미로 퍼즐 풀이를 통한 모델 체킹 도구의 특성 분석

박사천, 이건수, 권기현
경기대학교 컴퓨터과학과
{sachem, gslee, khkwon}@kgu.ac.kr

Analysis of Model Checking Tools with Maze Puzzles

Sachoun Park, Gunsoo Lee, Gihwon Kwon
Department of Computer Science, Kyonggi University

요 약

본 논문은 게임 풀이를 통해서 모델 체킹 도구의 특징을 분석한다. 도망자-추적자 게임은 격자 모양의 미로에서 도망자가 추적자를 따돌리고 탈출하는 게임이다. 도망자가 격자 상에서 한 칸 움직일 때, 추적자는 일정한 패턴을 가지고 두 칸 움직인다. 이 문제를 SMV 와 SPIN 모델 체커로 모델링하고 검증하는 과정을 통해서 SMV 와 Spin 모델 체커의 특징을 분석한다. 실험을 통해서 우리는 최단 경로를 찾을 경우는 SMV 모델 체커를 사용해야 하고, 가능한 경로를 빨리 찾는 경우는 Spin 모델 체커가 더 적합함을 확인할 수 있었다.

1. 서론

모델 체킹은 테스팅 및 시뮬레이션을 보완하는 검증 기법으로 주목 받아 지난 20 여 년간 발전에 발전을 거듭해 왔다. 모델 체킹 기술이 각광을 받던 초기에는 군사용 시스템, 원자력 시스템, 그리고 우주·항공 시스템과 같이 특정분야에 주로 적용되었었다. 그러한 미션-위험(mission-critical) 시스템이나 안전성-위험(safety-critical) 시스템의 오류는 심각한 혼란과 재정의 낭비를 초래할 수 있기 때문에 철저하게 검증되어야 할 우선 대상이 된 것이다.

그러나 현재는 오히려 최종 사용자들을 대상으로 하는 보안 시스템, 내장형 시스템, 웹 서비스 시스템 등 일반 어플리케이션 분야에서도 정형검증에 대한 요구가 증대되고 있다. 따라서 자동 검증을 기반으로 하고 있는 기존의 모델 체킹 도구들을 잘 선별해서 사용하는 것이 중요한 문제가 되었다. 물론 시스템을 정확하게 모델링하고, 검증 결과를 잘 분석해서 진단하는 과정도 매우 중요하다. 하지만, 어떤 문제에 어떤 도구를 적용할 것인가에 관한 사항도 정형 분석가에게는 반드시 필요한 능력이다.

따라서 본 논문은 모델 체킹 도구로 게임 풀이를 한 후 얻게 된 경험을 토대로 검증 목적에 따른 모델 체킹 도구 사용 방안에 대해서 기술한다. 우리가 다루는 모델 체킹 도구는 가장 폭 넓게 사용되고 있는 SMV[1]와 Spin[2]이다. 검증 대상은 도망자-추적자 게임이다. 도망자-추적자 게임은 원래 테세우스-미노스라는 이름으로 알려져 있는데 유명한 게임 디자이너인 Robert Abbott 에 의해서 1990 년에 만들어졌다. 이 게임의 규칙은 매우 단순하지만, 추적하는 미노스가 도망치는 테세우스 보다 2 배 빠르게 움직이기 때문에 추적자를 따돌리고 도망치는 일은 무척 어렵다. 그러나 추적자의 움직임은 예측가능하기 때문에 이 게임은 2-player 게임이 아니고 1-player 게임에 해당한다. 여기서 예측 가능하다라는 말은 추적자의 움직임이 도망자의 움직임에 대해 결정적이라는 뜻이다. 도망자-추적자 게임은 인터넷 웹 페이지(http://www.tnelson.demon.co.uk/mazes/)에 접속해서 직접 풀어볼 수 있다. 검증 대상을 모델 체커의 입력 언어의 특성을 잘 살려서 모델링 하는 것이 중요한데, 우리는 이 게임을 SMV 로 모델링 할 때, 변수로 선언된 두 개의 모듈을 사용 했고, Spin 으로 모델링 할 때는 하나의 프로세스로 모델링 했다.

이렇게 모델링 된 게임은 각각의 모델 체커를 통해서 크기가 6×6 인 첫 번째 문제로부터 9×14 인 마지막 열 다섯 번째 문제까지 모두 해를 구할 수 있었다. 가능한 해를 찾을 때는 Spin 이 SMV 보다 더 적은 메모리로 더 빨리 찾았지만, 그것은 최단 경로가 아니었다. 어떤 문제는 가능한 경로를 찾는 것으로 충분할 수 있겠으나 어떤 문제는 반드시 최단 경로를 찾아야 한다. 최단 경로를 찾을 때에는 반대로 SMV 가 Spin 보다 더 좋은 결과를 보임을 알 수 있었다.

본 논문의 구성은 다음과 같다. 2 장에서는 모델 체킹의 개요를 기술하고, 3 장과 4 장에서는 게임에 대한 SMV 와 Spin 에 대한 모델링 방법을 설명한다. 5 장에서는 실험 및 결과 분석을 서술하고 마지막으로 6 장에서 결론을 맺는다.

2. 모델 체킹

모델 체킹은 시스템을 철저하게 검사하는 자동화 된 정형 검증 방법이다. 모델 체킹의 입력은 모델과 속성인데, 모델 M 은 유한 상태 기계로 작성되고, 검사

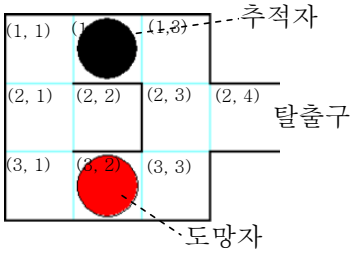
이 논문은 경기도지역협력연구센터(콘텐츠융합소프트웨어 연구센터)의 지원을 받아 수행된 연구임

하고 싶은 속성 ϕ는 시제논리로 기술된다. 모델 체커는 이렇게 기술된 모델과 속성을 가지고 모델이 속성을 만족($M \models \phi$)하는지 검사하게 된다. 이때 사용되는 시제논리가 CTL (Computation Tree Logic)[3]인지 LTL(Linear Temporal Logic)[4]인지에 따라 모델 체킹 기술은 크게 두 가지로 분류된다. CTL 을 사용하는 대표적인 도구는 SMV 인데 이 도구는 모델을 BDD (Binary Decision Diagram)[5]로 표현하고 고정점 계산으로 속성의 만족성 여부를 판정한다. 주로 하드웨어 시스템을 검증하는데 많이 적용되었고, 현재는 소프트웨어 모델 체킹을 위해서 SAT(Satisfiability) 기술을 사용하는 바운드 모델 체킹 방식이 적용되고 있다. 본 논문에서는 BDD 방식의 모델 체킹 기술을 적용했다.

LTL 을 사용하는 대표적인 도구는 Spin 이고 주로 프로토콜이나 멀티 스레드 모델과 같이 비동기적인 시스템을 검증하는데 많이 적용되고 있다. LTL 모델 체킹은 모델 M 과 속성의 부정 $\neg\phi$을 모두 오토마타로 변환하고 이 둘에 대한 곱 오토마타 $A_{M\otimes\neg\phi}$를 생성해서 인식되는 언어가 있는 검사한다. 만일 인식되는 언어가 있다면, 속성이 위배되는 경우를 발견한 것이 된다. 인식되는 언어의 존재 여부는 곱 오토마타에서 도달 가능한 사이클을 찾는 방식으로 판정한다. 특히 Spin 의 경우 오토마타에 대한 C 코드를 만들어 검사함으로써 C 컴파일러의 효율성을 이용했다. Spin 모델 체커는 또한 assert 구문을 지원해서 도달성 검사를 할 수 있도록 해준다. 게임 풀이는 도달성 분석과 같다. 따라서 Spin 으로 게임 풀이를 할 때는 assert 구문을 이용해서 도달성 분석을 적용했다.

3. SMV 게임 모델

도망자-추적자 게임은 아래 그림 1 과 같이 미로와 도망자, 추적자, 그리고 탈출 지점으로 구성된다. 아래 그림은 문제를 설명하기 위한 3×3 의 예제이다.



(그림 1) 도망자-추적자 게임

위 게임에서는 도망자가 한 칸 움직일 때마다 추적자는 두 칸을 움직일 수 있다. 그러나 추적자는 도망자와의 위치 차를 계산해서 가능한 한 먼저 좌우로 움직이고, 나중에 상하로 움직인다. 따라서 위의 그림에서 도망자가 좌측인 (3, 1)으로 움직이면 추적자는 (1, 1)을 거쳐 (2, 1)에 도달한다. 이때 도망자가 다시 우측인 (3, 2)지점으로 움직이면, 추적자는 도망자와의 위치를 계산하고 먼저 우측으로 이동한다. 그런데 (2, 2)의 아래 쪽은 벽이므로 더 이상 좇지 못하고 멈추게 된다. 이때 도망자는 (3, 3)과 (2, 3)을 지나 (2, 4) 지점으로 탈출하면 게임에서 이기는 것이다.

이를 모델 체킹 도구로 풀이하기 위해서 먼저 미로와 각 에이전트(도망자, 추적자)의 움직임을 모델로써 기술하고 “살아서 탈출구를 빠져나가는 길은 전혀 없다”라는 의미의 속성을 기술한다.

먼저 SMV 에서는 도망자와 추적자 모두 하나의 모듈로써 모델링 한다. 아래와 같이 두 개의 변수로써 각각의 에이전트들을 선언한다. 이때 `red()`는 도망자를 모델한 것이고, `black()`은 추적자를 모델한 것이다. 추적자는 도망자의 다음 위치를 받게 되는데, 이는 추적자가 도망자의 위치 정보를 받아서 자신의 위치를 결정하기 때문이다. 그리고 속성은 “도망자가 탈출 위치에 도달할 때까지 추적자가 도망자를 잡지 못하는 경우는 없다.”를 의미하는 CTL 식으로 기술한다.

```
MODULE main
VAR
  v1 : red();
  v2 : black(v1.nexts);
SPEC
  !E[!(v1.state=v2.state)U(v1.state=24)]
```

그리고 도망자의 모듈은 아래와 같이 도망자 자신의 위치를 의미하는 상태변수 `state` 와 추적자가 두 번 움직일 때 도망자는 한번만 움직이도록 하기 위해 계속해서 반전되는 부울변수 `move` 를 선언한다.

```
MODULE red()
VAR
  states : {    11,12,13,
                21,22,23,24
                31,32,33};
  move : boolean;
```

그런 후 도망자의 움직임을 모델링 한다. 도망자는 벽이 아닌 어디로든 갈 수 있어야 하기에 먼저 벽에 관한 정보를 `DEFINE` 으로 정의한 후, 움직임을 `ASSIGN` 으로 정의한다. SMV 의 `case` 구문은 다수의 조건이 동시에 참이 될 때, 가장 먼저 만족한 구문부터 실행하기 때문에, 참이 되는 서로 다른 조건들을 비결정적으로 선택할 수 없다. 따라서 앞의 코드와 같이 4 방향에 대한 모든 가능한 조건 즉, 16 가지의 경우를 나열해 주는 방식으로 모델링 한다.

```
ASSIGN
   init(move) := 1;
   next(move) := !move;

   init(state) := 32;
   next(state) := case
    move& !notDown& notRight& notUp& notLeft:{state+10,state};
    move& !notRight& notDown& notUp& notLeft:{state+1,state};
    move& !notUp& notDown& notRight& notLeft:{state-10,state};
    move& !notLeft& notRight& notUp& notDown:{state-1,state};
    move& !notLeft& !notDown& notRight& notUp : …;
    move& !notLeft & notDown & !notRight & notUp : …;
    move& !notLeft & notDown & notRight & !notUp : …;
    move& notLeft & !notDown & !notRight & notUp : …;
    move& notLeft & !notDown & notRight & !notUp : …;
    move & notLeft & notDown & !notRight & !notUp : …;
    move & !notLeft & !notDown & !notUp & notRight : …;
    move & notLeft & !notDown & !notUp & !notRight : …;
    move & !notLeft & notDown & !notUp & !notRight : …;
    move & !notLeft & !notDown & notUp & !notRight : …;
    move & !notLeft & !notDown & !notUp & !notRight : …;
    1 : state;
   esac;
```

```
DEFINE
  nexts := next(state);
  notLeft :=  state=11| state=21| state=31| state=23;
  notRight := state=13| state=22| state=33;
  notUp := state=11|state=12|state=13|state=22|state=23;
  notDown:= state=31|state=32|state=33|state=12|state=22;
```

SMV 에서 비결정성은 조건이 만족할 때 다음 값을 비 결정적으로 취할 수 있게 하는 것이지 조건 자체에 대한 비 결정성을 제공하지 않는다. 다음에 살펴보겠지만 Spin 에서는 다수의 조건이 참이 되면 그 중에 비결정적으로 선택하는 구문을 제공해 주고 있어서 도망자의 움직임에 대한 표현이 간단하다.

도망자에 관한 SMV 코드를 모두 살펴 보았다. 다섯 번째부터 `case` 구문의 값 배정이 생략되었는데 다섯 번째 경우는 왼쪽으로도 갈 수 있고, 아래로도 갈 수 있기에 `{(state+10),(state-1), state}` 로 기술하면 되고 나머지 부분도 같은 방식으로 기술하면 된다. 여기서 `state` 는 제자리에 멈춤을 의미한다. 추적자에 대한 모델은 도망자의 다음 위치를 받아서 자신의 다음 위치를 계산하도록 하는 코드로 구성되는데 도망자와 유사한 부분이 많기 때문에 지면상 생략 한다.

아래 그림은 이제까지 기술했던 SMV 모델을 Cadence SMV 로 검증한 결과이다. 도망자의 경로는 앞서 설명했던 것과 같고 `v1.state` 의 값 변화로 확인할 수 있다.

	1	2	3	4	5	6	7	8	9	10
v1.move	1	0	1	0	1	0	1	0	1	0
v1.nexts										
v1.notDown	1	1	1	1	1	1	1	0	0	0
v1.notLeft	0	1	1	0	0	0	0	1	1	0
v1.notRight	0	0	0	0	0	1	1	0	0	0
v1.notUp	0	0	0	0	0	0	0	1	1	0
v1.state	32	31	31	32	32	33	33	23	23	24
v2.notDown	1	0	0	1	1	1	1	1	1	1
v2.notLeft	0	1	1	0	0	0	0	0	0	0
v2.notRight	0	0	0	1	1	1	1	1	1	1
v2.notUp	1	1	0	1	1	1	1	1	1	1
v2.state	12	11	21	22	22	22	22	22	22	22

(그림 2) 그림 1 의 예제에 대한 SMV 실행 결과

4. Spin 게임 모델

Spin 모델 체커는 Promela 를 입력 언어로 사용하는데, 이것은 CSP(Communicating Sequential Processes)[6]에 많은 영향을 받았다. Promela 코드는 C 코드로 변환되어 검증되기 때문에 C 코드와도 매우 흡사하다. 데이터 타입 또한 bool, bit, byte, int 등 일반적인 프로그램 언어 비슷하고, 사용자 데이터 구조 및 프로시져 등을 선언해서 사용할 수 있다.

도망자-추적자 게임을 Spin 으로 모델링 하는 것은 SMV 의 경우와는 많은 차이가 있다. 일단 SMV 에서는 도망자와 추적자의 위치를 각각의 상태 변수로 선언했는데, 그것은 SMV 에서 필연적인 방식이다. 대신 Spin 에서는 두 에이전트의 위치를 두 개의 byte 형 변수로 선언하고, 위, 아래, 왼쪽, 오른쪽 등의 방향은 4 개의 bool 변수로 선언한다. 또한 Spin 은 참인 모든 조건을 비 결정적으로 선택하기 때문에 에이전트의 움직임을 4 가지 경우로 표현할 수 있다. 그리고 추적자의 움직임은 도망자에 의존적이기 때문에 독립된 프로세스로 선언하기 보다는 하나의 인라인 프로시져로 표현해서 도망자 프로세스에서 호출하는 형태로 모델링 한다. 벽에 대한 정보 또한 도망자 프로세스나 추적자 프로시져에서 호출하는 또 다른 인라인 프로시져로 선언해서 코드의 길이를 줄일 수 있다. Spin 코드는 검증되기 위해서 C 코드로 변환되어야 한다. 변환 된 pan.c 코드를 컴파일하면 pan.exe 파일이 생성된다. 이때 컴파일 옵션으로 –DREACH 를 사용하고 pan 옵션으로 –i 를 사용하면 에러로 가는 최단 경로를 구할 수 있다. 아래는 앞의 그림 1 에 대한 Spin 코드이다.

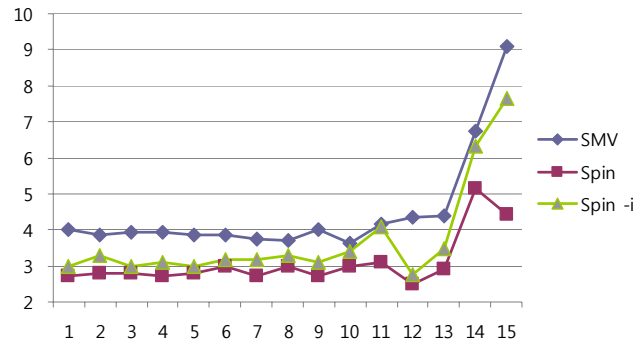
```
byte red = 32, black = 12;
bool right, left, up, down;

inline setDirection(obj){
    right = 1; left = 1; up = 1; down = 1;
    if
    :: (obj==11|obj==21|obj==31|obj==23)->left=0
    fi;
    if
    :: (obj==13|obj==22|obj==33)->right=0
    fi;
    if
    :: (obj==11|obj==12|obj==13|obj==22|obj==23)
        ->up=0
    fi;
    if
    :: (obj==31|obj==32|obj==33|obj==12|obj==22)
        ->down=0
    fi;
}

inline blackUpDownStop(){
    if
    :: (red/10<black/10)&&up -> black=black-10;
    :: (red/10>black/10)&&down -> black=black+10;
    fi;
}

inline Black(){
    red!=black;
    setDirection(black);
    if
    ::((red%10)<(black%10))-> if
          :: left -> black = black - 1;
          :: !left -> blackUpDownStop();
          fi;
    ::((red%10)>(black%10))-> if
          :: right -> black = black + 1;
          :: !right -> blackUpDownStop();
          fi;
    :: else -> blackUpDownStop();
    fi;
}

active proctype Red() {
    do
    :: red == 24 -> break;
    :: setDirection(red);
       If
       ::printf("stay::%d\n",red);
       ::left->red=red-1; printf("left::%d\n",red);
       ::right->red=red+1; pintf("right::%d\n",red);
       ::up->red=red-10; printf("left::%d\n",red);
       ::down->red=red+10; printf("left::%d\n",red);
       fi;
       Black();Black();
    od;
    assert(false)
}
```

`assert(false)`는 그 위치까지 도달할 수 없다는 것을 의미한다. 그 위치에 도달하려면 `red` 의 값이 24 즉 탈출 위치에 있어야 하고 그때까지 지켜져야 하는 가드조건은 `Black()`에서 선언한 `red!=black` 가 된다. 위의 Promela 코드를 Spin 으로 검증하면 기 같은 trail 파일을 생성하는데 이것은 SMV 로 검증한 결과와 동일하다.

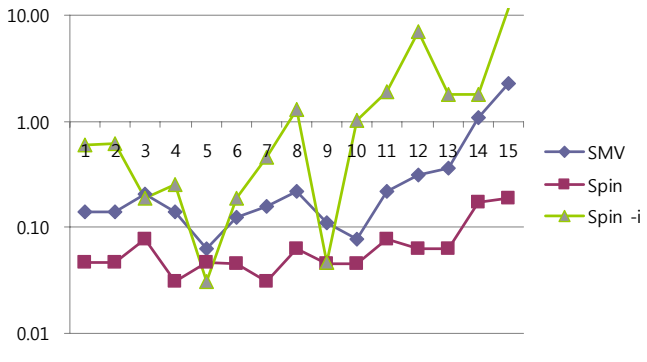
```
left::31->right::32->right::33->left::23
->right::24
```

5. 실험 및 분석

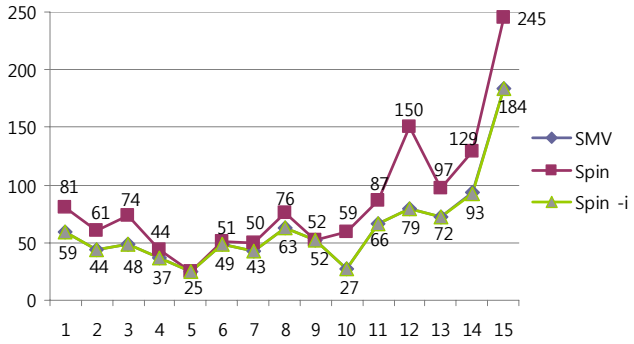
실험은 웹에 공개된 도망자-추적자 문제 15 개를 가지고 1.66GHz 듀얼코어 CPU 와 1GB 메모리에서 동작하는 윈도우 XP 상에서 수행했다. 먼저 SMV 로 모든 문제를 모델링하고 풀어서 문제의 최단경로임을 확인했고, 그 다음 Spin 모델 체커로 문제 풀이를 했다. Spin 의 경우 깊이 우선 탐색을 하기 때문에 주어진 최대 깊이 범위 내에서 신속하게 에러를 검출해 준다. On-the-fly 방식을 취하고 있어서 메모리와 검증 시간에서 SMV 보다 앞섰다. 그러나 깊이 우선 탐색이기 때문에 도망자가 최소의 움직임으로 탈출하는 최단 경로를 구할 수는 없다. Spin 에서 최단 경로를 구하려고 한다면 앞서 언급한 바와 같이 최단 경로를 구하는 옵션을 컴파일 시간과 실행 시간에 각각 추가해야 한다. 아래의 그림 3 과 4 는 각각 실험하면서 사용된 메모리와 검증 시간을 나타낸 것이다.



(그림 3) 사용된 메모리(MB)



(그림 4) 검증 시간(초)



(그림 5) 도망자의 움직임 횟수

그림 3 을 보면 메모리의 사용량은 SMV 가 가장 많고, 최단 경로 옵션을 사용한 Spin 이 그 다음이며, 아무 옵션을 사용하지 않았을 때의 Spin 이 가장 작은 메모리를 사용했다. 검증 시간 역시 최단 경로 옵션을 사용하지 않았을 때의 Spin 이 가장 빨랐지만, 최단 경로를 구하는 옵션을 주었을 때는 SMV 보다 대체적으로 검증 시간이 오래 걸렸다. 그림 5 는 도망자가 탈출하기 위해 움직여야 할 횟수를 나타낸다. Spin 에서 –i 옵션을 사용하면 SMV 와 동일한 결과를 얻을 수 있다.

6. 결론 및 향후 연구

시스템에 대한 정형 검증의 요구가 증대되고 있는 반면 모델 체킹 도구 사용에 대한 가이드는 미비한 상황이다. 본 논문에서는 널리 사용되고 있는 SMV 와 Spin 모델 체커에 게임 풀이를 적용해서 각 도구의 메모리 사용량과 처리 시간 및 반례의 길이 등을 비교·분석함으로써 찾고자 하는 반례의 특성에 따라 어떤 도구를 사용하는 것이 좋은지 가이드를 제시했다. 또한 모델링 함에 있어서 각각의 도구가 지향하는 바른 모델의 필요성을 기술하였다. 향후 보다 다양한 관점에서 모델 체커들의 특징을 분석하는 연구들이 수행되어야 할 것이다.

참고문헌

[1] K. L. McMillan, Symbolic Model Checking, PhD thesis, Carnegie Mellon University, 1993.

[2] G. Holzmann, The SPIN Model Checker, Addison-Wesley, 2003.

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

[4] M. Vardi and P. Wolper, “An automata-theoretic approach program verification,” In the Proceedings of the 1st IEEE Symposium on Logic in Computer Science, pp. 332-344, IEEE, 1986.

[5] R. E. Bryant, Graph-based algorithms for Boolean function manipulation, IEEE Transactions on Computers, C-35(8), pp. 677-691, 1986

[6] C. A. R, Hoare, Communicating Sequential Processes, Prentice Hall, 1985.