릴레이 모델 체킹을 이용한 상태 폭발 문제 해결
(Mitigating the State Explosion Problem using Relay Model Checking)

이 태 훈 † 권 기 현 ††
(Taehoon Lee) (Gihwon Kwon)

요 약 모델 체킹에서 고려해야 할 상태의 수는 모델의 크기에 따라 지수적으로 증가한다. 이것을 상태 폭발 문제라고 부르며 이를 해결하기 위한 방법으로 추상화, 반순서, 대칭성 등이 폭 넓게 사용되고 있다. 이들 방법들은 모델의 구조 정보를 이용하여 모델의 크기를 축소하는 데 목표를 두고 있다. 이와는 달리, 본 논문에서는 논리식을 순서적으로 분할하여 차례대로 모델 체킹을 수행하는 릴레이 모델 체킹을 제안한다. 그리고 기존 모델 체킹 기법으로 해결하지 못했던 상태 폭발 문제를 릴레이 모델 체킹으로 해결한 경험을 기술한다.

키워드 : 릴레이 모델 체킹, 시제 논리식, 상태 폭발 문제, 반례

Abstract In temporal logic model checking, the number of states is exponentially increased by the size of a model. This is called the state explosion problem. Abstraction, partial order, symmetric, etc. are widely used to avoid the problem. They reduce a number of states by exploiting structural information in a model. Instead, this paper proposes the relay model checking that decomposes a temporal formula to be verified into several sub-formulas and then model checking them one by one. As a result, we solve complex games that can't handle with previous techniques.

Key words : relay model checking, temporal logic formula, state explosion problem, counterexample

1. 서 론

Computation Tree Logic(CTL) 모델 체킹은 유한 상태 모델과 CTL 논리식을 받아서 모델과 논리식간의 만족성 관계를 검사한다[1]. 모델이 논리식을 만족하는 경우 모델 체킹은 참을 출력한다. 그렇지 않은 경우 모델 체킹은 거짓과 함께 반례를 출력한다. 예를 들어, CTL 논리식 $AG\neg\phi$를 살펴보자. 이 식의 의미는 '도달 가능한 모든 상태에서 항상 $\neg\phi$'이다. 만약 도달 가능한 상태들이 모두 $\neg\phi$라면 논리식 $AG\neg\phi$는 참이다. 그렇지 않은 경우, 즉 도달 가능한 상태 중에서 ϕ가 참인 상태가 하나라도 존재한다면 $AG\neg\phi$는 거짓이다. 이 경우, 모델 체킹은 초기 상태로부터 ϕ인 상태로 도달되는 경로를 반례로서 출력한다.

본 논문에서는 모델 체킹으로 생성된 반례를 이용하여 상자 이동 게임의 풀이 경로를 찾고자 한다. 상자 이동 게임의 목표는 주어진 상자(또는 공)를 목표 지점으로 모두 이동시키는 것이다. 푸쉬푸쉬[2], 소코반[3], 소코마인드[4] 등이 상자 이동 게임의 예이다. M을 상자 이동 게임의 유한 상태 모델이라 하고 $\phi_1,\cdots,\phi_n$을 목표 지점을 나타내는 단순 명제라고 하자. 예를 들어 ϕ_1 위치에 상자가 놓이면 명제식 ϕ_1은 참이고, 그렇지 않다면 ϕ_1은 거짓이다. 목표 상태, 즉 상자가 목표 지점에 모두 위치한 상태를 명제식으로 나타내면 $\phi_1\wedge\cdots\wedge\phi_n$이다. 본 논문의 주요 관심사는 모델 체킹을 통해서 게임의 초기 상태로부터 목표 상태로 가는 경로를 구하는 것이다. 이를 위해서 $M \not\models AG\neg(\phi_1\wedge\cdots\wedge\phi_n)$ 임을 보여야 한다. 왜냐하면 $AG\neg(\phi_1\wedge\cdots\wedge\phi_n)$이 거짓인 경우, 모델 체킹에 의해 생성된 반례는 초기 상태에서 목표 상태인 $\phi_1\wedge\cdots\wedge\phi_n$로 도달되는 경로 즉 게임을 풀 수 있는 경로이기 때문이다. 이러한 아이디어를 이용해서 푸쉬 푸쉬 50게임 중에서 37게임을 풀었다[5]. 그러나 나머지 13 게임은 상태 폭발 때문에 풀지 못했다. 상태 폭발을 방지하기 위해서 추상화 등 다양한 최적화 방법을 사용하여 모델의 상태 공간을 축소하였다. 그 결

· 본 연구는 과학기술부 목적기초연구(R05-2004-000-10612-) 지원으로 수행되었음
† 비 회 원 : 경기대학교 정보과학부
taehoon@kyonggi.ac.kr
†† 종신회원 : 경기대학교 정보과학부 교수
khkwon@kyonggi.ac.kr
논문접수 : 2004년 5월 19일
심사완료 : 2004년 9월 13일

과, 미 해결했던 13게임 중에서 12게임을 해결했다[6]. 추상화를 통해서 모델을 최적화했음에도 불구하고 맨 마지막 단계인 50번째 게임은 풀 수 없었다. 이것이 릴레이 모델 체킹을 연구하게 된 동기였다.

$M \not\models AG\neg(\phi_1 \wedge \cdots \wedge \phi_n)$을 보이는 경우를 살펴보자. 기존의 경우, 단 한번의 모델 체킹으로 $\phi_1,\cdots,\phi_n$이 동시에 모두 만족되는 상태를 찾으려고 하고, 거짓의 반례로서 초기 상태로부터 $\phi_1,\cdots,\phi_n$이 모두 만족되는 상태까지의 도달 경로를 출력한다. 만일 상태 폭발 문제가 발생해서 단 한번의 모델 체킹으로 $\phi_1,\cdots,\phi_n$이 모두 만족되는 상태를 구할 수 없다면, 차선책으로 모델 체킹을 여러 번 순차적으로 적용해서 $\phi_1,\cdots,\phi_n$이 동시에 만족되는 상태를 찾을 수 있다. 예를 들어 논리식 $AG\neg(\phi_1 \wedge \cdots \wedge \phi_n)$를 n개의 논리식 $AG\neg\phi_1$, $AG\neg(\phi_1 \wedge \phi_2)$,$\cdots$, $AG\neg(\phi_1 \wedge \cdots \wedge \phi_n)$으로 분할했다고 하자. 첫 번째 논리식을 모델 체킹해서 $M \not\models AG\neg\phi_1$를 보이면, 게임의 초기 상태에서 ϕ_1를 만족하는 상태까지 가는 경로를 얻는다. 이 경로를 바탕으로 모델의 초기 상태를 재구성한다. 즉 ϕ_1을 만족하는 상태를 모델의 초기 상태로 지정해서, 두 번째 논리식을 모델 체킹하면 $\phi_1 \wedge \phi_2$를 만족하는 상태까지 가는 경로를 얻는다. 이러한 과정을 반복하면 최종적으로 $\phi_1 \wedge \cdots \wedge \phi_n$를 만족하는 상태로 가는 경로를 얻는다. 마치 릴레이 경기에서 현재 주자가 다음 주자에게 배턴을 넘겨주는 것과 유사해서 릴레이 모델 체킹이라고 명명하였다.

모델 체킹시 상태 폭발 문제를 해결하기 위해서 추상화, 반순서 등의 기법이 사용되고 있다[7]. 이들의 목표는 모델의 구조 정보(이를테면 유사성, 대칭성, 계층성 등)를 이용하여 상태 공간을 축소하는 것이다. 이와는 달리 본 논문에서는 논리식 분할을 이용한 릴레이 모델 체킹을 제안하고, 기존의 모델 축소 기법으로 해결하지 못했던 상태 폭발 문제를 릴레이 모델 체킹으로 해결하였다. 특히 반복 횟수가 많아서 일반적인 모델 체킹을 사용할 수 없는 경우에, 본 논문에서 제안하는 기법을 적용할 수 있다. 그러나, 릴레이 모델 체킹으로 얻은 경로는 각 부분식을 만족하는 최단 경로일 뿐, 전체 논리식을 만족하는 최단 경로라고 할 수 없다. 따라서 릴레이 모델 체킹으로 얻은 경로를 게임을 풀 수 있는 힌트, 즉 초기 상태에서 목표 상태로 가는 힌트라고 여기고 이를 이용해서 좀더 적은 길이의 경로를 구하는 알고리즘을 제시한다.

본 논문의 구성은 다음과 같다. 2장에서는 모델 체킹에 관한 기본적인 내용을 살펴본다. 3장에서는 상태 폭발 문제 해결을 위해서 제안된 릴레이 모델 체킹을 설명하고, 4장에서는 릴레이 모델 체킹으로 얻어진 경로를 축소하는 알고리즘을 설명한다. 5장에서는 적용사례에 대해서 살펴보고, 6장에서는 결론과 향후 연구 과제에 대해서 살펴본다.

2. 배경 지식

2.1 모델

모델 체킹에 사용되는 모델의 핵심 요소는 상태와 상태간의 전이이며, 이들을 사용하여 시스템의 행위를 모델링 한다. CTL 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 $\pi = s_1 s_2 s_3 s_4 \cdots$는 전이 가능한 상태들을 차례대로 나열한 것으로서 $(s_i, s_{i+1}) \in R$, $i \geq 1$이며 그 길이는 무한이다.

2.2 논리식

모델에 관한 속성은 모델을 트리의 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 두 개의 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 얻는다. 전 장에서 언급했듯이 본 논문에서 사용하는 주요 CTL 논리식은 $AG\neg\phi$이며 이것의 쌍대는 $EF\phi$이다.[1] 따라서 본 절에서는 두 연산자에 국한해서 설명한다. $AG\phi$ 의미는 '도달 가능한 모든 상태에서 항상 ϕ'이며, $EF\phi$의 의미는 '언젠가는 ϕ인 상태로 도달 가능하다'이다. 모델 M의 상태 s에서 CTL논리식 ϕ가 참인 경우를 $M,s \models \phi$로 표시한다. $AG\phi$와 $EF\phi$의 정형적 의미는 다음과 같다:

$M,s \models AG\phi$ iff $\forall \pi = s_1,s_2,\cdots \bullet \forall i \geq 1 \bullet M,s_i \models \phi$

$M,s \models EF\phi$ iff $\exists \pi = s_1,s_2,\cdots \bullet \exists i \geq 1 \bullet M,s_i \models \phi$

여기서 $s = s_1$ 이다.

2.3 모델 체킹 알고리즘

모델 M의 모든 초기 상태에서 CTL논리식 ϕ가 참인 경우를 $M \models \phi$으로 표시하며 'M이 ϕ를 만족한다'라고

1) $\neg\phi \equiv \Psi$인 경우, ϕ와 Ψ는 쌍대(dual) 관계이다. $\neg(AG\neg\phi) \equiv EF\phi$이기 때문에 두 식은 쌍대이다.

읽는다(반대로, 만약 모델 M이 ϕ를 만족하지 않는 경우 $M \not\models \phi$로 표시한다). 모델 체킹은 M과 ϕ를 받아서 이들 간의 만족성 관계를 결정한다. 이를 위해서 ϕ를 만족하는 상태들의 집합을 구한 후, 이 상태 집합에 초기 상태가 포함되어 있는지를 검사한다. CTL 논리식 ϕ를 만족하는 상태 집합을 $[\![\phi]\!]$라고 표시하자. AGϕ, EFϕ를 만족하는 상태 집합은 다음과 같이 역 방향으로 계산된다:

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

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

여기서 μ, ν는 최소 고정점과 최대 고정점이다. 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.[2] AGϕ, EFϕ의 상태 집합 계산에 사용된 역 방향 탐색 함수는 다음과 같다:

$pre_{\forall}(Q) = \{s \in S \mid \forall_{s' \in S} \cdot (s,s') \in R \Rightarrow s' \in Q\}$

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

함수 $pre_{\forall}$는 집합 Q로만 도달되는 이전 상태들의 집합을 출력한다. 함수 $pre_{\exists}$는 $pre_{\forall}$와 비슷하지만 Q로 도달되는 이전 상태들을 모두 출력한다는 점에서 다르다. CTL 논리식 ϕ에 대한 상태 집합 $[\![\phi]\!]$을 구한 후, 모델 체킹 알고리즘의 핵심은 초기 상태 집합 I가 $[\![\phi]\!]$의 부분 집합인지를 검사하는 것이다. 즉 $M \models \phi$ iff $I \subseteq [\![\phi]\!]$이다. 일반적으로 M의 크기가 커질수록 $[\![\phi]\!]$을 계산하기 위해서 고려해야 할 전체 상태 수는 지수적으로 증가한다. 이를 상태 폭발 문제라고 하며, 이를 해결하는 것이 모델 체킹의 중요한 문제중의 하나이다.

2.4 반례 생성

$M \not\models \phi$인 경우, 다시 말해서 $I \not\subset [\![\phi]\!]$인 경우 모델 체킹은 그 이유를 담은 반례를 생성한다. 여기서는 $M \not\models$ AG$\neg\phi$의 반례 생성만을 설명한다. 위에서 설명한대로 AG$\neg\phi$의 쌍대는 EFϕ이기 때문에, AG$\neg\phi$의 반례는 EFϕ의 증거이다[5]. 그러므로 EFϕ의 증거로서 AG$\neg\phi$의 반례를 설명할 수 있다. 위에서 정의한대로 $[\![$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. 공집합으로부터 시작해서 함수 τ를 반복적으로 적용함으로서 최소 고정점을 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 1에서 보듯이 $S_n \cap I \neq \emptyset$이면 AG$\neg\phi$는 거짓이다.

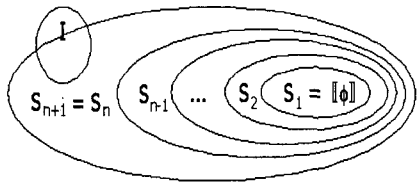

그림 1 $S_n \cap I \neq \emptyset$이면 AG$\neg\phi$는 거짓이다.

거짓인 경우, 그림 2에서 보는 것처럼 두 단계를 거쳐서 반례를 생성한다. 첫번째 단계는

$$S_1 = I$$
$$S_{i+1} = post_{\exists}(S_i) \cup S_i$$

와 같이 초기 상태에서 ϕ가 참인 상태까지 정방향 탐색을 진행하면서 도달 가능한 상태를 저장한다($S_i \cap [\![\phi]\!] \neq \emptyset$일 때 종료한다). 여기서 $post(Q) = \{s' \in S \mid \exists_{s \in Q} \cdot (s,s') \in R\}$는 Q로 부터 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 $\langle s_1, \cdots, s_n \rangle$를 찾는다.

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

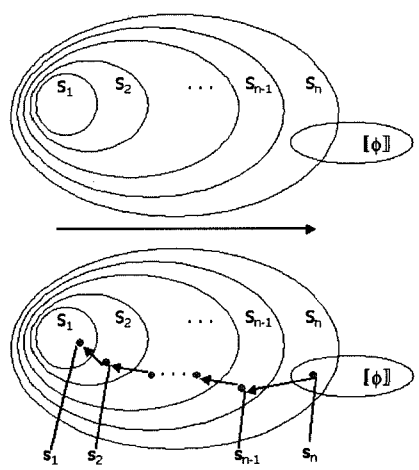

그림 2 정방향과 역방향 탐색을 통해서 반례 생성

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

3. 릴레이 모델 체킹

전장에서 언급했듯이 CTL 모델 체킹은 모델과 논리식을 받아서 모델과 논리식 간의 만족성 관계를 검사하기 때문에 모델 M=(S, I, R, L)에 대한 논리식 AG$\neg(\phi_1\wedge\cdots\wedge\phi_m)$의 모델 체킹을

$$MC(M,\mathrm{AG}\neg(\phi_1\wedge\cdots\wedge\phi_m))=\begin{cases}\varepsilon & \text{if } M\models\mathrm{AG}\neg(\phi_1\wedge\cdots\wedge\phi_m)\\ \pi & \text{if } M\not\models\mathrm{AG}\neg(\phi_1\wedge\cdots\wedge\phi_m)\end{cases}$$

함수로 표현할 수 있다. $M\not\models$AG$\neg(\phi_1\wedge\cdots\wedge\phi_m)$인 경우 반례 $\pi=<s_1,\cdots,s_n>$는 길이가 1이상인 유한 시퀀스로 다음 특성을 갖는다:

1. $s_1\in I$
2. $(s_i, s_{i+1})\in R$, where $i\geq 1$
3. $s_n\in [\![\phi_1\wedge\cdots\wedge\phi_m]\!]$

시퀀스를 조작하는 *tail*, *last*는 시퀀스의 첫번째 항목을 제외한 나머지 시퀀스와 시퀀스의 마지막 항목을 출력하는 함수이다. 즉, $\pi=<s_1,s_2,\cdots,s_n>$의 경우 $tail(\pi)$ = $<s_2,\cdots,s_n>$, $last(\pi)$ = s_n이다. 기호 $\wedge$는 시퀀스 결합 연산자로 $<s_1>\wedge<s_2,\cdots,s_n>$ = $<s_1,s_2,\cdots,s_n>$이다.

정리 1. 두 모델 M_i=(S, I_i, R, L)와 M_j=(S, I_j, R, L)를 차례대로 모델 체킹한다고 가정하자. 여기서 M_j의 시작 상태는 M_i에서 생성된 반례의 마지막 상태이다. 초기 상태 집합 I_i, I_j가 $|I_i|$ = $|I_j|$ = 1로서 단일 집합이라면 두 모델을 차례대로 모델 체킹하는 경우 다음이 성립한다.

$$\begin{pmatrix} MC(M_i,\mathrm{AG}\neg\phi_i)=\pi_i \\ \wedge \\ I_j=\{last(\pi_i)\} \\ \wedge \\ MC(M_j,\mathrm{AG}\neg(\phi_i\wedge\phi_j))=\pi_j \end{pmatrix}\Rightarrow\begin{pmatrix} MC(M_i,\mathrm{AG}\neg(\phi_i\wedge\phi_j)) \\ =\pi_i\hat{}\,tail(\pi_j)\end{pmatrix}$$

증명. $(x,u)\in R$ 그리고 $(v,w)\in R$라고 하자. 만약 u = v이라면, 추이적 성질에 의해서 $(u,w)\in R$이다. 이와 같은 관계 성질을 이용해서 정리를 증명한다. 반례 $\pi_i=<s_1^i,\cdots,s_{n-1}^i,s_n^i>$를 가정하자. 정의 1과 전제에 의해 다음이 만족된다:

1. $s_1^i=I_i$
2. $(s_k^i,s_{k+1}^i)\in R,\ k\geq 1$
3. $(s_{n-1}^i,s_n^i)\in R$
4. $last(\pi_i)=s_n^i=I_j$

마찬가지로 반례 $\pi_j=<s_1^j,s_2^j,\cdots,s_n^j>$를 가정하면 다음이 만족된다:

5. $s_1^j=I_j$
6. $(s_1^j,s_2^j)\in R$
7. $(s_k^j,s_{k+1}^j)\in R,\ k\geq 1$
8. $s_n^j\in [\![\phi_i\wedge\phi_j]\!]$

위의 3, 6에 의하면 $(s_{n-1}^i,s_n^i)\in R$, $(s_1^j,s_2^j)\in R$이다. 4, 5에서 보듯이 $s_n^i=s_1^j$이기 때문에 $(s_n^i,s_2^j)\in R$이다. 그러므로 AG$\neg(\phi_i\wedge\phi_j)$의 반례에 해당하는 시퀀스 $<s_1^i,\cdots,s_{n-1}^i,s_n^i,s_2^j,\cdots,s_n^j>=\pi_i\hat{}\,tail(\pi_j)$가 존재한다. □

정리 2. 모델 M_i=(S, I_i, R, L) 이 m개 있다고 하자. 모델의 초기 집합이 모두 단일 집합일 때($|I_i|$ = 1), M_1부터 M_m까지의 모델을 차례대로 모델 체킹하면 다음이 성립한다.

$$\begin{pmatrix} MC(M_1,\mathrm{AG}\neg\phi_1)=\pi_1 \\ \wedge \\ I_2=\{last(\pi_1)\} \\ \wedge \\ MC(M_2,\mathrm{AG}\neg(\phi_1\wedge\phi_2))=\pi_2 \\ \cdots\cdots\cdots \\ \wedge \\ I_m=\{last(\pi_{m-1})\} \\ \wedge \\ MC(M_m,\mathrm{AG}\neg(\phi_1\wedge\phi_2\wedge\cdots\wedge\phi_m))=\pi_m \end{pmatrix}\Rightarrow\begin{pmatrix} MC(M_1,\mathrm{AG}\neg(\phi_1\wedge\phi_2\wedge\cdots\wedge\phi_m)) \\ =\pi_1\hat{}\,tail(\pi_2)\hat{}\cdots\hat{}\,tail(\pi_m)\end{pmatrix}$$

증명. 정리 1의 따름 정리로서 정리 1의 증명을 이용해서 쉽게 증명될 수 있다 (지면상 여기서는 상세한 증명 과정을 생략한다). □

정리 2는 릴레이 모델 체킹의 이론적인 개념을 설명하는 것으로서, m번의 순차적인 모델 체킹으로 얻어진 반례는 원래 모델에 존재하는 반례이다.

```
Algorithm RelayModelChecking
input
  M = (S,I,R,L) is the original model
  AG¬(φ1 ∧ ... ∧ φm) is the formula to be checked
  {C1,...,Cm} is the set of constraints
output
  π is the counterexample for AG¬(φ1 ∧ ... ∧ φm)
begin
  M1 = (S,I1,R1,L) is constructed from M = (S,I,R,L)
                   with I1 = I, R1 = R ∧ C1;
  π = MC(M1, AG¬φ1);
  i = 1;
  repeat
    Mi+1 = (S,Ii+1,Ri+1,L) is constructed from Mi = (S,Ii,Ri,L)
                   with Ii+1 = {last(π)}, Ri+1 = Ri ∧ Ci;
    π = π^tail(MC(Mi+1, AG¬(φ1 ∧ ... ∧ φi+1)));
    i = i + 1;
  until(i < m)
  return π;
end
```

그림 3 릴레이 모델 체킹 알고리즘

그림 3은 릴레이 모델 체킹 알고리즘을 보이고 있다. 릴레이 모델 체킹 기법은 불변식 AG ¬($\phi_1 \wedge \cdots \wedge \phi_m$)의 정형 부정과 그것의 반례 생성을 목적으로 고안되었다. 특히 풀이 경로가 긴 게임을 해결할 때 유용하게 사용될 수 있다. 릴레이 모델 체킹에 관련된 관찰은 다음과 같다:

관찰 1. 릴레이 모델 체킹 기법을 이용해서 반례를 얻었다면, 생성된 반례는 실제 모델에서도 존재하는 반례이다. 왜냐하면 릴레이 모델 체킹 동안에 모델의 행위에 대한 변경이 없었기 때문이다. 하지만 릴레이 모델 체킹으로 반례를 찾지 못했다고 해서, 모델에 반례가 없다고 말할 수 없다. 다른 가능한 반례가 있는지 다시 검사해야 한다. 따라서 반례 관점에서 릴레이 모델 체킹은 보존적(conservative) 방법이지 완전한 방법은 아니다.

관찰 2. 기존의 상태 폭발 방지 기법들은 모델의 구조 정보(이를테면 유사성, 대칭성, 계층성 등)를 활용하였다. 반면에 릴레이 모델 체킹은 논리식 분할을 이용한다. AG ¬$\phi_1 \wedge \cdots \wedge \phi_m$가 주어졌을 때, 논리식을 분할하는 방법은 도메인에 따라서 다르다. 일반적으로 어떤 ϕ_i에 한번 상자가 놓인 이후에 계속해서 상자를 유지한다면, 그 ϕ_i를 먼저 모델 체킹할 수 있도록 앞쪽에 위치시키는 것이 좋다. 5 장에서 보겠지만 게임 풀이의 경우, 상자가 한 번 놓이면 다시는 상자를 꺼낼 수 없는 가장 안쪽에 있는 위치를 먼저 검사했다.

관찰 3. 그림 3의 릴레이 모델 체킹 알고리즘에는 제약 조건들 $C_1,\cdots,C_m$이 사용되었다. 비록 i번째 릴레이 모델 체킹이 성공했지만, i+1번째에서는 실패할 수도 있다. 왜냐하면 각 단계의 모델 체킹은 그 이후를 전혀 고려하지 않고 현재 단계에만 집중하기 때문이다. 마치 전체 숲을 보지 않고 나무만을 보면서 경로를 찾아가는 것과 같다. 따라서 올바른 경로를 찾을 수 있도록 힌트를 제공해야 한다. 즉 i+1번째의 성공을 위해서 i번째 해서는 안될 제약 사항을 표현한 것이 C_i이다. 제약 사항들을 찾아내기 위해서는 도메인 지식이 많이 요구된다. 제약 조건 C_i는 명제의 연접으로 구성된 논리식으로서, 성공적인 i+1번째 모델 체킹을 위해서 상자가 위치해서는 안되는 상태를 나타낸다. 따라서 i+1번째의 모델 체킹에 사용되는 전이 관계는 $R_{i+1} = R_i \wedge C_i$가 된다.

관찰 4. 릴레이 모델 체킹 기법으로 얻은 경로는 각 부분식을 만족하는 최단 경로이지, 전체 논리식을 만족하는 최단 경로라고 할 수 없다. 따라서 릴레이 모델 체킹으로 얻은 경로를 초기 상태에서 목표 상태로 가는 힌트라고 여기고, 이를 이용해서 좀더 적은 길이의 경로를 구하는 것이 필요하다.

4. 경로 최적화

전장의 관찰 4에서도 언급했지만, 릴레이 모델 체킹으로 결과를 얻었다고 해도 그 결과는 최단 경로가 아닐 수 있다. 따라서 릴레이 모델 체킹의 결과를 바탕으로 보다 적은 길이의 경로를 구할 필요가 있다. 이번 장에서는 좀더 적은 길이의 경로를 얻는 방법을 제안한다. 앞에서 살펴본 바와 같이, 기존 모델 체킹에서는 속성을 검사하는 경우 도달 가능한 상태를 모두 검사한다. 하지만 도달 가능한 상태 공간이 클 경우 상태 폭발이 발생한다. 이런 문제점을 방지하기 위하여 본 논문에서는 릴레이 모델 체킹을 제안했지만, 이 방법으로 얻어진 경로가 최단인지를 확신할 수는 없다. 최단 경로와 릴레이 모델 체킹으로 얻은 경로를 살펴보면, 두 경로에 공통으로 속하는 상태와 그렇지 않은 상태들로 구분할 수 있다. 본 논문에서는 이와 같은 구분을 이용해서 릴레이 모델 체킹에서 얻은 경로로부터 좀더 적은 길이의 경로를 얻는 방법을 제안한다.

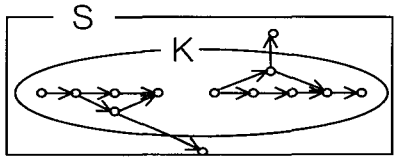

그림 4 경로 최적화

그림 4는 경로 최적화 알고리즘을 간략히 도식화하고 있다. 기존 모델 체킹에서는 전체 상태 공간 S를 탐색하려고 했기 때문에 상태 폭발이 발생했다. 전체 상태 공간 S 보다 적은 상태 공간을 고려하면서 좀더 적은 경로를 구하는 방법을 살펴보자. 릴레이 모델 체킹으로 초기 상태에서 목표 상태까지 가는 경로를 얻은 후, 이 경로를 바탕으로 해서 아래와 같이 상태 집합 K를 구한다면, 좀더 적은 길이의 경로를 찾을 수 있다.

릴레이 모델 체킹에서 얻어진 경로를 바탕으로 해서 K를 얻는 방법은 다음과 같다. P를 임의의 경로에 있는 상태 집합이라 하고, W를 릴레이 모델 체킹에서 나온 경로에 있는 상태 집합이라고 하자. 이때 P안에 있는 상태들과 W에 있는 상태들은 동일한 상태들도 있고 아닌 상태들도 있다. 이때 $d = \#(P \setminus W)$는 릴레이 모델 체킹으로 찾아내었지만 임의의 경로에 속하지 않은 상태들의 수이다. 이러한 d는 W에서 P를 얻을 수 있는 힌트가 된다. 도달 가능한 다음 상태 집합을 얻는 함수를 아래와 같이 정의하자:

$$post^n(Q) = post^{n-1}(Q) \cup \{s' \in S \mid \exists_{s \in S} \bullet (s,s') \in R \wedge s \in Q\}$$

$$post^0(Q) = Q$$

$$reachable\_state(X,n) = post^n(X)$$

K = *reachable_state*(W,d)를 상태 W에서 d번 이내에 도달 가능한 상태들의 집합이라 할 때, 도달 가능한 상태를 검사해 보면 $P \subset K$이고 $W \subset K$이다. 왜냐하면 P에 있는 상태들과 W에 있는 상태들은 모두 초기 상태에서 도달 가능한 상태이다. 즉 $I \subset P$이고 $I \subset W$이다. 따라서 P에 있는 상태들은 W에 있는 상태들에서 도달 가능하게 된다. W에서 d번 만큼 진행한 K를 얻으면 K는 P를 포함하고 있으며, 이때 P 상의 경로가 원래 경로보다 더 적은 경로라고 한다면, 원래 경로 W 보다 좀더 적은 길이의 경로를 얻을 수 있다. 하지만 일반적으로 주어진 경로의 상태들이 최단 경로상의 상태에 속하지 않는 숫자 d를 정확히 얻을 수는 없다. 따라서 본 이론을 이용해서 얻은 상태들의 집합 K에는 항상 최단 경로가 포함되어 있다고 할 수는 없다. 그러나 경험적으로 이것은 릴레이 모델 체킹을 이용해서 얻은 결과 보다 좀더 적은 결과를 얻을 수 있었고, 반복적으로 수행해서 원래보다 좀더 적은 길이의 경로를 얻을 수 있었다.

5. 적용 사례

5.1 푸쉬 푸쉬 게임

본 절에서는 릴레이 모델 체킹을 통해서 푸쉬 푸쉬 50번째 게임을 풀었던 경험을 설명한다. 그림 5에서 보듯이 10개의 상자(원으로 표시)와 10개의 목표 지점(빗금친 부분)이 있다. 상자들을 목표 지점으로 모두 이동시킬 수 있는 경로를 찾는 것이 게임의 목표이다.

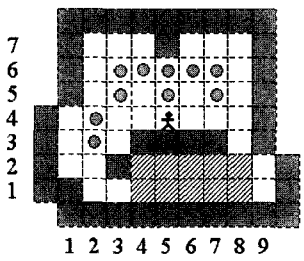

그림 5 푸쉬 푸쉬 50번째 게임

모델 체킹에 입력될 논리식은 AG ¬(41∧42∧51∧52∧61∧62∧71∧72∧81∧82)이다. 편의상 각 위치를 나타내는 숫자를 부울 변수로 간주한다. 예를 들어, 42는 해당 위치에 상자가 있음을 의미하는 반면에 ¬42는 해당 위치에 상자가 없음을 나타낸다. 그림 5의 게임을 유한 상태 모델로 변환한 후 모델 체킹을 시도했지만 실패했다[2]. 추상화를 통해서 게임의 상태 공간을 축소한 후에 모델 체킹했지만 그 역시 실패했다[3]. 그래서 아래와 같이 논리식을 4개로 분할하여 차례대로 모델 체킹을 하였다:

1회: AG ¬(**42**)

2회: AG ¬(42∧**52∧62**)

3회: AG ¬(42∧52∧62∧**72∧41∧51**)

4회: AG ¬(42∧52∧62∧72∧41∧51∧**61∧71∧81∧82**)

이전 단계에 비해 현재 단계에서 고려되는 부분은 글씨체를 진하게 표시했다. 전 장의 관찰 2에서 언급했듯이 안정된 상태를 갖는 위치를 전진 배치하였다. 릴레이 모델 체킹의 진행 과정은 그림 6과 같다. 처음에 상자 한 개를 42에 집어넣었고, 그리고 나서 상자 두 개를 52, 62에 넣었다. 그 다음에 상자 세 개를 72, 41, 51에 넣었고, 마지막으로 남아있는 모든 상자를 61, 71, 81, 82에 넣게 하였다. 그 결과 기존 모델 체킹 기법으로 풀 수 없었던 게임을 동일한 환경에서 릴레이 모델 체킹으로 풀 수 있었으며, 릴레이 모델 체킹 결과338 길이를 갖는 반례가 생성되었다. 즉 목표 지점으로 상자를 모두 이동하는데 338 번의 움직임이 요구되었다. 릴레이 모델 체킹에 소요된 시간과 메모리 사용량은 표 1에 있다. 1기가 메모리를 가진 펜티엄4에서 수행하였으며 모델 체커로는 NuSMV [9]을 사용하였다.

표 1 푸쉬 푸쉬 50번째 수행 결과

회수	시간(Sec)	메모리(KB)	BDD 수
1	102.09	20,909	360,190
2	644.35	69,413	1,527,837
3	1422.40	132,849	2,874,495
4	89.05	20,217	440,698

릴레이 모델 체킹으로 얻은 경로 상에 있는 상태들을 하나의 BDD로 표현했다. 이것을 확장한 K의 크기가 1기가 메모리에서 처리할 수 있는 용량인 9,000,000개 이상의 BDD 노드를 가질 때 까지 *pre*(Q) 혹은 *post*(Q)를 계산하도록 했다. 경로 최적화 알고리즘을 활용하여 확장된 BDD 내에서만 검사를 수행한 결과, 릴레이 모델 체킹에서 얻어낸 338번의 움직임을 16개 줄여322번

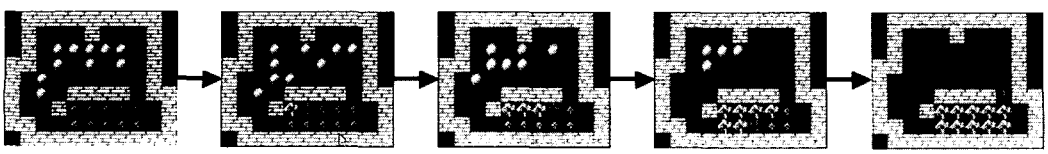

그림 6 푸쉬 푸쉬 50번째 게임의 릴레이 모델 체킹

움직임만에 푸쉬 푸쉬 50번째 게임을 풀 수 있었다. 다행히 50번째 게임을 최단 풀이 경로의 길이는 322으로 판명되었다.

5.2 소코반 게임

그림 7은 두 번째 사례로서 XSokoban 51번째 게임의 풀이 과정을 보여준다. 이 게임 역시 기존 모델 체킹 기술로는 해결할 수 없었다. 하지만 그림에서 보듯이 9단계로 나누어 릴레이 모델 체킹한 결과 371번의 움직임을 갖는 풀이 경로를 찾아내었다(수행 결과는 표 2를 참조). 371이 최단 길이가 아니라서, 경로 최적화 알고리즘을 이용하여 반복적으로 줄인 결과 339번의 움직임을 갖는 풀이 경로를 얻었다. 푸쉬 푸쉬와는 달리, 한번만에 줄인 것이 아니라8번 반복 수행한 끝에 339번의 움직임을 갖는 풀이 경로를 얻었다(움직임의 감소 과정은 371→358→354→350→346→344→342→341→339 와 같다). 경로 최적화를 통해서 경로의 길이를 32개 줄였지만, 안타깝게도 최적 풀이 경로의 길이는 339가 아니라 335로 판명되었다. 릴레이 모델 체킹으로 얻은 경로를 최적의 경로로 줄이는 연구가 필요하다.

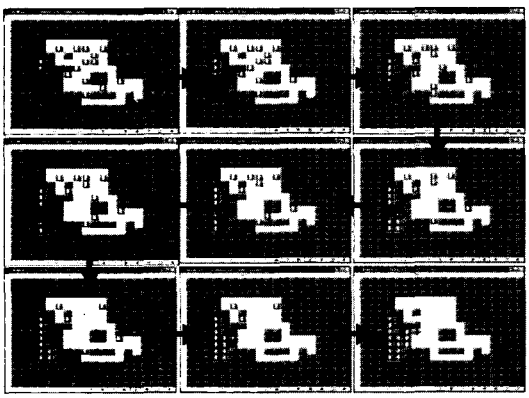

그림 7 XSokoban 51번째 게임의 릴레이 모델 체킹

표 2 XSokoban 51번째 수행 결과

회수	시간(Sec)	메모리(KB)	BDD 수
1	7.94	13,460	249,864
2	7.22	13,312	266,136
3	8.99	13,272	327,765
4	7.68	13,264	320,587
5	360.25	109,620	2,410,499
6	8.18	13,628	251,258
7	70.71	28,696	627,580
8	26.51	14,332	382,639
9	14.66	13,485	297,256

6. 결 론

상태 폭발을 해결하기 위해 추상화, 반순서 등이 사용되고 있다. 이들의 목표는 모델의 구조 정보(이를테면 유사성, 대칭성, 계층성 등)를 이용하여 상태 공간을 축소하는 것이다. 이와는 달리 본 논문에서는 논리식 분할을 이용한 릴레이 모델 체킹을 제안하였고, 기존의 모델 축소 기법으로 해결하지 못했던 복잡한 게임을 릴레이 모델 체킹으로 풀었다. 모델의 행위에 대해서는 변경을 하지 않기 때문에 결과가 나오면 그 결과는 실제 모델에도 존재한다. 따라서 복잡한 게임을 해결하는 중요 기술 중의 하나로 사용될 수 있다.

릴레이 모델 체킹은 일반적으로 반례를 기반으로 하고 일반적으로 초기 상태에서 어떤 상태에 도달할 수 있는지를 검사할 때 유용하게 사용될 수 있다. 최근 추상화 기법으로 반례 기반의 추상화 개선[10] 이 많이 사용되고 있다. 릴레이 모델 체킹 역시 반례 기반 추상화 기법과 같은 접근 방법에서 유용하게 사용될 수 있을 것이라 생각한다.

그러나 모델 체커인 NuSMV의 입력언어는 다양한 종류의 시스템을 모델링 하기에 불편한 점이 많이 있었고, 반례로서 하나의 경로만을 사용자에게 알려준다. 그리고 현재 AG 의 반례에 대해서만 릴레이 모델 체킹을 수행할 수 있었다. 따라서 좀더 쉽게 모델링을 할 수 있는 언어에 대한 연구와 반례 생성시 좀더 많은 정보를 제공할 수 있는 연구가 필요하다. 그리고 다른 CTL 속성에 적용 가능성도 살펴봐야 한다. 또한 게임 외의 다른 적용 사례 연구도 역시 필요하다.

참 고 문 헌

[1] E.M. Clarke, O. Grumberg, and D. Peled, Model Checking, MIT Press, 1999.
[2] http://kuic.kyonggi.ac.kr/~khkwon/game/pushpush.html
[3] http://www.cs.cornell.edu/andru/xsokoban.html
[4] http://www.sokominde.de
[5] 권기현, "모델 검증을 이용한 게임 풀이", 정보과학회 학회지, 제21권 제1호, pp.7-14, 2003년.
[6] G. Kwon, "Applying Model Checking Techniques to Game Solving," In the Proceedings of the SERA'03, pp.88-93, 2003.
[7] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Progress on the State Explosion Problem in Model Checking," in the Proceedings of 10 Years Dagstuhl, LNCS 2000, pp.154-169, 2000.
[8] E.M. Clarke, O. Grumberg, K.L. McMillan, and X. Zhao, "Efficient Generation of Counterexamples and Witness in Symbolic Model Checking," in the Proceedings of Design Automation Conference,

pp.427-432, 1995.
[9] A. Cimatti, et.al., "NuSMV 2: An OpenSource Tool for Symbolic Model Checking," in the Proceedings of CAV'02, 2002.
[10] E.M. Clark, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-Guided Abstraction Refinement," in the Computer Aided Verification, pp 154-169, 2000.

이 태 훈
2003년 경기대학교 전자계산학과(학사) 2003년~현재 경기대학교 전자계산학과 석사과정. 관심분야는 소프트웨어 공학, 소프트웨어 분석, 정형 기법 등

권 기 현
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 관심분야는 소프트웨어 모델링, 소프트웨어 분석, 정형 기법 등