한국게임학회 논문지
제4권 제2호 2004년 6월

모델 체킹을 이용한 도망자-추적자 게임 풀이

박사천*, 권기현**
경기대학교 정보과학부
sachem@kyonggi.ac.kr*, khkwon@kyonggi.ac.kr**

Solving Escapee-Chaser Game via Model Checking

Sa-Choun Park*, Gi-Hwon Kwon**
Information Science Dept. Kyonggi University

요 약

우리는 도망자-추적자 게임 풀이에 관심이 많다. 도망자가 추적자를 피해 미로를 탈출하는 게임이다. 도망자가 추적자를 유인하기도 하고 벽을 이용해 교묘히 피하기도 한다. 경험에 의하면 수작업으로 풀기에는 매우 어려운 단계들도 있었다. 게임을 풀기 위한 방법으로 모델 체킹 기법을 사용하였다. 모델 체킹은 게임의 모든 상태 공간을 넓이 우선 방식으로 철저하게 탐색하기 때문에 게임을 풀 수 있는 가장 짧은 경로인 최적의 답을 구할 수 있다. 다행히 풀이 과정에서 상태 폭발 문제는 일어나지 않았고, 게임 풀이 결과를 임베디드 시스템인 레고 마인드스톰에 응용하였다. 도망자, 추적자에 해당하는 두 대의 에이전트를 만들어 게임을 구현하고 실험하여 풀이가 정확한 지를 실제 확인할 수 있었다.

Abstract

We have been interested in solving escapee-chaser game. In this game, with avoiding chaser, the escapee must escape from given maze. The winning strategies of the escapee are driving the chaser to an intended place and closely evading from chaser by using some walls. According to our experience, some stages of the game are too difficult to solve manually. So we take the model checking method to get a solution of the game. Because the model checking with breadth first search manner exhaustively searches the all state space of the game, the solution using model checking is best solution, shortest path. Fortunately, during the process of finding solution path, the state space explosion problem didn' t occur, and the results of the game solving was applied to embedded system, Lego Mindstorm. Two agents, escapee and chaser, were implemented into robots and several experiments conformed the correctness of our solution.

Key Words : ModelChecking, GameSolving, Maze game, Lego Mindstorms

1. 서론

우리는 도망자-추적자 게임에 관심이 많다. 미로 속에 도망자와 추적자를 두고 도망자가 추적자를 피해 미로를 탈출하는 게임이다. 이 게임은 1단계부터 15단계까지 구성이 되어 있다. 경험에 의하면 14, 15단계는 손으로 풀기에는 매우 어려웠다. 따라서 도망자-추적자 게임을 풀기 위해 정형 검증 기술인 모델 체킹 기법을 사용하였다.

시스템은 사용자의 요구 사항을 반드시 만족시켜줘야만 한다(본 연구에서 시스템은 게임이 되고 요구 사항은 탈출

경로가 된다) 더욱이 안전 필수 시스템의 경우는 고 안정성을 요구하기 때문에 약간의 오류가 일어나도 치명적인 결과를 초래할 수가 있다. 따라서 안정성 보장을 위해 시스템의 요구 사항을 만족시키기 위한 검증 기법에는 크게 테스팅과 시뮬레이션 그리고 모델 체킹을 들 수 있다. 이들 모두 시스템의 오류를 찾아내지만 모델 체킹처럼 모든 상태 공간을 철저하게 탐색하면서 찾기 어려운 오류들을 찾아내는 방법은 없다. 테스팅은 일부의 테스트케이스에 의해서 시스템을 조사하며, 시뮬레이션도 시스템의 전체경로를 검사하는 것이 아니지만 모델 체킹은 시스템의 모든 행위를 검사하기 때문이다. 모델 체킹에서 만족해야 할 요구 사항을 속성이라 부르며 고 품질의 시스템 개발을 위해서 반드시 속성이 만족되는지를 검증해야 한다. 모델 체킹은 시스템과 요구 사항을 각각 유한 상태 모델 M과 명제 시제 논리식로 축소 표현하고 이를 이용해 모델이 논리식을 만족하는지에 대한 만족성 여부 $M \models$ 를 결정한다. 모델 체킹은 사용자가 관여하지 않고 자동으로 이루어지게 되고 시스템의 요구 사항 즉, 모델의 논리식을 만족하지 않으면 그 이유로 반례가 생성되어 디버깅 작업이 수월해진다 [1].

예를 들어 시제 논리식 AG가 거짓인 경우 e가 참이 되는 경로를 생성한다. 이러한 경로를 반례라고 부르는데 이를 이용하여 디버깅 작업뿐만 아니라 문제 풀이에도 사용할 수 있다. 문제를 유한 상태 모델로 표현하고 오답을 시제 논리식으로 표현해서 모델 체커에 입력하면 모델 체커는 반례로써 오답에 대한 거짓 즉 정답을 출력한다. 이를 통해 문제를 풀 수가 있다 [2, 3].

이러한 문제 풀이 아이디어를 게임 풀이에 적용해 보았다. 게임의 규칙을 제약 조건으로 준수하면서 초기 상태에서 목표 상태까지 도달하게 하는 것을 게임의 목표로 한다. 물론 반례로 초기 상태에서 목표 상태까지 가는 경로를 얻어내게 된다. 모델 체킹을 통해 도망자-추적자 게임을 풀었을 때 다행히 상태 폭발 문제가 발생하지 않았고 쉽게 문제를 해결할 수 있었다. 그리고 도망자-추적자 게임 풀이를 레고 마인드스톰이라는 임베디드 시스템에 응용해 보았다. 게임을 실제 에이전트가 동작하는 환경으로 구현하였고 게임 풀이가 정확한지를 눈으로 직접 테스팅하고 확인할 수 있었다. 본 논문의 구성은 다음과 같다. 2장에서는 모델 검증 기법을 간략히 설명하고 3장에서는 게임을 해결하는 과정을 기술한다. 그리고 4장에서는 게임을 임베디드 시스템에 응용한 과정을 기술하고 마지막으로 5장에서 결론을 맺는다 [4, 8].

2. 배경 지식

2.1 모 델

모델 검사에 사용되는 모델의 핵심 요소는 상태와 상태간의 전이이며, 이들을 사용하여 시스템의 행위를 모델링 한다. 특히 CTL(Computation Tree Logic) 모델 검증에서는 크립키 구조라 불리는 모델 $M=(S, I, R, AP, L)$ 을 사용한다[7]. 여기서,

- S는 상태들의 집합이다.
- $I \subseteq S$ 는 초기 상태들의 집합이다.
- $R \subseteq S \times S$ 은 상태들 간의 전이를 나타내는 관계이다.
- AP는 단순 명제들의 집합이다.
- $L : S \rightarrow 2^{AP}$ 은 각 상태에서 참이 되는 단순 명제들을 해당 상태에 배정하는 함수이다.

여기서 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 분기 시제 논리 언어인 CTL 로 표현한다. CTL은 모델을 트리의 관점에서 해석한다. 즉 초기 상태를 루트로 해서 모델을 풀어헤치면 트리를 얻을 수 있다. 트리는 모델의 가능한 행위를 모두 표현하며, 트리의 각 경로는 모델이 가질 수 있는 특정한 행위를 나타낸다. 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. CTL은 이들 연산자를 추가하여 기존의 명제 논리를 확장한 논리 언어이다. CTL 전체 구문을 BNF(Backus-Noar Form)형식으로 나타내면 다음과 같다.

$$\Phi := p \mid \bot \mid \neg\Phi \mid \Phi_1 \wedge \Phi_2 \mid \Phi_1 \vee \Phi_2$$
$$\mid \Phi_1 \Rightarrow \Phi_2 \mid \Phi_1 \Leftarrow \Phi_2 \mid \Phi_1 \Leftrightarrow \Phi_2 \mid AX\Phi \mid EX\Phi$$
$$\mid AF\Phi \mid EF\Phi \mid AG\Phi \mid EG\Phi \mid A(\Phi_1 U \Phi_2)$$
$$\mid E(\Phi_1 U \Phi_2)$$

여기서 $p \in AP$ 는 임의의 단순 명제를 나타내며 $\bot, \top$ 는 각각 참과 거짓을 나타내는 상수이다. CTL 식 Φ, Ψ 의 값이 모든 모델과 모든 상태에서 동일하다면 두 식을 동치라고 부르며 $\Phi \equiv \Psi$ 로 표시한다. 동치 관계에 있는 식은 다음과 같다.

$$\top \equiv \neg\bot$$
$$\Phi_1 \vee \Phi_2 \equiv \neg(\neg\Phi_1 \wedge \neg\Phi_2)$$
$$\Phi_1 \Rightarrow \Phi_2 \equiv \neg(\neg\Phi_1 \wedge \neg\Phi_2)$$

$$\Phi_1 \Leftrightarrow \Phi_2 \equiv \neg(\neg\Phi_1 \wedge \neg\Phi_2) \wedge \neg(\neg\Phi_2 \wedge \neg\Phi_1)$$
$$AX\Phi \equiv \neg EX\neg\Phi, \qquad AF\Phi \equiv \neg EG\neg\Phi$$
$$AG\Phi \equiv \neg EF\neg\Phi$$
$$A[\Phi_1 \cup \Phi_2] \equiv E[\neg\Phi_2 \cup (\Phi_1 \wedge \neg\Phi_2)] \wedge \neg EG\neg\Phi_2$$

두 식이 동치이기 때문에 좌변의 식은 우변의 식으로 대치 가능하다. 따라서 우변에 나오는 연산자의 모임이 CTL 식을 정의하는데 요구되는 연산자 집합이다. 위의 경우는 $\{\bot, \neg, \wedge, EX, EG, EF, EU\}$ 이다. 이들을 이용해서 CTL 구문을 정의하면 다음과 같다.

$$\Phi := p \mid \bot \mid \neg\Phi \mid \Phi_1 \wedge \Phi_2 \mid EX\Phi \mid EF\Phi \mid EG\Phi \mid E(\Phi_1 U \Phi_2)$$

이제 CTL의 의미를 살펴보자. 모델 M의 상태 s 에서 CTL 식 Φ가 참인 경우를 $M, s \models \Phi$로 표시한다. 그렇지 않다면 $M, s \not\models \Phi$로 표시한다. 상태 s에서 CTL 식 Φ의 값은 아래와 같이 재귀적으로 정의된다.

$$M, s \models p \quad \text{iff} \quad p \in L(s)$$
$$M, s \models \neg\Phi \quad \text{iff} \quad M, s \not\models \Phi$$
$$M, s \models \Phi_1 \wedge \Phi_2 \quad \text{iff} \quad M, s \models \Phi_1 \text{ and } M, s \models \Phi_2$$
$$M, s \models EX\Phi \quad \text{iff} \quad M, s' \models \Phi \text{ for some state } s' \text{ with } (s, s') \in R$$
$$M, s \models EF\Phi \quad \text{iff} \quad \exists_{i \geq 0} \cdot M, s_i \models \Phi$$
$$M, s \models EG\Phi \quad \text{iff} \quad \exists_{\pi = s0, s1, s2, \cdots} \cdot \forall_{k \geq 0} \cdot M, s_k \models \Phi$$
$$M, s \models E(\Phi_1 U \Phi_2) \quad \text{iff} \quad \exists_{\pi = s0, s1, s2, \cdots} \cdot (\exists_{k \geq 0} \cdot M, s_k \models \Phi_2 \wedge \forall_{0 \leq i < k} \cdot M, s_i \models \Phi_1)$$

2.3 모델 체킹 알고리즘

모델 M의 모든 초기 상태에서 CTL 식 Φ가 참인 경우를 $M \models \Phi$로 표시하며 "M이 Φ를 만족한다"라고 읽는다. 모델 검증 문제란 M과 Φ를 받아서 "M이 Φ를 만족하는지를 결정"하는 문제이다. 이를 위해서 Φ를 만족하는 상태들의 집합을 구한 후, 이 상태 집합에 초기 상태가 포함되어 있는지를 검증한다. CTL 식 Φ를 만족하는 상태들의 집합을 $\|\Phi\|$ 라고 표시하자. 모델 검증 알고리즘은 모델의 초기 상태 집합이 $\|\Phi\|$ 의 부분 집합인 것을 검증한다. 즉,

$$M \models \Phi \quad \text{iff} \quad I \subseteq \|\Phi\|$$

이다. 단순명제 p의 경우 집합 $\|p\|$를 쉽게 계산 할 수 있다. EX, EU, EG를 다루는데 함수 $pre_\exists$가 사용된다.

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

이 함수는 집합 X로 전이 할 수 있는 이전 상태(predecessor)들의 집합을 역 방향으로 계산한다. CTL식 Φ를 만족하는 상태 집합 $\|\Phi\|$은 다음과 같다.

$$\|p\| = \{s \in S \mid p \in L(s)\}$$
$$\|\bot\| = \emptyset$$
$$\|\neg\Phi\| = S \setminus \|\Phi\|$$
$$\|\Phi_1 \wedge \Phi_2\| = \|\Phi_1\| \cap \|\Phi_2\|$$
$$\|EX\Phi\| = pre_\exists(\|\Phi\|)$$
$$\|EG\Phi\| = \nu Z.(\|\Phi\| \cup pre_\exists(Z))$$
$$\|E(\Phi_1 \cup \Phi_2)\| = \mu Z.(\|\Phi_2\| \cup (\|\Phi_1\| \cap pre_\exists(Z)))$$

여기서 μ, ν는 각각 최소 고정점과 최대 고정점이다. 상태 집합 $\|\Phi\|$을 계산할 때, 최소 고정점 ν는 공집합을 초기값으로 하여 계속해서 증가되다가 더 이상 증가하지 않는 집합을 얻게 될 때를 말하고 EF, EU, AF, AU를 계산할 때 사용한다. 최대 고정점 ν은 반대로 전체 집합을 초기값으로 하여 계속해서 감소하다가 더 이상 감소하지 않는 집합을 얻게 될 때를 말하고 EG, AG를 계산할 때 사용한다[5],[6].

모델 검증의 수행 시간은 역 방향 함수와 고정점계산에 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 검증 알고리즘은 Φ 의 서브식을 재귀적으로 구하면서 길이가 짧은 식부터 긴 식 순서로 상태 집합을 계산한다. 따라서 알고리즘의 복잡도는 $O(|M|*|\Phi|)$ 로서, 모델의 크기 및 식의 길이에 모두 선형 비례한다 [7].

3. 게임 풀이

3.1 게임 설명

도망자-추적자게임[1]은 플레이어가 도망자가 되어 미로 속에서 추적자를 따돌리고 탈출하는 게임이다. 게임은 도망자와 추적자가 움직일 수 있는 보드 위에서 이루어진다. 보드는 바둑판 형태로 되어있고 그 크기는 6x6에서 9x14까지 다양하다. 도망자와 추적자는 처음에 각자의 출발점에 놓여지며 도망자부터 벽이 없는 방향으로 교대로 움직일 수 있다. 도망자는 상하좌우 4방향으로 한번에 한 칸 움직이거나 추적자를 유인하거나 피하기 위해 그 자리에 머물 수 있다. 추적자는 도망자를 향하여 한번에 두 칸 이하로 움직일 수 있다. 그런데 추적자는 제약 조건을 가지고 있다. 도망자를 향하여 먼저 좌우부터 움직이게 되어있고 도망자와 일직선상에 놓일 때는 그 선상을 벗어나지 못한다. 아래 도망자-추적자 게임에서 굵은 선은 벽이고 붉은색 원은 도망자, 검은색 원은 추적자이다. 도망자는 벽을 교묘히 이용해서 추적자를 따돌리고 goal로 탈출할 수 있게 된다.

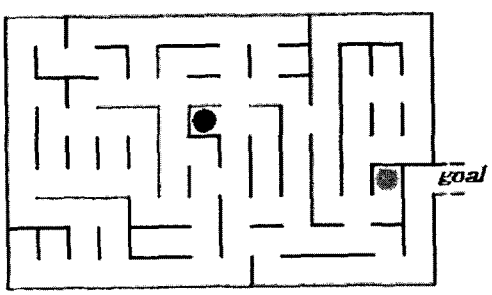

그림 1. 도망자-추적자 게임 15단계

1) http://www.tnelson.demon.co.uk/mazes/

게임은 1단계부터 15단계까지 구성되어있다. 단계가 증가할수록 게임의 복잡도는 증가하고 직접 풀이하기가 어려워진다. 도망자(플레이어)가 어떻게 움직이느냐에 따라 추적자의 이동 경로가 결정 되며 도망만 가는 것이 아니라 그 자리에 머물러 있을 수도 있고 추적자를 유인하여 교묘히 피해가기도 한다. 생각을 많이 해야 풀 수 있는 게임이다.

3.2 게임의 정형 모델

본 연구에서는 모델 체킹 기법을 게임 풀이에 적용했다. 게임 규칙을 준수하면서 도망자(플레이어)가 초기 상태에서 목표 상태까지 도달하는 탈출 경로를 찾아낸다. 탈출 경로를 찾아내기 위해서는 유한 상태 모델과 시제 논리식으로 속성을 기술하여 반례를 생성한다. 게임 G는 다음과 같은 튜플로 이루어진다.

$G = \langle A_e, A_c, C, D, goal \rangle$

- A_e : 도망자 에이전트
- A_c : 추적자 에이전트
- C : $(x, y) \rightarrow 2D$ 임의의 위치를 입력받아 그 위치에서 이동 가능한 방향을 돌려주는 함수이다.
- D : {*up, down, right, left, delay*} 에이전트의 이동방향 을 나타낸다.
- *goal* : (x_g, y_g)탈출 지점의 좌표를 순서쌍으로 표현한다.

먼저 도망자 에이전트의 행위를 유한 상태 모델 $Ae = (Se, le, \delta e, clock)$로 모델링 하면 다음과 같다.

- $S_e = \{(x_e, y_e) \mid 0 < x = n, 0 < y = m\}$는 $n \times m$의 도망자의 이동 가능한 위치 집합이다.
- $l_e \in S_e$는 도망자 에이전트의 초기 위치이다.
- $\delta_e : S_e \times D \rightarrow S_e$ 는 도망자의 전이 함수이다.
- *clock* : {0, 1}은 toggle된다. 에이전트는1일 때 움직이고 0일 때 움직이지 않는다.
  (도망자가 한 번 움직일 때 추적자는 두 번 움직이게 된다.)

움직일 수 있는 방향은 매번 하나이기 때문에 방향을 집합으로 모델링 하였다.

전이 함수 $e((x_e, y_e), d) = (x_e, y_e)$는 아래와 같이 정의된다. x'은 x의 다음 값이다.

$$\delta_e((x_e, y_e), d) =$$
$$\begin{cases} (x_e' = x_e - 1, y_e' = y_e) \\ \quad If\ d=left \wedge left \in C(x_e, y_e) \wedge clock=1 \\ (x_e' = x_e + 1, y_e' = y_e) \\ \quad If\ d=right \wedge right \in C(x_e, y_e) \wedge clock=1 \\ (x_e' = x_e, y_e' = y_e - 1) \\ \quad If\ d=up \wedge up \in C(x_e, y_e) \wedge clock=1 \\ (x_e' = x_e, y_e' = y_e + 1) \\ \quad If\ d=down \wedge down \in C(x_e, y_e) \wedge clock=1 \\ (x_e' = x_e, y_e' = y_e) \\ \quad If\ d=delay \wedge clock=0 \end{cases}$$

다음으로 추적자 에이전트를 유한 상태 모델 $A_c = \{S_c, I_c, \delta c\}$로 모델링 하면 다음과 같다.

- $S_c = \{(x_c, y_c) \mid 0 < x = n, 0 < y = m\}$ 는 $n \times m$ 추적자의 이동 가능한 위치 집합이다.
- $I_c \in S_c$는 추적자의 초기 위치이다.
- $\delta_c : S_c \times S_e \rightarrow S_c$ 는 추적자의 전이 함수이다.

추적자 역시 움직일 수 있는 방향은 매번 하나이기 때문에 방향을 집합으로 모델링 하였다. 위에서 설명했듯이 움직일 때 도망자와는 다른 제약 조건을 가지고 있다.

전이 함수 $S_c((x_c, y_c),(x_e, y_e)) = (x_c', y_c')$는 아래와 같이 정의된다.

$$S_c((x_c, y_c),(x_e, y_e)) =$$
$$\begin{cases} (x_c' = x_c - 1, y_c' = y_c) \\ \quad If\ (x_c < x_e) \wedge left \in C(x_c, y_c) \\ (x_c' = x_c + 1, y_c' = y_c) \\ \quad If\ (x_c > x_e) \wedge right \subset C(x_c, y_c) \\ (x_c' = x_c, y_c' = y_c - 1) \\ \quad If\ ((left \notin C(x_c, y_c) \vee (x_c = x_s) \vee right \notin C(x_c, y_c)) \\ \qquad \wedge (y_c < y_s) \wedge up \in C(x_c, y_c)) \\ (x_c' = x_c, y_c' = y_c + 1) \\ \quad If\ ((left \notin C(x_c, y_c) \vee (x_c = x_s) \vee right \notin C(x_c, y_c)) \\ \qquad \wedge (y_c > y_e) \wedge down \in C(x_c, y_c)) \\ (x_c' = x_c, y_c' = y_c) \\ otherwise \end{cases}$$

이제 반례 생성을 위한 속성을 기술 한다. 게임을 풀기 위해 시제 논리식으로 오답을 표현하고 반례로서 도망자의 초기 상태부터 목표 상태까지의 탈출 경로인 정답이 출력된다. 속성은 다음과 같은 시제 논리식으로 나타낸다.

$$\neg E[\neg((x_e = x_c) \wedge (y_e = y_c) \cup (x_e = x_g) \wedge (y_e = y_g))]$$

식의 의미는 '도망자가 목표 지점에 도달할 때까지 추적자와 만나지 않는 경로는 존재하지 않는다'이다. 이 식은 실제 오답을 기술한 것으로 모델 체커를 통해 검증하는 과정에서 반례 즉 정답을 출력한다. 출력된 정답의 의미는 '도망자가 목표 지점에 도달할 때까지 추적자와 만나지 않는 경로가 존재한다.'이다. 따라서 출력된 탈출 경로를 통해 도망자는 목표 지점까지 무사히 도달할 수 있다.

3.3 게임 풀이 분석

아래 그림은 도망자-추적자 게임의 마지막 15단계를 보인다. 초기 상태에서 게임 규칙을 적용한 게임 풀이를 통해 목표 상태에 이르는 경로를 구했다. 이 경로를 통해 도망자는 추적자를 유인해내어 벽을 이용해 무사히 목표 지점까지 탈출할 수 있었다.

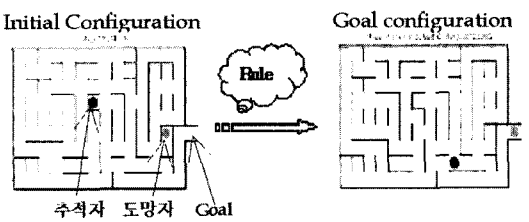

그림 2. 도망자-추적자 게임 구성

그림 3과 같이 유한 상태 모델에 게임과 시작 상태를 표현하고 목표 상태의 부정을 시제 논리식으로 표현하여 모델 체커에 입력하면 모델 체커는 부정에 대한 반례를 생성하여 초기 상태부터 목표 상태까지의 탈출 경로를 구하게 된다.

모델 체커로는 CadenceSMV(Symbolic Model Verifier)를 사용하여 입력 언어를 통해 모델링하고 15단계까지 모두 풀었다. 일반 PC(펜티엄 IV, 1.5Ghz, 메모리 384M)로 실험한 결과 상태 폭발 문제[8]는 일어나지 않았다. [9]의 푸쉬푸쉬 게임 50판의 경우 48개의 셀로 구성되고 237의 상태 공간을 사용했던 반면 도망자-추적자 게임15판의 경우 126개(149)의 셀로 구성되고 214의 상태 공간(메모리: 16,372KB,

시간: 16.694초, BDD 수: 529,694)에서 해결되었다.

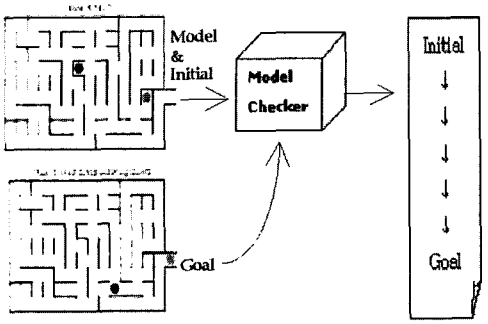

그림 3. 모델 체킹을 이용한 게임 풀이

CadenceSMV는 너비 우선 탐색으로 초기 상태부터 단계별 모든 다음 상태를 구하게 된다. 따라서 CadenceSMV를 통해 생성된 반례는 최적 해라고 할 수 있다. 그러므로 도망자는 가장 빠른 탈출 경로를 얻게 된다. 도망자-추적자 게임 마지막 15단계의 도망자 탈출 경로의 시퀀스는 다음과 같다.

〈down, left, up, up, right, right, up, up, up, up, left, left, left, down, down, down, left, up, left, up, up, left, left, left, left, left, left, down, left, up, left, down, down, up, up, right, down, right, up, right, right, right, right, right, down, down, down, right, down, down, down, right, left, up, up, up, left, down, down, left, down, left, left, up, left, left, left, left, down, right, right, right, down, down, right, right, right, left, left, left, left, up, up, left, left, up, right, up, up, right, up, right, right, right, right, left, right, right, right, right, down, right, up, up, up, right, right, right, down, down, left, left, down, down, down, down, down, up, left, up, up, up, up, up, up, right, right, left, left, down, down, down, left, up, left, left, left, left, left, left, left, down, down, down, left, left, down, up, up, up, down, down, down, right, right, down, down, up, up, left, left, up, right, right, right, right, down, right, right, up, right, down, right, right, down, right, right, right, down, right, up, up, up, right〉

도망자-추적자 게임15단계의 스텝 수는 184스텝이다. 이 경로를 통해 도망자가 한번 움직일 때마다 추적자는 2번 움직이게 된다.

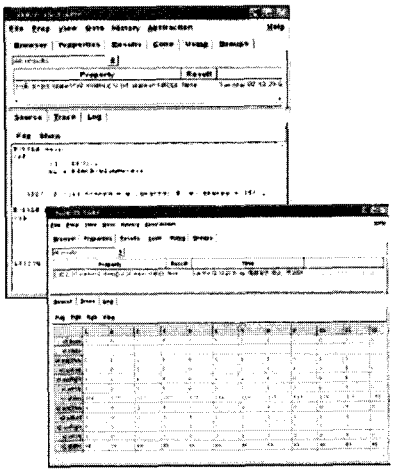

그림 4. CadenceSMV의 모델과 반례

4. 응용

게임 풀이한 도망자(플레이어)의 탈출 경로를 이용해 임베디드 시스템인 실제 에이전트를 만들어 테스팅 하였다. 레고 마인드스톰 킷 안에는 수백 개의 레고 브릭, 각종 센서, 모터 그리고 입출력 장치를 갖는 마이크로 컴퓨터인 RCX로 구성되어 있다. 이를 이용하여 두 대의 에이전트를 만들었다. 각RCX 내부에 Firmware로써 LeJos(Lego Java OS)를 선택하였고 그 위에 에이전트들이 센싱하고 동작할 수 있도록 제한된 자바 언어로 응용 프로그램을 작성하여 적재하였다 [10].

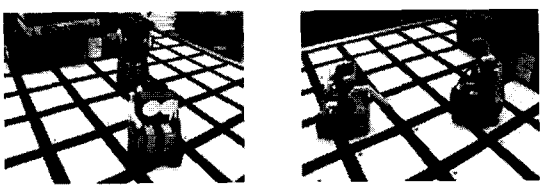

그림 5. 로봇으로 구현한 도망자-추적자 게임

그림 5와 같이 외부 환경은 흰색 보드에 검은색 테이프로 셀을 만들어 구성했고 각각의 에이전트들이 지면의 색을 구별하여 센싱, 동작하도록 하였다. 그리고 에이전트들과 컴퓨터에 ID를 부여하여 컴퓨터에 연결된 Infra-red Tower

를 통해 커넥션을 형성하고 메시지를 전송하는 통신 방법을 사용하였다. 그림 6은 레고 마인드스톰으로 구현한 도망자-추적자 게임의 실행 과정이다. 먼저 도망자와 추적자는 각각 초기 상태의 위치에 놓여지게 된다.

1) CadenceSMV를 통해 얻어진 탈출 경로를 한 스텝씩 Infra-red Tower를 통해 도망자에게 전달한다.
2) 도망자는 1step 움직이거나 delay한 후에 Tower에게 reply를 보낸다.
3) Tower는 추적자에게 도망자의 움직인 위치를 전달한다.
4) 추적자는 도망자의 위치를 계산하여 2step이하로 움직이거나 delay한 후에 Tower에게 reply를 보낸다.
5) 위 과정은 도망자가 탈출 지점에 도달 할 때까지 계속 된다.

여기서 우리는 도망자가 추적자를 만나지 않고 무사히 목표 지점까지 도달하는 것을 볼 수 있었다.

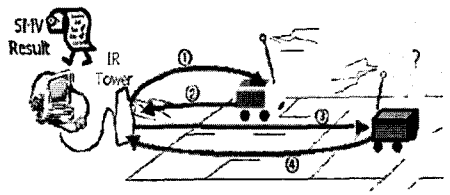

그림 6. 도망자-추적자 게임의 실행 과정

5. 결론 및 향후 연구

본 논문은 모델 체킹을 게임 풀이에 적용했고 임베디드 시스템에 응용하여 유용한 사례를 보였다. 이를 위해 도망자-추적자 게임을 선정하여 모델 체커인 CadenceSMV를 이용해 게임 풀이를 하였고 레고 마인드스톰이라는 임베디드 시스템을 이용해 응용하였다. 그 결과 CadenceSMV는 너비 우선 탐색이기 때문에 가장 짧은 탈출 경로를 얻었다. 게임을 해결하는 최단 경로가 15판의 경우 184스텝이나 되었다. 이것은 사람이 짧은 시간에 생각하기는 힘든 길이이다. 따라서 게임의 크기가 커갈 수 록 더욱 어렵다고 할 수 있다. 또한 모델 체킹은 상태공간을 탐색하는데 기호적 방법을 사용하기 때문에 선행연구[9]에서는 최신의 A*알고리즘으로도 해결할 수 없었던 소코반 51게임을 모델 체킹을 이용해서 풀 수 있었다. 본 연구는 앞으로 현실 세계의 다양한 게임을 풀고 임베디드 시스템에 적절히 응용하는데 도움이 될 것으로 보인다.

향후 연구 방향은 다양한 응용 방법을 연구하여 협력 에이전트나 멀티 에이전트간의 문제 풀이에 모델 체킹을 응용하는 한편 임베디드 소프트웨어의 코드 수준에서의 검증으로 확장하려고 한다.

참고문헌

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

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

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

[4] 권기현, "모델 검증을 이용한 게임풀이", 정보과학회학회지, 제21권 제1호, pp.7-14, 2003.

[5] K. L. McMillan, Symbolic Model Checking, Kluwer Aca-demic Publishers, 1993.

[6] W. Chan, Symbolic Model Checking for Large Software Specifications, Ph.D. thesis, University of Washington, Com-puter Science and Engineering, 1999.

[7] E. M. Clarke, E.A. Emerson, A.P. Sistla, Automatic Verification of Finite-State Concurent Systems using Temporal Logic Specifications, ACM Transactions on Programming Languages and Systems, Vol. 8, No. 2, pp.244-263, 1986.

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

[9] 이태훈, 권기현, "상태폭발 문제 해결을 위한 릴레이 모델 체킹," 한국 소프트웨어공학 학술대회 논문집, Vol. 6, No. 1, pp.298~305, 2004.

[10] G. Farrari, A. Gombos, S. Hilmer, J. Stuber,

"Programming Lego Mindstorms with Java: The Ultimate Tool for Mindstorms Maniacs," Syngress, April 2002

박 사 천

2001년 경기대학교 전자계산학과(학사)
2003년 경기대학교 전자계산학과(석사)
2004~현재 경기대학교 전자계산학과 박사과정
관심분야: 정형검증, 정보 보안, 게임 모델링

권 기 현

1985년 경기대학교 전자계산학과(학사)
1987년 중앙대학교 전자계산학과(이학 석사)
1991년 중앙대학교 전자계산학과(공학 박사)
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