSMV를 사용하여 푸쉬 푸쉬 게임을 풀었다. 게임 풀이에 소요된 시간과 메모리 사용량이 <표 1>에 있다. 표에서 기호 ∞는 3시간 이상 경과해도 결과를 볼 수 없었음을 나타낸다. 실험은 펜티엄 4 1.5GHz, Linux 운영체제, 그리고 1기가 메모리를 가진 컴퓨터에서 수행하였다. 전장에서 설명한 방법들을 적용해서 얻은 결과를 표로 정리했다. 기존 NuSMV를 사용하여 얻은 결과, 상태 공간을 축소한 후 기존 NuSMV를 사용하여 얻은 결과, 탐색 횟수를 축소하도록 NuSMV를 수정한 후 얻은 결과, 그리고 하드 디스크에 중간 결과를 저장하도록 NuSMV를 수정한 후 얻은 결과를 차례대로 정리했다. 기존 NuSMV를 사용하여 원래 모델을 풀이한 결과, 전체 50게임 중에서 42게임은 풀었으나 8게임은 풀지 못했다. 그러나 제안한 추상화 기법을 적용하여 모델의 상태 공간을 축소한 후 기존 NuSMV로 풀이한 결과 48게임을 풀었고 추상화 이전에 비해서 반례 생성에 소요된 시간과 메모리 사용량이 각각 42%와 23% 개선되었다. 또한 중간 결과를 재사용하여 탐색 횟수를 줄인 결과 총 49게임을 풀었고, 기존 NuSMV에 비해서 반례 생성에 소요된 시간과 메모리 사용량이 각각 75%와 40% 개선되었다. 그러나 마지막 50번째 게임의 경우는 추상화를 수행하고 동시에 반례 생성 방법을 개선했음에도 불구하고 풀지 못했다. 그래서 계산 결과를 하드 디스크에 저장하도록 NuSMV를 수정한 결과 반례 생성에 소요된 시간과 메모리 사용량이 기존 NuSMV에 비해서 70%와 55% 개선되었다. 뿐만 아니라 결국 50게임 전체를 모두 풀었다. 또한 다양한 방법으로 상태 공간을 탐색하도록 NuSMV를 수정하였다. <표 2>는 역방향, 정방향, 그리고 양방향으로 상태 공간을 탐색한 결과이다. 각 게임을 풀 때 세 가지 탐색 방향 중에서 성적이 가장 우수한 결과만을 선택해서 마지막 열에 나타내었다. 그 결과, 전체 50게임을 모두 풀었고 반례 생성에 소요된 시간과 메모리 사용량은 기존 NuSMV에 비해서 86%와 62% 개선되었다. 실험 결과에서 보듯이 본 논문에서 제안한 기법은 기존 NuSMV에 비 해서 반례 생성에 요구되는 시간과 메모리 사용량을 크게 개선했을 뿐만 아니라, 상태 폭발 문제 때문에 기존 NuSMV로 찾아낼 수 없었던 반례를 수정된 NuSMV로 찾을 수 있었다.

5. 관련 연구들

반례는 모델 디버깅 및 모델 이해에 유용하기 때문에, 반례 생성에 관한 연구가 많이 수행되었다. 이들 연구들은 크게 반례 생성에 관한 연구, 반례 표현에 관한 연구, 그리고 반례 활용에 관한 연구로 구분할 수 있다. 먼저 반례 생성에 관한 연구들을 살펴보자. Clarke[12]은 반례 생성에 관한 효율적인 알고리즘을 제시했고, 이 알고리즘은 NuSMV 등 대부분의 모델 체킹 도구에 구현되어 있다. 그러나 NuSMV는 정형 긍정과 정형 부정에 범용적으로 사용되기 때문에, Chan[9]의 지적과 같이 정형 부정 목적으로 모델 체킹을 사용하는 경우 반례 생성이 비효율적인 경우가 가끔 있었다. 그래서 [9, 15]의 연구에서는 정형 부정인 경우에 효율적으로 반례를 생성하도록, 본 논문 3.2절에 소개한 것과 같이 상태 공간 탐색 횟수를 2회로 축소하도록 NuSMV를 수정하였다. 본 논문에서는 탐색 횟수 축소 이외에도 저장 공간 변경과 다양한 탐색 방법을 제시하고 있다.

반례를 사용하여 모델을 디버깅하기 위해서는 반례의 가독성이 높아야 한다. NuSMV에서는 반례를 경로로 표현하기 때문에, 이해하기 어렵다는 지적이 있었다. 이를 개선하기 위해서 반례를 트리[17] 또는 증명[18] 형태로 표현하는 연구들이 수행되었다. 트리 형태의 표현은 중첩된 논리식이 위반된 경우에 반례 이해에 도움을 주며, 증명 형태의 표현은 반례가 생성되는 과정을 논리적으로 이해하는데 도움을 준다. 트리와 증명은 2차원 그림 표현인 것에 비해서 경로는 1차원 텍스트 표현이다. 본 논문에서는 NuSMV의 출력과 같이 반례를 경로로 표현한다.

반례의 생성과 표현에 관한 연구뿐만 아니라 반례를 활용하는 연구들도 많다. 예를 들어, Ammann[19]은 반례를 활용하여 테스트 케이스를 추출하는 방법을 제안했으며, Clarke[20]은 주어진 모델을 자동으로 추상화할 때 반례를 활용하였다. 이들과 다르게 본 논문에서는 반례를 활용하여 푸쉬 푸쉬 게임을 풀었다. 즉, 반례에 의해서 생성된 경로가 푸쉬 푸쉬 게임에서 주어진 공을 목표 지점으로 모두 이동하는 최단 경로이다.

6. 결 론

우리는 모델 체킹에서 안전성 속성이 위반되었을 때 그에 대한 반례를 효율적으로 생성하도록 NuSMV를 수정하였고, 제안된 기법의 효용성을 실제 확인하기 위해서 푸쉬 푸쉬 게임 풀이에 적용하였다. 푸쉬 푸쉬 게임을 선택한 이유는, 이 게임은 정형 부정의 대표적인 사례로서 모델

체킹에서 생성된 반례가 게임의 최단 풀이 경로이기 때문이다. 기존 NuSMV로는 전체 50게임 중에서 42게임만을 풀었으나, 본 논문의 방법으로는 푸쉬 푸쉬 전체 50게임을 모두 풀었다. 뿐만 아니라 반례 생성에 소요된 시간과 메모리 사용량이 기존 NuSMV에 비해서 각각 86%와 62% 개선되었다. 즉, 기존 NuSMV에 비해서 12%의 시간과 34%의 메모리만을 사용하고서도 반례를 구할 수 있었다.

기존 NuSMV는 정형 긍정과 정형 부정에 모두 사용되는데 반해서, 수정된 NuSMV는 정형 부정에 사용할 경우에만 효율적이다. 따라서 주어진 문제가 정형 긍정이거나 또는 판단이 애매할 때는 기존 NuSMV를 사용하여 모델 체킹하는 것이 좋을 것이다. 한편, 주어진 문제가 정형 부정이라는 확신이 있다면 본 논문의 방법을 사용하는 것이 유리하다.

언급했듯이 <표 2>의 마지막 열은 정방향, 역방향, 그리고 양방향 세 가지 탐색 방법 중에서 가장 우수한 결과를 모아 놓은 것이다. 예를 들어, 25번째 게임은 양방향으로 탐색하는 것이 가장 우수했지만, 48번째 게임은 정방향 탐색이 가장 우수했다. 탐색 방법 별로 우수한 성적을 거둔 게임 수는 정방향이 7개, 역방향은 18개, 그리고 양방향은 25개이다. 그러나 아직까지 왜 25번째 게임을 양방향으로 탐색하는 것이 가장 좋은 결과를 내었는지를 이해하지 못했다. 이러한 이해를 바탕으로, 모델 체킹 전에 주어진 문제를 해결하기에 가장 좋은 탐색 방법을 미리 결정하는 것이 향후 연구 과제이다.

참 고 문 헌

[1] 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, Vol.8, No.2, pp.244-263, 1986.

[2] T. Kropf, Introduction to Formal Hardware Verification, Springer, 1999.

[3] K. Laster, and O. Grumberg, "Modular model checking of software," In Proceedings of TACAS'98, LNCS 1384, pp.20-35, 1998.

[4] N. Heintze, J. D. Tygar, J. Wing, and H. C. Wong, "Model checking electronic commerce protocols," In Proceedings of the USENIX 1996 Workshop on Electronic Commerce, pp.147-164, 1996.

[5] 권기현, "모델 검증을 이용한 게임 풀이," 정보과학회학회지, 제21권 1호, pp.7-14, 2003년.

[6] A. Cimatti, and M. Roveri, "Conformant Planning via Symbolic Model Checking," Journal of Artificial Intelligence Research, Vol.13, pp.305-338, 2000.

[7] E. M. Clarke, O. Grumberg, and D. Peled, Model Checking, MIT Press, 1999.

[8] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV 2: An OpenSource Tool for Symbolic Model Checking," In the Proceedings of CAV'02, 2002.

[9] W. Chan, Symbolic Model Checking for Large Software Specifications, Ph.D. thesis, University of Washington, Computer Science and Engineering, 1999.

[10] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Property specification patterns for finite-state verification," In Proceedings of the Workshop on Formal Methods in Software Practice, 1998.

[11] G. Kwon, "Applying Model Checking Techniques to Push Push Game Solving," In Proceedings of SERA2003, LNCS 3026, pp.290-303, 2003.

[12] E. M. Clarke, O. Grumberg, K. L. McMillan, and X. Zhao, "Efficient Generation of Counterexamples and Witness in Symbolic Model Checking," In Proceedings of Design Automation Conference, pp.427-432, 1995.

[13] E. M. Clarke, O. Grumberg, and D. E. Long, "Model Checking and Abstraction," ACM Transactions on Programming Languages and Systems, Vol.16, No.5, pp.1512-1542, 1994.

[14] 권기현, "모델 체킹에서 상태 투영을 이용한 모델의 추상화," 정보처리학회논문지 D, 제11-D권 제6호, pp.1295-1300, 2004.

[15] 권기현, 이태훈, "게임 풀이를 위한 NuSMV의 효율적인 반례 생성," 정보처리학회논문지 D, 제10-D권 제5호, pp.813-820, 2003.

[16] R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," IEEE Transactions on Computer, Vol.35, No.8, pp.677-691, 1986.

[17] E. M. Clark, Y. Lu, S. Jha, and H. Veith, "Tree-Like Counterexamples in Model Checking." In Proceedings of LICS'02, pp.19-29, 2002.

[18] M. Chechik, and A. Gufinkel, "Proof-Like Counterexamples," In Proceeding of TACAS'03, LNCS 2619, pp.160-175, 2003.

[19] P. E. Ammann, P. E. Black, and W. Majurski, "Using Model Checking to Generate Tests from Specifications," In Proceedings of ICFEM '98, pp.46-54, 1998.

[20] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-Guided Abstraction Refinement," In Proceedings of CAV'2000, pp.154-169, 2000.

이 태 훈

e-mail : taehoon@kyonggi.ac.kr
2003년 경기대학교 전자계산학과(학사)
2003년~현재 경기대학교 전자계산학과 석사과정
관심분야 : 소프트웨어 공학, 소프트웨어 분석, 정형 기법 등

권 기 현

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