# 자바 메모리 모델을 이용한 멀티 스레드 자바 코드 검증

이 민[†] · 권 기 현[††]


## 요 약

최신의 컴파일러는 실행 속도를 높이기 위해서 최적화 작업을 수행한다. 그러나 최적화 작업 중에 프로그램 구문의 실행 순서가 바뀔 수 있다. 단일 스레드 소프트웨어 에서는 최적화가 실행 결과에 영향을 주지 않지만 멀티 스레드 소프트웨어에서는 최적화로 인해서 기존의 실행 과정을 계산하는 방법으로는 설명할 수 없는 실행 결과가 발생할 수 있다. 이 문제점을 해결하기 위해서 자바 메모리 모델이 제안되었다. 자바 메모리 모델은 구문의 재배치를 고려하여 멀티 스레드 소프트웨어의 가능한 실행 과정을 명세하고 있다. 현재 자바 메모리 모델은 자바의 표준 메모리 모델로 정의되어 있다. 하지만 대부분의 멀티스레드 소프트웨어 검증 도구는 자바 표준 메모리 모델인 자바 메모리 모델 대신에 순차 일관성메모리 모델만을 고려하고 있다. 순차 일관성 메모리모델에서는 구문의 재배치를 고려하지 않는다. 본 논문에서는 자바 메모리 모델을 이용한 소프트웨어 모델 체킹 기법을 설명한다. 이를 이용하여 기존 소프트웨어 검증 도구인 JavaPathFinder 에서 오류가 없다고 한 소프트웨어의 오류를 찾아내었다.


**키워드 : 자바 메모리 모델, 검증, 멀티스레드 소프트웨어, 만족성 문제**

# Verification for Multithreaded Java Code using Java Memory Model

Min Lee[†] · Gihwon Kwon[††]


## ABSTRACT

Recently developed compilers perform some optimizations in order to speed up the execution time of source program. These optimizations require the reordering of the sequence of program statements. This reordering does not give any problems in a single-threaded program. However, the reordering gives some significant errors in a multi-threaded program. State-of-the-art model checkers such as JavaPathfinder do not consider the reordering resulted in the optimization step in a compiler since they just consider a single memory model. In this paper, we develop a new verification tool to verify Java source program based on Java Memory Model. And our tool is capable of handling the reordering in verifying Java programs. As a result, our tool finds an error in the test program which is not revealed with the traditional model checker JavaPathFinder.


**Key Words : Java Memory Model, Verification, Multi-Threaded Software, Satisfiability Problem**

## 1. 서 론

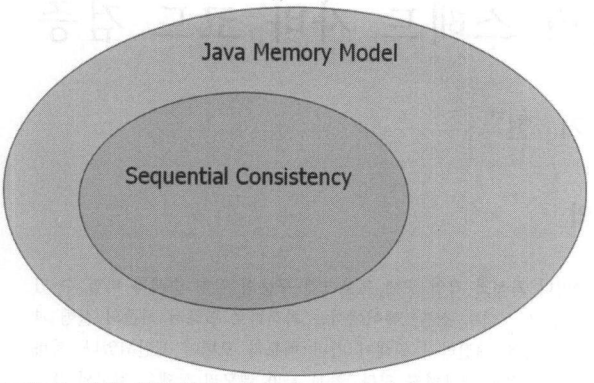
컴퓨터 시스템이 발전함에 따라서 듀얼 CPU 혹은 듀얼 코어와 같은 2개 이상의 CPU 를 보유한 시스템이 널리 보급되고 있다. 기존에 순차적으로 작성된 소프트웨어는 여러 개의 CPU를 사용하지 못하고 하나의 CPU만을 사용하게 된다. 다중 CPU를 가지고 있는 시스템의 효율적인 사용을 위해 병행성을 가진 소프트웨어 개발이 보편화 되고 있다. 소프트웨어의 병행성을 지원하는 대표적인 방법은 멀티 스레드 기법이다. Java와 C# 같은 고수준언어는 언어수준에서 멀티스레드를 지원하고 있다. 멀티 스레드는 효율적이지만 오류가 발생할 가능성도 높다. 특히 멀티 스레드에서 오류가 발생했을 때 재연의 어려움 때문에 테스트를 수행하기도 쉽지 않다. 따라서 멀티스레드 소프트웨어의 실행 의미는 정확한 실행 결과를 알기 위해서 중요하다.

프로그램의 실행 과정과 결과를 설명하기 위해서 순차 일관성(sequential consistency)이라 불리는 메모리 모델이 널리 사용되고 있다[1]. 순차 일관성에서는 멀티 스레드에서 동작하는 소프트웨어가 단일 스레드에서 동작하는 소프트웨어와 동일한 결과를 얻어야한다. 순차 일관성에서 모든 스레드의 작업은 정의된 순서에 따라서 실행되어야하고 각각의 스레드의 작업은 소프트웨어에서 정의된 순서대로 실행되어야 한다.

최신의 프로세서와 가상기계와 컴파일러는 소프트웨어의 실행 속도를 높이기 위해서 최적화를 수행한다. 단일 스레드 소프트웨어에서는 최적화를 수행한 것이 실행 결과에 영향을 미치지 않지만 멀티 스레드 소프트웨어에서는 최적화를 수행한 것 때문에 원래의 경우에서는 발생 할 수 없는

※이 논문은 경기대학교 대학원 연구원 장학생 수혜와 2007년도 정부(과학기술부)의 재원으로 한국과학재단의 지원을 받아 수행된 연구임(R01-2005-000-11120-0).
† 준 회 원 : 경기대학교 전자계산학과 석사과정
†† 종신회원 : 경기대학교 정보과학부 교수
논문접수 : 2007년 11월 7일, 심사완료 : 2007년 12월 7일

DOI: 10.3745/KIPSTD.2008.15-D.1.99




(그림 1) 순차 일관성과 자바 메모리 모델에서 가능한 실행 결과의 관계

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; | |

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 2) 소프트웨어 최적화의 결과

결과가 발생할 수 있다. 순차 일관성에서 정의된 실행 의미는 사람이 이해하고 소프트웨어의 실행 결과 분석에 사용하기 쉽지만 최적화를 허용하지 않기 때문에, 멀티 스레드 소프트웨어에서 소프트웨어의 구문의 재배치와 같은 최적화 작업이 수행되면 순차 일관성에서는 설명하지 못하는 결과가 발생될 수 있다. 이런 문제 때문에 실제 소프트웨어에서 발생할 수 있는 실행 결과를 순차 일관성 메모리 모델에서는 발생할 수 없다고 판단하는 경우가 존재한다.

이와 같은 단점을 보안하기 위해서 많은 연구가 진행되었고 자바 언어에서는 자바 메모리 모델(Java Memory Model)을 통해서 소프트웨어의 모든 최적화를 지원하는 멀티 스레드 소프트웨어의 실행 의미를 정형적으로 명세했다[2]. 자바 메모리 모델은 현재 Java 5.0 표준으로 지정되었다[3]. (그림 1)에서는 순차 일관성과 자바 메모리 모델 간의 관계를 표현 한다. 주어진 소프트웨어에서 순차 일관성 메모리 모델에서 발생 가능한 수행 결과의 집합을 $S$라고 하고, 자바 메모리 모델에서 발생 가능한 수행 결과의 집합을 $JMM$ 이라고 하면, 모든 자바 소프트웨어에서 $S$는 $JMM$의 부분집합 $S \subseteq JMM$이 된다. 따라서 순차 일관성 메모리 모델에서 도달 불가능 했었던 오류가 있을 때 그 오류가 자바 메모리 모델에서는 도달 가능한 오류일 수도 있다. 따라서 소프트웨어의 모든 가능한 결과를 검사하기 위해서는 순차 일관성 메모리 모델 대신에 자바 메모리 모델을 이용해서 검사를 수행해야 한다.

(그림 2)에서는 이러한 예제를 보여준다. 예제에서는 최적화를 수행하기 전과 최적화를 수행한 후를 나타낸다. (그림 2)의 예제에서는 공유 변수 $p$와 $q$가 있다. 초기 값으로 $p$와 $q$는 동일한 객체를 나타내고 있고 $p.x$의 값은 0으로 할당되어 있다. 오른쪽 그림의 스레드 1에서는 총 5개의 지역 변수가 선언되어 있고 스레드 2에서는 1개의 지역 변수가 선언 되어있다. 기존 방법으로 나올 수 있는 변수의 값은 $r4$가 0이 나오고 $r5$가 0 이 나오는 경우와 $r4$가 0이 나오고 $r5$가 3이 나오는 경우와 $r4$가 3이 나오고 $r5$가 3이 나오는 경우만 존재한다. 하지만 최적화를 수행 한 후 (그림 2)의 오른쪽 같은 소프트웨어로 변경되고 최적화를 수행하기 전에는 발생할 수 없었던 $r4$는 3이고 $r5$는 0인 결과가 발생할 수 있다. 이런 결과가 나오는 이유를 순차 일관성으로는 설명 불가능하다.

현재 자바 소프트웨어를 검증하기 위한 도구들이 많이 개발 되었다. 하지만 대부분의 도구는 순차 일관성 메모리 모델만을 지원하고 있다. 모델 체킹을 이용해서 멀티스레드 소프트웨어를 검증하는 경우 순차 일관성만을 고려할 뿐 다른 메모리 모델을 고려하지 않는다. 특히 대표적인 자바 모델 체킹 도구인 JavaPathFinder[5] 의 경우 순차 일관성만을 고려한다. 그 결과 자바 메모리 모델에서 허용하는 다양한 행위를 검사하지 못한다. 이러한 이유로 인해서 실제 오류를 갖는 소프트웨어를 JavaPathFinder로 검증하면 오류가 없다고 판정하는 경우가 발생할 수 있다. 본 논문에서는 자바 소프트웨어에 대해서 자바 메모리 모델을 이용한 정형 검증 기법을 소개한다. 그리고 제안된 기법을 이용하여 SAT 기반 검증 도구를 개발했다. 검증 도구를 이용해서 JavaPathFinder에서 assert 위반이 없다고 한 소프트웨어의 오류를 찾아내었다.

본 논문의 구성은 다음과 같다. 2장에서는 자바 메모리 모델에 대해서 설명하고 3장에서는 SAT 기반 검증 기법에 대해서 설명한다. 4장에서는 기존 소프트웨어 검증 도구인 JavaPathFinder와의 비교를 한다. 그리고 5장에서 결론을 맺는다.

## 2. 배경 지식

### 2.1 자바 메모리 모델

자바 메모리 모델은 소프트웨어와 실행 경로가 주어졌을 때 실행경로가 소프트웨어에서 올바른 가능한 경로인지 판단한다[2]. 자바 언어에서 메모리 모델은 변수 값의 읽기와 쓰기가 규칙대로 수행되었는지 판단하여 소프트웨어의 가능한 행위를 설명한다.

만일 하나의 스레드만 동작할 경우, 주어진 소프트웨어의 구문 순서대로 실행하여도 문제가 발생하지 않는다. 때문에 메모리 모델이 필요 없다. 하지만 다중 스레드일 경우 컴파일러와 하드웨어가 최적화를 수행하면 예상하지 못한 결과가 나올 수 있다. 게다가 C 혹은 C++과 같은 언어에서는 표준 메모리 모델이 존재하지 않기 때문에 같은 소프트웨어 에 따라 다른 결과를 도출할 수 있다. 이런 상황 때문에 자바에서는 JSR133에서 표준 메모리 모델을 정의 하고 현재 Java 5.0의 표준으로 제정 하였다.

이 내용을 요약하면 아래와 같다. 자바 메모리 모델의 정형 명세는 행위(Action)에 대한 정의와 수행 결과(Execution)에 대한 정의로 구성된다.

행위 $A=<t,q,v,u>$는 다음과 같이 구성된다.

- $t$는 행위를 수행하는 스레드를 나타낸다.
- $q$는 행위의 종류를 의미한다. action의 종류에는 읽기와 쓰기작업 메소드 호출 등이 있다
- $v$는 행위에서 포함되는 variable 혹은 monitor 을 의미한다.
- $u$는 행위의 ID를 의미한다.

수행 결과 $E=<P,A,\xrightarrow{po},\xrightarrow{so},W,V,\xrightarrow{sw},\xrightarrow{hb}>$는 다음과 같이 구성된다.

- $P$는 프로그램을 의미한다.
- $A$는 행위 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-with를 나타낸다.
- $\xrightarrow{hb}$는 happen-before를 나타낸다.

여기서 동기화 행위(Synchronization action)는 lock, unlock, volatile 변수에 대한 읽기와 쓰기 작업이다. 동기화 순서(Synchronization order)는 동기화 행위간의 완전 순서(total order)이다. happen-before는 동일한 스레드에서 동작하는 동일한 종류의 행위 x, y에 대해서 x가 y이전에 존재한다면 $x\xrightarrow{hb}y$로 표현한다. 또한 동기화 행위들 간의 순서인 synchronized-with도 happen-before에 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 수행(committed) 되었다면 주어진 수행 결과의 집합 $E$에 대해서 자바 메모리 모델의 요구사항을 만족한다. 정형 의미 정의는 [2]에 나와 있다. 간단하게 설명하면 수행된 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 수행 되었고 외부 행위 $x$가 $y$이전에 수행되어야 한다면 모든 외부 행위 $x$는 수행된 행위 집합에 포함되어 있다. 다음 장에서는 이런 규칙이 어떻게 SAT으로 변하는지를 설명한다.

### 2.2 SAT 기반 정형 검증 기법

대표적인 정형 검증 기법인 모델 체킹은 시스템을 유한 상태 모델로 표현하고 그 시스템이 만족해야하는 속성을 시제논리식으로 표현하여 시스템이 속성에 맞게 동작하는지를 자동으로 검증해 주는 기법이다. 주어진 모델 $M$과 속성 $\phi$를 입력받아서 $M\models\phi$를 만족하는지 자동으로 검사한다. 모든 상태 공간을 검사하기 때문에 찾기 힘든 오류를 찾아낼 수 있다. 하지만 모델에 크기에 따라서 탐색해야하는 상태 공간의 크기는 지수적으로 증가한다. 이러한 문제를 상태폭발 문제라고 한다. 상태폭발 문제를 해결하기 위해 제시된 방법들 중에 하나가 SAT 기반 모델 체킹 이다. SAT 기반 모델 체킹은 $M\models\phi$의 검사를 주어진 범위 $k$에서 반례가 존재함을 보임으로서 검사를 수행한다. 모델 $M$을 명제논리식으로 변환하고 속성 $\phi$의 부정인 $\neg\phi$를 명제논리식으로 변환한 후 SAT 해결기를 이용하여 명제논리식이 만족하는지 만족하지 않는지를 검사한다. SAT 해결기는 CNF 형식의 명제논리식을 입력받아서 만족(Satisfiable) 혹은 불만족(Unsatisfiable)을 되돌려 준다. 만일 SAT 해결기가 변환된 명제논리식을 입력받았을 때 만족(Satisfiable)을 되돌려주면 모델 $M$은 속성 $\phi$를 만족하지 않는다. 만일 SAT 해결기에서 불만족(Unsatisfiable)을 되돌려주면 모델 $M$은 속성 $\phi$를 만족한다. 모델 체킹에서 모델 $M=(S,I,R,L)$은 다음과 같은 튜플로 구성된다[4].

- $S$ 상태들의 집합을 의미 한다
- $I\subseteq S$ 초기 상태를 의미한다.
- $R\subseteq S\times S$ 상태에서 상태로의 전의를 의미한다.
- $L:S\rightarrow 2^{AP}$ 각 상태에서 참이 되는 단순명제들을 해당 상태에 배정하는 함수이다. 여기서 AP는 단순 명제들의 집합이다.

모델 $M$과 속성식 $\phi$ 가 주어졌을 경우 SAT 기반 모델 체킹의 논리식 $[[M]]$은 다음과 같이 구성된다.

$$[[M,\phi]]=I\wedge\bigwedge_{i=0}^{k}R(s_i,s_{i+1})\wedge[[\neg\phi]]$$

여기서 $I$는 모델 $M$의 초기 상태이고, $R$은 $M$의 유한 전이 시스템이다. $\phi$는 검사할 시제 논리식이다. $[[\phi]]$는 시제 논리식 $\phi$를 명제 논리 식으로 변환한 식이다. 변환 방법은 [6]에 나와 있다. $k$는 검사 범위를 나타낸다. SAT 기반 정형 검증 기법을 이용하여 메모리 모델을 이용한 소프트웨어 검증에 적용 하였다.

## 3. 메모리 모델을 이용한 SAT 기반 소프트웨어 검증

### 3.1 자바 소프트웨어의 구조

소프트웨어의 정형 검증을 수행하기 위해서 소프트웨어의 *for*와 *while*과 같은 제어 흐름은 모두 *if-goto* 형식으로 변환한다. *if-goto* 형식으로 변환된 구문에서 이전 구문으로 가는 *goto* 구문은 모두 풀어 헤치는(unwinding) 과정을 거치게 된다. 메소드 호출 구문은 *inline*방법을 이용해서 스레드 관련된 메소드 호출을 제외한 모든 메소드 호출 구문을 제거한다. 이 과정은 SOOT 과서 프레임워크[7]를 이용

하여 작업을 수행한다. 만일에 풀어헤치게 되는 횟수를 알 수 없을 경우 사용자가 입력한 횟수만큼 풀어헤치게 된다. 또한 효율적인 검증을 수행하기 위해서 SOOT을 이용하여 정적 단일 할당 구문(Static Single Assignment)[8]으로 변환 후 검증을 수행한다. 정적 단일 할당 구문은 모든 변수 값의 할당이 한번만 일어나는 형식이다. 모든 함수와 루프에 대해서 풀어헤치기(Unwinding)를 수행하여 검증을 쉽게 수행할 수 있다. 만일 몇 번을 풀어 헤쳐야할지 정확한 횟수를 알 수 없을 경우 임의의 횟수만큼 풀어헤치기를 수행한다. 자바에서 변수를 표현하기 위한 방법으로 메소드 영역 과 힙 영역이 존재한다. 메소드 영역은 풀어 헤치기 때문에 단 하나의 메소드만 존재하는 소프트웨어로 변환 되고 힙 영역의 구분은 없어지게 된다. 이렇게 변환된 자바 소프트웨어를 받아들여서 동기화 행위를 찾아내고, 동기화 행위들 과 소프트웨어의 구문들 간에 happen-before 관계를 설정 한다. 동기화 행위를 기준으로 각각의 클래스에 구문들을 여러 개의 그룹으로 분리하고 그룹 간의 실행되는 구문은 happen-before에 위반되지 않을 경우 선택될 수 있다.

소프트웨어의 실행 과정 중에 assert 구문이 위반되는 경우가 있는지 검사한다. 이러한 과정을 통해서 검사대상인 자바 소프트웨어를 다음과 같은 구조로 변환한다.

$$P=< Class, Th.L_C, I, \xrightarrow{order}, Var, Statement, O >$$

- $Class$는 스레드로 선언되지 않은 클래스의 집합이다. 소프트웨어에 $j$개의 클래스가 존재한다면 이는 다음과 같이 정의된다.

$$Class = \{c_1,...,c_j\}$$

- $Th$는 스레드로 선언된 클래스의 집합이다. 만일 소프트웨어에 스레드로 선언된 클래스가 $n$개 있다고 하면 다음과 같이 정의 할 수 있다.

$$Th = \{\mathrm{th}_1,...,\mathrm{th}_n\}$$

- $L_C$는 각각의 클래스에 해당되는 값을 되돌려주는 매핑 함수 이다.

$$L_C : C \cup Th \Rightarrow 2^{Var_{static}} \times 2^{Var^1_{nonStatic}} \times ... \times 2^{Var^{|C|}_{nonStatic}} \times Active \times Th$$

변수는 크게 두 가지 종류로 나눠진다. 하나는 정적 변수와 다른 하나는 일반 변수이다. 자바에서 정적 변수는 인스턴스의 개수와 상관없이 1개만 존재하고 일반 변수는 클래스의 개수만큼 존재한다.

여기서 $Var_{static}$는 정적 변수들의 집합을 의미하고 $Var_{nonStatic}$는 동적 변수들의 집합을 의미한다. $Active$는 현재 클래스의 인스턴스가 생성이 되어있는 상태인지 아닌지를 나타낸다. $Active \in \mathrm{active}, nil$ 이다. 따라서 생성 되어있는 상태에서만 변수의 값이 변경되고 생성이 되어있지 않은 상태에서는 변수의 값이 변경되지 않는다. 만일 클래스 생성 구문이 실행되면 $Active$의 값이 변경된다. 자바에서는 synchronized와 같은 구문은 모니터로서 구현된다. 이를 검증 도구에서는 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.

- $I \in L_C(Th_1) \times ... \times L_C(Th_n) \times L_C(c_1) \times ... \times L_C(c_j)$는 소프트웨어의 초기 상태를 되돌려 준다.
- $\xrightarrow{order}$는 자바 소프트웨어를 분석하여 각각의 관계를 만들어 낸다. $\xrightarrow{order}$는 monitor의 unlock 이후의 lock를 획득하는 것과 스레드의 종료, 메소드 초기값의 할당 등이다. 이와 같은 구문들은 실행순서가 지켜져야 하는 것들이다.
- $Var$ - 은 변수와 클래스의 매핑 함수 이다. 각각의 클래스의 인스턴스 $c$에 대해서, $Var(c)$는 변수들의 집합을 되돌려 준다.
- $Statement$ - 구문들의 집합이다.
- $O$ - 클래스에 대해서 생성되는 인스턴스의 개수를 되돌려주는 함수이다.

순차 일관성에서는 소프트웨어의 실행 순서는 소스코드에 명시된 순서대로 진행이 되지만 자바 메모리 모델에서는 happen-before 순서에 위반되지 않을 경우 소스코드에 명시되지 않은 순서대로 소프트웨어가 실행될 수 있다. 이를 본 논문에서는 하나의 실행 가능한 소프트웨어 구문들의 그룹으로 표현한다. 하나의 스레드 에서 실행 가능한 구문은 스레드 클래스 내부의 구문만 실행할 수 있는 것이 아니라 다른 클래스의 구문도 역시 실행할 수 있다. 이 때문에 스레드 클래스의 그룹에서 포함 가능한 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[6]의 접근방법을 사용한다. 이번 절에서는 메모리을 이용한 소프트웨어 검증을 수행하기 위해서 자바 소프트웨어 구조에서 CNF 형식으로의 변환 기법을 제안한다. CNF로 변경된 식을 SAT 해결기에 입력하여 만족이 나오게 되면 오류가 존재하는 것이고 불만족이 나오게 되면 오류가 존재하지 않는 것이다. 하나의 상태는 다음과 같이 표현된다.

$$s \in L_{Th}(Th_1) \times ... \times L_{Th}(Th_n) \times L_C(c_{1_1}) \times ... \times L_C(c_{i_j}) \times Th$$

여기서 $n=|Th|$이고 $j=|C|$이고 $l=O(c_j)$이다. $c_i$는 I 클래스를 의미하고 $c_{j_j}$는 $i$ 클래스의 $j$번째 인스턴스를 의미한다. 미리 계산된 스레드의 생성 개수만큼의 소프트웨어의 실행 구문과 클래스별 인스턴스의 생성 개수만큼의 변수를

가지게 된다. 그리고 현재 어떤 스레드가 실행되는지에 대한 정보를 하나의 상태에서 가지고 있다.

검증을 수행하기 위해 변환된 CNF 식을 살펴보면 다음과 같다.

$$[[E]]_k = [[INIT]] \wedge [[TRANS]]_k \wedge [[PROPERTY]]_k$$

여기서 $[[INIT]]_k$는 초기상태를 의미 하는 명제논리 식이다. 변수의 값은 초기 값이 지정된 경우 초기 값을 할당하고 그 외의 경우에는 자바 언어에 정의된 초기 값을 입력하였다. 그리고 각각의 스레드들은 $main$을 제외하고 모두 실행중이 아닌 상태로 초기 값이 할당된다. 여기서 $k$는 구문의 개수이다.

$main$ 스레드는 소프트웨어의 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 나와 있는 알고리즘[6]을 이용하여 LTL과 같은 시제논리식을 검사하도록 쉽게 확장 가능하다.

$TRANS$는 다음과 같은 변환 규칙을 이용하여 변환한다.

$$[[TRANS]]_k = \bigwedge_{i=0}^{k} [[Ordering]]_i \wedge [[Selection]]_i \wedge [[Act]]_i$$

우선 $\xrightarrow{order}$ 의 변환을 수행한다. 모든 $\xrightarrow{order}$ 관계는 명제 논리식으로 변환되어야한다. $\xrightarrow{order}$관계는 다음과 같은 논리식으로 변환된다.

$$[[Ordering]]_i = \forall (x,y) \in \xrightarrow{order} \cdot \bigvee_{j=1}^{|Th|} statement^i_{yj} \rightarrow executed_{xj}$$

여기서 $statement^i_{yj}$는 구문 $statement_y$가 스레드 $j$에 의해 $i$번째에 실행이 되었다는 것을 나타내는 이진 변수이다. $executed_{xj}$는 구문 $statement_x$가 스레드 $j$에 의해 이전에 실행이 되었다면 참을 되돌려주고 아니라면 거짓의 값을 가지게 된다. 이 변환규칙을 이용해서 모든 $\xrightarrow{order}$ 관계에 대해서 명제논리식으로 변환을 수행한다.

검증을 수행하기 위해서 실행되는 구문은 하나의 구문만 선택되어 실행된다. 하나의 구문만 실행되는 것을 아래와 같이 표현한다.

$$[[Selection]]_i = \bigwedge_{y=0}^{|Statement||Statement|} \bigwedge_{k=j} \bigvee_{j=1}^{|Th|} statement^i_{yj} \rightarrow \neg statement^i_{kj}$$

$statement^p_l$는 어떤 스레드 이든 상관없이 구문 $statement_l$가 $p$번째 실행되었다는 의미 이다. 이것은 아래와 같이 정의 된다.

구문의 종류에는 변수 값의 할당, 메소드 호출, 분기, 인스턴스 생성, 스레드 시작, 동기화 구문 등이 있다. 소프트웨어의 구문이 실행되었다면 그에 따라서 해당 하는 변수의 값이 변경된다. 변수 값의 할당 에서는 변수의 값을 단순히 읽어오거나 변수에 값을 쓸 수도 있고 읽어온 값에 대해서 사칙연산 혹은 다양한 연산을 수행할 수 있다. 메소드 호출은 현재 실행되고 있는 구문의 위치를 변경시킨 후 메소드가 종료되었을 경우 호출한 메소드 위치로 변경한다. 인스턴스의 생성은 미리 정의된 인스턴스중에 생성이 안 된 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.

$$[[Act]]_i = [[CICE]]_i \wedge [[Assign]]_i$$

행위에 관련된 구문은 크게 클래스 인스턴스 생성 구문 $[[CICE]]_i$과 변수의 값을 할당하는 구문 $[[Assign]]_i$, 소프트웨어의 동기화 행위 $[[Sync-action]]_i$로 구성된다. 클래스 생성 구문 $[[CICE]]_i$는 아래와 같이 정의된다.

$$[[CICE]]_i = \forall s_j \in \mathrm{Proj}_I(Statement) \cdot \bigvee_{k=0}^{|Thread|} statement^i_{jk} \Rightarrow \bigvee_{l=0}^{O(S(s_j))} (\neg c^i_{jl} \wedge c^{i+1}_{jl} \wedge ( \bigwedge_{p=0}^{O(S(s_j))} (p \neq l) \rightarrow \neg c^i_{jp} \rightarrow \neg c^{i+1}_{jp}))$$

여기서 $\mathrm{Proj}_I(Statement)$는 구문들의 집합에서 클래스 생성 구문만을 되돌려준다. $S(statement)$ 클래스 생성 구문에서 대상이 되는 클래스를 되돌려 준다. 기본적으로 클래스의 인스턴스는 생성이 안 되어 있는 상태로 되어있다. 하지만 $new$ 키워드를 만났을 경우 클래스의 인스턴스는 생성이 된 상태로 변경되게 된다. 여기서 $c_{jl}$은 인스턴스 $c_j$의 $l$번째 인스턴스가 생성된 상태라는 것을 의미한다.

$$[[Assign]]_i = (\bigvee_{k=0}^{|Thread|} statement^i_{jk}) \Rightarrow executed^i_j$$

하나의 구문 $statement_j$이 스레드 $k$에 의해서 $i$번째에 실행되면 그 구문이 실행 되었다는 것을 나타내는 보조 변수 $executed^i_j$가 참이된다. $[[Assign]]$에서는 어떠한 구문이 실행이 되었는지를 나타낸다.

본 장에서 정의된 규칙을 이용해서 자바 메모리 모델의 정의대로 자바 소프트웨어의 진행 순서를 모델링할 수 있다. 검사해야하는 속성의 CNF 식은 assert 구문 $a_{assert}$과 조건 $Guard$를 이용해서 정의 할 수 있다.

$$[[PROPERTY]]_k = \bigvee_{j=0}^{k} statement^j_{assert} \wedge \neg Guard(statement^j_{asert})$$

assert 위반이 발생한다는 것은 assert 구문에 도달 가능하고 $Guard$ 에서 명세된 조건을 만족하지 못한다는 것이다. 위의 규칙대로 생성된 CNF 식을 SAT 해결기에 입력하

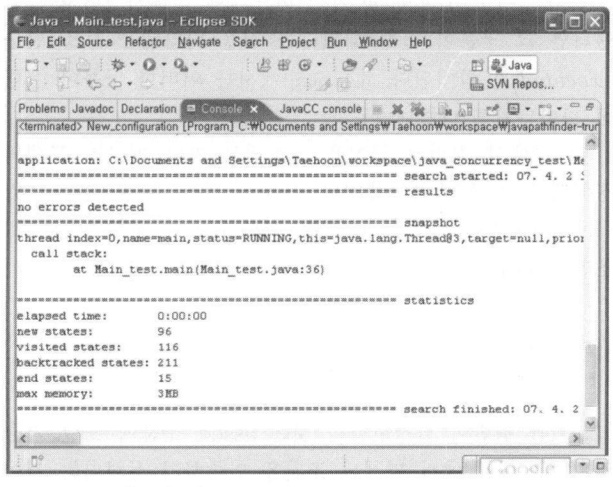
였을 때 만족인 결과로 나오게 된다. 여기서 만족하는 인스턴스는 위반하는 경로를 반례로서 보여준다. 만일 assert 위반이 k 범위 이내에서 발생하지 않는다면 SAT 해결기의 결과는 불만족(Unsatisfiable)이 나오게 된다. 이를 통해 멀티 스레드 소프트웨어에 대한 assert 위반 검사를 수행할 수 있다.

## 4. 사례 연구

3장에서 설명한 변환 규칙을 바탕으로 자바 소프트웨어의 assert 위반을 검사하는 검증 도구를 개발하였다. 검증도구의 기본 구조는 범위 모델 체킹[6]을 기반으로 하고 있다. 검증 도구는 3단계로 소프트웨어를 검증한다. 첫 번째 단계는 자바로 구현된 소프트웨어의 소스 코드를 입력받아서 소프트웨어의 정보를 추출 한다. 소프트웨어의 정보에는 동기화 작업, 동기화 순서, 클래스별 객체의 생성 수, 스레드의 개수, 그룹의 생성, 전의 관계 생성, 메소드 호출 정보 생성 등을 수행한다. 두 번째 단계에서는 각각의 정보의 이진화를 수행하고 CNF 형식으로 변환한다. 세 번째 단계에서는 생성된 CNF 화일을 SAT 해결기에 입력하여 만족, 불만족 관계를 찾는다. 도구의 파싱과정은 Soot을 이용하여 구현하였고 정보 추출및 변환과정은 자바 언어로 구현 하였다. 그리고 SAT 해결기는 Minisat[9]을 이용해서 CNF 의 만족성 검사를 수행하였다. 현재 개발된 도구는 몇 가지 가정을 하고 있다. 메소드의 재귀 호출은 없다고 가정하고 소스코드가 제공되지 않는 메소드에 대해서 부작용(side effect) 이 존재하지 않는다고 가정한다.

본 논문에서는 (그림 2)에 있는 예제 모델을 자바 소프트웨어로 구현하고 변수 $r4$가 3이 나오면서 변수 $r5$가 0이 나오는 경우가 발생하면 안 된다고 assert 구문으로 명세하였다. (그림 2)을 구현한 소프트웨어는 총 4개의 클래스로 구성되어 있고 각각의 클래스는 약 20-60 줄 정도의 코드로 구성되어 있다.

(그림 2)를 구현한 소프트웨어를 JavaPathFinder를 이용하여 검사한 결과 assert 위반은 발생하지 않는다고 출력하였다.

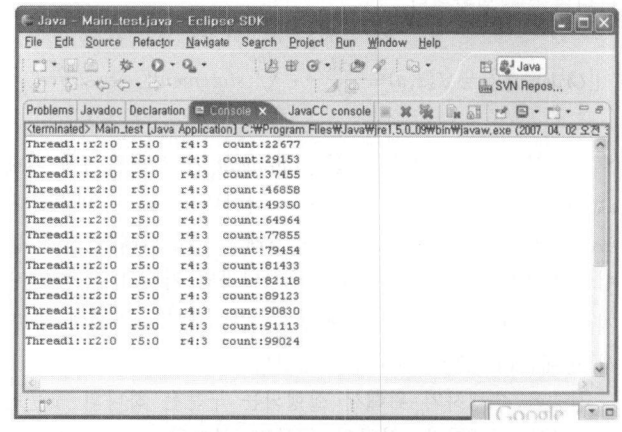



(그림 2) JavaPathFinder 의 검증 결과

하지만 프로그램을 동작 시키면 (그림 4)와 같이 assert 구문을 위반하는 경우가 가끔씩 발생한다. (그림 1)을 구현한 소프트웨어를 여러번 수행할 경우 변수 $r4$의 값이 3 이고 $r5$의 값이 0인 결과가 나올 수 있다는 의미다. 여기서 몇가지 알아두어야 할 사항은 이런 값으로 변경되는 경우가 가끔 발생된다는 것이다. 또한 단일 CPU를 가진 시스템에서는 이런 경우가 1시간 동안 동작시켜도 발생하지 않았다. 하지만 2개 이상의 CPU를 가진 시스템에서는 상황에 따라서 다르지만 가끔씩 발생하게 된다. 이런 점들은 찾기 힘들 소프트웨어 오류로 분류 될 수 있다.

동일한 소프트웨어를 본 논문에서 제안한 방법으로 검증을 수행해 보았다. 범위 모델 체킹을 기반으로 하고 있기 때문에 범위값 $k$를 입력해야한다. 본 논문에서는 범위값 $k$의 경우 구문의 개수이다. 검증 도구를 이용해서 CNF 식을 생성한 결과 약 11M 정도의 CNF 식이 생성되었고 Minisat 을 이용해서 검사를 수행한 결과 약 0.1초 만에 15M 정도의 메모리 사용량으로 assert 구문이 위반되는 경로를 되돌려 주었다.

(그림 5)에서는 본 논문의 아이디어를 구현한 도구를 사용하여 몇 가지 소프트웨어에 적용을 하였다. 여기에서 변수의 개수는 생성된 CNF 식의 변수의 개수를 의미하고 절의 개수는 생성된 CNF 식의 절의 개수를 의미한다. 첫 번째 JMM_test는 (그림 2)에 있는 예제 모델을 구현한 것이다. 첫 번째 예제에서는 총 24100개의 변수가 생성되었고 949003개의 절이 생성되었다. 두 번째 예제는 많이 알려진 식사하는 철학자 예제이다. 총 6명의 철학자가 있다고 설정하고 검증을 수행하였다. 프로그램의 크기는 이전 예제에 비해서 작지만 많은 스




(그림 3) Assert을 위반하는 결과

| 프로그램 | 변수의 개수 | 절의 개수 | 코드 길이 | 클래스 개수 | 검증 수행 시간 |
|---|---|---|---|---|---|
| JMM_test | 24100 | 949003 | 117 | 4 | 0.1 sec |
| DiningPhil 6 | 68500 | 5377923 | 56 | 4 | 0.7 sec |
| Crossing | 53250 | 3376422 | 277 | 3 | 0.3 sec |
| DEOS | ∞ | ∞ | 1794 | 21 | ∞ |

(그림 4) 사례 연구

레드가 동시에 동작하기 때문에 변수와 절이 필요하다. 세 번째 예제는 JavaPathFinder에서 예제로 사용된 프로그램이다. 이 예제를 검증 도구를 이용하여 CNF 형식으로 변환할 경우 53250개의 변수가 생성되고 3376422개의 절이 생성된다. 절의 개수는 무척 크지만 SAT 해결기에서는 쉽게 해결 가능하다. 모든 예제를 약 1초 이내에 초기상태에서 오류로 도달하는 경로를 알려준다. 하지만 마지막 예제인 DEOS 예제의 경우 검증을 수행할 수 없었다. 생성되는 명제논리식이 너무 크기 때문에 SAT 해결기에서 처리할 수 없었다. DEOS 와 같은 문제나 더 큰 문제를 다루기 위해서 효율적인 변환방법이 필요하다.

## 5. 결 론

현재 많은 소프트웨어에서 멀티스레드가 동작하고 있다. 특히 앞으로는 듀얼 코어와 같은 다중 CPU의 확산으로 멀티 스레드의 중요성이 더욱 커질 것이다. 멀티 스레드 소프트웨어에서는 오류가 발생할 경우 오류를 다시 재현하기 힘들기 때문에 단일 스레드 소프트웨어에 비해 오류없는 소프트웨어를 개발하기 힘들다. 이를 위해서 멀티 스레드 소프트웨어의 안전성을 검사하는 검증 도구가 필요하다.

단일 스레드 소프트웨어 에서는 소프트웨어의 최적화가 실행 결과에 영향을 주지 않는다. 하지만 멀티 스레드 소프트웨어에서는 소프트웨어 최적화로 인해서 단일 스레드 소프트웨어에서 동작 할 때와는 다른 결과가 나올 수 있다. 이를 해결하기 위해 자바 언어에서는 자바 메모리 모델을 표준 메모리 모델로 지정하였다. 자바 메모리 모델에서는 구문의 재배치를 고려하여 멀티 스레드 소프트웨어의 가능한 실행 과정을 정형 명세 한다. 하지만 현재 나와 있는 대부분의 멀티스레드 자바 소프트웨어 검증 도구는 자바 메모리모델에 대해서 고려를 하지 못한다. 본 논문에서는 자바 메모리 모델을 이용하여 소프트웨어의 제약 사항 위반 검사 기법을 제안하였고 이를 이용하여 최신의 자바 소프트웨어 검증 도구인 JavaPathFinder 에서 찾지 못한 멀티 스레드 소프트웨어의 오류를 찾았다.

하지만 현재는 프로그램의 흐름을 표현하기 위해 많은 수의 변수가 필요하고 이는 큰 규모의 시스템 검증에 걸림돌로 작용된다. 본 도구를 이용하여 현재까지는 수백 라인 정도의 소프트웨어 검증에 성공적으로 적용이 가능하지만 좀 더 큰 시스템의 검증은 불가능하다. 따라서 좀 더 적은 크기의 CNF 식을 생성하는 프로그램 개발이 필수적이다. 이를 위해 추상화 기술을 적용하여 검증해야 하는 소프트웨어의 크기를 줄이는 기술을 향후 적용해야 한다. 또한 현재의 변환 기술을 효율적으로 구현하여 같은 의미를 가지는 적은 크기의 CNF 식을 생성하는 방법을 개발해야 한다. 프로그램 순서와 같은 부분을 계산하기 위한 효율적인 기술도 필요하다. 본 논문의 기술을 확장하여 다양한 속성을 검사할 수 있도록 하는 것도 필요하다. 자바 언어에서 제공하는 다양한 API 들의 사용규칙을 명세하고 검증하는 프로그램 개발이 필요하다. 현재의 기술을 바탕으로 대형 소프트웨어에 대한 정형 검증을 수행하여 향후 정형 검증이 소프트웨어 개발에 유용하게 사용될 수 있도록 해야 한다. 특히 멀티 스레드 소프트웨어가 많이 사용되는 네트워크 응용 프로그램과 게임 응용 프로그램 등의 실제 사용되는 소프트웨어에 직접적으로 검증 기술을 적용하는 방법에 대한 연구가 필요하다.

## 참 고 문 헌


[1] Lamport, L. "How to make a multiprocessor computer that correctly executes multiprocess programs," IEEE Transactions on Computers 9, No.29, pp.690-691, 1979.

[2] Jeremy Manson, William Pugh, Sarita V. Adve " The Java Memory Model," In the Proceedings of the POPL 2005 , pp.378-391, 2005.

[3] J. Gosling, B. Joy, G. Steele and G. Bracha, "The Java Language Specification Third Edition," Addison-Wesley, 2005.

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

[5] W. Visser, K. Havelund, G. Brat, S. Park and F. Lerda., "Model Checking Programs," Automated Software Engineering Journal.Vol.10, No.2, April, 2003.

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

[7] 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.

[8] Muchnick, "Advanced Compiler Design & Implementation," Morgan Kaufman, 2005.

[9] Niklas Een and Niklas Sorensson. "MiniSat - a SAT solver with conflict-clause minimization". In SAT 2005, 2005. 8.



이 민

e-mail : leemin@kyonggi.ac.kr

2006년 경기대학교 수학과(학사)

2006년~현 재 경기대학교 전자계산학과 석사과정

관심분야 : 모델 검증, 소프트웨어 공학, 만족성 문제 등


## 권 기 현

e-mail : khkwon@kyonggi.ac.kr


1985년 경기대학교 전산학과(학사)
1987년 중앙대학교 전산학과(이학석사)
1991년 중앙대학교 전산학과(공학박사)
1998년~1999년 독일 드레스덴대학
전자계산학과 방문교수
1999년~2000년 미국 카네기멜론대학 전자계산학과 방문교수
2006년~2007년 미국 카네기멜론대학 전자계산학과 방문교수
1991년~현 재 경기대학교 정보과학부 교수
관심분야 : 소프트웨어 모델링, 소프트웨어 분석, 정형 기법 등