임베디드 자바 프로그램의 정형 검증

이 태 훈[†] · 권 기 현[††]

요 약

임베디드 소프트웨어는 크기나 기능면에서 복잡하기 때문에 에러가 숨어있을 확률이 높다. 그래서 에러를 효과적으로 찾아낼 수 있는 방법이 절실히 요구된다. 임베디드 소프트웨어에서 잠재적인 오류를 찾아내는 기술중의 하나가 모델 체킹이며, 대표적인 연구로서 SLAM 이 있다. SLAM은 임베디드 C 프로그램의 안전성 속성을 검증하는 소프트웨어 모델 체킹 도구이다. 본 논문에서는 임베디드 자바 프로그램을 검증하는 모델 체킹 도구를 개발했다. SLAM과는 달리 자바 프로그램의 안전성뿐만 아니라 궁극성 속성을 모두 검증할 수 있다. 개발된 모델 체킹 도구를 이용하여 레고 로봇을 제어하는 임베디드 자바 프로그램의 속성을 검증하였다.

키워드 : 정형 검증, 모델 체킹, 추상화, 자바 프로그램

Formal Verification of Embedded Java Program

Taehoon Lee[†] · Gihwon Kwon[††]

ABSTRACT

There may be subtle errors in embedded software since its functionality is very complex. Thus formal verification for detecting them is very needed. Model checking is one of formal verification techniques, and SLAM is a well-known software model checking tool for verifying safety properties of embedded C program. In this paper, we develop a software model checker like SLAM for verifying embedded Java program. Compared to SLAM, our tool allows to verify liveness properties as well as safety ones. As a result, we verify some desired properties in embedded Java program for controlling REGO robot.

Key Words : Formal Verification, Model Checking, Abstraction, Java Program

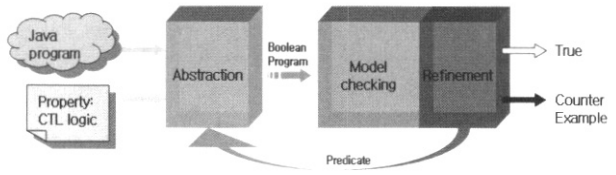
1. 서 론

모델 체킹은 정형명세 언어를 사용하여 시스템을 유한상태모델로 표현하고 그 시스템이 만족해야하는 안전성 속성 등을 논리식으로 표현하여 시스템이 속성을 만족하는 지를 자동으로 검증해주는 기법이다[1]. 지금까지 모델 체킹은 모델이 커짐에 따라 검사해야 할 상태공간의 크기가 지수적으로 증가하는 상태폭발문제 때문에 작은 규모의 상태공간을 갖는 하드웨어나 프로토콜 검증에 주로 사용되어져 왔다[2]. 그러나 최근에는 상태폭발문제를 다루기 위한 다양한 연구와 술어추상화[3] 등과 같은 추상화 기술의 발전으로, 다룰 수 있는 상태공간의 크기가 증가하고 정리증명기의 성능이 향상됨에 따라 소프트웨어 자체의 논리적 오류를 검증하려는 움직임이 활발하다. 기존의 모델 체킹은 사용자의 실수로 모델이 올바로 작성되지 않았다면 속성 검증 시 잘못된 결과를 가져올 수 있다. 또한 비전문가는 정형명세 언어로 모델을 표현하기가 쉽지 않지만 소프트웨어 자체를 검증하게 되면 별도로 정형명세를 할 필요 없다. 또한 잘못된 코드 작성은 소프트웨어의 논리적 오류 발생 가능성이 매우 높고 이것은 시스템의 신뢰도에 직접적인 영향을 미치게 된다. 따라서 소프트웨어 자체를 검증하면 개발자 입장에서 더 많은 논리적 오류들을 원천적으로 막을 수 있고 높은 신뢰도의 시스템을 구현할 수 있게 된다.

소프트웨어는 무한한 상태공간을 갖는다. 예를 들어, 정수형 변수 하나만 하더라도 무수히 많은 상태를 가질 수 있고 이런 정수형 변수들의 조합은 더욱 무한한 상태공간을 발생시킨다. 또한 자바와 같은 고수준 언어의 다양한 문법은 모델 체킹을 더욱 어렵게 한다. 따라서 소프트웨어를 모델 체킹하기 위해서는 추상화가 필수적이고 매우 중요하다. 또한 추상화는 자동으로 수행되어야한다. 소프트웨어를 추상모델로 변환 시 사용자가 개입하면 코드와 추상모델간의 추적성이 떨어지고 많은 부분 성능 감소를 가져오기 때문이다. 여러 추상화 기법들 중 술어 추상화는 소프트웨어 모델 체킹 분야에서 프로그램을 자동으로 추상화시키는데 많이 활용되

※ 본 연구는 과학기술부 기초과학 연구(R01-2005-000-11120-0) 지원으로 수행되었음.
† 준 회 원 : 경기대학교 전자계산학과 박사과정
†† 종신회원 : 경기대학교 정보과학부 교수
논문접수 : 2005년 9월 22일, 심사완료 : 2005년 10월 20일

어지고 있다. 대표적인 연구로는 마이크로소프트사의 C 검증도구인 SLAM[4], 버클리 대학의 C검증도구인 BLAST[5], 카네기 멜론 대학의 C검증도구인 MAGIC[6], Java 검증 도구인 Bandera[7]등이 있다. 특히 SLAM은 실제 윈도우의 디바이스 드라이버의 오류를 검증한 가장 실용적인 도구로 평가된다. 하지만 SLAM은 검사할 수 있는 속성이 단순한 assert 위반에 대해서만 검사를 수행할 수 있다. 다른 부분의 소프트웨어 모델 체킹 도구 역시 간단한 속성에 대해서만 검사를 수행할 수 있다. 따라서 좀 더 복잡한 행위에 대한 검사를 수행하기는 힘들다. 본 논문에서는 자바 프로그램을 받아들여서 이를 추상화 하고 모델 체킹을 수행하는 도구를 개발하였다. 이를 이용한 검증 과정은 (그림 1)과 같이 일반적으로 사용되는 추상화 - 모델 체킹 - 추상화개선으로 이루어진다. 변환된 이진 프로그램은 CTL식(Computation Tree Logic, [8])으로 표현된 속성과 함께 모델 체커의 입력이 된다. 모델 체킹을 수행한 후 속성을 만족하면 true, 만족하지 않으면 반례를 생성하고 반례가 실제 코드에서 도달가능 한 경로이면 false, 그렇지 않으면 가짜 반례로 판단한다. 추상화된 이진프로그램은 상위 근사화 된 모델이기 때문에 원래 프로그램에는 없는 행위를 가질 수 있다. 이런 경우 술어를 추가하여 이진프로그램을 개선한 후 다시 프로세스를 반복 수행한다.



(그림 1) 자바 모델 체커

모델 체커에서 검증해야 할 속성에는 안전성(Safety)과 궁극성(Liveness)이 있다. 안전성은 시스템 실행 도중에 나쁜 상태에 도달할 수 없음을 의미한다. 안전성의 대표적인 예는 데드락 부재이다. 궁극성은 시스템 도중 언젠가는 좋은 상태에 도달할 수 있음을 의미한다. 궁극성의 대표적인 예는 자원 사용 허가이다. 시스템이 정확하다는 것을 보장하기 위해서는 두 가지 속성이 만족됨을 보여야 한다. SLAM과 BLAST등과 같은 소프트웨어 모델 체킹 도구에서는 검증할 수 있는 속성은 안전성에 국한되어 있다. 또한 BANDERA와 같은 모델 체킹 도구에서는 안전성과 궁극성 모두를 검사할 수 있지만, 기존 모델 체킹 도구를 이용하기 때문에 소프트웨어 모델 체킹을 수행하기는 비효율적이다. 따라서 본 논문에서는 간단한 자바 프로그램을 대상으로 추상화를 수행한 후 생성된 이진 프로그램에서 안전성과 궁극성을 모두 검증할 수 있는 방법을 제안한다.

제안한 방법에 대한 사례 연구로서 레고에 들어가는 소프트웨어에 대한 검증을 수행하였다. 레고에서 개발한 마인드스톰을 이용해서 주어진 대로 행동을 하는 로봇을 개발하였고, 내부에서 동작하는 프로그램을 자바를 이용해서 개발하였다. 마인드 스톰에는 8Bit CPU 와 32 Kb 의 메모리를 가지고 있고 C 혹은 Java 와 같은 언어를 이용해서 센서에서 감지된 정보를 통해 동작을 수행할 수 있다. 레고에서 사용된 자바 소프트웨어의 특징은 일반적인 자바 프로그램의 부분집합으로 구성된다. JDK1.2 를 기준으로 할당문과 조건문 메소드 호출과 객체 생성과 같은 부분은 동일하게 적용된다. 하지만 자바에서 자용되는 라이브러리는 제공되지 않고 메소드의 재귀호출은 제공하지 않고 멀티 스레드를 지원하지 않는다. 또한 배열 혹은 가비지 콜렉션과 같은 기능 역시 지원 하지 않는다. 레고 마인드 스톰을 이용하여 로봇을 개발하는데 사용된 자바 프로그램을 추상화 모듈을 이용해서 이진 프로그램으로 변환한 후 여기서 제시한 방법을 이용하여 모델 체킹을 수행했다. 이를 통해서 자바 프로그램의 안전성 속성과 긍극성 속성에 대해서 검사를 수행했다.

본 논문의 구성은 다음과 같다. 2장에서는 기존 소프트웨어 모델 체킹 도구를 비교하고 3장에서 추상화된 이진 프로그램을 대상으로 모델 체킹을 수행하는 방법을 설명한다. 4장에서는, 사례연구로서 자바언어로 작성된 로봇 소프트웨어에 대해서 안전성과 궁극성을 검증한다. 그리고 마지막으로 5장에서 결론을 맺는다.

2. 기존 소프트웨어 모델 체킹 도구

기존 소프트웨어 모델 체킹 도구의 특징을 살펴보고 장단점을 살펴본다. <표 1>에서는 기존 소프트웨어 모델 체킹 도구의 특성을 보여준다.

〈표 1〉 소프트웨어 모델 체킹 도구 특징 비교

도구 이름	추상화 방법	검사할수 있는 속성
SLAM	술어 추상화	Assert 위반
BLAST	Lazy 추상화	Assert 위반
Bandera	자료 추상화	CTL, LTL
JavaPathFinder	PartialOrderReduction	Assert 위반

2.1 SLAM

SLAM은 마이크로소프트에서 개발 중인 소프트웨어 모델 체킹 도구이다. SLAM에서는 C로 작성된 Device Driver의 행위를 검증할 수 있다. SLAM의 검사 과정은 3단계로 나뉘어져 있다. 추상화 - 모델 체킹 - 추상화 개선이다. 추상화 방법은 술어 추상화를 이용해서 이진 프로그램을 생성한다. 이 이진 프로그램은 원래 C 프로그램의 행위를 포함하는 더 많은 행위를 가지고 있다. 모델 체킹은 이진 프로그램 상의 도달성 분석을 수행하여 오류가 발생하는 위치에 도달 가능한지를 검사한다. 검사 결과 오류에 도달 불가능하다면 원래 프로그램에서도 오류에 도달 불가능하다. 하지만 오류에 도달 가능하다면 실제 프로그램에서도 도달 가능한지를 검사해야 한다. 검사 결과 오류에 도달 가능하다면 프로그램의 오류를 찾은 것이고 도달 불가능한 오류라면 새

로운 술어를 추가하여 추상화를 개선한다. 마이크로 소프트에서는 실제 동작중인 Device Driver 의 오류를 찾아내었다. 하지만 검사할 수 있는 속성의 종류는 ASSERT 위반으로 제한되어 있다.

2.2 BLAST

SLAM의 술어추상화는 한 번의 추상화 과정에서 전체 프로그램을 대상으로 추상화를 수행하게 된다. 따라서 더 이상 추상화를 수행하지 않아도 되는 부분도 추상화를 수행하기 때문에 불필요한 계산을 하게 된다. BLAST에서는 LAZY 추상화를 이용해서 추상화 개선이 필요한 부분만을 계산하는 방법을 제안했다. BLAST 에서는 C 언어를 검사할 수 있다. 하지만 검사할 수 있는 오류의 종류는 간단한 ASSERT 위반으로 제한되어있다.

2.3 Bandera

Bandera는 Java 프로그램을 위한 모델 체킹 환경이다. Bandera는 Java 프로그램을 받아들여서 이를 SMV, SPIN 등과 같은 기존 모델 체킹 도구로 변환 한다. 하지만 Java 의 여러 속성을 올바르게 검증하지 못하고 기존 모델 체킹 도구들은 소프트웨어를 검증하기엔 비효율적이다. 기본적으로 자료 추상화를 지원하고 있으며 SPIN 모델 체킹 도구로 변환을 수행한다. 또한 CTL 속성을 검사하기 위해 SMV 모델 체킹 도구도 지원을 하고 있다. 하지만 SMV 와 SPIN 과 같은 모델 체킹 도구가 소프트웨어 모델 체킹을 위한 모델 체킹 도구가 아니기 때문에 큰 프로그램에 대한 검증 사례는 존재하지 않는다.

2.4 JavaPathFinder

JPF는 NASA에서 개발 중인 모델 체킹 도구이다. 기존 도구들은 소스 코드 상에서 적용을 할수 있지만 JPF 의 경우 자바 소스코드를 컴파일한 바이트 코드 상에서 모델 체킹을 수행한다. 자체적인 가상 머신을 구현하여 바이트 코드를 실행하여 오류가 있는지 없는지를 검사해준다. 실제 동작하는 바이트 코드를 검사하기 때문에 좀 더 믿을 수 있는 결과 이지만 검사할 수 있는 속성은 데드락과 같은 오류와 ASSERT 위반으로 제한된다.

3. 이진 프로그램을 위한 CTL 모델 체킹

술어추상화의 결과로서 생성되는 이진 프로그램은 변수의 타입이 이진(boolean)형식만 사용이 가능한 프로그램이다. 본 연구에서는 간단한 자바 프로그램을 대상으로 한다. 간단한 자바 프로그램의 특성은 JDK1.2를 기본으로 하고 있고 재귀호출을 처리하지 않고 멀티스레드를 지원하지 않고 라이브러리의 사용을 제외한 것이다. 술어 추상화과정에서 for, while 같은 구문은 if ,goto 구문으로 변환된다. 간단한 자바 프로그램을 술어 추상화한 결과인 이진 프로그램은 다음과 같은 문법을 가지게 된다.

```
BP:= class*
Class:=var* method*
Var:= predicate STRING
Method:= STRING(VAR *) {  Statement *}
Statement := skip;
          | goto STRING;
          | STRING( expr+);
          | if(expr) then statement* else statement*
          | STRING = expr
          | assume( expr+ );
expr : =   | STRING( expr+);
          | STRING = expr
          | expr op expr
          | ( expr )
          | ! Expr
Op := "&" | "|"
```

이진 프로그램에서 가능한 구문은 총 6가지 구문이 가능하다. skip은 이진 프로그램의 제어흐름에 영향을 미치지 않을 경우를 의미한다. goto 구문은 주어진 위치로 이동하게 된다. 그리고 메소드 호출과 if 구문을 이용한 조건문이 존재한다. 그리고 각각의 변수의 값을 변경해 줄 수 있는 할당문이 존재하고, if 구문을 처리하기 위해서 이진 프로그램에서는 assume 구문이 존재 한다. assume 구문의 의미는 프로그램이 assume 구문의 조건을 만족한다고 가정한다는 의미이다. 만일 만족하지 않을 경우 assume 구문에서 더 이상 진행해 가지 못한다. 이진 프로그램에서 모델 체킹을 수행하기 위해서 제어흐름 그래프로 변환해야 한다. 제어 흐름 그래프는

$$CFG=(S,I,R)$$

로 구성되어 있다. 여기서 S는 상태들의 집합이고, 하나의 상태 $s=PC\times 2^{\gamma}$이다. $I\in S$는 초기 상태이고 $R\subseteq S\times S$ 은 현재 상태에서 다음 상태로의 전이를 나타내는 함수이다. 여기서 PC 는 프로그램의 위치를 나타내고 γ는 변수에 대한 변수의 값을 매칠 시키는 함수다.

이진 프로그램을 CFG 로 변환하기 위해서 이진 프로그램의 구문과 변수를 표현하는 함수를 정의한다. γ는 변수에 대한 변수의 값을 매핑 시키는 함수다. 간단한 예는 다음과 같다.

$$V=\{x,y,v\},\ \gamma=\{(x,true),(y,false),(z,true)\}$$

각각의 구문에 따른 상태 전이를 정의 하기 위해 필요한 함수들을 정의 한다. $post(pc)$는 현재 위치의 다음 위치를 되돌려 준다. $post_{T}(pc)$는 조건문에서 then 구문의 위치를 되돌려주고 $post_{F}(pc)$는 조건문의 else 구문의 위치를 되돌려준다.

Σ는 해당되는 프로그램 위치의 assume 구문의 변수의 값을 매핑 시켜준다. Δ_{cm}은 호출하는 부분과 호출되는 부분간의 변수 이름을 매핑 시켜준다. $First_{M}(pc)$는 현재 위치에서 호출하는 메소드의 위치를 되돌려준다. $Return_{M}(cp)$는 M 을 호출한 구문의 위치를 되돌려준다.

CFG의 초기 상태는 이진 프로그램의 클래스에 속해있는 메소드 중 main이라는 이름의 메소드의 맨 처음 위치에서 변수의 값은 모두 true로 설정된 상태이다.

상태의 전이는 각각의 구문에 따라서 상태 전이가 추가된다. 표로 정리하면 다음과 같다.

〈표 2〉 각각의 구문에 따른 상태 전이

구문	상태 전이
x=true	$((pc,\gamma),(post(pc),\gamma[x/true])$
IF	$((pc,\gamma),(post_T(pc),\gamma))$ $((pc,\gamma),(post_F(pc),\gamma))$
Assume	$((pc,\gamma),(post(pc),\gamma))\ where \forall v \in V.\Sigma(pc)(v)=\gamma(v)$ $((pc,\gamma),(pc,\gamma))\ where \exists v \in V.\Sigma(pc)(v)\neq\gamma(v)$
메소드 호출	$((pc,\gamma),(First_m(pc),\gamma))$
return	$((pc,\gamma),(Return_m(pc),\gamma))$
skip	$((pc,\gamma),(post(pc),\gamma))$

변수 값의 변경이 있을 경우 프로그램의 현재 위치는 다음으로 이동하게 되고 변수 값은 해당되는 값으로 변경한다. 조건문의 경우 현재 조건문이 만족할 경우와 만족하지 않을 경우로 나누어서 처리한다. Assume 의 경우 조건이 만족될 경우와 만족 하지 못할 경우로 나누어서 검사를 수행한다. 메소드 호출은 파라메터가 존재하지 않을 경우 프로그램의 위치만 변경하고 파라메터가 존재할 경우 Δ_{cm}을 이용해서 파라메터의 이름을 매핑 시켜준다. 또한 리턴 되는 값이 있을 경우 Δ_{cm}을 이용해서 원래의 변수 값으로 변경한다.

예를 들어 다음과 같은 간단한 이진 프로그램이 존재한다고 하면

```
class a {
1.   public void main( ) {

        Predicate C1: m <= 10;
        Predicate C2: x == 0;
        Predicate C3: r == true;

2.      C1 = true;
3.      C2 = true;
4.      C3 = false;

5.      L1:
6.      if( * ) {
7.              assume( !(C3));
8.              if( * ) {
9.                      assume( C1);
10.                     C1 =!C1;
                }else {
11.                     assume( !(C1));
12.                     C2 = false;
13.                     C3 = true;
                }
14.             goto L1;
        }
     }
}
```

주어진 이진 프로그램에서 다음과 같은 프로그램 전이 관계를 생성한다.

(1, (true,c2,c3)) , (2,(true,c2,c3))
(1, (false,c2,c3)) , (2,(true,c2,c3))
(2,(c1,true,c3)) , (3,(c1,true,c3))
(2,(c1,false,c3)) , (3,(c1,true,c3))
(3,(c1,c2,true)) , (4,(c1,c2,false))
(3,(c1,c2,false)) , (4,(c1,c2,false))
(4,(c1,c2,c3)) , (5,(c1,c2,c3))
(5,(c1,c2,c3)) , (6,(c1,c2,c3))
(6,(c1,c2,c3)) , (7,(c1,c2,c3))
(6,(c1,c2,c3)) , (15,(c1,c2,c3))
(7,(c1,c2,c3)) , (8,(c1,c2,c3))
(8, (c1,c2,c3)) , (9,(c1,c2,c3))
(8, (c1,c2,c3)) , (11,(c1,c2,c3))
(9, (false,c2,c3)) , (9,(false,c2,c3))
(9, (true,c2,c3)) , (10,(true,c2,c3))
(10, (true,c2,c3)) , (14,(false,c2,c3))
(10, (false,c2,c3)) , (14,(true,c2,c3))
(11, (false,c2,c3)) , (12, (false,c2,c3))
(11, (true,c2,c3)) , (11, (true,c2,c3))
(12, (c1, true,c3)) , (13, (c1, false,c3))
(12, (c1, false,c3)) , (13, (c1, false,c3))
(13, (c1, c2, true)) , (14, (c1,c2,true))
(13, (c1, c2, false)) , (14, (c1,c2,true))
(14, (c1,c2,c3)) , (6,(c1,c2,c3))
(15, (c1,c2,c3)) , (16,(c1,c2,c3))

예제 프로그램에서 각각의 프로그램 구문은 프로그램 전이와 대응된다. 프로그램 전이는 프로그램의 위치를 정의하는 PC와 현재 프로그램 구문에서 변수의 값의 변경으로 구성된다. c1=true와 같은 구문의 경우, 현재 PC에서 c1 변수가 true 혹은 false중에 어떤 값이 있더라도 현재 값에 머물러 있게 된다. 조건문을 만났을 경우에는 변수 값의 이동 없이 then 부분 혹은 else 부분으로 프로그램 위치가 이동하게 된다. assume 구문의 경우 현재 assume 구문의 값이 참이 되는 경우에만 다음 구문으로 이동하고 거짓일 경우 현재에서 계속 남아 있다. 예제 프로그램에서의 초기 상태는 프로그램 위치가 1 이면서 모든 변수의 값이 참인 상태이다.

CTL 모델 체킹을 수행하기위해 $pre_{\exists}$ 연산자를 이 장에서는 이진 프로그램에 알맞게 재정의 한다.

$$pre_{\exists}(i,j)=\{(x,y)|(x,i)\in PC\times PC\wedge(y,j)\in 2^{\gamma}\times 2^{\gamma}\wedge((x,y),(i,j))\in R\}$$

재 정의된 $pre_{\exists}$을 이용해서 이진 프로그램을 위한 CTL 모델 체킹 알고리즘을 아래와 같이 정의한다.

$$[\![(pc,\gamma)]\!]=\{(pc,\gamma')\mid \gamma'=\gamma\wedge pc\in PC\}$$
$$[\![\perp]\!]=\emptyset$$
$$[\![\neg(pc,\gamma)]\!]=\{(pc',\gamma')|pc'\in PC\setminus pc\wedge\gamma'\in 2^{\gamma}\}\cup\{(pc,\gamma')|\forall a\in V.\gamma'[a]=\neg\gamma[a]\}$$

$$\| EX(pc,\gamma) \| = pre_{\exists}(pc,\gamma)$$

$$\| EF(pc,\gamma) \| = \mu Z.(\| (pc,\gamma) \| \cup pre_{\exists}(Z))$$

$$\| EG(pc,\gamma) \| = \nu Z.(\| (pc',\gamma') \| \cap pre_{\exists}(Z))$$

$$\| AG(pc,\gamma) \| = \| \neg EF \neg (pc,\gamma) \|$$

$$\| AF(pc,\gamma) \| = \| \neg EG \neg (pc,\gamma) \|$$

$$\| AX(pc,\gamma) \| = \| \neg EX \neg (pc,\gamma) \|$$

$$\| E((pc_1,\gamma_1)U(pc_2,\gamma_2)) \| = \mu Z.(\| (pc_2,\gamma_2) \| \cup (\| (pc_1,\gamma_1) \|) \cap pre_{\exists}(Z))$$

모든 상태는 (pc,γ)의 형태로 표현된다. pc 는 현재 프로그램의 실행 위치를 나타내고 γ 는 pc에서 가질 수 있는 변수 값들의 집합을 나타낸다. 만일 현재 프로그램 위치와 변수의 값이 들어오게 되면 해당 되는 상태를 되돌려 준다. 사용자가 입력한 변수의 값이 일부분에 대해서만 설정이 되어있다면 나머지 변수들의 값은 참일 수도 있고 거짓일수도 있다. 현재 주어진 상태에 대한 부정은 두 부분으로 나누어진다. 주어진 프로그램 위치를 제외한 모든 프로그램위치에서 모든 변수의 값이 가능하거나, 현재 프로그램 위치에 변수의 값이 반대이다. EF 와 EG에서 사용된 μ, ν는 각각 최소고정점(least fixed point)과 최대 고정점(greatest fixed point)을 나타낸다. 위의 식을 이용해서 주어진 CTL 식을 만족하는 상태들의 집합을 계산한다. 주어진 CTL 식 ϕ에 대해서 초기상태 $I \in \| \phi \|$ 이라면 주어진 이진 프로그램에서 속성은 만족한다. 만일에 $I \notin \| \phi \|$ 이라면 주어진 이진프로그램에서 속성은 만족하지 않는다.

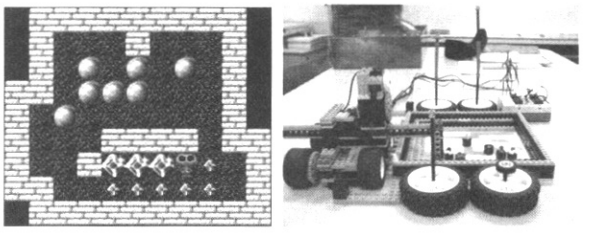
본 연구에서는 술어추상화를 수행한 이진 프로그램 상에서 프로그램을 모델 체킹을 수행한다. 술어추상화는 원래 프로그램보다 좀 더 많은 행위를 가지고 있는 이진 프로그램을 생성한다. 이진 프로그램과 원래 자바 프로그램간의 관계는 상위 근사화 관계에 있다. 술어 추상화를 수행한 모델이 더 많은 행위를 가지고 있기 때문에 모든 CTL 식을 기준으로 검사를 수행할 수 없다. ACTL 식에 대해서만 검사가 가능하다. ACTL 식은 CTL 식에 항상 A 경로 한정자가 있는 CTL 식을 의미한다.

4. 사례 연구

위의 술어추상화 기법을 통해 얻은 이진 프로그램과 와 속성을 기술한 CTL식을 자체 모델체커에 입력해서 특정 자바프로그램의 속성을 검증한다. 이진 프로그램은 모델 체커 내에서 제어흐름 그래프로 변환된다. 이 변환된 그래프의 상태공간을 탐색하며 주어진 속성이 만족하는 지를 검사한다. 검증해야 할 속성에는 크게 안전성(Safety)과 궁극성(Liveness)이 있다. 안전성은 시스템 실행 도중, 나쁜 상태에 도달할 수 없음을 의미한다. 안전성의 대표적인 예는 데드락 부재이다. 궁극성은 시스템 도중 언젠가는 좋은 상태에 도달할 수 있음을 의미한다. 궁극성의 대표적인 예는 자원 사용 허가이다. 시스템이 정확하다는 것을 보장하기 위해서는 두 가지 속성이 만족됨을 보여야 한다. SLAM에서는 검증할 수 있는 속성이 안전성에 국한되어 있지만 우리는 CTL식을 통해 궁극성도 검증한다.

4.1 문제

먼저, 검증 대상으로 아래 (그림 3)의 핸드폰에 탑재되는 푸시푸시 게임 50판을 선정하였다.



(그림 3) 푸시푸시 50판과 이를 구현한 크레인 시스템

푸시푸시 게임은 플레이어가 보드위에 흩어져있는 공들을 목적지에 모두 옮겨놓으면 클리어된다. 이 게임을 교육용 로봇제작 툴킷인 LegoMindstorm을 사용해 실제 구현하였다[9]. 모터와 센서, 레고 블럭을 이용해 크레인을 만들고 RCX라는 임베디드 시스템을 연결하였다. 이 크레인은 Arm을 이용해 공을 상, 하, 좌, 우로 움직이며 터치센서를 통해 Arm이 움직이는 거리를 측정하고 다음 움직임을 결정한다. 또한 RCX 내부에는 LeJos로 만들어진 자바프로그램을 탑재하였다. LeJos는 레고로봇을 위한 자바언어로 JDK1.1 문법의 일부분만을 지원하기 때문에 검증이 수월했다. 그리고 이 게임의 해답인 경로는 모델체커 SMV에서 반례생성을 통해 얻어내었다. 따라서 우리는 크레인이 경로를 따라 Arm을 움직여서 모든 공을 목적지로 옮기는 자바프로그램을 작성하였다.

4.2 속성

작성한 자바프로그램에서 검증하려는 안전성 속성과 궁극성 속성을 아래와 같이 CTL식으로 기술한다.

1) AG !(*game* == "Clear")

이 안전성 속성의 의미는 "모든 가능한 경우에 *game*이 Clear되지 않는다"이다. 다시 말해서, 크레인이 모든 공을 목적위치에 옮겨놓고 게임이 클리어 되는 경우는 존재하지 않는다는 것이다. 검증을 위해 프로그램 내, 게임이 클리어 되어야 하는 지점에 *game* = "Clear"; 를 삽입하고 술어 추상화하여 이 구문이 참이 되는지 분석한다. 위 식이 거짓이면 게임이 클리어 되는 경로를 보여준다.

2) AG (*sensing* == true ⇒ AF (*lr_cycle_cnt*++))

이 궁극성 속성의 의미는 "모든 가능한 경우에 *sensing*이 일어나면 언젠가는 *lr_cycle_cnt*가 1증가 한다" 이다. 다시 말해서, 언제든지 크레인의 터치센서가 감지되면 움직인 거

리가 측정되고 다음 움직임을 결정한다. 만약 터치센서가 감지 되도 *lr_cycle_cnt*가 증가하지 않으면 다음 움직임 없이 Arm은 한 방향으로만 무한히 진행된다. SLAM 과 BLAST와 같은 기존 소프트웨어 모델 체킹 도구는 이와 같은 속성을 검증하기가 힘들다.

위 두 가지 속성을 검증한 결과, 크레인 시스템 프로그램이 주어진 경로에 따라 오작동 없이 공을 모두 목적지까지 옮길 수 있다는 것을 알 수 있었다. 따라서 실제 시스템을 실행한 결과도, 시스템이 올바르게 동작하여 모든 공을 목적지까지 옮기고 게임이 종료되었다.

이 술어 추상화를 수행한 이진 프로그램을 대표적인 모델 체킹 도구인 NuSMV 와 우리의 모델 체킹 도구와 성능 비교를 수행하였다. 이진 프로그램의 NuSMV 로의 변환 기법은[10]을 참조하였다. NuSMV에서 첫 번째 속성의 검사에 걸린 시간이 1.2 초에 메모리 사용량은 12메가바이트를 사용하였다. 하지만 우리의 방법은 0.5초 만에 6메가의 메모리만으로 속성을 검증하였다.

5. 결 론

오늘날, 소프트웨어 자체의 논리적 오류를 찾아내기 위해 소프트웨어 모델체커에 관한 연구가 매우 활발하다. 일반적으로 소프트웨어 검증 프로세스는 추상화-모델 체킹-추상화 개선이 사용된다. 하지만 이 프로세스는 C프로그램만을 대상으로 적용되어져 왔다. 또한 대부분의 도구에서 검사할 수 있는 속성은 간단한 수준의 assert 위반에 대해서만 검사 가능했다. 따라서 우리는 이 프로세스를 바탕으로 자바 프로그램을 대상으로 한 모델체커를 개발했다. 그리고 간단한 자바 프로그램을 검증하기 위해서 기존 CTL 모델 체킹 알고리즘을 확장하였다. 실제 레고 크레인 시스템을 구현하여 그 시스템에 탑재된 자바프로그램이 속성을 만족하는지 검증하였다. 속성은 CTL식을 이용하기 때문에 안전성 속성과 궁극성 속성 모두 표현이 가능하다. 따라서 두 가지 속성을 검증한 결과, 레고 프로그램이 주어진 경로에 따라 오작동 없이 공을 모두 목적지까지 옮길 수 있다는 것을 알 수 있었다. 또한 실제 시스템을 실행한 결과도, 시스템이 올바르게 동작하여 모든 공을 목적지까지 옮기고 게임이 종료된다는 것을 확인할 수 있었다. 하지만 현재는 간단한 자바프로그램에 대해서만 모델 체킹이 적용되고 있다. 따라서 자바의 멀티 쓰레드나 실행시간 동적 객체 생성에 대해 다룰 수 있는 연구가 필요하다. 또한 검증 도구를 기반으로 다양한 시스템에서 적용해 봐야 한다.

참 고 문 헌

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

[2] 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

[3] S. Graf and H. Saidi, "Construction of Abstraction State Graphs with PVS," in Proceedings of Computer Aided Verification, pp.72-83, 1997.

[4] T. Ball, R. Majumdar, T. Millstein and S.K. Rajamani, "Automatic Predicate Abstraction of C programs," SIGPLAN Notices, Vol.36, No.5, pp.203-213, 2001.

[5] T.A. Henzinger, R. Jhala, R. Majumdar and G. Sutre, "Lazy Abstraction," in Proceedings of Principles of Programming Languages, pp.58-70, 2002.

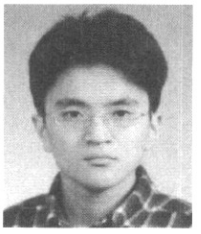
[6] S. Charki, E.M. Clarke, A. Groce, S. Jha and H. Veith, "Modular Verification of Software Components in C," IEEE Transactions on Software Engineering, Vol.30, No.6, pp.388-402, 2004.

[7] James Corbett, Matthew Dwyer, John Hatcliff, Corina Pasareanu, Robby, Shawn Laubach, Hongjun Zheng, "Bandera : Extracting Finite-state Models from Java Source Code", in Proceedings of the 22nd International Conference on Software Engineering, June, 2000.

[8] 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.

[9] G. Farrari, A. Gombos, S. Hilmer, J. Stuber, "Programming Lego Mindstorms with Java: The Ultimate Tool for Mindstorms Maniacs," Syngress, April, 2002.

[10] C. Eisner "Model checking the garbage collection mechanism of SMV", Electronic Notes in Theoretical computer Science Vol.55, Elsevier Science Publishers, 2001.



이 태 훈

e-mail : taehoon@kyonggi.ac.kr
2003년 경기대학교 전자계산학과(학사)
2003년~2005년 경기대학교 전자계산학과 석사과정
2005년~현재 경기대학교 전자계산학과 박사과정
관심분야 : 모델 체킹, 정형기법, 소프트웨어 공학

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