논·문 18-1-?

# 자바 메모리 모델 을 이용한 SAT 기반 멀티 스레드 자바 소프트웨어 검증 [1]

*SAT based Verification for Multithread Java software using Java Memory model*

이태훈 · 권기현
경기대학교 정보과학부
{taehoon,khkwon}@kyonggi.ac.kr

요 약

최신의 컴파일러는 프로그램의 실행 속도를 높이기 위해서 최적화 작업을 수행한다. 최적화 작업중에 프로그램의 구문의 실행 순서가 바뀔수 있다. 단일 스레드 소프트웨어 에서는 구문의 재배치가 결과에 영향을 주지 않지만 멀티 스레드 소프트웨어에서는 구문의 재배치로 인해서 예상하지 못한 결과값이 나올 수 있다. 이런 점은 멀티 스레드 소프트웨어의 오류를 찾기 힘들게 한다. 자바 메모리 모델에서는 구문의 재배치를 고려하여 멀티 스레드 소프트웨어의 가능한 실행 과정을 정형 명세 하였다. 하지만 현재 나와있는 대부분의 멀티스레드 소프트웨어 검증 도구는 메모리모델에 대해서 고려를 하고 있지 못하다. 본 논문에서는 자바 메모리 모델을 이용하여 소프트웨어의 제약 사항 위반 검사 기법을 설명한다. 이를 이용하여 기존 소프트웨어 검증 도구인 JavaPathFinder 에서 오류가 없다고한 소프트웨어의 오류를 찾아내었다.

## 1 서 론

Java 와 C# 같은 고수준언어는 언어수준에서 멀티스레드를 지원하고 있다. 이런 멀티스레드 소프트웨어의 실행 의미는 정확한 동작을 알기 위해서 중요하다. 기본적으로 sequential consistency[1] 로 프로그램의 실행을 설명할수 있지만, 프로그램 최적화를 수행하기에는 너무 강한 제약을 가지고 있다. 대부분의 최적화 기술은 구문의 재배열(reordering) 을 수행한다. 하지만 sequential consistency 의 경우 프로그램 구문의 재배열을 수행할수 없다.

그림 1 에서는 이러한 예제를 보여준다. 그림의 왼쪽에서는 컴파일러 최적화를 수행하기전과 컴파일러 최적화를 수행한후를 살펴볼수 있다. 그림의 예제에선 최적화를 수행하기전에는 프로그램의 실행 결과가 r2==r5==0이고 r4==3 인 결과가 나올수가 없다. 하지만 실제 소프트웨어로 구현하였을 경우에 왼쪽과 같이 최적화가 수행되어서 발생할수 없는 결과가 발생하는 경우가 있다.

이 단점을 보안하기 위해서 많은 연구가 진행되었고 자바 언어에서는 자바 메모리 모델(Java Memory Model)[2]을 통해서 프로그램 최적화를 지원하면서 멀티 스레드 소프트웨

[1]이 논문은 2007년도 정부(과학기술부)의 재원으로 한국과학재단의 지원을 받아 수행된 연구임(R01-2005-000-11120-0)

컴파일러 최적화 수행 전

초기값 : p==q , p.x==0

| Thread 1 | Thread 2 |
|---|---|
| r1=p; | r6=p |
| r2=r1.x | r6.x=3 |
| r3=q | |
| r4=r3.x; | |
| r5=r1.x; | |

컴파일러 최적화 수행 후

초기값 : p==q , p.x==0

| Thread 1 | Thread 2 |
|---|---|
| r1=p; | r6=p |
| r2=r1.x | r6.x=3 |
| r3=q | |
| r4=r3.x; | |
| r5=r2; | |

[그림 1] 프로그램 구문 재배열의 결과

어의 가능한 실행 의미를 정형적으로 명세했다. 자바 메모리 모델은 현재 Java 5.0 표준으로 지정되었다.

현재 자바 소프트웨어를 검증하기 위한 검증도구들이 많이 개발 되고 있다. 하지만 대부분의 도구는 sequential consistency 만을 고려하고 있다. 많이 연구되는 모델 체킹 분야 역시 멀티스레드 소프트웨어의 여러 특성들을 검증할때 sequential consistency 에 대해서만 고려를 하고 있고 다른 메모리 모델에 대해서는 고려를 하지 않고 있다. 기존 자바 검증 도구인 JavaPathFinder[3] 와 같은 경우 assert 위반을 자동적으로 검사하지만 sequential consistency 만을 고려하고 있다. 따라서 JavaMemoryModel상에서 나올수 있는 다양한 구문 재배열 행위를 검사하지 못한다. 그래서 소프트웨어 상에서 오류 상황이 발생하지만 JavaPathFinder 에서는 오류가 없다고 하는 경우가 발생 한다. 본 논문에서는 자바 소프트웨어에 대해서 자바 메모리 모델을 이용한 assert 위반 검증 기법을 소개한다. 그리고 알고리즘을 이용하여 SAT 기반 검증 도구를 개발했다. 검증 도구를 이용해서 JavaPathFinder 에서 assert 위반이 없다고 한 소프트웨어의 오류를 찾아내었다.

본 논문의 구성은 다음과 같다. 2장에서는 Java Memory Model 에 대해서 설명하고 3장에서는 SAT 기반 검증 기법에 대해서 설명한다. 4장에서는 JavaPathFider 와의 비교를 한다. 그리고 5장에서 결론을 맺는다.

# 2 배경 지식

## 2.1 Java Memory Model

자바 메모리 모델은 프로그램과 실행 경로가 주어졌을때 프로그램에서 올바른 실행 경로인지 판단한다.[2] 자바 언어에서 메모리 모델은 변수값의 읽기와 쓰기가 규칙대로 수행되었는지 판단하여 프로그램의 가능한 행위를 설명한다.

만일 하나의 스레드만 동작할경우 주어진 프로그램의 구문순서대로 실행하여도 문제가 발생하지 않는다. 때문에 복잡한 메모리 모델이 필요없다. 하지만 다중 스레드일경우 컴파일러와 하드웨어가 최적화를 수행할 경우 프로그램의 실행이 예상하지 못한 결과가 나올수 있다. 게다가 C 혹은 C++ 과 같은 언어에서는 표준 메모리 모델이 존재하지 않기 때문에 같은 프로그램이 환경이 바뀌었을경우 다른 결과가 나오는 상황이 발생할수 있다. 자바 에서는 JSR133 에서 표준 메모리 모델을 정의 하고 현재 Java 5.0 의 표준으로 제정 하였다.

자바 메모리 모델의 정형 명세는 Action 에 대한 정의와 Execution 에 대한 정의로 구성된다.

Action 은 다음과 같이 구성된다.

$$A =< t, k, v, u >$$

– $t$는 action 수행하는 스레드를 나타낸다.
– $k$는 action 의 종류를 의미한다.
– $v$는 action 에서 포함되는 variable 혹은 monitor 을 의미한다.
– $u$는 action 의 ID를 의미한다.

Execution 은 다음 튜플로 구성된다.

$$E =< P, A, \xrightarrow{po}, \xrightarrow{so}, W, V, \xrightarrow{sw}, \xrightarrow{hb} >$$

– $P$는 프로그램을 의미한다.
– $A$는 action 의 집합을 의미한다.
– $\xrightarrow{po}$ 는 프로그램 순서(Program Order) 을 의미한다.
– $\xrightarrow{so}$ 는 동기화 순서(Synchronization order) 을 의미한다.
– $W$ 는 각각의 변수값 읽기 행위 r 에 대해서 $W(r)$ 은 쓰기 행위를 되돌려준다.
– $V$ 는 각각의 쓰기 행위 w 에 대해서 $V(w)$ 는 값을 읽어오는 읽기 행위를 되돌려준다.
– $\xrightarrow{sw}$ 는 synchronized-with 를 나타낸다.
– $\xrightarrow{hb}$ 는 happen-before 를 나타낸다.

여기서 동기화 행위(Synchronization action)는 lock , unlock, volatile 변수에 대한 읽기와 쓰기 작업이다. 동기화 순서(Synchronization order)는 동기화 행위간의 완전순서(total order)이다. happen-before는 동일한 스레드에서 동작하는 동일한 종류의 행위 x,y 에 대해서 x 가 y 이전에 존재한다면 $x \xrightarrow{hb} y$ 로 표현한다. 또한 동기화 행위들 간의 순서인 synchronized-with 도 happen-before 에 포함된다. 만일 $x$ 가 $y$ 에 의해 동기화(synchronized-with) 되었다면$x \xrightarrow{hb} y$ 로 표현 할 수 있다.

모든 행위 $A$ 가 규칙에 맞게 실행(committed) 되었다면 주어진 Execution 의 집합 $E$ 에 대해서 자바 메모리 모델의 요구사항을 만족한다. 정형 정의는 [2] 에 나와있다. 간단하게 설명하면 실행된 행위의 집합 $C_i$는 행위의 집합 $A_i$에 포함되어야하고 $C_i$의 행위는 $E$ 에 정의된 happen-before 순서와 동기화 순서와 동일한 관계를 가져야 한다. 그리고 모든 읽기 행위는 happen-before 관계 이전에 있는 쓰기행위의 값을 읽어야 한다. 만일 행위 $y$ 가 실행 되었고 외부 행위 $x$ 가 $y$ 이전에 실행되어야 한다면 모든 외부 행위 $x$는 실행된 행위 집합에 포함되어 있다.

# 3 메모리 모델을 이용한 소프트웨어 검증

## 3.1 자바 소프트웨어의 구조

본 논문에서는 자바 소프트웨어에 대한 assert 위반을 메모리 모델의 의미를 이용하여 검사한다. 우선 자바로 작성된 소프트웨어를 받아들여서 동기화 행위를 찾아내고, 동기화 행위들 과 프로그램 구문들간에 happen-before 관계를 설정 한다. 동기화 행위를 기준으로 각각의 클래스에 구문들을 여러개의 그룹으로 분리하고 그룹 간의 실행되는 구문은 happen-before에 위반되지 않을경우 선택될수 있다. 소프트웨어의 실행 과정중에 assert 구문이 위

반되는 경우가 있는지 검사한다. 검사대상인 자바 소프트웨어를 다음과 같은 구조로 변환한다.

$$P =< A_{sync}, A_{Nonsync} Th, C, I, M, \xrightarrow{sw}, \xrightarrow{hb}, Tr, L >$$

- $A_{sync}$ 는 동기화 구문들의 집합이다. $A_{Nonsync}$는 동기화 구문이 아닌 구문들의 집합이다 $A = A_{sync} \cup A_{Nonsync}$ 이다.
- $Th$는 스레드로 선언된 클래스의 집합이다.
- $C$ 는 스레드로 선언되지 않은 클래스의 집합이다.
- $I$ 는 주어진 클래스에 대해 생성되는 인스턴스의 개수를 되돌려 주는 함수이다. 임의의 $c \subseteq Th \cup C$ 에 대해서 $I(c)$ 는 소프트웨어에서 생성 가능한 인스턴스의 개수를 되돌려준다. 모든 소프트웨어에서는 생성되는 인스턴스의 개수가 항상 동일하지 않고 상황에 따라서 다른 개수의 인스턴스가 생성될수 있다. 만일 상황에 따라서 생성되는 인스턴스의 개수가 다를때에는 현재 클래스가 다른 클래스에서 참조되는 횟수를 계산하여 알려준다.
- $M$ 은 주어진 클래스와 메소드 이름에 대해서 속해있는 구문의 그룹을 되돌려주는 함수이다.
- $\xrightarrow{sw}$ 와 $\xrightarrow{hb}$ 는 이전장에서 설명했던 synchronized-with 와 happen-before 이다.
- $Tr$ 은 구문의 전이의 집합을 나타낸다. 여기서 구문에서 구문으로의 전이가 발생하는것이 아니라 구문의 그룹에서 구문의 그룹으로의 전이가 발생한다. 이것을 다음과 같이 표현할수 있다. $Tr \subseteq g_i \times g_j$ 이다.
- $L$은 소프트웨어의 구성요소를 받아들여서 CNF 에 알맞는 이진 변수로 변환하는 함수이다.

자바 소프트웨어에서 각각의 클래스 $C$는 메소드와 변수를 가지고 있다. 이중에 메소드는 구문의 집합$A$에서 정의되고 변수의 값을 가지고 있게 된다. 변수는 크게 두가지 종류로 나뉘어 진다. 하나는 정적 변수와 다른 하나는 일반 변수이다. 자바에서 정적 변수는 인스턴스의 개수와 상관없이 1개만 존재하고 일반 변수는 클래스의 개수만큼 존재한다. 이는 다음과 같이 정의된다.

$$C =< Var^c_{static}, Var^c_{nonStatic}, Active, Th >$$

여기서 $Var^c_{static}$는 클래스 $c$ 의 정적 변수를 의미하고 $Var^c_{nonStatic}$는 클래스 $c$의 동적 변수를 의미한다. $Active$는 현재 클래스의 인스턴스가 생성이 되어있는 상태인지 아닌지를 나타낸다. 따라서 생서잉 되어있는 상태에서만 변수의 값이 변경되고 생성이 되어있지않은 상태에서는 변수의 값이 변경되지 않는다. 만일 클래스 생성 구문이 실행되면 $Active$의 값이 변경된다. 자바에서는 synchronized 와 같은 구문은 모니터로서 구현된다. 이를 검증도구에서는 추가 변수를 두어서 어떤 스레드에서 객체의 모니터를 소유하고 있는지 표현한다. 이를 통해 자바의 synchronized 와 같은 동기화 작업을 지원한다. $Th$에서 어떤 객체에서 모니터를 소유하고 있는지 표현한다.

각각의 스레드 클래스 $Th$ 는 클래스 와 비슷하지만 추가적으로 수행하야하는 구문들의 집합이 추가된다. 이는 다음과 같이 정의된다.

$$Th =< Var^c_{static}, Var^c_{nonStatic}, G^c >$$

$G^c = \{g^c_1, ..., g^c_n\}$ 는 스래스 클래스 $c$에서 실행될수 있는 구문들의 그룹의 집합을 의미한다. 여기서 구문들의 집합의 개수 n 은 $A_{sync}$ 의 개수에 2를 곱한수에 1을 더한수와 같다. $n =\mid A^c_{sync} \mid *2 + 1$ 이다. 여기에 추가로 특수한 그룹으로 생성되기 전 을 나타내는 그룹

과 실행이 종료를 나타내는 그룹을 추가한다. 하나의 그룹 $g_i^c$는 $g_i^c \subseteq A_{sync} \cup A_{Nonsync} \cup \{Active, Null, Terminate\}$ 이다. 소스 코드 에서 volatile 변수의 읽기 와 쓰기 행위, 동기화(synchronized) 메소드 내의모든 구문 monitor 의 획득과 해지구문은 각각의 하나의 구문이 하나의 그룹으로 구성된다. 또한 하나의 그룹에는 동일한 변수에 대한 쓰기 작업이 2개 이상 포함되어 있으면 안된다. 이 제약 규칙을 이용해서 소스 코드를 그룹으로 분류 한다.

그리고 $\xrightarrow{sw}$와 $\xrightarrow{hb}$는 자바 소프트웨어를 분석하여 각각의 관계를 만들어 낸다.$\xrightarrow{hb}$는 monitor 의 unlock 이후의 lock 를 획득하는것과 스레드의 종료 감지 메소드 초기값의 할당 등이다. 이와같은 구문들은 실행순서가 지켜져야 하는 것들이다.

sequential consistency 에서는 소프트웨어의 실행 순서는 소스코드에 명시된 순서대로 진행이 되지만 자바 메모리 모델에서는 happen-before 순서에 위반되지 않을경우 소스코드에 명시되지 않은 순서대로 소프트웨어가 실행될수 있다. 이를 본 논문에서는 하나의 실행 가능한 소프트웨어 구문들의 그룹으로 표현한다. 하나의 스레드 에서 실행 가능한 구문은 스래드 클래스 내부의 구문만 실행할수 있는것이 아니라 다른 클래스의 구문도 역시 실행할 수 있다. 이때문에 스레드 클래스의 그룹에서 포함가능한 구문은 전체 소프트웨어 구문의 부분집합이다.

## 3.2 자바 소프트웨어 검증

자바 소프트웨어에 대해 검증을 수행하기 위해 본 논문에서는 SAT 기반 검증 기법을 이용한다. 범위 모델 체킹[4]의 접근방법을 사용하지만 메모리 모델 검증을 수행하기 위해서 자바 소프트웨어 구조에서 CNF 형식으로의 변환 기법을 제시한다. 우선 본 논문에서 하나의 상태는 다음과 같이 표현된다.

$$s \in Th_1 \times ... \times Th_n \times Var_{static}^{c_1} \times ... \times Var_{static}^{c_j} \times Var_{Nonstatic}^{c_{1_1}} \times ... \times Var_{Nonstatic}^{c_{j_k}} \times G^1 \times ... \times G^j$$

여기서 $n = I(Th)$ 이고 $j = |C|$ 이고 $k = I(c_i)$ 이다. $c_i$는 i 클래스를 의미하고 $c_{i_j}$는 i 클래스의 j 번째 인스턴스를 의미한다.

검증을 수행하기 위해 변환된 CNF 식을 살펴보면 다음과 같다.

$$[\![P]\!]_k = [\![INIT]\!] \wedge [\![TRANS]\!]_k \wedge [\![PROPERTY]\!]_k$$

여기서 $[\![INIT]\!]$는 초기상태를 의미 하는 명제논리 식이다. 변수의 값은 초기값이 지정된경우 초기값을 할당하고 그외의 경우에는 자바 언어에 정의된 초기값을 입력하였다. 그리고 각각의 스레드들은 *main* 을 제외하고 모두 실행중이 아닌상태로 초기값이 할당된다. *main* 스레드는 프로그램의 최초 실행을 표현하는 특수한 스레드 이다. *main* 스레드의 최초 시작위치는 소프트웨어의 시작 구문과 동일하다. 그리고 검증해야할 속성을 나타내는 $[\![PROPERTY]\!]$는 현재 assert 구문에 위한한 상태에서 도달 가능한지만 검사 가능하다. 하지만 기존에 나와있는 알고리즘[4]을 이용하여 LTL 과 같은 시제논리식을 검사하도록 쉽게 확장 가능하다.

$TRANS$ 는 다음과 같은 변환 규칙을 이용하여 변환한다.

$$[\![TRANS]\!]_k = \bigwedge_{i=0}^{k} [\![Hb]\!]_i \wedge [\![Group]\!]_i \wedge [\![Trans]\!]_i \wedge [\![Act]\!]_i \wedge [\![Thread\_selection]\!]_i$$

**Definition 3.2.1** *Happen-before 의 변환*

*모든 Happen-before 관계는 명제 논리식으로 변환되어야한다. 이를 위해 이전에 구문이 실행 되었는지를 나타내는 변수 $e_x$를 추가한다. 이전에 변수 $x$ 가 실행 되었다면 $e_x$가 참*

*의 값을 가지고 실행 되지 않았다면 거짓의 값을 가진다. Happen-before 관계는 다음과 같은 논리식으로 변환된다.*

$$[\![Hb]\!]_i = \forall(y,x) \in \xrightarrow{hb} \cdot L_i(y) \implies L_i(e_x)$$

*여기서 $L_i(y)$와 $L_i(e_x)$는 $y$ 변수에 대해서 $i$ 번째 해당되는 이진 변수로 변환한다. 이 변환규칙을 이용해서 모든 happen-before 관계에 대해서 명제논리식으로 변환을 수행한다.*

**Definition 3.2.2** *구문의 선택*

*$g_i$ 가 현재 진행되는 그룹이라면 $g_i$ 의 하나의 구문이 한번씩만 선택이 되어야 한다. $g_i$에 있는 구문 $\{a_1, ..., a_n\}$ 에 대해서 $n =\mid g_i \mid$ 라고 할때 구문 $a_i$가 실행되었는지 아닌지를 나타내는 변수 $e_i$를 만든다. 현재 실행될 구문은 지금까지 실행되지 않았던 구문중에 하나의 구문만이 실행된다. 실행되는 구문은 $a_i$ 로 표현하고 실행되지 않는 구문은 현재 실행 되지 않는 구문은 $\neg a_i$ 로 표현한다. 이를 다음과 같이 표현할 수 있다.*

$$[\![Group]\!]_i = L_i(g_i) \implies (\bigwedge_{i=0}^{n} L_i(e_i) \implies \neg L_i(a_i)) \wedge (\bigwedge_{j=0}^{n-1} \bigwedge_{k=j}^{n} L_i(a_j) \implies \neg L_i(a_k))$$

**Definition 3.2.3** *그룹의 이동*

*만일 임의의 그룹$g_i$에서 모든 구문의 선택이 되었다면 다음 정의된 그룹으로 이동해야 한다.이를 위해 현재 그룹이 다 수행되었는지를 나타내는 특수한 변수 $g_i_complete$를 이용해서 구문이 끝난것을 표현한다. 만일에 complete 참의 값을 가진다면 그룹이 종료된것이기 때문에 다음 그룹이 실행되어야 한다. 이를 다음과 같이 표현한다.*

$$[\![Trans]\!]_i = L_i(g_i\_complete) \implies (L_i(Tr(g_i)') \wedge \neg L_i(g_i))$$

*여기서 Tr 은 주어진 그룹에 대해 다음 가능한 그룹을 되돌려주는 함수이다. 만일 다음 가능한 그룹이 여러개일경우 여러개의 구문중에 하나만 선택되어야 한다.*

**Definition 3.2.4** *구문에 의한 값의 변화*

*구문의 종류에는 변수값의 할당, 메소드 호출, 분기, 인스턴스 생성, 스레드 시작, 동기화 구문 등이 있다. 프로그램 구문이 실행 되었다면 그에 따라서 해당 하는 변수의 값이 변경된다. 변수 값의 할당 에서는 변수의 값을 단순히 읽어오거나 변수에 값을 쓸수도 있고 읽어온 값에 대해서 사칙연산 혹은 다양한 연산을 수행할수 있다. 메소드 호출은 현재 실행되고 있는 구문의 위치를 변경시킨후 메소드가 종료되었을경우 호출한 메소드 웨치로 변경한다. 인스턴스의 생성은 미리 정의된 인스턴스중에 생성이 안된 상태로 되어있는 인스턴스를 생성이 된 상태로 변경한다. 스레드의 시작은 클래스와 비슷하게 실행이 되어있지 않은 스레드에 대해서 동작하도록 변수의 값을 변경한다. 각각의 해당되는 구문에 따라서 현재 상태의 값을 변경한다.*

**Definition 3.2.5** *실행 스레드의 선택*

*스레드가 여러개일때 어떤 스레드의 구문을 실행할지 결정해야한다. 하나의 스레드가 결정되면 해당 스레드의 현재 그룹에서 하나의 구문을 실행한다. $n$ 개의 스레드가 있을경우 하나의 스레드를 선택하는 구문은 다음과 같다.*

$$[\![Thread\_selection]\!]_i = (\bigvee_{i=0}^{n} L_i(Th_i\_active) \wedge L_i(Th_i)) \wedge (\bigwedge_{j=0}^{n-1} \bigwedge_{k=j}^{n} L_i(Th_j) \implies \neg L_i(Th_k))$$

*여기서 $Th_i_active$는 스레드 $i$가 실행중인지 아닌지를 나타낸다. 그리고 $Th_i$ 가 참이면 소속된 그룹중에 참인 그룹의 구문이 실행 된다. 위의 규칙을 바탕으로 자바 메모리 모델의 정의 대로 자바 프로그램의 가능한 값들의 집합을 구할 수 있다.*

본 장에서 정의된 규칙을 이용해서 자바 메모리 모델의 정의대로 자바 소프트웨어의 진행 순서를 모델링할수 있다.

## 4 사례 연구

3장에서 설명한 변환 규칙을 바탕으로 자바 소프트웨어의 assert 위반을 검증 도구를 개발하였다. 검증도구의 기본 구조는 범위 모델 체킹 [4] 을 기반으로 하고 있다. 검증 도구는 3단계로 소프트웨어를 검증한다. 첫번째 단계는 자바로 구현된 소프트웨어의 소스 코드를 입력받아서 소프트웨어의 정보를 추출 한다. 소프트웨어의 정보에는 동기화 작업, 동기화 순서, 클래스별 객체의 생성 수, 스래드의 개수, 그룹의 생성, 전의 관계 생성, 메소드 호출 정보 생성등을 수행한다. 두번째 단계에서는 각각의 정보를 이진화를 수행하고 CNF 형식으로 변환한다. 세번째 단계에서는 생성된 CNF 화일을 SAT 해결기에 입력하여 만족,불만족 관계를 찾는다. 도구의 파싱과정은 Soot[5]을 이용하여 구현하였고 정보 추출및 변환과정은 자바 언어로 구현 하였다.그리고 SAT 해결기는 minisat 을 이용해서 CNF 의 만족성 검사를 수행하였다. 현재 개발된 도구는 몇가지 가정을 가지고 있다.메소드의 재귀호출은 없다고 가정하고 소스코드가 제공되지 않는 메소드에 대해서 부작용(side effect) 가 존재하지 않는다고 가정한다.

본 논문에서는 그림 1에 있는 예제 모델을 자바 프로그램으로 구현하고 sequential consistency 에서는 발생할수 없는 결과가 발생할수 있는지 assert 구문으로 명세하였다. 예제 프로그램 에서는 r2 변수와 r5 변수가 0 이고 r4 변수가 3 이 나오는 경우가 발생하지 않는다고 assert 구문으로 명세하였다. 구현된 소프트웨어는 총 4개의 클래스로 구성되어 있고 각각 약 40-60 줄 정도의 코드로 구성되어 있다.

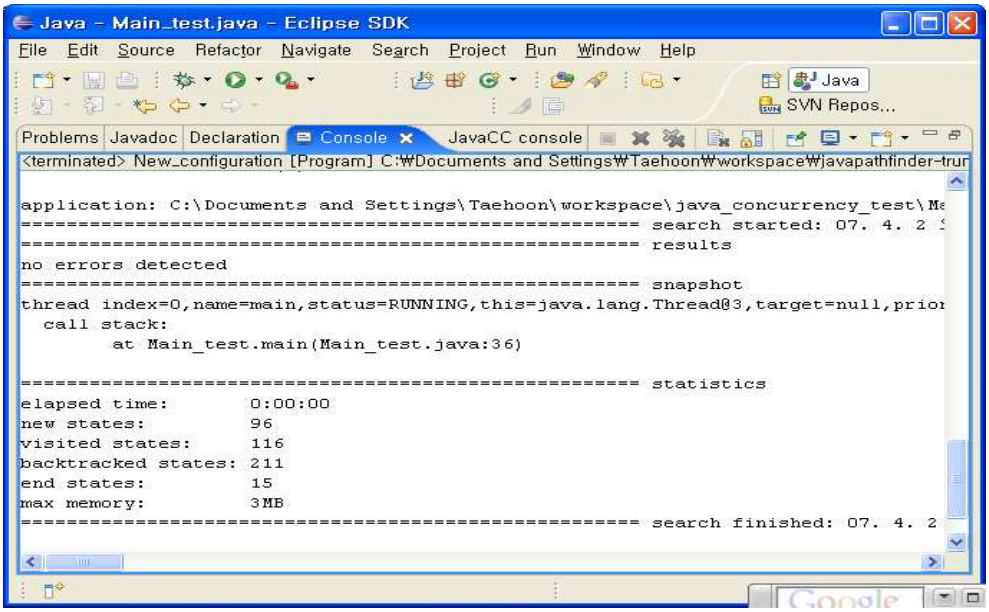
그림 2를 참고하면 이 소프트웨어를 JavaPathFinder를 이용하여 검사한 결과 assert 위반은 발생하지 않는다고 출력하였다.

하지만 실제로 프로그램을 동작 시키면 그림 3과 같이 assert 구문을 위반하는 경우가 가끔씩 발생한다. 그림 3의 출력 결과에서 마지막의 count 는 프로그램의 반복 횟수이다. 맨 첫번째 줄의 22677 이란 숫자의 의미는 그림 1의 작업을 22677 번 수행할경우 r2 의 값이 0 이고 r5 의 값이 0 이고 r4 의 값은 3인 결과가 나올수 있다는 의미다.

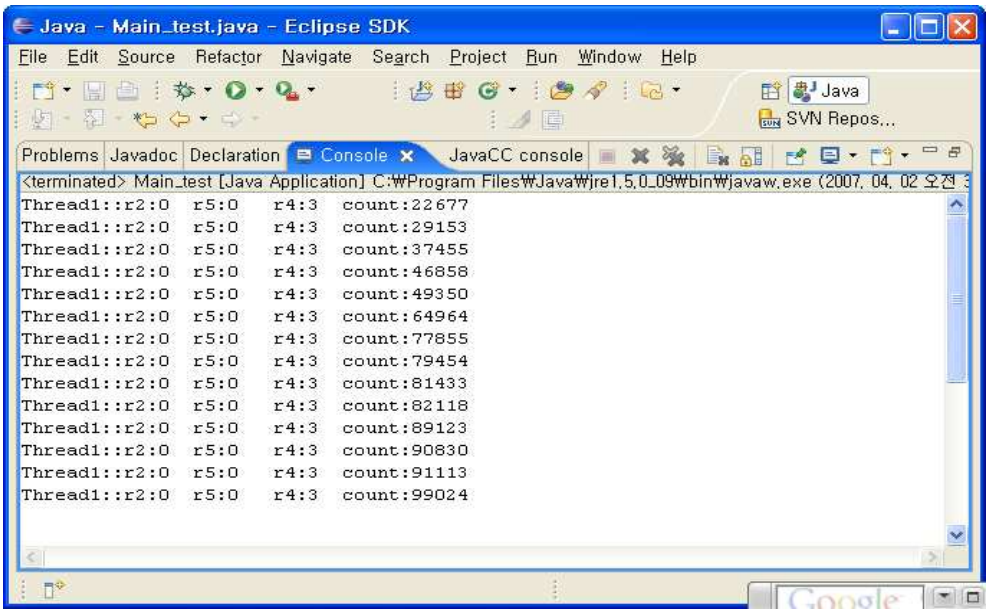
동일한 소프트웨어를 본 논문에서 제안한 방법으로 검증을 수행해 보았다. 범위 모델 체킹을 기반으로 하고 있기 때문에 범위값 k 를 입력해야한다. 사례연구에서는 예제 프로그램의 범위값 k 을 50 으로 했다. 검증 도구를 이용해서 CNF 식을 생성한 결과 약 11M 정도의 CNF 식이 생성되었고 Minisat 을 이용해서 검사를 수행한 결과 약 0.1 초만에 15M 정도의 메모리 사용량으로 assert 구문이 위반되는 경로를 되돌려 주었다.

## 5 결론

현재 많은 소프트웨어에서 멀티스레드가 동작하고 있다.특히 앞으로는 듀얼 코어 와 같은 CPU 의 개발로 멀티 스레드의 중요성이 더욱 커질 것이다. 하지만 멀티스레드는 일반적인 프로그램에 비해 오류 없는 소프트웨어를 개발하기 힘들다. 특히 최적화 작업중에 프로그램의 구문의 실행 순서가 바뀔수 있고 단일 스레드 소프트웨어 에서는 구문의 재배치가 결과에 영향을 주지 않지만 멀티 스레드 소프트웨어에서는 구문의 재배치로 인해서 예상하지 못한 결과값이 나올 수 있다. 이런 점은 멀티 스레드 소프트웨어의 오류를 찾기 힘들




[그림 2] JavaPathFinder 의 검증 결과




[그림 3] assert을 위반 하는 결과

게 한다. 자바 메모리 모델에서는 구문의 재배치를 고려하여 멀티 스레드 소프트웨어의 가능한 실행 과정을 정형 명세 하였다. 하지만 현재 나와있는 대부분의 멀티스레드 소프트웨어 검증 도구는 메모리모델에 대해서 고려를 하고 있지 못하다. 본 논문에서는 자바 메모

리 모델을 이용하여 소프트웨어의 제약 사항 위반 검사 기법을 제안하였고 이를 이용하여 기존 소프트웨어 검증 도구인 JavaPathFinder 에서 오류가 없다고한 소프트웨어의 오류를 찾아내었다.

하지만 현재는 프로그램의 흐름을 표현하기 위해 많은 수의 변수가 필요하고 이는 큰 규모의 시스템 검증에 걸림돌로 작용된다. 따라서 좀더 적은 크기의 CNF 식을 생성하는 프로그램 개발이 필수 적이다. 또한 현재 범위 k 를 입력 해야한다. 하지만 주어진 범위 이후에 오류가 발생할수 있다. 따라서 소프트웨어에 대해서 최적의 범위를 계산하는 방법이 필요하다. 본 논문의 기술을 확장하여 다양한 속성을 검사할수 있도록 하는것이 필요하다. 자바 언어에서 제공하는 다양한 API 들의 사용규칙을 명세하고 검증하는 프로그램 개발이 필요하다. 또한 현재의 기술을 바탕으로 좀더 큰 소프트웨어에 대한 정형 검증을 수행하여 향후 정형 검증이 소프트웨어 개발에 유용하게 사용될수 있도록 하는 것이다.

## 참고문헌

[1] Lamport, L."Hwo to make a multiprocessor computer that correctly executes multiprocess programs", IEEE Transactions on Computers 9, 29, 690-691, 1979.

[2] Jeremy Manson, William Pugh, Sarita V. Adve " The Java Memory Model", In the Proceedings of the POPL 2005 , 2005.

[3] W. Visser, K. Havelund, G. Brat, S. Park and F. Lerda.,"Model Checking Programs", Automated Software Engineering Journal.Volume 10, Number 2, April 2003.

[4] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman and Y. Zhu, "Bounded Model Checking,", Vol. 58 of Advances in Computers, 2003.

[5] Raja Vallee-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon and Phong Co, "Soot - a Java Optimization Framework","Proceedings of CASCON 1999",pp.125–135,1999.

이태훈

– 1997년 - 2003년 경기대학교 전자계산학과(학사)
– 2003년 - 2005년 경기대학교 대학원 전자계산학과(석사)
– 2005년 - 현재 경기대학교 대학원 박사과정

<관심분야> 정형 검증, 모델 체킹, 소프트웨어 공학

권기현

– 1985년 경기대학교 전자계산학과(학사)
– 1987년 중앙대학교 전자계산학과(이학 석사)
– 1991년 중앙대학교 전자계산학과(공학박사)
– 1998년 - 1999년 독일 드레스덴 대학 전자계산학과 방문교수
– 1999년 - 2000년 미국 카네기 멜론 대학 전자계산학과 방문교수
– 2006년 - 2007년 미국 카네기 멜론 대학 전자계산학과 방문교수
– 1991년 - 현재 경기대학교 정보과학부 교수

<관심분야> 소프트웨어 모델링 , 소프트웨어 분석, 정형 기법, 소프트웨어 공학