Promela 모델에서 C 코드를 끼워서 추상화하기

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

Embedding C Code into Promela Model and its Abstraction

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

요 약

SPIN은 소프트웨어 정확성 검사에 널리 사용되는 모델 검증 도구이다. 특히 C 코드로 작성된 소프트웨어를 효율적으로 검사하기 위해서 SPIN의 입력 언어인 Promela 모델에 C 코드를 끼워 넣는 기능이 버전 4.0 이상에서 지원되고 있다. 본 논문에서는 이러한 기능을 미로 게임 풀이에 적용하였다. 그 결과, Promela 모델만을 사용해서 풀이한 것보다도 모델에 C 코드를 끼워 풀이한 것이 메모리 사용 및 처리 시간에서 월등히 우수했다. 메모리와 시간과 같은 객관적인 성능 향상과 더불어서, 이러한 사례 연구는 모델 검증 도구 및 추상화 학습에도 유용함을 경험했다.

1. 서 론

모델 체킹은 시스템의 버그를 철저하게 검증하는 정형 검증 방법이다[1,2]. 처음에는 소프트웨어 모델을 검증하는 연구가 많았지만 최근에는 프로그램 자체를 검증하는 연구가 활발하다. 전통적인 소프트웨어 모델 체킹 방식은 소프트웨어에 대한 모델을 구축하고 구축된 모델 위에서 속성을 검사한다. 그러나 모델을 구축하는 과정은 매우 어렵고 주관적이어서 오류가 개입될 수 있는 결정적인 한계를 갖고 있다. 따라서 현재 소프트웨어 모델 체킹 연구의 주요 관심은 소프트웨어의 모델을 자동으로 추출하는 것이다.

마이크로소프트사의 SLAM 프로젝트로부터 시작된 연구들은 술어 추상화를 통해서 소스코드에서 이진 프로그램을 자동으로 추출한다. 이렇게 추상화된 모델인 이진 프로그램은 모델 체킹의 입력이 된다. 여기서는 존재 추상화 기법을 사용하기 때문에 추출된 모델에서 버그가 없다면 그대로 믿을 수 있지만, 버그가 존재하면 실제 버그인지 다시 조사해야 한다. 만일 실제의 버그가 아니라면 정제 과정을 거쳐 새로운 모델이 만들어진다. 이렇게 추상-정제를 반복하면서 소스코드의 버그를 검증하게 된다[3,4].

또 다른 연구의 흐름은 소스코드를 직접 논리식으로 변환하는 연구이다[5]. 이 방법은 미리 정의한 범위 내에서 모델 체킹하기 때문에 바운드 모델 체킹이라고 하는데, 검증 결과 버그가 존재한다면 실제의 버그이지만 만일 버그가 없다고 하면 그 범위 밖에서도 버그가 없다고 보증할 수 없다. 따라서 이 방법은 버그의 부재를 증명하는 것이 매우 어렵다.

모델 체킹은 연구 초기부터 서로 다른 두 그룹에서 각각 진행되었다. 앞의 두 방법은 내부 표현으로 BDD(Binary Decision Diagram)나 CNF(Conjunctive Normal Form)를 이용하는 기호적 모델 체킹 방법[6]을 사용한다. 이와는 다르게 그래프를 탐색해서 명시적으로 모델 체킹하는 오토마타 기반의 모델 체킹 연구가 있다[7]. 여기서 대표적인 도구가 SPIN 모델 체커이다. SPIN은 비결정적이며 비 동기적인 시스템을 모델링하고 검증하기에 적합한데 Promela 라는 모델링 언어를 사용한다. Promela 코드는 C 코드로 변환되어 검증되기 때문에 C 코드와도 매우 흡사하다. 데이터 타입 또한 bool, bit, byte, int 등으로 일반적인 프로그램 언어와 비슷하고, 사용자 데이터 구조 및 프로세스, 프로시져 등을 선언해서 사용할 수 있다. 그렇기에 프로그램 모델 체킹 도구로써 많이 사용되고 있다[8].

SPIN 버전 4.0 이상부터는 입력언어인 Promela 에 직접 C 코드를 삽입할 수 있는 기능이 지원된다. 이 기능은 프로그램 코드를 검증할 때, 해당 코드에 대한 모델을 만드는 것이 아니라 직접 그 코드를 수행하도록 하는 방식을 취한다. 따라서 보다 정확한 모델을 만들도록 돕는다[9]. 본 논문에서는 SPIN에 C 코드 끼워 넣기 기능을 사용해서 미로 게임을 풀이했다. 미로 게임은 도망자가 여러 명의 추적자들을 따돌리고 목적지까지 안전하게 도달하는지를 묻는 도달성의 문제이다. 실험을 통해서 Promela 모델만을 사용해서 풀이한 것보다도 모델에 C 코드를 끼워 풀이한 것이 메모리 사용 및 처리 시간에서 월등히 우수함을 확인할 수 있었다. 또한 끼워 넣은 C 코드를 활용해서 상태공간을 추상화 함으로써 보다 큰 문제를 빠르게

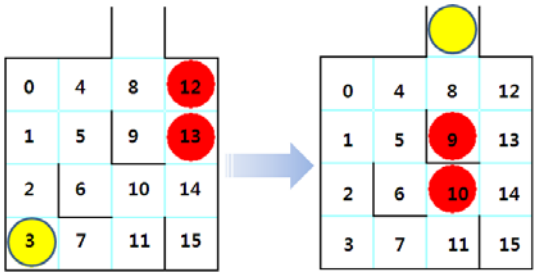
* 본 연구는 경기도의 경기도지역협력연구센터사업의 일환으로 수행하였음.[2007-081-8, 메타 데이터 기반의 디지털 콘텐츠 관리도구 개발]

해결할 수 있었다. 이러한 사례 연구를 통해서 우리는 SPIN에서 직접 C 코드를 삽입하는 모델의 특성을 분석할 수 있었고, C 소스코드에서 추상화 기법을 적용함으로써 소프트웨어 모델 체킹에서 추상화에 대한 이해를 높일 수 있었다. 논문에서 우리는 두 가지 추상화 방법을 정의하고 실험했는데, 이를 통해 안전한 추상화와 극단적인 추상화의 특성을 비교할 수 있었다.

본 논문의 구성은 다음과 같다. 2장에서는 다루려는 문제인 미로 게임과 함께 그에 대한 일반적인 Promela 모델을 설명하고, 3장에서는 SPIN에 C 코드를 끼워 넣는 방법 미로 게임으로 설명한 후 추상화 방법에 대해서 기술한다. 4장에서는 실험 결과와 함께 극단적인 추상화와 안전한 추상화 방법을 비교하고 5장에서 결론 맺는다.

2. 미로 게임에 대한 Promela 모델

미로 게임은 아래 그림 1 과 같이 미로와 도망자, 추적자들, 그리고 탈출 지점으로 구성된다. 아래 그림은 문제를 설명하기 위한 4×4 의 예제이다. 이 예에서 도망자는 초기에 그림의 왼쪽과 같이 3 의 위치에 있고, 추적자들은 각각 12 와 13 의 위치에 있다. 최종적으로는 추적자들을 따돌리고 그림의 오른쪽과 같이 탈출해야 한다.



(그림 1) 4×4 미로 게임

위 게임에서 도망자가 한 칸 움직일 때 추적자도 한 칸을 움직일 수 있다. 추적자는 도망자와의 위치 차를 계산해서 가능한 한 먼저 좌우로 움직이고, 좌우로 움직일 수 없을 때 상하로 움직인다. 따라서 위의 그림에서 도망자가 위쪽인 2 로 움직이면 추적자들은 각각 8 과 9 에 도달한다. 추적자들끼리는 겹쳐질 수 없다. 이때 도망자는 자신은 움직이지 않으면서 추적자가 움직일 수 있도록 제자리에 “멈춤”을 선택할 수 있다. 그러면 추적자들의 위치는 4 와 9 가 된다. 이렇게 계속해서 도망자는 추적자들에게 잡히지 않는 범위 내에서 움직이면서 탈출하게 된다. 이 게임에서 도망자의 움직임을 미로의 숫자로 나타내면 ⟨3, 1, 2, 3, 3, 7, 11, 10, 6, 5, 4, 5, 6, 10, 14, 13, 9, 8⟩가 된다.

그림 1의 예를 모델링 해보자. 미로 게임을 Spin에서 모델링 하기 위해 우선 미로에 대한 정보와 도망자와 추적자의 위치를 두 개의 byte 형 변수로 선언한다.

```
#define N 4
#define M 2
#define GOAL 8
byte red = 3;
byte black[2];
byte i, turn;
bool right, left, up, down;
byte p[4];
```

위에서 N은 미로의 크기를 나타내는 숫자로 정의된다. M은 추적자의 수이다. GOAL은 목적지의 위치를 나타내며, red는 도망자의 위치로 초기화된다. i, turn은 각 도망자들이 겹쳐지지 않게 하기 위한 변수들이고, 위, 아래, 왼쪽, 오른쪽 등의 방향은 4개의 bool 변수로 선언한다. 또한 Spin은 참인 모든 조건을 비결정적으로 선택하기 때문에 도망자의 움직임을 아래와 같이 5가지 경우로 표현할 수 있다. 여기에는 제자리에 멈추는 동작도 포함된다.

```
active proctype Red() {
    initialization();
    do
    :: red == GOAL -> break;
    ::     setDirection(red);
           if
           :: skip;
           :: left -> red -= N;
           :: right -> red += N;
           :: down -> red++;
           :: up -> red--;     ;
           fi;
           play();
    od;
    assert(false)
}
```

그리고 추적자의 움직임은 도망자에 의존적이기 때문에 독립된 프로세스로 선언하기 보다는 하나의 인라인 프로시져로 표현해서 도망자 프로세스에서 호출하는 형태로 모델링 한다.

```
inline initialization(){
    black[0] = 12; black[1] = 13;
    p[0] = 111; p[1]= 91;
    p[2] = 55;   p[3]= 42;
}

inline setDirection(obj){
    right = 1; left = 1; up = 1; down = 1;
    if::((p[obj/4]>>2*(obj&3))&2)==0->down=0; fi;
    if::((p[obj/4]>>2*(obj&3))&1)==0->right=0; fi;
    if:: obj % N == 0 -> up = 0;
       :: else -> if
          :: ((p[(obj-1)/4]>>2*((obj-1)&3))&2)==0->up=0;
    fi; fi;
    if:: obj < N   -> left = 0;
       :: else -> if
          :: ((p[(obj-N)/4]>>2*((obj-N)&3))&1)==0->left=0;
    fi; fi;
}

inline movable(b, k, d){
    i = 0;
    do
       :: i<M -> if
             :: i != turn && b+k == black[i] -> d=0;
             fi;
             i++;
             :: else -> break;
    od;
}
```

```
inline blackUpDownStop(b){
    if
    :: (red%N < b%N) && up -> b--;
    :: (red%N > b%N) && down -> b++;
    fi;
}

inline Black(b){
    setDirection(b);
    if :: up -> movable(b, -1, up); fi;
    if :: down -> movable(b, 1, down); fi;
    if :: left -> movable(b, -N, left); fi;
    if :: right -> movable(b, N, right); fi;
    if :: ((red/N)<(b/N)) ->
        if
        :: left -> b = b - N;
        :: !left -> blackUpDownStop(b);
        fi;
        :: ((red/N)>(b/N)) ->if
            :: right -> b = b + N;
            :: !right -> blackUpDownStop(b);
            fi;
        :: else -> blackUpDownStop(b);
        fi;
        (red != b);
}

inline play(){
    turn=0; Black(black[0]); turn=1; Black(black[1]);
}
```

Initialization()은 벽 정보와 추적자들의 위치를 초기화한다. setDirection()은 해당 셀에서 이동 가능한 위치를 알려준다. movable()은 추적자들이 겹치지 않고 이동할 수 있는지를 계산한다. play()는 추적자들을 움직이는 함수이다. 여기서 호출되는 Black()은 먼저 좌우로 움직이게 하고 좌우로 움직일 수 없을 때, 아래 위로 움직이기 위해서 blackUpDownStop()를 호출한다.

Spin 코드는 검증되기 위해서 C 코드로 변환되어야 한다. 변환 된 pan.c 코드를 컴파일하면 pan.exe 파일이 생성된다. 이때 컴파일 옵션으로 -DREACH를 사용하고 pan 옵션으로 -i를 사용하면 에러로 가는 최단 경로를 구할 수 있다.

3. C 코드를 끼워 넣은 Promela 모델

SPIN 4.0부터 Promela에 C 코드를 직접 사용할 수 있도록 하는데, 이렇게 Promela에 끼워진 C코드는 SPIN에서 따로 구문 검사하지는 않는다. 다음은 C 코드를 사용을 위한 SPIN의 예약어 이다.

- c_expr : 임베디드 C로 표현된 조건문 정의
  ```
  c_expr { /* c code */ }
  c_expr '[' /* c expr */ ']' { /* c code */ }
  ```

- c_decl : 임베디드 C형태의 데이터 타입을 정의
  ```
  c_decl { /* c declaration */ }
  ```

- c_track : Promela코드 밖에서 선언이 되며 현재의 상태에서 해당 객체의 정보를 기록하며 2개 또는 3개의 따옴표 스트링을 이용한다. 첫 번째 부분은 데이터의 위치를 나타내고 두 번째 부분은 데이터의 크기를 나타낸다. 마지막은 Matched 와 UnMatched 둘 중에 하나를 선택하게 되는데 Matched를 선택할 경우에는 해당 변수의 값이 스텍에 저장되며 상태 기술자(state descriptor)안에 저장이 된다. 하지만 UnMatched일 경우에는 스텍에만 저장이 되며 상태 기술자안에는 저장되지 않는다.
  ```
  c_track  "location of value" "sizeof(type)"
                              " Matched / UnMatched"
  ```

- c_code : Promela 코드안에서 임베디드 C 사용을 제공해주는 예약어
  ```
  c_code { /* c code */ }
  c_code '['/* c expr */ ']' {/* c code */ ; }
  ```

- c_state : 2개 또는 3개의 따옴표 스트링을 이용하여 데이터를 선언한다. 첫 번째 부분은 데이터의 타입과 이름을 나타내며 두 번째 부분은 데이터의 범위를 나타낸다. 이 범위는 "Global", "Local", "Hidden" 으로 정의된다. 세 번째 부분은 데이터 객체에 대한 초기치를 나타낸다.
  ```
  c_state "type and name" "scope" "initial value"
  ```

이와 같은 예약어로 앞의 4×4 미로 게임에 대한 Promela 모델을 나타내면 아래와 같다. 여기서는 Promela 코드 부분과 C 코드 부분으로 나뉘는데, 비결정적인 요소를 가지고 있는 도망자는 Promela 언어로 정의하고, 도망자에 위치에 따라 자신의 위치가 결정되는 추적자들은 C 코드에서 계산하도록 구성했다. 아래에서 c_track 다음에 “Matched” 혹은 “UnMatch”가 생략되면 기본적으로 “Matched”가 된다. 또한 Promela의 전역 변수는 C 코드 내에서 now 예약어 뒤에 첨부되어 사용된다.

```
#define GOAL 8
#define N 4
c_decl {
        extern short setDirection(short);
        extern short play(short);
        extern short black[];
};

byte x;   byte red = 3;

c_track "&black[0]" "sizeof(short)"
c_track "&black[1]" "sizeof(short)"

active proctype Red() {
    do
    ::red == GOAL -> break;
    ::c_code {now.x = setDirection(now.red);};
        if
        :: skip;
        :: ((x/2)*2 != x) -> red -= N;
        :: (((x/2)/2)*2 != (x/2)) -> red+=N;
        :: (x>7) -> red++;
        :: (((x/4)/2)*2 != (x/4)) -> red--;
        fi;
        c_code{now.x = play(now.red);};
        x==0;
    od;
    assert(false)
}
```

```
#include <stdio.h>
#define N 4
#define M 2

short black[2] = {12, 13};
short p[3] = {111, 91, 55, 42};
short turn, right, left, up, down;

short setDirection(short obj)
{
  right = 1; left = 1; up = 1; down = 1;
  if (((p[obj/4] >> 2 * (obj&3)) & 2) == 0) down=0;
  if (((p[obj/4] >> 2 * (obj&3)) & 1) == 0) right=0;
  if (obj % N == 0) up = 0;
  else if(((p[(obj-1)/4]>>2*((obj-1)&3))&2)==0) up=0;
  if (obj < N) left = 0;
  else if(((p[(obj-N)/4]>>2*((obj-N)&3))&1)==0) left=0;
  return left*1 + right*2 + up*4 + down*8;
}

void movable(short k, short* d){
  short i;
  for(i=0;i<M;i++)
    if(i!=turn && black[turn]+k == black[i]) *d=0;
}

void blackUpDownStop(short r, short* b){
  if ((r%N < *b%N) && up) *b -= 1;
  else if ((r%N > *b%N) && down) *b += 1;
}

short Black(short r, short* b)
{
  setDirection(*b);
  if(left) movable(-N, &left);
  if(right) movable(N, &right);
  if(up) movable(-1, &up);
  if(down) movable(1, &down);

  if((r/N) < (*b/N))
    if(left) *b -= N;
    else  blackUpDownStop(r, b);
  else
    if ((r/N) > (*b/N)){
        if(right) *b += N; else blackUpDownStop(r, b);
        else blackUpDownStop(r, b);

  if (r != *b) return 1; else return 0;
}
void sort(short* i){
  int j, k, temp;
  for (j=0; j<M; j++)
       for (k=j+1; k<M; k++)
           if(i[j] > i[k]){
                temp = i[j]; i[j] = i[k]; i[k] = temp;
           }
}

short play(short red){
  short i;
  for(i=0;i<M;i++){
      turn=i; if(!Black(red, &black[i])) return i+1;
  }
  return 0;
}
```

위의 코드는 Promela에서 호출되는 setDirection() 함수와 play()함수에 대한 C 코드이다. 각각의 함수들은 Promela로 원래 정의된 인라인 함수에 각각 대응된다. 실험 결과에서 언급하겠지만, 이렇게 Promela 코드 중 일부를 C 코드로 교체하는 것 만으로도 성능에 향상을 가져올 수 있다.

이제 추상화 방법을 다뤄본다. 추상화를 위해서는 앞의 예에서 다음과 같은 문장들로 변경·추가하면 된다. 여기서는 도망자 변수들에 대해서는 “UnMatched”로 선언하고 새로 정의한 abstract 변수는 “Matched”로 선언했다. 즉, 도망자의 위치는 상태 벡터 정보로서 저장되지만, 상태 공간 탐색을 위한 정보로는 활용되지 못한다는 것을 의미한다. 따라서 상태 공간은 오직 abstract 변수와 Promela에서 정의된 도망자(Red) 변수에 의해서만 탐색된다.

```
c_decl {
          extern short setDirection(short);
          extern short play(short);
          extern short black[];
          extern short abstract;
};

c_track "&black[0]" "sizeof(short)" "UnMatched";
c_track "&black[1]" "sizeof(short)" "UnMatched";
c_track "&abstract" "sizeof(short)";
```

이제 추상화된 C 코드에 대해서 살펴본다. 추상화된 C 코드는 원래의 C 코드에서 다음과 같은 부분만 추가된다.

```
void sort(short* i){
  int j, k, temp;
  for (j=0; j<M; j++)
       for (k=j+1; k<M; k++)
           if(i[j] > i[k]){
                temp = i[j]; i[j] = i[k]; i[k] = temp;
           }
}

void ABS(){
  short i[2];
  i[0] = black[0]; i[1] = black[1];

  sort(i);
  abstract = i[0];
  abstract = i[1]*N*N;
}

short play(short red){
  short i;
  for(i=0;i<M;i++){
      turn=i; if(!Black(red, &black[i])) return i+1;
  }
  ABS(); return 0;
}
```

추상화는 실제로 ABS()라는 함수를 통해서 이루어지는데, 이 함수가 하는 일은 abstract 변수가 새로운 값을 갖도록 하는 일이다. 추상화는 미로에 놓인 추적자들을 서로 구별하지 않음으로써 가능하다. 만일 추상화 하지 않는다면 이론적인 상태 공간의 크기는 $O(n^{2M+2})$이다. 여기서 n과 M은 모델에서 정의된 미로의 크기와 도망자의 개수이다. 이것을 그림 1의 4×4 미로 게임에 적용하면, $4^{2\times2+2}$ 즉, 4,096이다. 그런데 추상화를 하게 되면, 각 추적자들의 순서가 무시 된다. 예를 들어 그림 1에 대한 원래 모델에서는 추적자0과 추적자1이 각각 12, 13에 위치하는 것과 13, 12에 위치하는 것이 서로 다른 상태를 갖게 되지만, 추상화된 모델에서는 동일한 상태로 표현된다. 즉 원래 모델에서 도망자들은 시퀀스로 표현되지만 추상화된 모델에서는 집합으로 표현된다. 따라서 4×4 미로의 경우 추상화된 모델의 상태공간이 원래 모델보다 2배 작다. 이를 일반화 하면 $O(n^{2M+2}/M!)$ 이 된다. 따라서 미로가 커지고 추적자들이

많아질수록 추상화의 효과는 더욱 커진다.

4. 실험 및 분석

실험을 위해서 우리는 웹에 공개된 미로 문제를 바탕으로 3×3부터 12×12까지 10개의 새로운 문제를 만들었다. 실험은 1.66GHz 듀얼코어 CPU와 1GB 메모리에서 동작하는 윈도우 XP 상에서 수행했다. 원래 Promela 모델, C를 끼워 넣은 모델, 추상화를 적용한 모델 순으로 실험했다. 그림 2는 메모리 사용량을 나타낸 것이다. 원래의 Promela 모델은 7×7까지만 해결할 수 있었다. 그 이후에는 1G 이상의 메모리를 사용하고도 문제를 해결할 수 없었다. C 코드를 끼워 넣은 모델은 훨씬 좋은 결과를 보여준다. 그러나 마지막 문제는 해결할 수 없었다. 역시 1G 이상의 메모리를 사용하고도 문제를 해결할 수 없었다. 그에 비해서 추상화를 적용한 모델은 모든 문제를 비교적 짧은 시간에 해결할 수 있었다. 그래프의 세로축이 로그 형태임을 감안하면 문제의 크기가 커질수록 추상화한 모델이 거의 10배 작은 메모리를 사용하고 문제를 해결함을 확인할 수 있다.

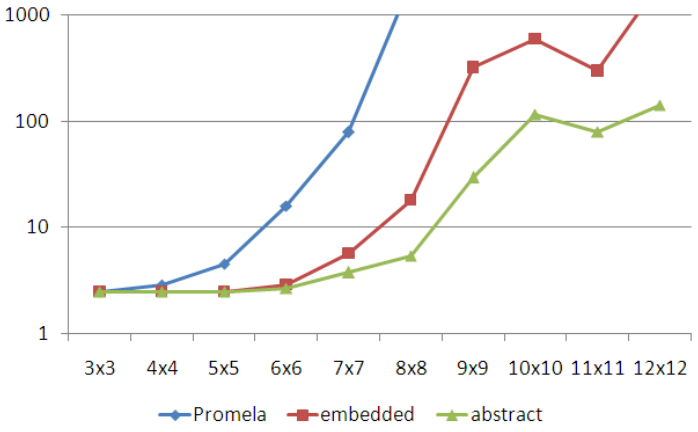


그림 2. 메모리 사용량(Mb)

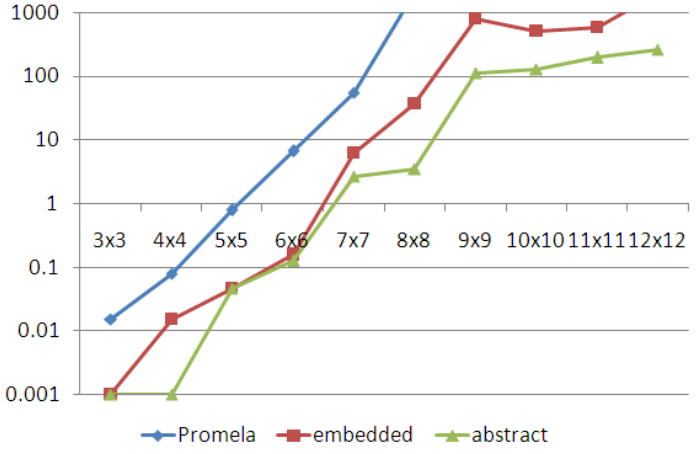


그림 3. 처리 시간(초)

그림 3은 처리시간에 대한 그래프인데 1000초를 timeout 시간으로 설정했다. 그러나 그래프 상에서 해결되지 않은 문제들은 실제로 30분 이상의 시간을 쓰고도 해결할 수 없었다. 이로써 우리는 추상화의 이점을 확인할 수 있었다. 왜냐하면 앞서 언급했던 바와 같이 모델이 커지고 추적자의 수가 많아질수록 M! 배의 상태공간이 줄어들기 때문이다.

이러한 추상화 방법은 미로의 모든 형상을 나타내기 때문에 안전한(Sound) 추상화 방법이라고 할 수 있다. 즉 추적자의 위치만 관심 있을 뿐 개별적인 추적자들의 순서 정보는 무시해도 되기 때문이다. 그런데 문제의 크기가 커지면 이러한 추상화 방법도 해결할 수 없는 문제를 만나게 된다. 모델 체킹을 수행하면서 상황에 따라 안전성을 잃더라도 문제를 해결하는 것이 더 중요할 때가 있다. 그때에는 극단적인 추상화의 방법이 적용될 수 있다. 예를 들어 이 문제에서 도망자와 추적자의 거리로써 상태공간을 구별한다면 상태공간은 *도망자의 위치*×$(2n-2)^M$ 즉 $O(n^{M+2})$의 크기를 갖게 된다. 여기서 도망자의 위치가 n^2이 되기 때문이다. 그러나 이러한 방법은 하나로 볼 수 없는 상태들을 하나로 묶기 때문에 너무 많은 상태 정보를 잃게 된다. 그리고 여기에 앞서 적용했던 추상화 방법을 추가적으로 적용하면 $O(n^{2M+2}/M!)$의 상태 공간에서 검사할 수 있게 된다. 다음은 극단의 추상화를 적용한 C 코드이다.

```
int distance(int red, int x){
        return (abs(red%N - x%N)+abs(red/N - x/N));
}

void ABS(int red){
        short i[2];
        i[0] = black[0];
        i[1] = black[1];

        i[0] = distance(red, i[0]);
        i[1] = distance(red, i[1]);

        sort(i);

        abstract = i[1];
        abstract += i[0]*(2*N-2);
        abstract *= (red+1);
}

short play(short red){
        short i;
        for(i=0;i<M;i++){
                turn=i;
                if(!Black(red, &black[i])) return i+1;
        }
        ABS(red);
        return 0;
}
```

즉 도망자와 각 추적자들의 거리를 계산하고 그들을 구별하지 않는 방식으로 추상화 되었다. 이러한 추상화와 첫번째 추상화의 결과를 비교하면, 그림 4, 그림 5와 같다. 그림의 그래프를 보고 알 수 있듯이 안전한 추상화 방법에 비해서 극단적인 추상화 방법이 훨씬 좋은 성능을 보임을 알 수 있다. 그러나 여기서 반드시 밝혀야할 사실은 7×7을 제외한 비교적 작은 크기의 모델 즉 3×3부터 8×8 까지의 모델에서는 탈출 경로를 찾을 수 없었다. 그리고 그림 6에서 보이는

바와 같이 찾아진 경로가 최단경로는 아니었다. 어떤 문제는 최단 중요하지 않을 수도 있지만 대부분의 문제에서 최단 경로는 매우 중요하다. 따라서 극단적인 추상화 방법은 적은 자원을 사용하지만, 바람직하지 못한 결과를 가져올 수도 있음을 알 수 있다. 따라서 모델 체킹을 할 때에는 선택적으로 추상화 수준을 변경할 수 있어야 한다.

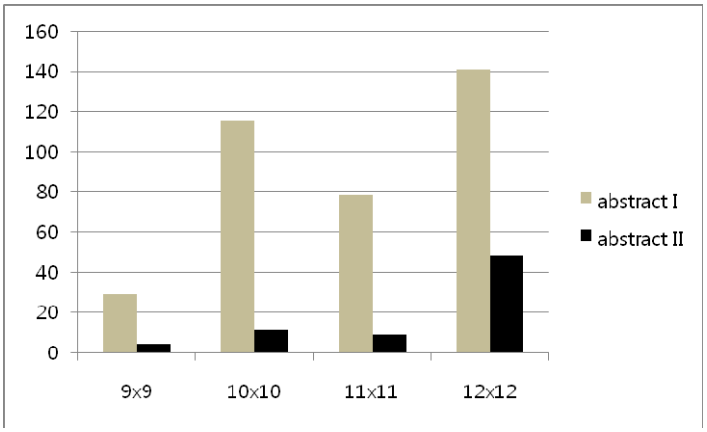


그림 4. 두 추상화 모델의 메모리 사용량 비교(Mb)

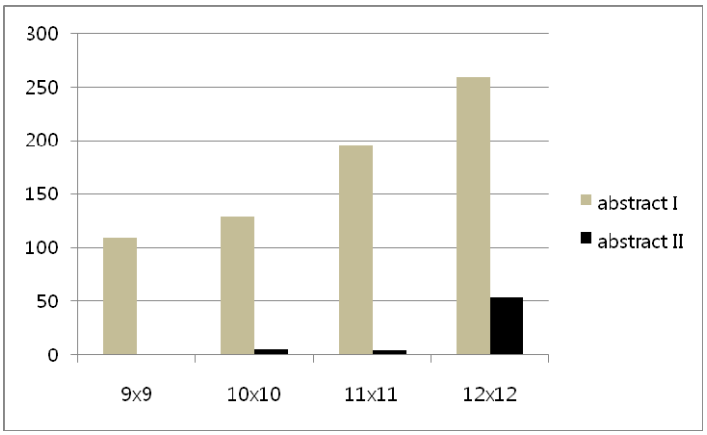


그림 5. 두 추상화 모델의 처리 시간 비교(초)

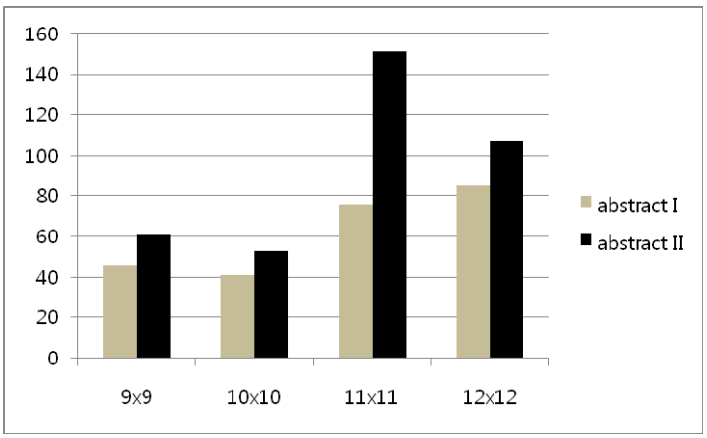


그림 6. 두 추상화 모델에서 찾아진 탈출 경로 비교

5. 결론 및 향후 연구

모델 체킹은 시스템의 모든 경로를 탐색해서 시스템 내에 버그가 존재하는지 검사하는 정형 검증 기법이다. 전통적인 소프트웨어 모델 체킹은 주로 상태 전이도와 같은 소프트웨어의 행위 모델을 다루었다. 최근에는 프로그램 소스코드를 직접 다루려는 연구들이 활발하다. 그러한 연구의 일환으로 널리 사용되는 SPIN 모델 체커에서는 4.0 버전부터 C 소스코드를 모델 내에 끼워넣고 쓸 수 있는 구문들을 제공하고 있다.

본 논문에서는 미로 게임을 통해서 Promela 모델에 C 소스코드를 끼워 넣는 방식에 대해서 설명했다. 또한 추상화를 통해서 훨씬 적은 메모리 및 CPU 자원을 가지고 모델 체킹이 가능함을 보였고, 또한 극단적인 추상화 방법과 안전한 추상화 방법을 소개함으로써 모델 체킹시 이 두 방식의 추상화를 함께 사용하는 것이 유용함을 설명했다.

향후 연구로는 본 연구의 경험을 토대로 실제 소스코드의 검증을 수행하는 것과 소스 코드 수행시의 추상화 기법을 고안하고 기존의 추상화 기법과 비교하는 연구가 남았다.

참고문헌

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

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

[3] Thomas Ball, Rupak Majumdar, Todd Millstein, Sriram K. Rajamani, "Automatic Predicate Abstraction of C Programs", In the Procedings of PLDI , SIGPLAN Notices 36(5), pp. 203-213, 2001.

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

[5] D. Kroening, K. Yorav, and E. Clarke, "Behavioral Consistency of C and Verilog Programs Using Bounded Model Checking," In the Proceedings of DAC 2003, 368-371, 2003.

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

[7] 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.

[8] G. Holzmann, "Logic Verification of ANSI-C code with SPIN," In the Proceedins of SPIN 2000, 2000.

[9] G. Holzmann and R. Joshi, "Model-Driven Software Verification," In the Proceedins of SPIN 2004, pp. 76-91, 2004.