게임 풀이를 위한 NuSMV의 효율적인 반례 생성

권 기 현† · 이 태 훈††

요 약

모델 검사는 모델이 속성을 만족하는지를 판정하기 위해서 모델의 상태 공간을 철저하게 조사한다. NuSMV는 모델 검사를 자동으로 수행하는 도구로서 본 논문에서는 이와 같은 NuSMV를 이용하여 푸쉬 푸쉬 게임을 해결한다. 모델이 속성을 만족하지 않는 경우 NuSMV는 그 이유를 설명하는 반례를 생성하게 되는데 NuSMV에 구현되어 있는 반례 생성 방식은 상태 공간을 2번 탐색하기 때문에 게임 풀이에 비효율적이다. 본 논문에서는 반례 생성시 상태 공간을 한 번만 탐색하도록 NuSMV를 재 구현 하였다. 그 결과 게임 풀이에 있어서 원래 NuSMV 보다 약 62%의 시간 절감과 11%의 공간 절감이 있었다.

Efficient Counterexample Generation for Game Solving in NuSMV

Gihwon Kwon† · Taehoon Lee††

ABSTRACT

This paper solves Push-Push game with the model checker NuSMV which exhaustively explores all search space to determine whether a model satisfies a property. In case a model doesn't satisfy properties to be checked, NuSMV generates a counterexample which tells where this unsatisfaction occurs. However, the algorithm for generating counterexample in NuSMV traverses a search space twice so that it is inefficient for solving the game we consider here. To save the time to be required to complete the game, we revise the part of counterexample generation so that it traverses a search space once. As a result, we obtain 62% time improvement and 11% space improvement in solving the game with modified NuSMV.

키워드 : 모델 검사(Model Checking), 속성(Property), 고정점 계산(Fixed-point Computation), 반례(Counterexample)

1. 서 론

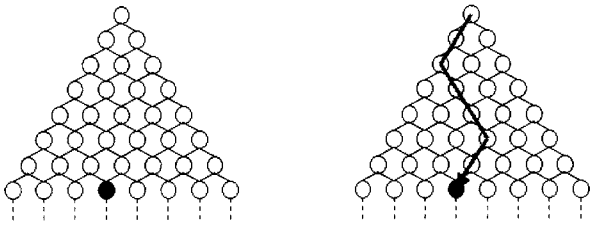
시스템의 기능이 복잡해지고 규모가 커질수록, 위험한 에러들이 발견되지 않은 채 시스템에 숨어있을 가능성이 높다. 만약 숨어있던 에러들이 시스템 실행 중에 출현한다면, 그 피해 규모는 적지 않다. 일상 업무와 생활이 컴퓨터 시스템에 의존적일수록, 피해 규모는 당연히 증가할 것이다. 일상 업무와 생활이 컴퓨터 시스템에 매우 의존적인 디지털 시대에서, 소프트웨어 엔지니어가 추구해야 할 주요 목표 중의 하나가 에러 없는 고품질의 소프트웨어를 개발하는 것이다.

모델 검사는 효과적인 에러 식별 기법으로 인식되고 있으며, 산업체에서의 활용이 점차 증대되고 있다[1]. 모델 검사는 시스템을 다루기 보다는 시스템의 추상 표현인 모델을 다룬다. 원하는 속성이 모델에서 만족되는지를 검사하기 위해, 모델이 갖는 모든 상태 공간을 철저하게 탐색한다. 그러므로 기존의 테스팅이나 시뮬레이션과 같은 비정형 검증 기법으로 찾을 수 없는 여러 가지 오류를 찾을 수 있다. 그리고 검사 과정이 자동으로 이루어지기 때문에 정리 증명과 같은 다른 정형 검증 기법에 비해 많은 이점을 제공한다. 한편, 모델이 속성을 만족하지 않는 경우 그 이유를 담은 반례를 생성하기 때문에 시스템의 디버그 작업을 도와줄 수 있다. 이런 이점 덕분에 하드웨어 검증, 소프트웨어 검증, 그리고 프로토콜 검증 분야에서 모델 검사 기법이 활발히 사용되고 있다.

언급한 바와 같이 모델 검사의 이점중의 하나는 반례 생성이다[2]. 만일 시스템이 속성을 만족하지 않는다면 시스템이 어떻게 속성을 만족하지 않게 되는지를 보여주는 것이 반례이다. 예를 들어, (그림 1)과 같이 CTL(Computation Tree Logic, [3]) 속성 AG $\neg\phi$가 거짓인 경우를 살펴보자. AG $\neg\phi$는 '도달 가능한 모든 상태에서 ϕ가 항상 거짓'임을 의미한다. 만약 도달 가능한 상태 중에서 ϕ가 참이

※ 본 연구는 정보통신부(MIC)와 한국소프트웨어공학협회(KSEA)의 한·카 네기멜론 S/W 전문인력 교육 국내보급사업의 지원으로 수행되었음.
† 종신회원 : 경기대학교 정보과학부 교수
†† 준 회 원 : 경기대학교 대학원 전자계산학과
논문접수 : 2003년 5월 29일, 심사완료 : 2003년 6월 17일

되는 상태가 최소한 하나라도 존재한다면, AG ¬ϕ는 거짓이 된다. AG ¬ϕ가 거짓인 경우 초기 상태로부터 ϕ가 참인 상태로 도달되는 경로를 보여준다. 이러한 경로를 반례라고 하고 반례를 통해서 AG ¬ϕ가 만족되지 않는 이유를 알 수 있다.



(○는 ϕ가 거짓인 상태, ●는 ϕ가 참인 상태를 나타냄)

(그림 1) AG ¬ϕ가 거짓인 경우(왼쪽 그림)와 그에 대한 반례 (오른쪽 그림)

반례는 유용한 정보를 제공하기 때문에 반례를 이용한 연구들이 많다. Clarke[4]은 반례를 이용한 자동 추상화 기법을 연구하였고, Ammann[5]은 반례로부터 테스트 케이스 생성 방법을 연구했다. 본 연구에서는 반례를 이용해서 푸쉬 푸쉬 게임을 풀려고 한다. 주어진 공을 목표 지점으로 이동하는 경로를 찾는 것이 푸쉬 푸쉬 게임의 목표이다. 모델 검사 도구인 NuSMV(New Symbolic Model Verifier)[6]를 사용하여 푸쉬 푸쉬 게임을 풀 수 있는 최단 경로를 구했지만, 시간과 메모리를 많이 소비해야만 했었다. NuSMV와 같은 범용 모델 검사 도구로 푸쉬 푸쉬와 같은 게임을 풀기 위해서는 최적화된 기법이 반드시 필요했고, 그러한 기법을 연구하던 도중 NuSMV에 구현되어 있는 반례 생성 방식이 게임 풀이에 비효율적임을 발견하게 되었고, 그래서 불필요한 상태 공간 탐색을 수행하는 것을 알게 되었다. 게임 풀이에 소요되는 시간을 줄이기 위해서, 본 논문에서는 NuSMV의 반례 생성 부분을 효율적인 방법으로 재 구현하였다. 그 결과 푸쉬 푸쉬 게임을 풀 수 있는 최단 경로를 기존 NuSMV 보다 빠르게 찾아내었다.

본 논문의 구성은 다음과 같다. 2장에서는 본 논문의 배경 지식이라고 할 수 있는 모델 검사 기법을 설명한다. 3장에서는 모델 검사를 이용한 푸쉬 푸쉬 게임 풀이에 대해서 설명하고, 4장에서는 새로 구현한 반례 생성 기법을 설명하고, 이를 푸쉬 푸쉬 게임에 적용한 실험 결과를 제시한다. 마지막으로 5장에서 결론을 맺는다.

2. 배경 지식

2.1 모 델

모델 검사에 사용되는모델의 핵심 요소는 상태와 상태간의 전이이며, 이들을 사용하여 시스템의 행위를 모델링 한다. 본 논문에서 사용할 CTL 모델 검사에서는 크립키(Kripke) 구조라 불리는 모델 $M=(S,I,R,X,L)$을 사용한다[7]. 여기서,

- S는 상태들의 집합이다.
- $I \subseteq S$는 초기 상태들의 집합이다.
- $R \subseteq S \times S$은 상태들간의 전이를 나타내는 관계이다.
- X는 단순 명제들의 집합이다.
- $L : S \rightarrow 2^X$은 각 상태에서 참이 되는 단순 명제들을 해당 상태에 배정한 함수이다.

모델 검사는 시스템이 갖는 무한 행위에 대해서 조사를 하기 때문에 상태들간의 전이를 나타내는 R은 전체 관계(total relation)라고 가정한다. 즉 $\forall s \in S \cdot \exists s' \in S \cdot (s,s') \in R$로서, 모든 상태마다 전이할 수 있는 다음 상태가 최소한 하나 이상 존재한다. 경로 $\pi = s_0 s_1 s_2 s_3 s_4 \cdots$는 전이 가능한 상태들을 차례대로 나열한 것으로서 $(s_i, s_{i+1}) \in R, i \geq 0$이며 그 길이는 무한이다.

2.2 속 성

모델에 관한 속성은 모델을 트리의 관점에서 해석하는 CTL 시제 논리로 표현한다. 모델의 초기 상태를 루트로 해서 모델을 풀어헤치면 트리를 얻게 되며, 트리는 모델의 가능한 모든 행위를 표현한다. 모델의 속성을 정형적으로 기술하기 위해서 CTL은 두 개의 경로 한정자 A(All), E(Exists)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 얻는다.

CTL 식 ϕ, ψ의 값이 모든 모델과 모든 상태에서 동일하다면 두 식을 동치라고 부르며 $\phi \equiv \psi$로 표시한다. 동치 관계에 있는 식은 다음과 같다.

$$T \equiv \neg \bot$$
$$\phi_1 \vee \phi_2 \equiv \neg(\neg \phi_1 \wedge \neg \phi_2)$$
$$\phi_1 \Rightarrow \phi_2 \equiv \neg(\neg \phi_1 \wedge \neg \phi_2)$$
$$\phi_1 \Leftrightarrow \phi_2 \equiv \neg(\neg \phi_1 \wedge \neg \phi_2) \wedge \neg(\neg \phi_2 \wedge \neg \phi_1)$$
$$AX\phi \equiv \neg EX \neg \phi$$
$$AF\phi \equiv \neg EG \neg \phi$$
$$AG\phi \equiv \neg EF \neg \phi$$
$$A[\phi_1 \cup \phi_2] \equiv \neg E[\neg \phi_2 \cup (\phi_1 \wedge \neg \phi_2)] \wedge \neg EG \neg \phi_2$$

두 식이 동치라면, 좌변의 식은 우변의 식으로 대치 가능하다. 따라서 우변에 나오는 연산자의 모임이 CTL 식을 정의하는데 요구되는 연산자 집합이다. 위의 경우는 { ⊥, ¬, ∧, EX, EG, EF, EU} 이다. 이들을 이용해서 CTL 구문을 정의하면 다음과 같다.

$$\phi := p \mid \bot \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid EX\phi \mid EF\phi \mid EG\phi \mid E(\phi_1 U \phi_2)$$

CTL의 의미를 살펴보자. 모델 M의 상태 s에서 CTL 식 ϕ가 참인 경우를 $M,s \models \phi$로 표시한다. 그렇치 않다면 $M,s \not\models \phi$로 표시한다. 상태 s에서 CTL 식 ϕ의 값은 다음과 같이 재귀적으로 정의된다.

$M,s \not\models \bot$

$M,s \models p$ iff $p \in L(s)$

$M,s \models \neg\phi$ iff $M,s \not\models \phi$

$M,s \models \phi_1 \wedge \phi_2$ iff $M,s \models \phi_1$ and $M,s \models \phi_2$

$M,s \models \mathrm{EX}\phi$ iff $M,s' \models \phi$ for some state s' $with(s,s') \in R$

$M,s \models \mathrm{EF}\phi$ iff $\exists_{i\geq 0} \cdot M,s_i \models \phi$

$M,s \models \mathrm{EG}\phi$ iff $\exists_{\pi=s0,s1,s2,\cdots} \cdot \forall_{k\geq 0} \cdot M,s_k \models \phi$

$M,s \models \mathrm{E}(\phi_1 \mathrm{U} \phi_2)$ iff $\exists_{\pi=s0,s1,s2,\cdots} \cdot (\exists_{k\geq 0} \cdot M,s_k \models \phi_2 \wedge \forall_{0\leq i<k} \cdot M,s_i \models \phi_1)$

2.3 만족성 검사 알고리즘

모델 M의 모든 초기 상태에서 CTL 식 ϕ가 참인 경우를 $M \models \phi$로 표시하며 "M이 ϕ를 만족한다"라고 읽는다. 모델 검사는 M과 ϕ를 받아서 "M이 ϕ를 만족하는지를 결정"하는 문제이다. 이를 위해서 ϕ를 만족하는 상태들의 집합을 구한 후, 이 상태 집합에 초기 상태가 포함되어 있는지를 검사한다. CTL 식 ϕ를 만족하는 상태들의 집합을 $[\![\phi]\!]$라고 표시하자. 모델 검사 알고리즘의 핵심은 상태 집합 $[\![\phi]\!]$를 구한 후 초기 상태 집합이 $[\![\phi]\!]$의 부분 집합인 것을 검사하는 것이다. 즉,

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

이다. 만족성을 검사하기 위해서는 모델이 갖는 상태 공간을 탐색해야 하는데, 여기에는 2가지 탐색 방법이 있다. 초기 상태에서 출발해서 목표 상태를 찾아나가는 정 방향 탐색과 목표 상태에서 출발해서 거꾸로 초기 상태를 찾아가는 역 방향 탐색 방법이 있다. 다음과 같이

$$pre_\exists(Q) = \{s \in S \mid \exists_{s' \in Q} \cdot (s,s') \in R\}$$
$$post_\exists(Q) = \{s' \in S \mid \exists_{s \in Q} \cdot (s,s') \in R\}$$

함수 $pre_\exists$는 집합 Q로 도달할 수 있는 이전 상태들의 집합을 역방향으로 구하며, $post_\exists$는 현재 상태 Q로 부터 도달 가능한 다음 상태들의 집합을 정방향으로 찾는다. 모델 검사의 핵심 부분인 ϕ를 만족하는 상태 집합 $[\![\phi]\!]$은 다음과 같이 구한다.

$[\![p]\!] = \{s \in S \mid p \in L(s)\}$

$[\![\bot]\!] = \varnothing$

$[\![\neg\phi]\!] = S \setminus [\![\phi]\!]$

$[\![\phi_1 \wedge \phi_2]\!] = [\![\phi_1]\!] \cap [\![\phi_2]\!]$

$[\![\mathrm{EX}\phi]\!] = pre_\exists([\![\phi]\!])$

$[\![\mathrm{EF}\phi]\!] = \mu Z.([\![\phi]\!] \cup pre_\exists(Z))$

$[\![\mathrm{EF}\phi]\!] = \nu Z.([\![\phi]\!] \cap pre_\exists(Z))$

$[\![\mathrm{E}(\phi_1 \mathrm{U} \phi_2)]\!] = \mu Z.([\![\phi_2]\!] \cup ([\![\phi_1]\!]) \cap pre_\exists(Z))$

여기서 μ,ν는 각각 최소 고정점과 최대 고정점이다[8]. 상태 집합 $[\![\phi]\!]$을 계산할 때, 최소 고정점 μ는 공집합을 초기값으로 하여 계속해서 증가되다가 더 이상 증가하지 않는 집합을 구할 때 사용하며, 반대로 최대 고정점 ν은 전체 집합을 초기값으로 하여 계속해서 감소하다가 더 이상 감소하지 않는 집합을 계산할 때 사용한다[1].

살펴본 바와 같이, 모델 검사의 방향은 역방향 이다. 모델 검사의 수행 시간은 역 방향 함수와 고정점 계산에 좌우된다. 함수 $pre_\exists$와 고정점을 계산하는데 소요되는 시간은 모델의 크기에 선형 비례한다. 여기서 모델의 크기는 $|M| = |S| + |R|$로서 상태 수와 전이 수를 합한 것이다. 주어진 식 ϕ에 대한 상태 집합 $[\![\phi]\!]$을 계산하기 위해서, 모델 검사 알고리즘은 ϕ의 서브 식을 재귀적으로 구하면서 길이가 짧은 식부터 긴 식 순서로 상태 집합을 계산한다. 따라서 알고리즘의 복잡도는 $O(|M| \times |\phi|)$로서, 모델의 크기 및 식의 길이에 모두 선형 비례한다.

3. 반례를 이용한 게임 풀이

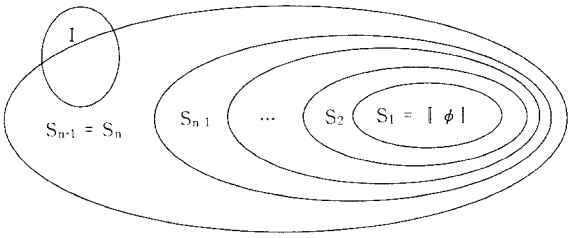
3.1 반례 생성

나중에 설명하겠지만, 본 논문에서 사용하는 주요 CTL 속성은 $\mathrm{AG}\neg\phi$이다. 따라서 반례 생성 과정을 $\mathrm{AG}\neg\phi$에 국한해서 설명하고자 한다. 2장의 동치 관계에 의하면, $\mathrm{AG}\neg\phi$의 대칭(dual)은 $\mathrm{EF}\phi$이다. NuSMV는 대칭인 $\mathrm{EF}\phi$를 이용해서 $\mathrm{AG}\neg\phi$를 검사한다. 전장에서 정의한 바와 같이 $[\![\mathrm{EF}\phi]\!] = \mu Z.([\![\phi]\!] \cup pre_\exists(Z))$이기 때문에 $[\![\mathrm{EF}\phi]\!]$는 함수 τ의

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

최소 고정점이다. 공집합으로부터 시작해서 함수 τ를 반복적으로 적용함으로서 최소 고정점을 구할 수 있다. 즉, 함수를 적용한 순서 $\tau^1(\varnothing) \subseteq \cdots \subseteq \tau^n(\varnothing) \subseteq \cdots$는 언젠가 더 이상 증가되지 않는 상태 $\tau^n(\varnothing) = \tau^{n+1}(\varnothing)$에 이르게 되는데, 이때 $\tau^n(\varnothing)$이 함수 τ의 최소 고정점이 된다. 함수 τ의 중간 계산 값을 $S_1 = \tau^1(\varnothing)$, $S_2 = \tau^2(\varnothing)$, $\cdots$, $S_n = \tau^n(\varnothing)$라고 하자. 사실, $S_1 = \tau^1(\varnothing) = [\![\phi]\!]$이다. 왜냐하면 $pre_\exists(\varnothing) = \varnothing$이기 때문이다. (그림 2)에서 보듯이 $S_n \cap I \neq \varnothing$이면 $\mathrm{AG}\neg\phi$는 거짓이다.

1) 최소 고정점 μ는 EF, EU, AF, AU를 계산할 때 사용되며, 최대 고정점 ν은 EG, AG를 계산할 때 사용한다.



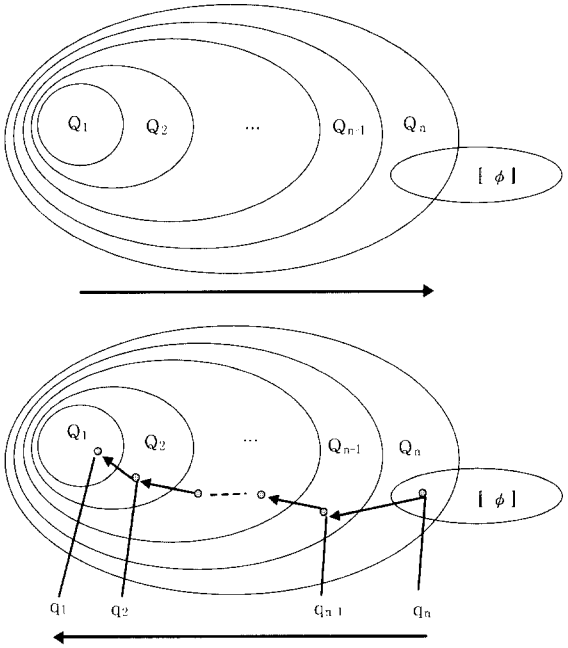
(그림 2) $S_n \cap I \neq \emptyset$이면 AG $\neg\phi$는 거짓이다

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

$$Q_1 = I$$
$$Q_{i+1} = post_{\exists}(Q_i) \cup Q_i$$

와 같이 초기 상태에서 ϕ가 참인 상태까지 정방향 탐색을 진행하면서 도달 가능한 상태를 저장한다($Q_i \cap \llbracket \phi \rrbracket \neq \emptyset$일 때 종료한다). 두 번째 단계는 ϕ가 참인 상태에서 초기 상태로 역 방향으로 오면서 초기 상태에서 가 참인 상태로 도달 가능한 경로, 즉 반례 $\langle q_1, \cdots, q_n \rangle$를 찾는다.

$$q_n \in Q_n \cap \llbracket \phi \rrbracket$$
$$q_{i-1} \in pre_{\exists}(Q_i) \cap Q_{i-1}$$



(그림 3) 원래 NuSMV에서는 두 단계를 거쳐서 반례를 생성한다

3.2 게임 풀이

본 논문에서는 범용 모델 검사 도구인 NuSMV를 이용해서 푸쉬 푸쉬 게임을 해결한다. 이 게임은 일종의 블록 이동 게임으로서, 주어진 공을 목표 지점으로 모두 옮기는 게임이다. 그러므로 게임의 목표는 목표 지점으로 옮기는 이동 경로를 찾는 것이다. 이동 경로를 모델 검사 기법으로 찾아내기 위해서는 유한 상태 모델과 CTL 속성을 기술해야 한다. 먼저 에이전트의 행위를 유한 상태 기계 $T_{agent} = (S_{agent}, I_{agent}, \delta_{agent})$로 모델링 하면 다음과 같다.

- $S_{agent} = \{(x, y)\}$는 이동 가능한 위치들의 집합이다. 위치는 순서쌍 (x, y)로 표현된다.
- $I_{agent} \in S_{agent}$는 에이전트의 초기 위치이다.
- $\delta_{agent} : S_{agent} \times D \rightarrow S_{agent}$는 전이 함수이다.

여기서 $D = \{left, right, up, down\}$는 에이전트의 이동 방향을 나타낸다. 움직일 수 있는 방향은 매번 하나이기 때문에 방향을 집합으로 모델링 하였다. 전이 함수 $\delta_{agent}((x, y), d) = = (x', y')$는 아래와 같이 정의된다.

$$\delta_{agent}((x,y),d) = \begin{cases} (x' = x-1,\ y' = y) \\ \quad \text{if } d = left \wedge movableToLeft(x,y) \\ (x' = x+1,\ y' = y) \\ \quad \text{if } d = right \wedge movableToRight(x,y) \\ (x' = x,\ y' = y+1) \\ \quad \text{if } d = up \wedge movableToUp(x,y) \\ (x' = x,\ y' = y-1) \\ \quad \text{if } d = down \wedge movableToDown(x,y) \\ (x' = x,\ y' = y) \\ \quad \text{otherwise} \end{cases}$$

여기서, 술어 $movableToLeft(x, y)$는 (x, y)에 위치한 에이전트가 바로 왼쪽 위치 $(x-1, y)$로 이동 가능함을 나타내며 다음과 같이 정의된다.

$$borderLeftBallLeft(x,y) = (x-1,y) \notin S_{agent}$$
$$borderLeftBallLeft(x,y) = (x-2,y) \notin S_{agent} \wedge \exists_{cell} \cdot pos(cell) = (x-1,y)$$
$$ballLeftBallLeft(x,y) = \exists_{cell_1, cell_2} \cdot (post(cell_1) = (x-2,y) \wedge cell_1) \wedge (post(cell_2) = (x-1,y) \wedge cell_2)$$
$$movableToLeft(x,y) = \neg borderLeftBallLeft(x,y) \wedge \neg borderLeftBallLeft(x,y) \wedge \neg borllLeftBallLeft(x,y)$$

여기서, 보조 함수 pos는 셀을 받아서 셀의 현재 위치를 출력한다. 나머지 $movableToRight$, $movableToUp$, $movableToDown$ 술어들도 비슷하게 정의된다.

게임에는 셀들이 있다. 셀은 비어있거나 또는 공을 갖고 있다. 매번 게임이 진행될 때 마다 셀의 상태가 변경된다. 임의의 셀 $cell_i$의 행위를 상태 기계 $T_{cell_i} = (S_{cell_i}, I_{cell_i}, \delta_{cell_i})$로 모델링 하면 다음과 같다.

- $S_{cell_i} = B$는 부울 변수이다. 즉 1이면 공을 갖고 있고, 0이면 비어 있다.
- $I_{cell_i} \in S_{cell_i}$는 $cell_i$의 초기 상태를 나타낸다.
- $\delta_{cell_i} : S_{cell_i} \times D \rightarrow S_{cell_i}$는 전이 함수를 나타낸다.

여기서 전이 함수 $\delta_{cell_i} : B \times D \rightarrow B$ (왜냐하면 $S_{cell_i} = B$)는 다음과 같이 정의된다.

$$\delta_{cell_i}(c,d) = \begin{cases} 0 \text{ if } c \wedge d = left \wedge \exists_{agent,cell} \cdot pos(agent) \\ \quad = (x+1,y) \wedge pos(cell) = (x-1,y) \wedge \neg cell \\ 1 \text{ if } \neg c \wedge d = left \wedge \exists_{agent,cell} \cdot pos(agent) \\ \quad = (x+2,y) \wedge pos(cell) = (x+1,y) \wedge cell \\ 0 \text{ if } c \wedge d = right \wedge \exists_{agent,cell} \cdot pos(agent) \\ \quad = (x-1,y) \wedge pos(cell) = (x+1,y) \wedge \neg cell \\ 1 \text{ if } \neg c \wedge d = right \wedge \exists_{agent,cell} \cdot pos(agent) \\ \quad = (x-2,y) \wedge pos(cell) = (x-1,y) \wedge cell \\ 0 \text{ if } c \wedge d = up \wedge \exists_{agent,cell} \cdot pos(agent) \\ \quad = (x,y-1) \wedge pos(cell) = (x,y+1) \wedge \neg cell \\ 1 \text{ if } \neg c \wedge d = up \wedge \exists_{agent,cell} \cdot pos(agent) \\ \quad = (x,y-2) \wedge pos(cell) = (x,y+1) \wedge cell \\ 0 \text{ if } c \wedge d = up \wedge \exists_{agent,cell} \cdot pos(agent) \\ \quad = (x,y+1) \wedge pos(cell) = (x,y-1) \wedge \neg cell \\ 1 \text{ if } \neg c \wedge d = up \wedge \exists_{agent,cell} \cdot pos(agent) \\ \quad = (x,y+2) \wedge pos(cell) = (x,y+1) \wedge cell \\ c \text{ otherwise} \end{cases}$$

에이전트의 상태 기계와 셀의 상태 기계를 동기식으로 결합하면 게임 전체의 상태 기계 $T_{game} = T_{agent} \otimes T_{cell_1} \otimes \cdots \otimes T_{cell_n}$를 얻는다. 여기서 기호 $\otimes$는 동기식 결합 연산자이다. 즉, 게임의 행위는 상태 기계 $T_{game} = (S_{game}, I_{game}, D, \delta_{game}, F_{game})$로 모델링 할 수 있다.

- $S_{game} = S_{agnet} \times S_{cell_1} \times \cdots \times S_{cell_n}$는 게임이 갖는 상태 공간이다.
- $I_{game} = (I_{agnet}, I_{cell_1}, \cdots, I_{cell_n})$는 게임의 초기 상태이다.
- $\delta_{game} : S_{agent} \times S_{cell_1} \times \cdots \times S_{cell_n} \times D \rightarrow S_{agnet} \times S_{cell_1} \times \cdots \times S_{cell_n}$는 상태 전이 함수이다. 즉, $\delta_{game}((x,y), c_1, \cdots, c_n, d) = \delta_{agent}((x,y), d), \delta_1(c_1, d), \cdots, \delta_n(c_n, d))$이다.
- F_{game}는 목표 셀들의 집합이다.

지금까지 유한 상태 기계로 게임을 모델링 하였다. 게임에 있는 개체와 그들의 행위를 모델링 하였다. 이와 같은 모델의 의미는 크립키 구조로 $M = (S, I, R, X, L)$ 정의 할 수 있다.

- $S = S_{agnet} \times S_{cell_1} \times \cdots \times S_{cell_n} \times D$는 상태 공간의 집합이다.
- $I = I_{agent} \times I_{cell} \times \cdots \times I_{cell_n} \{d \in D\}$는 초기 상태 집합이다.
- $R((x,y), c_1, \cdots, c_n, d, \delta_{agnet}((x,y),d), \delta_1(c_1,d), \cdots, \delta_n(c_n,d), d \in D)$는 전이 관계이다.
- $X = \{cell_1, \cdots, cell_n\}$는 단순 명제들의 집합이다.
- $L(s) = L((x,y), c_1, \cdots, c_n, d) = \{cell_1 \mid c_1\} \cup \cdots \cup \{cell_n \mid c_n\}$는 라벨 함수이다.

전술한 바와 같이, 푸쉬 푸쉬 게임 풀이란 초기 상태에서 목표 상태로 공을 모두 이동시키는 경로를 찾는 것이다. 목표 상태는 $F_{game} = \{cell_1, \cdots, cell_n\}$이다. 즉 게임마다 목표 상태에 도달하는 경로가 최소한 하나이상 존재한다. 만약 그러한 경로가 결코 존재하지 않는다고 하면 이는 틀린 것이 되고 모델 검사는 이에 대한 반례로 초기 상태에서 목표 상태로 가는 경로를 생성한다. 이러한 경로가 푸쉬 푸쉬 게임을 푸는 답이 되는 것이다. 게임을 풀 수 있는 답을 유도하기 위해서 '목표 상태로 갈 수 있는 경로가 결코 존재하지 않는다'를 CTL로 표현하면 AG$\neg\phi$이다. 여기서 $\phi \equiv \bigwedge_{i=1}^{|F_{game}|} cell_i \in F_{game}$이다.

4. 효율적인 반례 생성 기법

푸쉬 푸쉬와 같은 경로 찾기 게임을 해결하는 데에는 여러 가지 풀이 방법이 존재할 것이다. 인공 지능 분야의 연구와 같이 경로 찾기를 위한 탐색 프로시져를 처음부터 개발하기 보다는, 소프트웨어 공학 분야에서 소프트웨어의 정확성을 검증하는데 널리 사용되는 범용 모델 검사 도구인 NuSMV를 사용하여 푸쉬 푸쉬 게임을 풀었다[9]. 하지만 메모리와 시간을 많이 소비해야만 했었다. 설명한 바와 같이 NuSMV는 범용 모델 검사 도구로서, 정형 검증과 정형 부정에 모두 사용된다[2]. NuSMV의 첫 번째 단계는 (그림 2)와 같이 모델의 상태 공간을 역방향으로 탐색하면서 모델이 속성을 만족하는지를 검사한다. 만약 만족하는 경우 참을 출력하고 종료한다. 그러나 모델이 속성을 만족하지 않는 경우, 에러가 어디에서 발생되었는지를 보이는 반례를 생성해야 한다. 반례를 생성하기 위해서 (그림 3)과 같이 정방향과 역방향으로 상태 공간을 2번 탐색한다. 다시 말해서, 속성을 검사할 때 이미 탐색한 상태 공간을 반례 생성시 다시 탐색하고 있다. 이와 같은 중복 탐색은 메모리 사용량을 증가시킬 뿐만 아니라 반례 생성 시간을 크게 지연시킨다. 이와 같은 이유로, 모델이 속성을 만족하지 않는다는 것은 불과 몇 초 만에 알 수 있지만 반례 생성에는 여러 시간이 소요되는

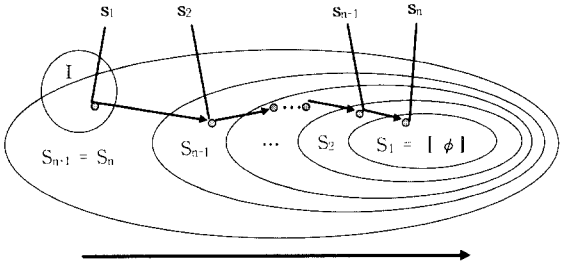
2) 정형 검증(formal verification)의 의도는 모델이 속성을 만족하는지를 보여주는 것이고($M \models \phi$), 정형 부정(formal falsification)의 의도는 모델이 속성을 만족하는 않는지를 보여주는 것이다($M \not\models \phi$).

경우들이 있다[10]. 본 논문에서 대상으로 하는 게임 풀이는 정형 부정 문제이다. 즉 모델을 검사하면 틀림없이 속성이 만족되지 않는다. 본 논문에서는 NuSMV를 정형 부정에 효율적으로 사용할 수 있도록, NuSMV의 반례 생성 부분의 소스 코드를 수정하고자 한다.

기존의 NuSMV에서는 Q_i를 구하기 위해서 다시 한번 계산을 했지만, 본 논문에서 구현한 방법은, 맨 처음 CTL 속성을 검사할 때 방문했던 상태들의 집합을 저장한 후 반례 생성시 이를 재사용하는 것이다. 게임 풀이에 사용할 목적이기 때문에 $AG\neg\phi$의 반례 생성에만 초점을 맞추어 NuSMV를 재구현 하였다. 3장에서 살펴본 바와 같이 첫 번째 역방향 탐색에서 함수 τ의 계산 값을 집합 $S_1 = \tau^1(\varnothing)$, $S_2 = \tau^2(\varnothing)$, …, $S_n = \tau^n(\varnothing)$에 저장한다. 중간 계산 결과를 저장한의 순서 $\langle S_1, S_2, \cdots, S_n \rangle = stg(\mathrm{EF}\,\phi)$를 고정점 계산의 스테이지라고 부른다[11]. 그런 후, 두 번째 단계인 정방향 탐색에서는 저장된 스테이지를 이용하여

$$s_1 \in S_n \cap I$$
$$s_{i+1} \in post_3(S_{n(i-1)}) \cap S_{n-1}$$

와 $i < n$ 일 때까지 계속해서 반례를 생성한다. 반례 생성시, 예전에는 (그림 3)에서 설명한대로 정방향 탐색과 역방향 탐색을 2번 수행하였지만, 수정을 한 이후의 반례 생성은 (그림 4)와 같이 역방향 탐색만 한 차례 수행해서 반례를 얻는다.

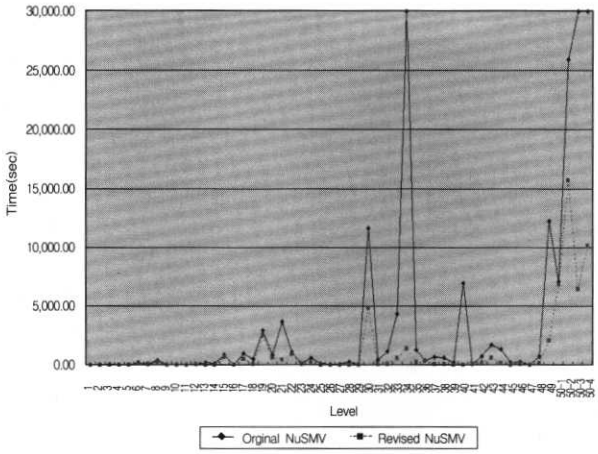


(그림 4) 수정된 NuSMV에서는 정방향 탐색 한번만으로 반례를 생성한다

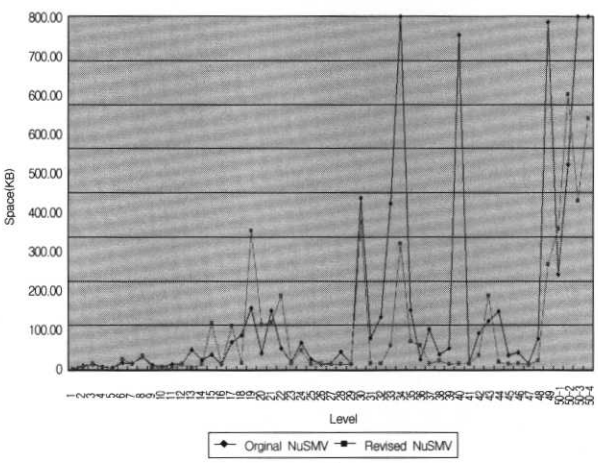
수정된 NuSMV가 원래 NuSMV에 비해서 얼만큼의 성능이 향상되었는가를 확인하기 위해서 각각의 NuSMV로 푸쉬 푸쉬 게임을 풀었다. 그 결과, <표 1>과 (그림 5)에서 보듯이 원래 NuSMV보다 수정한 NuSMV에서 반례 생성 시간이 평균 62% 줄어들었다. 시간 개선 이외에도, <표 1>과 (그림 6)에서 보듯이 수정한 NuSMV에서 메모리 사용량이 평균 11% 줄어들었다. 뿐만 아니라 원래 NuSMV를 이용해서는 34, 50-3, 50-4 레벨을 3시간 이내에 풀지 못했지만, 수정한 NuSMV를 이용해서는 1387초, 6474초, 10086초에 각각 해결하였다.

〈표 1〉 실험 결과[3)]

Level	Original NuSMV		Modified NuSMV	
	Time(sec)	Space(KB)	Time(sec)	Space(KB)
1	0.10	2,253	0.06	2,213
2	2.42	8,745	1.31	5,433
3	37.27	12,889	24.75	14,073
4	1.64	6,909	1.44	6,469
5	0.83	4,117	0.63	3,569
6	198.70	14,877	96.69	23,221
7	56.86	13,669	34.47	14,517
8	376.80	29,501	254.06	31,201
9	3.19	11,001	0.71	4,477
10	1.32	6,653	0.47	3,733
11	4.83	12,405	0.34	3,485
12	39.73	12,929	37.00	12,717
13	206.16	44,585	0.71	4,545
14	94.46	21,309	9.52	12,793
15	861.56	34,041	696.34	104,549
16	13.80	12,965	4.65	12,273
17	984.10	62,249	459.75	99,089
18	486.15	76,597	64.74	14,693
19	2,934.21	139,597	2,491.26	313,025
20	852.25	37,553	591.74	99,889
21	3,625.96	132,929	423.24	106,313
22	1,122.81	47,285	942.44	166,921
23	145.61	17,569	66.91	14,085
24	614.78	60,189	253.20	42,889
25	114.67	22,605	18.74	13,117
26	31.59	13,057	9.93	13,545
27	36.04	13,801	17.50	12,781
28	254.96	40,073	29.86	12,165
29	10.77	12,761	5.24	12,549
30	11,636.67	387,109	4,777.93	333,641
31	443.24	70,625	33.32	13,709
32	1,140.57	119,013	34.74	13,489
33	4,299.92	374,885	564.66	53,621
34	∞	∞	1,387.42	284,505
35	1,294.72	133,613	218.11	62,909
36	375.26	23,421	278.04	54,217
37	654.11	91,621	36.84	13,209
38	596.70	34,529	38.40	20,361
39	191.31	47,321	15.71	12,793
40	6,930.02	758,857	17.29	13,805
41	78.25	13,237	42.34	14,569
42	752.17	82,613	175.46	32,209
43	1,694.43	109,133	583.22	166,877
44	1,322.75	131,856	115.55	16,417
45	187.69	32,613	9.61	12,297
46	237.43	38,933	51.76	13,377
47	19.39	13,321	4.08	10,357
48	736.09	69,057	114.31	20,337
49	12,213.79	787,345	2,060.93	238,237
50-1	7,080.22	214,941	6,787.82	318,445
50-2	25,905.74	462,829	15,725.64	622,497
50-3	∞	∞	6,474.10	381,253
50-4	∞	∞	10,086.04	568,233



(그림 5) 수정된 NuSMV에서 반례 생성 시간이 평균 62% 절감되었다



(그림 6) 수정된 NuSMV에서 메모리 사용량이 평균 11% 절감되었다

5. 결 론

본 논문의 초점은 범용 모델 검사 도구인 NuSMV를 이용하여 푸쉬 푸쉬 게임을 효과적으로 푸는 것이었다. 이를 위해서 NuSMV의 반례 생성 부분을 게임 풀이(더 엄밀히 얘기하자면 정형 부정)에 맞게 수정하였다. 그 결과 수정된 NuSMV는 기존 NuSMV에 비해서 62%의 시간 절감과 11%의 공간 절감을 보였다. 뿐만 아니라 기존 NuSMV로는 3시간이 경과되어도 풀지 못했던 34, 50-3, 50-4 레벨을, 수정한 NuSMV로 1387초, 6474초, 10086초 만에 각각 해결하였다.

앞으로의 연구 방향은 다음과 같다. 모델 검사의 가장 큰 문제점은 상태 폭발 문제이다[12]. 특히 게임 풀이의 경우 고려해야 할 상태 수는 매우 거대하다. 예를 들어, 푸쉬 푸쉬 게임 50번째 레벨의 경우 상태 공간이 매우 커서 한 번에 풀 수 없었다. 그래서 50판의 상태 공간을 4개의 적은 상태 공간으로 분할해서 차례대로 해결해야 했었다 [9]. 비록 이러한 기법을 이용해서 50번째 판을 해결할 수 있었지만, 얻어진 경로가 최단 경로임을 보장하지는 못한다. 따라서 상태 공간을 줄일 수 있는 다양한 방법을 연구하여, 50판과 같은 복잡한 게임을 한 번에 풀어 최단 경로를 얻는 연구를 하려고 한다.

3) 푸쉬 푸쉬 게임은 50단계로 구성되어 있다. 마지막 단계인 50판은 모델 검사로 풀기에는 매우 복잡해서 50-1, 50-2, 50-3, 50-4와 같이 4개로 나누어 차례대로 풀었다(자세한 사항은 [9]를 참조. [9]에서는 Cadence SMV로 실험하였음). 한편, ∞표시는 3시간 이상 수행했어도 결과가 나오지 않은 경우이다.

참 고 문 헌

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

[2] 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.

[3] E. A. Emerson, Temporal and modal logic, in the Handbook of Theoretical Computer Science : Formal Models and Semantics, J. van Leeuwen, editor, Elsevier, pp.995-1072, 1990.

[4] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu and H. Veith, "Counterexample-Guided Abstraction Refinement," in Proceedings of Computer Aided Verification, pp.154-169, 2000.

[5] 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.

[6] 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 Proceedings of CAV '02, 2002.

[7] K. L. McMillan, Symbolic Model Checking, Kluwer Academic Publishers, 1993.

[8] M. Huth, M. Ryan, Logic in Computer Science : Modeling and Reasoning about System, Cambridge University Press, 2000.

[9] 권기현, "모델 검증을 이용한 게임 풀이", 정보과학회지, 제21권 제1호, pp.7-14, 2003.

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

[11] Y. Lu, Automatic Abstraction in Model Checking, Ph.D. thesis, Carnegie Mellon University, Department of Electrical and Computer Engineering, 2000.

[12] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu and H. Veith, "Progress on the State Explosion Problem in Model Checking," in Proceedings of 10 Years Dagstuhl, LNCS 2000, pp.154-169, 2000.

권 기 현

e-mail : khkwon@kyonggi.ac.kr

1985년 경기대학교 전자계산학과(학사)

1987년 중앙대학교 전자계산학과(이학석사)

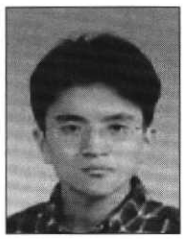
1991년 중앙대학교 전자계산학과(공학박사)

1998년~1999년 독일 드레스덴 대학 전자계산학과 방문교수

1999년~2000년 미국 카네기 멜론 대학 전자계산학과 방문교수

1991년~현재 경기대학교 정보과학부 교수

관심분야 : 소프트웨어 모델링, 소프트웨어 분석, 정형 기법 등



이 태 훈

e-mail : taehoon@kyonggi.ac.kr

2003년 경기대학교 전자계산학과(학사)

2003년~현재 경기대학교 전자계산학과 석사과정

관심분야 : 모델체킹, 소프트웨어 모델링