BIR 모델의 바운디드 모델 검증
(Bounded Model Checking BIR Model)

조 민 택 † 이 태 훈 † 권 기 현 ††
(Mintaek Cho) (Taehoon Lee) (Gihwon Kwon)

요 약 하드웨어 검증에서 성공적으로 적용되었던 모델 검증 기법을 소프트웨어 검증에 활용하는 연구가 활발하다. 이러한 연구 중의 하나가 바운디드 모델 검증이다. 바운디드 모델 검증에서는 모델이 갖는 상태 공간을 한꺼번에 모두 탐색하기 보다는 모델의 탐색 범위를 점진적으로 넓혀가면서 에러를 찾는다. 본 논문에서는 이러한 바운디드 모델 검증을 이용하여 BOGOR의 입력 언어인 BIR를 검증하였다. 그 결과 BOGOR에서 제공되는 명시적 모델 검증 기능 보다 우수한 성능을 보였다. 본 논문에서는 BIR 언어를 CNF 논리식으로 변환하는 방법과 단계적 절차를 기술한다.

키워드 : 바운디드 모델 검증, Bandera Intermediate Representation, Satisfiability, Conjunctive Normal Form, Linear Temporal Logic

Abstract Model checking has been successfully applied to hardware verification. Software is more subtle than hardware with respect to formal verification due to its infinite state space. Although there are many research activities in this area, bounded model checking is regarded as a promising technique. Bounded model checking uses an upper bound to unroll its model, which is the main advantage of bounded model checking compared to other model checking techniques. In this paper, we applied bounded model checking to verify BIR which is the input model for the model checking tool BOGOR. Some BIR examples are verified with our technique. Experimental results show that bounded model checking is better than explicit model checking provided by BOGOR. This paper presents the formalization of BIR and the encoding algorithm of BIR into CNF.

Key words : Bounded Model Checking, Bandera Intermediate Representation, Satisfiability, Conjunctive Normal Form, Linear Temporal Logic

1. 서 론

모델 검증(model checking)은 모델과 속성을 받아서 속성이 모델 내에서 만족하는지를 자동으로 결정한다[1]. 속성의 만족 여부를 결정하기 위해서 모델 검증은 모델이 갖는 상태 공간을 모두 탐색한다. 상태 공간을 전부 다 조사해야 하기 때문에 모델 검증은 유한 상태를 갖는 시스템 검증에 주로 사용되어 왔다.

최근에 모델 검증을 이용해서 소프트웨어를 검증하는 연구가 많이 시도되고 있다[2]. 그러나 소프트웨어가 갖는 상태 공간은 무한이라서, 모든 상태 공간을 전부 다 조사할 수는 없다. 이것을 '상태 폭발 문제'라고 부른다[3]. 소프트웨어 모델 검증에서 상태 폭발 문제를 극복하기 위해서 술어 추상화[4], 추상-정제 프레임워크[5], 그리고 바운디드(bounded) 모델 검증[6] 기법들이 사용되고 있다. 특히 바운디드 모델 검증은 검사하려는 모델의 크기를 유한 범위로 제한해서 에러를 검사한다. 만약 제한된 범위 내에서 에러를 찾지 못하면 점진적으로 범위를 넓혀가면서 에러를 찾는다.

모델 검증은 상태들의 집합 S, 초기 상태 집합 $I \subseteq S$, 전이 관계 $R \subseteq S \times S$, 그리고 레이블 함수 L: $S \rightarrow P(AP)$로 구성된 모델 $M=(S, I, R, L)$과 시제 논리식으로 표현된 속성 ϕ를 입력으로 받는다. 여기서 AP는 단순 명제의 집합이다. 바운디드 모델 검증은 모델과 속성 이외에도 검사 범위를 나타내는 정수 k를 입력으로 받는다. k는 초기 상태로부터 도달 가능한 경로의 최대 길이를 나타내는 수치로서 모델의 검사 범위를 제한한다. 제한

· 이 논문은 2007년도 정부(과학기술부)의 재원으로 한국과학재단의 지원을 받아 수행된 연구임(R01-2005-000-11120-0)
† 학생회원 : 경기대학교 전자계산학과
tesnoi@kyonggi.ac.kr
taehoon@kyonggi.ac.kr
†† 종신회원 : 경기대학교 전자계산학과 교수
khkwon@kynoggi.ac.kr
논문접수 : 2007년 4월 25일
심사완료 : 2007년 6월 28일

된 범위 내에서 모델이 속성을 만족하는지를 결정하기 위해서 바운디드 모델 검증은 모델과 속성을 논리곱(Conjunctive Normal Form, 이하 줄여서 CNF) 형태의 명제 논리식으로 변환한다. 그런 다음, 명제 논리식의 만족 여부를 결정하는 도구인 SAT 처리기를 이용해서 변환된 CNF 논리식을 검사한다. 만약 SAT 처리기의 실행 결과가 참이면 모델은 속성을 위반한 것으로 간주한다. 이 경우 바운디드 모델 검증은 에러(속성 위반)를 설명하는 증거로서 반례를 생성한다.

본 연구에서는 캔자스 주립대에서 개발한 소프트웨어 모델 검증 도구인 BOGOR[7]에서 사용되는 중간 표현 언어인 BIR(Bandera Intermediate Representation)[8]모델을 검사하는 바운디드 모델 검증 도구를 개발한다. BOGOR를 선택한 이유와 바운디드 모델 검증 도구를 개발하려는 이유는 다음과 같다. BOGOR는 모델 검증을 위한 탐색 기법을 다양하게 선택할 수 있어서 유연성이 높다. 그러나 BOGOR는 모델을 명시적으로 표현하기 때문에 모델의 크기가 커질수록 모델을 표현하는데 요구되는 메모리 양도 증가한다. 이를 개선하기 위해 다양한 BOGOR의 확장이 이루어지고 있다. 이러한 확장 연구의 일종으로서 바운디드 모델 검증 도구를 개발하여 기존 도구에 비해서 더 낮은 성능을 보이고자 한다.

검증 도구의 전체 개요는 그림 1과 같다. 입력으로는 BIR 모델, 속성을 나타낸 LTL(Linear Temporal Logic) 시제 논리식, 그리고 정수 k를 받는다. 그런 후, 인덱스 i를 1 부터 최대 k 까지 증가시켜가면서 모델과 속성을 CNF로 변환해서 SAT 처리기를 실행한다. 만약 범위 i 내에서 변환된 CNF 식을 조사해서 에러를 발견하지 못한 경우 i 를 점진적으로 증가시켜가면서 에러를 발견할 때까지 이러한 과정을 반복한다.

바운디드 모델 검증에서 핵심 분야 중의 하나가 BIR 모델과 LTL 속성을 CNF로 변환하는 것이다. LTL을 CNF로 변환하는 방법은 이미 제시되었다[6]. 그래서 LTL에 대한 변환은 기존 방법을 따르기로 했다. 그러나 우리가 알고 있기로는 BIR 모델을 CNF로 변환하는 연구는 거의 없어서, 본 논문에서는 이 부분에 촛점을

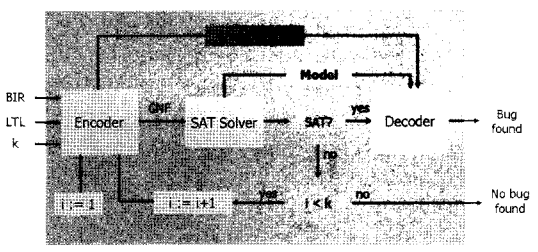

그림 1 BIR 모델의 바운디드 모델 검증 흐름

맞춘다. 그렇다고 해서 모든 BIR 구문을 CNF로 변환하려는 것은 아니다. 특히 스레드의 동적 생성, 재귀 호출, 실수 등은 CNF로 변환하기 어렵다. 그래서 본 연구에서는 이들 부분을 제외하고서 스레드 정적 생성, 정수, 덧셈, 뺄셈 연산등과 같이 BIR의 기본 구문들을 CNF로 변환하는 방법을 제안한다.

본 논문의 구성은 다음과 같다. 2장에서는 바운디드 모델 검증을 소개하고, 3장에서는 적용 대상인 BIR의 구문 및 의미에 대해서 살펴본다. 그리고 4장에서는 BIR 모델을 CNF로 변환하는 방법을 기술하며, 5장에서는 적용 사례와 실험 결과를 설명한다. 그런 다음 6장에서 결론 및 향후 연구 주제를 언급한다.

2. 바운디드 모델 검증

바운디드 모델 검증은 모델 검증 문제를 SAT(satisfiability) 문제로 간주하는 것으로서 Biere에 의해 소개되었다[9]. 바운디드 모델 검증은 BDD(Binary Decision Digaram) 기반의 모델 검증을 보완할 방법으로 주목을 받고 있다. 핵심적인 아이디어는 주어진 모델의 검사 범위를 제한한 후 이를 SAT 문제로 변환하여 제한된 모델의 상태 공간에서 주어진 속성을 위반하는 반례를 찾는 것이다. 바운디드 모델 검증은 속성을 나타낸 LTL 논리식 ϕ과 유한 상태 모델 M, 그리고 검사 범위를 제한하는 정수 k를 입력으로 받는다. 여기서 k는 모델 M이 갖는 모든 가능한 수행 경로들 중에서 검사할 최대 길이를 의미한다. M, ϕ, 그리고 k를 SAT 문제로 표현하면 다음과 같다.

$$[[M_{sat}]]_k = [[M]]_k \wedge [[\neg\phi]]_k$$

$[[M_{sat}]]_k$은 CNF 식을 나타내며 이 식을 SAT 처리기로 실행해서 불만족(UNSAT)이라는 결과가 나올 경우 k 이내에서 모델 M은 속성 ϕ를 만족한다고 결론내릴 수 있다. 만일 SAT 처리기 실행 결과가 만족(SAT)이라면 k 범위 내에서 모델 M은 속성 ϕ를 위반한다고 판정할 수 있다. 이 경우 반례가 생성된다. 범위가 k로 제한된 모델을 CNF 식으로 변환하는 과정은 다음과 같다.

$$[[M]]_k = I(s_0) \wedge \bigwedge_{i=1}^{k} R(s_{i-1}, s_i)$$

위의 식에서 $I(S_0)$는 M의 초기 상태를 나타내고, $R(S_{i-1},S_i)$는 M이 가질 수 있는 전이 관계를 나타낸다. 그래서 CNF 식 $[[M]]_k$는 초기 상태로부터 최대 k 길이 내에 도달 가능한 상태 집합을 의미한다.

바운디드 모델 검증은 반례를 찾아내는 것이 목적이다. 따라서 $[[\neg\phi]]_k$는 범위 k로 한정된 모델 M에서 위반 되어야 하는 속성 ϕ를 표현한다. 여기서 ϕ는 모델이

만족해야 하는 안정성과 궁극성 속성으로서, 예를들어 안전성과 궁극성을 나타낸다. 궁극성 속성은 "언젠가는 원하는 상태에 도달한다"를 의미한다. Fp 형태로 표현되는 궁극성 속성은 k로 제한된 모델의 범위 내에서 루프를 나타내는 경로가 있는가를 확인함으로써 검사된다. 즉, 루프가 존재함으로 인해서 결국 p를 만족하는 상태에 도달하지 못하기 때문이다. 한편, Gp 형태로 표현된 안전성 속성은 "도달 가능한 모든 상태가 원하는 상태이다"를 나타낸다. 이 경우 k 길이 내에서 p를 만족하지 않는 상태에 도달하는 경로가 있음을 보이면 된다.

바운디드 모델 검증은 주어진 속성을 위반하는 경우만을 찾는 보전적인(conservative) 방법이다. 그림 1과 같이 모델의 가능한 상태를 모두 검사했음을 의미하는 k 최대값이 있어야 속성을 위반하는 에러가 모델내에 없음을 보장할 수 있다. 하지만 최대값을 쉽고 간단하게 구하는 일반적인 방법은 아직 없으며 이를 효율적으로 계산하는 것이 바운디드 모델 검증의 중요한 연구주제이다[10]. 이러한 특징 때문에 바운디드 모델 검증은 일반적으로 속성의 위반을 찾기 위해 많이 쓰이며, 모델 검증 수행 시 사용자가 지정한 한계 값에 도달하거나 혹은 계산에 필요한 자원(메모리, 실행시간)이 초과될 때까지 k 값을 증가시켜 가며 모델에 대한 검사를 수행한다.

3. BIR 모델

BIR는 자바 프로그램을 모델 검증하는 BANDERA 프로젝트의 중간 표현 언어로 개발되었다[8].

3.1 BIR 구문

BIR의 구문은 선언부와 제어부로 크게 구성된다. 그림 2의 (2)에 있는 <system-member>에서 보듯이 선언부는 BIR 모델이 가지는 변수와 스레드 등을 선언하고 초기값을 할당하는 부분이다(초기값을 선언하지 않는 경우 해당 타입에 따라 초기 값이 자동으로 할당된다). 제어부는 모델이 가지는 행위를 표현하는 부분으로 그림 2의 <fsm>, <thread>, <function>으로 정의된다. <thread>는 독립적으로 동작하는 스레드를 정의하며, 다수의 스레드가 존재할 경우 병렬적으로 동작한다. BIR 모델에서는 스레드가 최소한 하나 이상 존재해야 한다. 하나의 스레드만 존재하는 경우 이 스레드는 main을 의미한다. <function>은 함수를 정의하며, 함수는 스레드에 의해서 호출되며, 호출한 스레드에 종속적이다.

BIR 표현은 제어부인 스레드와 함수 표현 방식에 따라 상위 표현과 하위 표현 두 가지 표현 방식을 지원한다. 상위 표현은 일반적인 프로그래밍 언어와 비슷한 구문형태를 가지며 프로그래밍 언어와 같은 수준의 표현이 가능하다. 이에 반해 하위 표현은 단순하게 만족해야

```
(1) <system> ::= "system" <system-id> "{"
                 <system-member>* "}"
(2) <system-member> ::= <const> | <enum>
          | <record> | <extension> | <type-alias>
          | <global-var> | <fsm>
          | <virtual-table> | <fun>
(3) <fsm> ::= <thread> | <function>
(4) <thread> ::= ("active" ("[" <num-active> "]")?)?
          "thread" <thread-id> <params>? ")"
          "{" <local-var> * (<low-level-body>
          | <high-level-body>) "}"
(5) <function> ::= "function" <thread-id>
          "(" <params>? ")" ("returns"
          <basic-type>)? "{" <local-var> *
          (<low-level-body> | <high-level-body>) "}"
```

그림 2 BIR 모델의 구문

```
1. system Example {
2.   int  x = 0;
3.   active thread MAIN() {
4.       x := x + 1;
5.   }
6. }
```

(a) 상위 표현 모델

```
1.  system Example {
2.    int  x = 0;
3.    active thread MAIN() {
4.       int temp$0;
5.       loc loc0: do { temp$0 := x; } goto loc1;
6.       loc loc1: do { x := temp$0 + 1; } goto loc2;
7.       loc loc2: do { } return;
8.    }
9. }
```

(b) 하위 표현 모델

그림 3 BIR 모델의 표현 방식

할 '조건 구문'과 조건이 만족할 경우 실행되어질 '명령 구문'의 형태를 가지는 조건-명령 형태로 표현된다. 이들 두 표현은 내부적으로 호환이 가능하며 자유롭게 변환이 가능하다.

모델 검증을 수행할 때에는 상위 표현 모델의 경우 하위 표현 모델로 변환하여 모델 검증을 수행하며, 하위 표현 모델의 경우는 그대로 모델 검증을 수행한다. 따라서 모델 검증으로 반례를 찾게 된다면 이 반례는 하위 표현 모델에 대한 경로로 주어진다. 그림 3의 (a)는 상위 표현 모델의 예를 보여주며, (b)는 (a) 표현에 대한 하위 표현 모델을 나타낸다.

3.2 BIR 의미

M_{bir}는 스레드의 집합 Th와 변수의 집합 V로 구성된다.

$$M_{bir} = (Th, V)$$

Th는 병행적으로 동작하는 스레드의 집합이다. 주어진 BIR 모델에서 n개의 스레드가 병행적으로 동작할 경우는 다음과 같이 표현한다.

$$Th = \{th_1, th_2, \ldots, th_n\}$$

임의의 스레드 $th \in Th$는 상태 집합 S, 초기 상태 I, 그리고 전이 관계 R로 구성된다.

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

$S = \{s_1,\ldots,s_n\}$는 스레드가 가지는 제어 위치 즉, 제어 구문의 위치 집합을 의미한다. 예를 들어, 그림 3의 (b)에서 5, 6, 7번째 행이 제어 위치들 이다. I는 스레드의 초기 제어 위치를 의미한다(4번 행과 같은 선언 구문은 제어 위치에 포함되지 않는다). $R=\{r_1,\ldots,r_n\}$은 스레드의 전이 관계 집합을 의미한다. 여기서 각각의 전이 r은 다음과 같다.

$$r \in S \times G \times Act \times S'$$

S는 현재 상태 집합, $G=\{g_1,\ldots,g_n\}$는 스레드의 전이를 표현하는 구문에서 만족해야하는 조건문을 나타내는 조건식들의 집합이다. $Act=\{a_1, \ldots, a_n\}$는 전이가 발생했을 때 실행되는 구문들의 집합이며, S'은 다음 단계에서 상태 집합을 나타낸다. 예를 들어 스레드는 현재 제어 위치가 s_i에 있고 조건 g_i를 만족한다면 a_i를 실행한 후에는 제어 위치가 s_i' 변경된다. 여기서 현재 상태 s_i와 조건 구문 g_i는 스레드의 전이 발생을 유발하는 요소가 된다.

V는 BIR 모델에서 사용하는 공유변수와 지역변수들의 집합이다. 모델에 m 개의 변수가 존재한다고 하면 V에 대한 구성은 다음과 같이 표현할 수 있으며, 각각의 변수 $v \in V$는 해당 도메인 D에 속한다.

$$V = \{v_1, v_2, \ldots, v_n\}$$

v는 타입에 의해 변수가 가지는 범위와 초기 값을 할당받는다. 예를 들어 정수형 변수의 경우는 16bit의 범위를 가진다. v는 그림 3의 (b)에서 2번과 4번 라인과 같이 선언된다. 초기 값을 직접 할당 할 수도 있지만, 만약 할당하지 않을 경우는 타입별로 자동으로 초기 값이 할당된다(정수나 실수 변수의 경우는 0, 이진 변수 경우는 *false* 가 할당된다).

선언문에 의해서 초기 값과 범위를 가지게 되는 변수들은 스레드의 전이가 발생할 때 실행되어지는 구문들의 집합 Act에 의해 값이 변하게 된다. 그러므로 변수 값의 변화 즉, 변수의 전이는 스레드의 전이 발생에 의해 유발된다.

BIR 모델의 행위는 $M_{bir}=(S_{bir}, I_{bir}, R_{bir}, L_{bir})$로 구성된다. 여기서 S_{bir}는 상태 집합이다. 상태는 $th \in Th$와 $v \in V$의 조합이다. 초기 상태 I_{bir}는 초기 상태를 나타내며 $th \in Th$와 $v \in V$의 초기 상태들의 조합이다. R_{bir}는 전이 관계를 나타내면 $th \in Th$와 $v \in V$들의 전이 집합의 조합이다. 마지막으로 $L:S_{bir} \rightarrow P(A)$은 레이블 함수이다.

4. CNF 변환

4.1 모델의 변환

바운디드 모델 검증 기법을 적용하여 BIR 모델을 검증하기 위해서는 BIR 모델이 가지는 의미를 바운디드 모델 검증의 입력 형태인 CNF 식으로 변환하여야 한다. 본 연구에서는 다음과 같은 절차로 BIR 모델을 CNF 식으로 변환한다.

- 상태 전이 테이블 생성
- 초기상태 특성함수 생성
- 전이 특성함수 생성
- 구문의 CNF식 생성
- 상태 유지를 위한 CNF식 생성
- 스레드 스케쥴링 CNF식 생성

4.1.1 상태 전이 테이블 생성

BIR는 (Th,V)를 구성 요소로 가진다. 변수들은 BIR 모델에서 선언된 초기 값으로부터 해당 변수의 초기 값 즉, 초기 상태를 구할 수 있다(만약 초기값이 없을 경우는 내부적으로 변수 타입에 따라 초기값이 할당 된다). 변수를 표현하는 이진 변수들의 집합은 기본적으로 선언된 타입으로 정해진다. 예를 들어 8-bit를 가지는 byte 타입으로 선언된 변수의 경우 이진 변수 8개로 상태를 표현한다.

스레드의 경우 상태 집합과 초기 상태를 명시적으로 구하기 어렵다. 이를 해결하기 위해 BIR 모델의 명세로부터 그림 4의 (b)와 같이 상태 전이 테이블을 작성 한다. 상태 전이 테이블은 BIR 모델의 스레드 th의 전이 관계인 $r \in R$ 들을 테이블 형태로 표현한 것이다. 전이 r 별로 프로그램 위치 s, 만족해야 하는 조건 g, 전이 발생 시 수행해야하는 액션 a와 전이 발생 후 다음 상태 s'를 전이들 별로 테이블 형태로 조직화 해 놓은 것이다. 이렇게 조직화된 상태 전이 테이블들은 초기 상태 특성함수 $I(s_0)$, 전이 관계 특성함수 $R(s_{i-1},s_i)$를 생성하기 위한 기반을 마련한다. 그림 4는 간단한 BIR 모델에 해당하는 상태 전이 테이블이다.

그림 4(a)의 예제 BIR 모델에는 1개의 스레드가 존재하고, 이 스레드는 3개의 프로그램 위치를 가지고 있다. 1번 라인은 *foo*라는 BIR 모델을 생성한다. 2번 라인은 *foo* 모델의 main 스레드를 선언한다. 3-5번 라인은 *foo* 모델에서 사용하게 될 변수들을 선언해 준다. 특히, 5번 라인은 byte 타입의 변수 x의 크기를 0부터 3으로 정의한 것이다. 즉, 2-bit로 크기를 제한한 것이다. 6-8번 라인은 BIR 모델이 가지게 되는 전이 관계를 의미한다. 즉, main 스레드의 전이인 $r \in R$을 의미한다. 그림 4의 (b)는 각 프로그램 위치 $r \in R$에 대해서 상태 전이 테이블로 정리해 놓은 것이다.

```
1. system foo {
2.   active thread MAIN() {
3.   Boolean temp$0 := true;
4.   Boolean temp$1 := false;
5.   byte wrap(0, 3) x;
6.   loc loc0: when temp$0 do { x:= x+1 } goto loc1;
7.   loc loc1: when !temp$1 do { temp$1 := true }
       goto loc2;
8.   loc loc2: do { } goto loc0;
9.   }
10. }
```

(a) BIR 모델

현재상태	가드조건	액 션	다음상태
loc0	temp$50	x := x + 1	loc1
loc1	!temp$1	temp$1 := true	loc2
loc2			loc0

(b) main 스레드의 상태 전이 테이블

그림 4 BIR 모델과 상태 전이 테이블

스레드의 총 상태 수 즉, 전이의 수를 j라고 했을 때 스레드를 표현하기 위해서는 $log_2 j$ 만큼의 이진 변수가 필요하다. 그림 4의 경우 총 3개의 상태가 있으므로 이를 표현하기 위해서는 2개의 이진 변수가 필요하다.

4.1.2 초기 상태 특성함수 생성

BIR 모델에서 초기 상태 특성함수 $I(s_0)$는 모델이 가지고 구성요소들의 초기 위치와 초기 값을 통해서 구해진다. 그림 4(a)의 BIR 모델은 main 스레드와 *temp$0*, *temp$1* 그리고 x의 4개 구성요소를 가진다. 이 중 변수는 선언문의 타입과 초기값을 통해서 쉽게 범위와 초기상태를 구할 수 있다. 예를 들어 *temp$1*는 *boolean* 타입이므로 {*true*, *false*}의 2개 상태를 가지며, 초기상태는 *false* 임을 알 수 있다. 또한 2개의 상태가 존재하므로 이진 변수를 하나로 모든 상태를 표현할 수 있다. 하지만 스레드 main의 경우는 초기상태와 총 상태의 수가 명시적으로 정의가 되어있지 않다. 이를 해결하기 위해서 본 연구에서는 상태 전이 테이블이라 명명한 테이블을 만든다. 그림 4(b)와 같이 만들어진 상태 전이 테이블을 통해서 스레드 main은 총 상태가 {*loc0*, *loc1*, *loc2*}로 구성되며 3개의 상태 범위를 가지며 임의의 이진 변수 *p1.a*, *p1.b*를 사용하여 모든 상태의 표현이 가능하다. 또한 초기상태는 가장 처음 상태 전이 테이블에 나타나는 *loc0*가 된다. 이렇게 구해진 상태 전이 테이블과 초기값을 기반으로 다음과 같은 수식을 사용하여 초기 상태 특성 함수를 구할 수 있다.

$$I(s_0)_{bir} = (\bigwedge_{i=1}^{n} init(th_i)) \wedge (\bigwedge_{j=1}^{m} init(v_j))$$

*init()*은 해당 구성요소의 초기 상태를 표현하는 이진 변수들의 논리곱을 돌려주는 함수이다 (n은 스레드의 수이고 m은 변수의 수이다).

4.1.3 전이 관계 특성함수 생성

전이 관계 특성 함수는 앞서 다룬 초기 상태 특성함수와 비슷하게 각 구성 요소들의 전이 집합들에 대한 논리곱 형태로 구성되며, 아래와 같이 구할 표현할 수 있다.

$$R(s_{i-1}, s_i)_{bir} = (\bigwedge_{i=1}^{n} trans(th_i)) \wedge (\bigwedge_{j=1}^{m} trans(v_j))$$

(1) 제어 전이 특성 함수의 구성

스레드의 상태 전이 테이블에서 각 행은 하나의 전이를 의미한다. 스레드는 한 스탭 동안 단 하나의 전이만이 발생해야 하며, 비결정적인 전이의 발생을 허용하지 않기 때문에 논리합 형태로 표현이 가능하다. 만약 교착상태 이거나 다른 현재 스텝에서 다른 스레드가 선택되어졌을 경우 스레드는 기존의 상태를 유지하고 있어야 한다. *stay*(th)는 현재 상태를 유지하도록 하는 식을 생성하는 함수이다. 이는 상태 유지 CNF 생성에서 자세히 다루기로 한다 (nR은 스레드가 가지고 있는 전이의 수, $th.tr$은 스레드의 전이들을 대표하는 변수이다).

$$trans(th) = (th.tr \leftrightarrow (\bigvee_{i=1}^{nR} r_i))$$

각각의 전이 $r \in R$ 들은 상태 전이 테이블을 사용하여 표현한다.

$$\bigwedge_{j=1}^{nR} (r_j \leftrightarrow (s_j \wedge g_j \wedge a_j \wedge s'_j))$$

전이 표현의 간결성을 위해 조건 g_j 와 액션 a_j는 대표 변수를 사용하여 표현하며, 각 조건과 액션에 대한 구문은 따로 정의한다. 예를 들어 g_j가 x>5일 경우 $g_j \leftrightarrow (x>5)$의 전이 r을 표현하는 수식에서는 대표 변수 g_j를 g_j에 대한 내용은 $g_j \leftrightarrow (x>5)$의 식을 추가한다. 만약 상태 전이 테이블에서 조건과 액션이 없는 경우는 공허한 참으로 이들을 표현한다.

(2) 변수 전이 특성 함수의 구성

변수들은 스레드의 전이 발생에 의해 실행되는 액션 a로 인해 값이 변하게 된다. 즉, 변수 값은 상태 전이 테이블에서 변수가 종속되어 있는 전이에 의해 값이 변하게 된다. 그림 4(a)의 예제에서 사용되는 변수 x는 스레드의 r_1의 전이가 발생하면 값이 변하게 된다. 다시 말해 변수 x는 전이 r_1에 종속되어 있다. 이러한 변수의 전이는 종속되어 있는 전이발생에 따라 값의 변화가 결정된다. 이는 다음과 같이 표현할 수 있다($dep(v)$는 모든 스레드의 전이들 중 해당 변수가 종속되어 있는 전이의 집합을 돌려주는 함수이다).

$$trans(v) = (v.tr \leftrightarrow (\bigvee_{j \in dep(v)} r_j))$$

변수 또한 자신이 종속된 전이가 발생하지 않을 경우에는 현재의 값을 유지해야 하므로 현재 상태를 유지하는 전이 $stay(v)$를 추가한다.

4.1.4 구문의 CNF식 생성

BIR에서 사용되는 대표적인 구문 중 변수의 크기를 비교하는 구문과 덧셈, 뺄셈에 대한 CNF식 생성은 다음과 같다.

(1) 동치 비교: $l == r$

변수 l과 변수 r의 동치 비교는 다음식과 같다. 여기서 l_i과 r_i는 각각 변수 l과 r을 표현하는 이진 변수들을 의미한다.

$$\bigwedge_{1 \le i \le n} (l_i \leftrightarrow r_i)$$

(2) 크기 비교: $l < r$

변수 l과 변수 r의 크기 비교는 가장 큰 비트를 표현하는 이진 변수부터 비교를 시작한다(n은 두 변수를 표현하는 이진 변수들의 수).

$$(L_n \leftrightarrow (\neg l_n \wedge r_n)) \wedge$$
$$(E_n \leftrightarrow (l_n \leftrightarrow r_n)) \wedge$$
$$\bigwedge_{1 \le i < n} (L_{n-i} \leftrightarrow (E_{n-i+1} \wedge (\neg l_{n-i} \wedge r_{n-i}))) \wedge$$
$$\bigwedge_{1 \le i < n} (E_{n-i} \leftrightarrow (E_{n-i+1} \wedge (l_{n-i} \wedge r_{n-i}))) \wedge$$
$$\bigvee_{1 \le i < n} L_i$$

(3) 덧셈: $l + r$

두 변수에 대한 덧셈은 전가산기를 명제 논리식으로 변환하여 구현이 가능하다(c는 캐리지리턴 값).

$$(x_1 \leftrightarrow (l_1 \oplus r_1)) \wedge$$
$$(c_1 \leftrightarrow (l_1 \wedge r_1)) \wedge$$
$$\bigwedge_{1 \le i < n} (x_i \leftrightarrow (l_i \oplus r_i \oplus c_{i-1})) \wedge$$
$$\bigwedge_{1 \le i < n} (c_i \leftrightarrow ((l_i \wedge r_i) \vee (c_{i-1} \wedge (l_i \vee r_i))))$$

(4) 뺄셈: $l - r$

두 변수의 뺄셈은 다음과 같이 구현이 가능하다.

$$(x_1 \leftrightarrow (l_1 \oplus r_1)) \wedge$$
$$(c_1 \leftrightarrow (\neg l_1 \wedge r_1)) \wedge$$
$$\bigwedge_{1 \le i < n} (x_i \leftrightarrow (l_i \oplus r_i \oplus c_{i-1})) \wedge$$
$$\bigwedge_{1 \le i < n} (c_i \leftrightarrow ((l_i \wedge r_i \wedge c_{i-1}) \vee (\neg l_i \wedge (r_i \vee c_{i-1}))))$$

4.1.5 상태 유지 CNF식 생성

스레드의 전이가 발생하게 되면 해당하는 스레드의 상태는 변하게 되며, 전이에 의해 실행된 액션으로 인해 해당하는 변수의 값은 변하게 된다. 하지만 전이가 발생하지 않은 스레드들과 전이에 영향을 받지 않은 변수들의 경우는 현재의 상태를 유지해야 한다. 즉, 전이가 발생하지 않을 경우는 모든 구성 요소들은 현재 상태를 유지해야 한다. 이는 다음과 같이 표현 할 수 있다(nS는 해당 스레드의 상태 수를 의미한다). 변수 전이에 대한 상태 유지도 CNF식으로 다음과 같이 표현한다.

$$stay(th) = stay.th \leftrightarrow (\neg th.tr \wedge \bigwedge_{j=1}^{\log_2 nS} stay.x_j)$$
$$stay.x_j \leftrightarrow ((x_j \wedge x_j') \vee (\neg x_j \wedge \neg x_j'))$$

4.1.6 스레드 스케쥴링 CNF식 생성

멀티 스레드 모델의 경우 한 스텝동안 단 하나의 스레드의 전이만이 발생하여야 하며, 최대 하나의 변수의 값만이 변해야 한다. 변수의 경우는 스레드의 전이에 종속적이기 때문에 특별하게 제어를 해 줄 필요는 없다. 하지만 스레드의 경우는 한 스텝 동안 단 하나의 스레드만이 동작하고 단 하나의 전이만이 발생해야 하므로 이는 다음과 같은 표현으로 제어가 가능하다(n은 스레드의 수이다).

$$(\bigvee_{i=1}^{n} th_i) \wedge$$
$$\bigwedge_{i=1}^{n-1} \bigwedge_{j=i+1}^{n} (\neg th_i \vee \neg th_j)$$

첫 번째 식은 모든 스레드들 중 하나는 꼭 선택되어야 한다는 것을 의미한다. 두 번째 식은 모든 스레드들 중 최대한 하나의 스레드만이 선택되어야 함을 의미한다.

4.2 속성식의 변환

LTL 속성식의 변환 방법과 정리는 앞서 언급했던 것처럼 [6]에 소개가 되어 있으므로 간단한 소개와 변환 방법에 대한 내용 정리로 대체한다.

바운디드 모델 검증은 2장에서 살펴본 바와 같이 주어진 속성식에 대한 반례를 찾는 것이 목적이다. 검사하고자 하는 속성이 궁극성 속성일 경우 반례는 속성식을 만족하지 않는 루프가 된다. 만약 루프가 아닐 경우에는 주어진 범위 내에서 속성을 만족하지 않더라도 그 다음 범위에서 속성을 만족할 수 있기 때문에 정확하게 속성에 대해 만족, 불만족을 말할 수 없게 된다. 하지만 안전성 속성의 경우는 초기상태로부터 속성을 만족하지 않는 상태까지의 경로가 반례가 되기 때문에 루프가 존재하지 않아도 명확하게 만족, 불만족을 말할 수 있다. 그렇기 때문에 바운디드 모델 검증의 LTL 속성은 루프의 존재 여부에 따라서 검사 할 수 있는 속성이 달라진다. 즉, 루프가 존재하지 않을 경우 궁극성 속성에 대해서는 검사할 수 없지만, 안전성 속성에 대해서는 검사가 가능하다.

(1) 루프가 존재하는 경우 : f, g는 LTL식이며, k는 주어진 바운드, l은 루프 발생 위치, i는 현재 진행 위치이다($k,l,i \ge 0$이며 $l,i \le k$이다.)

$_l[[p]]^i_k := p(s_i)$

$_l[[\neg p]]^i_k := \neg p(s_i)$

$_l[[f \vee g]]^i_k := {}_l[[f]]^i_k \vee {}_l[[g]]^i_k$

$_l[[f \wedge g]]^i_k := {}_l[[f]]^i_k \wedge {}_l[[g]]^i_k$

$_l[[Gf]]^i_k := {}_l[[f]]^i_k \wedge {}_l[[Gf]]^{succ(i)}_k$

$_l[[Ff]]^i_k := {}_l[[f]]^i_k \vee {}_l[[Ff]]^{succ(i)}_k$

$_l[[fUg]]^i_k := {}_l[[g]]^i_k \vee ({}_l[[f]]^{succ(i)}_k \wedge {}_l[[fUg]]^{succ(i)}_k)$

$_l[[fRg]]^i_k := {}_l[[g]]^i_k \wedge ({}_l[[f]]^{succ(i)}_k \vee {}_l[[fRg]]^{succ(i)}_k)$

$_l[[Xf]]^i_k := {}_l[[f]]^{succ(i)}_k$

i가 루프의 안에 있을 경우는 $succ(i)=i+1$로 정의되고, $i=k$일 경우 $succ(i)=l$ 즉, 루프가 시작되는 위치를 향하게 된다.

(2) 루프가 존재하지 않는 경우

$[[p]]^i_k := p(s_i)$

$[[\neg p]]^i_k := \neg p(s_i)$

$[[f \vee g]]^i_k := [[f]]^i_k \vee [[g]]^i_k$

$[[f \wedge g]]^i_k := [[f]]^i_k \wedge [[g]]^i_k$

$[[Gf]]^i_k := [[f]]^i_k \wedge {}_l[[Gf]]^{i+1}_k$

$[[Ff]]^i_k := [[f]]^i_k \vee [[Ff]]^{i+1}_k$

$[[fUg]]^i_k := [[g]]^i_k \vee ([[f]]^i_k \wedge [[fUg]]^{i+1}_k)$

$[[fRg]]^i_k := [[g]]^i_k \wedge ([[f]]^i_k \vee [[fRg]]^{i+1}_k)$

$[[Xf]]^i_k := [[f]]^{i+1}_k$

만약 현재 진행 위치 $i=k$일 경우의 $k+1$을 나타낼 수 없기 때문에 값은 다음과 같이 정의한다.

$$[[Ff]]^{k+1}_k := 0$$

k로 제한된 모델 M에 루프가 존재할 경우 정의 (1)의 변환 규칙을 적용하며, 루프가 존재하지 않을 경우 (2)의 변환 규칙을 적용하여 LTL 속성식을 CNF로 변환한다. 예를 들어 루프가 존재하지 않는 모델 M에서 k가 2일 때 속성 Fp를 만족해야 한다면 $[[Fp]]_2$는 다음과 같은 식들로 구성된다.

$$[[Fp]]^0_2 := p_0 \vee [[Fp]]^1_2 \quad [[Fp]]^1_2 := p_1 \vee [[Fp]]^2_2$$

$$[[Fp]]^2_2 := p_2 \vee [[Fp]]^3_2 \quad [[Fp]]^3_2 := 0$$

이렇게 구성된 식들을 정리하면 $[[Fp]]_2$는 다음과 같이 CNF 형태의 명제 논리식으로 변환된다.

$$[[Fp]]_2 := (p_0 \vee p_1 \vee p_2)$$

5. 사례연구

사례로는 널리 알려진 식사하는 철학자 문제를 사용하였다. 그림 5는 2명의 철학자와 2개의 포크를 사용한 BIR 모델이다. 각각의 철학자는 스레드로 표현되었으며,

```
system TwoDiningPhilosophers{
    boolean fork1 := false;
    boolean fork2 := false;
    active thread Philosopher1() {
        loc loc0: live {} // take first fork
            when !fork1 do { fork1 := true; }
                goto loc1;
        loc loc1: live {} // take second fork
            when !fork2 do { fork2 := true; }
                goto loc2;
        loc loc2: live {} // put second fork
            do { fork2 := false; }
                goto loc3;
        loc loc3: live {} // put first fork
            do { fork1 := false; }
                goto loc0;
    }
    active thread Philosopher2() {
        loc loc0: live {} // take second fork
            when !fork2 do { fork2 := true; }
                goto loc1;
        loc loc1: live {} // take first fork
            when !fork1 do { fork1 := true; }
                goto loc2;
        loc loc2: live {} // put first fork
            do { fork1 := false; }
                goto loc3;
        loc loc3: live {} // put second fork
            do { fork2 := false; }
                goto loc0;
    }
}
```

그림 5 식사하는 철학자 BIR 모델

포크는 두 스레드가 공유하는 공유변수로 정의하였다. 각 철학자 스레드들은 4개의 상태 집합을 가진다.

5.1 예제의 변환

이번 절에서는 그림 5의 예제를 앞서 제안한 방법을 적용하여 CNF 식으로 변환한다. 그림 6은 예제로부터 구해진 철학자 스레드들에 대한 상태 전이 테이블이다 (상태 전이 테이블에서 현재 상태와 다음 상태 중 *Philosopher1*, *Philosopher2*는 스레드의 구분자이며 수식표현의 복잡성을 피하기 위해 *p1*, *p2*로 줄여서 기술하기로 한다). 이를 기반으로 해서 1번 철학자가 가지는 구성요소를 다음과 같이 구할 수 있다.

- 상태 집합 $S=\{loc0, loc1, loc2, loc3\}$
- 초기 상태 $I=\{loc0\}$
- 전이 관계 $R=\{r_1, r_2, r_3, r_4\}$

총 상태 수는 4이므로 이진 변수 2개로 표현이 가능하며 이들을 *p1.lv1*과 *p1.lv2*라 하기로 한다. 이들을 사용하여 상태 집합을 표현하면 다음과 같다.

$$S = \{(\neg p1.lv1 \wedge \neg p1.lv2), (\neg p1.lv1 \wedge p1.lv2),$$
$$(p1.lv1 \wedge \neg p1.lv2), (p1.lv1 \wedge p1.lv2)\}$$

공유변수로 사용되고 있는 *fork1*은 그림 5의 선언문을 통해 하나의 이진 변수로 모든 상태의 표현이 가능

현재상태	가드조건	액 션	다음상태
Philosopher1 loc0	!fork1	fork1 := true	Philosopher1 loc1
Philosopher1 loc1	!fork2	fork2 := true	Philosopher1 loc2
Philosopher1 loc2		fork2 := false	Philosopher1 loc3
Philosopher1 loc3		fork1 := false	Philosopher1 loc0

(a) 1번 철학자에 대한 상태 전이 테이블

현재상태	가드조건	액 션	다음상태
Philosopher2 loc0	!fork2	fork2 := true	Philosopher2 loc1
Philosopher2 loc1	!fork1	fork1 := true	Philosopher2 loc2
Philosopher2 loc2		fork1 := false	Philosopher2 loc3
Philosopher2 loc3		fork2 := false	Philosopher2 loc0

(b) 2번 철학자의 상태 전이 테이블

그림 6 식사하는 철학자 문제의 상태 전이 테이블

하며, 이를 *var1*이라 하기로 한다. 또한 초기상태가 *false*라는 것도 선언문을 구할 수 있다. *fork2*번 *var2*로 표현하고 초기상태도 *false*이다.

(1) 초기상태 특성함수 생성

변수들의 초기상태는 *false*이며, 각각 하나의 이진 변수로 표현이 가능하므로 아래와 같이 변수들의 초기 상태를 구할 수 있다.

$$\neg var1_0 \wedge \neg var2_0$$

각 철학자 스레드들은 이진 변수 2개로 표현되며 두 스레드 모두 초기 상태는 *loc0*이다. 그러므로 아래와 같이 초기 상태를 구할 수 있다.

$$\neg p1.lv1_0 \wedge \neg p1.lv2_0 \wedge \neg p2.lv1_0 \wedge \neg p2.lv2_0$$

(2) 전이 특성 함수의 CNF식 생성

1번 철학자의 전이 집합 R={$p1.r_1$, $p1.r_2$, $p1.r_3$, $p1.r_4$}이며 다음과 같이 전이 집합을 표현하며,

$$p1.tr \leftrightarrow (p1.r_1 \vee p1.r_2 \vee p1.r_3 \vee p1.r_4)$$

각각의 전이들은 아래와 같다. 이들을 k 번만큼 사용한다.

$$p1.r_1 \leftrightarrow (loc0 \wedge g_1 \wedge a_1 \wedge loc1')$$
$$p1.r_2 \leftrightarrow (loc1 \wedge g_2 \wedge a_2 \wedge loc2')$$
$$p1.r_3 \leftrightarrow (loc2 \wedge g_3 \wedge a_3 \wedge loc3')$$
$$p1.r_4 \leftrightarrow (loc3 \wedge g_4 \wedge a_4 \wedge loc0')$$

*fork1*은 1번 철학자의 {p1.r_1, p1.r_4}, 2번 철학자의 {p2.r_2, p2.r_3}전이에 의해 값이 변하게 되므로 아래와 같다.

$$v1.tr \leftrightarrow (p1.r_1 \vee p1.r_4 \vee p2.r_2 \vee p2.r_3)$$

(3) 구문의 CNF식 생성

1번 철학자의 $p1.r_1$의 가드 조건 g_1과 a_1은 아래와 같이 구할 수 있으며, 다른 가드 조건과 액션도 아래와 같다.

$$g_1 \leftrightarrow \neg fork1$$
$$a_1 \leftrightarrow fork1$$

(4) 상태 유지를 위한 CNF식 생성

1번 철학자의 상태를 표현하는 이진 변수에 대한 상태 유지식은 아래와 같으며, 2번 철학자와 *fork* 변수 들도 아래와 같이 구할 수 있다.

$$stay.p1 \leftrightarrow (\neg p1.tr \wedge stay.p1.lv1 \wedge stay.p1.lv2)$$
$$stay.p1.lv1 \leftrightarrow ((p1.lv1 \wedge p1.lv1') \vee (\neg p1.lv1 \wedge \neg p1.lv1'))$$
$$stay.p1.lv2 \leftrightarrow ((p1.lv2 \wedge p1.lv2') \vee (\neg p1.lv2 \wedge \neg p1.lv2'))$$

(5) 스레드 스케쥴링 CNF식 생성

두 개의 스레드가 병렬적으로 동작하는 예제이므로 한번의 실행 시 마다 하나의 스레드 전이가 실행되어야 한다. 이들은 아래와 같이 표현되며 초기 상태부터 검사 범위 k 까지 k+1 만큼 생성한다.

$$(p1.tr \vee p2.tr) \wedge (\neg p1.tr \vee \neg p2.tr)$$

(6) 속성식의 CNF식 변환

BIR 모델 검증 도구인 BOGOR에서는 기본적으로 모델을 구성하고 있는 모든 구성 요소들(*Th, V*)가 정상적인 종료 조건이 아닌 상황에서 상태 변화 없이 현재 상태를 유지하고 있는 경우를 교착상태로 정의하고 있다. 이는 모든 철학자가 하나씩 포크를 들고 있는 상태인 *loc1* 에 머무르는 경우가 교착상태가 된다. 이는 아래와 같이 표현할 수 있다.

$$G\neg(Philosohper1.loc1 \wedge Philosohper2.loc1)$$

2장에서 살펴 본 바와 같이 교착상태에 도달하는지를 정의하기 위해 속성식을 아래와 같이 부정을 취하여 적용한다.

$$F(Philosopher1.loc1 \wedge Philosopher2.loc1)$$

p=(*Philosopher1.loc1* $\wedge$ *Philosopher2.loc1*)라 하고 범위 k를 2라 하였을 때 앞장에서 다루었던 LTL 변환 규칙을 적용하면 다음과 같은 CNF 식을 구할 수 있다.

$$[[Fp]]_2 := p_0 \vee p_1 \vee p_2$$

5.2 실험 결과 및 고찰

만약 위와 같은 과정으로 변환된 CNF 식을 SAT 처리기에 입력해서 실행한 결과가 만족(SAT)이라면, 교착상태에 도달할 수 있음을 의미한다. 그림 5의 예제를 BOGOR에서 검사한 결과 시간은 0.32초가 걸렸으며 반례의 길이가 7스텝이 나왔다. 이 예제를 바운디드 모델 검증을 적용하여 수행한 결과 k가 2일 때 에러를 찾았으며, 시간은 0.001초가 걸렸다. 이러한 차이를 보이는 이유는 BOGOR의 경우 기본적으로 깊이 우선 탐색 전

략을 적용한 명시적인 모델 검증 기법을 사용한다. 깊이 우선 탐색은 탐색에 필요한 자원을 적게 사용하는 대신 반례의 최단 경로를 보장하지 않으며, 에러의 위치에 따라 차이는 있지만 상태 공간의 탐색에 시간이 많이 소요된다. 하지만 바운디드 모델 검증은 너비우선 탐색의 방식을 취하므로, 최단 길이의 반례를 보장하며 명시적인 모델 검증 보다 더 나은 모델 검증 성능을 보장한다. 표 1은 식사하는 철학자 문제에서 철학자와 포크의 수를 증가시켜 가면서 실험한 결과이다.

표 1 실험 결과

실 험	BOGOR		바운디드 모델 검증			
	검증 시간	반례 길이	검증 시간	반례 길이	변수 의수	절의 수
철학자 2 / 포크 2	0.32초	7	0.001초	2	155	479
철학자 5 / 포크 5	15초	89	0.003초	5	721	2647
철학자 10 / 포크 10	6분25초	3055	0.014초	10	1841	10762

위 실험은 우분트 리눅스(메모리 640Mbyte, CPU 1.5G)에서 실험을 진행하였으며, SAT 처리기로는 zChaff를 사용하였다.

6. 결론 및 향후 연구

최근 들어서 하드웨어의 논리적 오류를 찾 위해서 고안되었던 모델 검증 기술을 소프트웨어 검증에 적용시키는 연구가 활발하다. 이러한 연구 중 하나가 모델의 검사 범위를 점진적으로 늘려가면서 속성의 만족성 여부를 검사하는 바운디드 모델 검증 기법이다.

본 논문에서는 이러한 바운디드 모델 검증 기법을 BIR 모델에 적용하기 위해 BIR 모델을 CNF식으로 변환하는 방법을 제안하였다. 제안한 방법을 통해 BIR 모델을 CNF로 변환해서 검사한 결과 기존의 BOGOR에서 제공하는 깊이 우선 탐색을 사용한 명시적인 모델 검증 방식보다 더 좋은 성능을 보였고 최단 경로의 반례를 찾을 수 있었다.

하지만 동적인 스레드 생성이나, 복잡한 수식, 재귀 호출에 관해서는 처리하지 못했다. 향후 연구로는 이러한 부분을 CNF 식으로 표현하는 연구가 필요할 것이다.

참 고 문 헌

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

[2] B. Schlich and S. Kowalewski, "Model Checking C Source Code for Embedded Systems," in Proceedings of IEEE/NASA Workshop on Leveraging Applications of Formal Methods, Verification, and Validation, 2005.

[3] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Progress on the State Explosion Problem in Model Checking," Informatics, pp. 176-194, 2001.

[4] S. Graf and H. Saidi, "Construction of Abstract State Graphs with PVS," in Proceedings of CAV, pp. 72-83, 1997.

[5] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-Guided Abstraction Refinement," in Proceedings of CAV, pp. 154-169, 2000.

[6] A. Biere, A. Cimatti, E.M. Clarke, O. Strichman, and Y. Zhu, "Bounded Model Checking," Advances in Computers, Vol.58, pp. 118-149, 2003.

[7] R. Matthew, B. Dwyer, and J. Hatcliff, "Bogor: An Extensible and Highly-Modular Software Model Checking Frame work," in Proceedings of ACM SIGSOFT international symposium on Foundations of software engineering, pp. 267-276, 2003.

[8] J. Corbett, Bandera Intermdiate Representation(BIR) Specification, Version 6.0, Available at the website http://santos.cis.ksu.edu/bandera/birdocs.

[9] A. Biere, A. Cimatti, E.M. Clarke, and Y. Zhu, "Symbolic Model Checking without BDDs," in Proceedings of TACAS, pp. 193-207, 1999.

[10] D. Kroening and O. Strichman, "Efficient Computation of Recurrence Diameters," in Proceedings of VMCAI, pp. 298-309, 2003.

조 민 택

2004년 경기대학교 전자계산학과(학사)
2007년 경기대학교 전자계산학과(석사)
관심분야는 모델검증, 소프트웨어 모델링

이 태 훈

2003년 경기대학교 전자계산학과(학사)
2005년 경기대학교 전자계산학과(석사)
2005년~현재 경기대학교 대학원 전자계산학과 박사과정. 관심분야는 모델검증, 소프트웨어 공학

권 기 현

정보과학회논문지 : 소프트웨어 및 응용
제 34 권 제 7 호 참조