자바 언어를 위한 예외 상황 분석 비교 연구

Comparative Study on Exception Analyses for Java

조 장 우
Jang-Wu Jo
컴퓨터전자공학부
부산외국어대학교
jjw@taejo.pufs.ac.kr

창 병 모
Byeong-Mo Chang
정보과학부
숙명여자대학교
chang@cs.sookmyung.ac.kr

요 약

JDK 자바 컴파일러의 예외 상황 분석은 프로그래머의 선언에 의존하는 프로시저-내 분석이다. 이 분석은 프로그래머가 실제 발생하지 않는 예외들이나 실제 발생하는 예외보다 광범위한 예외들을 선언하는 경우 적절한 예외 처리를 하지 못하는 문제가 발생한다. 이러한 문제의 해결을 위해 [2,6]에서 프로그래머의 선언과 무관한 프로시저-간 예외 상황 분석기가 제시되었다. 본 논문에서는 이 두 분석 방법을 설계 구현하고 실험을 통해서 비교한다.

1. 서론

견고한(robust) 소프트웨어 개발을 용이하게 하기 위해서 현대의 대부분의 프로그래밍 언어에서는 명시적인 예외 처리 메커니즘을 지원한다. 인터넷 프로그래밍 언어로서 세계적으로 많이 사용되는 자바 언어에서도 예외처리 메커니즘을 지원하고 있다. 자바에서 예외는 Exception 클래스와 이의 하위 클래스들로 제공되며 또한 프로그래머가 Exception 클래스의 하위클래스로 새로운 예외 클래스를 정의할 수 있다[1].예외에 관련된 자바 구문으로는 예외를 발생시키는 **throw** 문장이 있고 발생된 예외를 처리하는 **try-catch** 문장이 있다. 그리고 메소드 선언 부분에서 처리되지 않는 예외를 선언하는 **throws** 구문이 있다.

예외 메커니즘을 이용하여 프로그램의 안전에 허점이 발생하지 않기 위해서는, 프로그래머가 발생 가능한 예외 상황에 대한 적절한 예외 처리기를 설치해야 한다. 그렇지 못한 경우 컴파일러가 미리 분석해서 프로그램의 안전도를 증진시키는 것이 필요하다. JDK 자바 컴파일러는 프로그래머가 각 메소드에 대해 선언한 예외 상황 명세를 기반으로 프로시저-내 예외 상황 분석을 한다. 그러나 이 분석은 프로그래머가 실행 시 발생 가능한 예외보다 광범위한 예외를 선언하거나 실제 발생하지 않는 예외들을 선언하는 경우 정확한 분석을 할 수 없다는 문제점을 가지고 있다. 이러한 문제점을 개선하기 위해서 [2,6]에서는 프로그래머의 선언과 무관한 프로시저-간(interprocedural) 예외 상황 분석을 제시하였다. 특히 이 분석은 선언을 이용하지 않고 예외 상황을 분석함으로써 보다 정확한 처리되지 않는 예외 상황 정보를 줄 수 있을 뿐만 아니라 프로그래머가 선언한 예외 상황 명세를 보다 정확히 검증할 수 있다.

본 연구에서는 자바 예외상황 분석을 위한

* 본 연구는 한국과학재단 목적기초연구(2000-1-30300-009-2) 지원으로 수행되었음

위의 두 방법을 실험적으로 비교함을 목표로 한다. 이를 위해서 프로그래머의 선언에 의존하는 프로시저-내 예외 상황 분석 및 선언과 무관한 프로시저-간(interprocedural) 예외 상황 분석을 설계 구현하였으며 실험을 통해서 실제 프로그램에 적용시켜 봄으로써 두 분석 방법을 비교하였다.

본 연구에서는 Heintze가 [4]에서 제안한 집합-기반 분석(set-based analysis) 기법을 기반으로 분석기를 설계하였으며 구현에 있어서 [2,5,6]에서 제시된 것처럼 모든 식에 대해서 집합-변수를 만드는 대신 try-catch와 메소드 같은 예외에 관련된 구문에 대해서만 집합-변수를 정의함으로써 집합-변수의 수를 줄이고 이를 통해 분석 속도를 향상시켰다. 이렇게 구현된 분석기는 속도 면에서 보다 효율적이면서 각 메소드에 대해서 수식 수준에서 설계된 분석기와 동등한 예외 상황 정보를 제공한다.

본 논문의 구성은 다음과 같다. 2절에서는 연구배경 및 동기를 설명하고 3절에서는 예외 상황 분석기 설계에 대해서 기술한다. 4절에서는 분석 속도 향상을 위한 방법에 대해서 설명한다. 5절에서는 실험을 통해서 두 분석 방법을 비교하고 6절에서 결론을 맺는다.

2. 연구 배경

프로그램 실행 중에 처리되지 않는 예외가 발생하면 프로그램의 실행이 비정상적으로 종료하므로 컴파일 과정에서 처리되지 않는 예외를 검증하는 것은 매우 중요하다. 현재 JDK의 자바 컴파일러에서도 이와 같은 예외 상황 분석을 수행하지만 자바 컴파일러는 프로그래머가 선언한 throws 구문에 의존하는 프로시저-내(intraprocedural) 분석을 한다. 자바는 메소드 내에서 처리되지 않는 예외들을 메소드 정의 시 명세처럼 선언하도록 하고 있다. 이러한 조건은 프로그래머에 부담이 되어 필요 이상으로 광범위한 선언을 하거나 실제 발생하지 않는 예외들을 선언할 수 있다. 따라서 프로그래머가 불필요한 예외들을 선언하거나 광범위하게 선언하는 경우 처리되지 않는 예외에 대한 정확한 분석 결과를 내지 못하며 이로 인해 정확한 예외 처리를 하지 못하는 문제점이 있다. 이와 같은 상황의 간단한 예가 그림 1의 프로그램이다.

```
class demo1{
  void demoProc1 ( ) throws Exception {
       try {
         demoProc2();
       } catch (Exception e) { ; }
  }

  void demoProc2 ( ) throws Exception {
       throw new IOException();
  }
}
```

그림 1 프로그램 예

[그림1]의 메소드 demoProc1()에서는 실제 발생하지 않는 예외 Exception을 선언하므로 메소드 demoProc1()을 호출하는 코드에서는 불필요한 catch-절을 추가해야 한다. 그리고 메소드 demoProc2()에서는 실제 발생하는 예외 IOException 보다 상위의 예외 Exception을 선언하므로 메소드 demoProc2()을 호출하는 메소드에서는 정확한 예외 처리가 어려운 경우가 발생한다. 이러한 문제는 자바 컴파일러가 프로그래머의 선언에 의존하는 프로시저-내 분석을 수행하기 때문이다.

따라서 본 연구에서는 프로그래머의 선언과 무관한 프로시저-간(interprocedural) 예외 상황 분석을 설계한다. 특히 이 분석은 선언을 이용하지 않고 예외 상황을 분석함으로써 보다 정확한 처리되지 않는 예외 상황 정보를 줄 수 있을 뿐만 아니라 프로그래머가 선언한 예외 상황 명세를 보다 정확히 검증할 수 있다. 또한 JDK 분석 방법과 비교를 위해 프로그래머의 선언에 의존하는 자바 컴파일러 방식의 예외 상황 분석도 설계한다.

3. 예외분석기의 설계

본 연구의 예외 상황 분석은 집합-기반 분석 기법을 기반으로 한다[4]. 예외 상황을 분석하기 위해서는 클래스 분석(class analysis) 정보가 필요하다. 클래스 분석이란 어떤 식이 가리키는 객체의 타입(클래스)을 정적으로 유추하는 분석이다[3]. 자바에 대한 집합-기반 클래스 분석은

[2]에 기술되어 있다. 본 연구는 클래스 분석 정보가 존재한다고 가정하고 예외 분석기를 설계한다.

3.1 입력 프로그램

본 연구에서 예외 상황 분석의 명료한 설명을 위해 예외 기능을 갖는 가상의 자바 언어 ExnJava를 정의하였다. [그림2]는 ExnJava에 대한 추상 구문(abstract syntax)이다. [그림2]에서 정의된 각 구문의 의미는 자바 언어의 의미와 동일하다.

P	::= C^*	program
C	::= class c text c {var x^* M^*}	class definition
M	::= $m(x)$= e[throws c^*]	method definition
e	::= id	variable
	| id := e	assignment
	| new c	new object
	| this	self object
	| e ; e	sequence
	| if e then e else e	branch
	| throw e	exception raise
	| try e catch *(c x e)*	exception handle
	| $e.m(e)$	method call
id	::= x	method parameter
	| $id.x$	field variable
c		class name
m		method name
x		variable name

그림 2 ExnJava 의 추상 구문

3.2 식-단위의 프로시저-간 예외 상황 분석

예외 상황 분석의 목표는 각 식에 대해서 실행 중 발생될 수 있는 예외 상황들을 분석을 통해서 구하는 것이다. 따라서 이 분석에서는 모든 식 e 에 대해 처리되지 않는 예외들을 위한 집합 변수 Xe 를 정의하고 $Xe \supseteq se$ 형태의 집합-관계식(set-constraints)을 구성한다. 집합 변수 Xe 는 식 e 에서 처리되지 않는 예외의 클래스 이름들을 포함하게 된다. 집합-관계식 $Xe \supseteq se$의 의미는 집합 변수 Xe 가 se가 나타내는 집합을 포함한다는 것을 의미한다.

집합-기반 분석은 집합-관계식을 생성하는 생성 규칙들(derivation rules)을 정의함으로써 설계한다. [그림3]는 예외 상황 분석을 위해 각 식에 대해 집합-관계식을 생성하는 생성 규칙이다. [그림3]의 "$\triangleright e : C$" 는 집합-관계식 C 는 수식 e 에서 생성된다는 것을 의미한다. 그리고 **Class**(e_1)는 식 e_1 에 대한 클래스 분석의 결과로 식 e_1 이 가리키는 객체의 클래스를 나타낸다.

[New] $> \text{new } c : \phi$

[FieldAss] $$\frac{> e_1 : C_1}{> id.x := e_1 \{\chi_e \supseteq \chi_{e_1}\} \cup C_1}$$

[Paramss] $$\frac{> e_1 : C_1}{> x := e_1 : \{\chi_{e_1}\} \cup C_1}$$

[Seq] $$\frac{> e_1 : C_1 \quad > e_2 : C_2}{> e_1 ; e_2 : \{\chi_e \supseteq \chi_{e_1} \cup \chi_{e_2}\} \cup C_1 \cup C_2}$$

[Cond] $$\frac{> e_0 : C_0 \quad > e_1 : C_1 \quad > e_2 : C_2}{> \text{if } e_0 \text{ then } e_1 \text{ else } e_2 : \{\chi_e \supseteq \chi_{e_0} \cup \chi_{e_1} \cup \chi_{e_2}\} \cup C_0 \cup C_1 \cup C_2}$$

[FieldVar] $$\frac{> id : C_{id}}{> id.x : C_{id}}$$

[Throw] $$\frac{> e_1 : C_1}{> \text{throw } e_1 : \{\chi_f \supseteq \text{Class}(e_1)\} \cup C_1}$$

[Try] $$\frac{> e_0 : C_0 \qquad > e_1 : C_1}{> \text{try } e_0 \text{ catch } (c_1\, x_1\, e_1) : \{\chi_e \supseteq (\chi_{e_0} - \{c_1\}^*) \cup \chi_{e_1}\} \cup C_0 \cup C_1}$$

[MethCall] $$\frac{> e_1 : C_1 \qquad > e_2 : C_2}{> e_1.m(e_2) : \{\chi_e \supseteq \chi_{c.m} \mid c \in \text{Class}(e_1), m(x) = e_m \in c\} \cup \{\chi_e \supseteq \chi_{e_1} \cup \chi_{e_2}\} \cup C_1 \cup C_2}$$

[MethDef] $$\frac{> e_m : C}{> m(x) = e_m : \{\chi_{c.m} \supseteq \chi_{e_m}\} \cup C}$$

[ClassDef] $$\frac{> m_i : C_i \quad i = 1, \ldots, n}{> \text{class } c = \{\text{var } x_1, \ldots, x_k, m_1, \ldots, m_n\} : C_1 \cup \ldots \cup C_n}$$

[Program] $$\frac{> c_i : C_i \quad i = 1, \ldots, n}{> c_1, \ldots, c_n : C_1 \cup \ldots \cup C_n}$$

그림 3 식-단위의 프로시저-간 예외 상황 분석

[그림3]의 주요 수식에 대한 생성규칙의 의미는 다음과 같다.

[Throw] $$\frac{> e_1 : C_1}{> \text{throw } e_1 : \{\chi_f \supseteq \text{Class}(e_1) \cup \chi_{e_1}\} \cup C_1}$$

throw-식에서 처리되지 않는 예외 클래스는 식 e_1 의 결과가 나타내는 객체의 클래스와 식 e_1 에서 처리되지 않는 예외 클래스들이다.

[Try] $$\frac{> e_0 : C_0 \qquad > e_1 : C_1}{> \text{try } e_0 \text{ catch } (c_1\, x_1\, e_1) : \{\chi_e \supseteq (\chi_{e_0} - \{c_1\}^*) \cup \chi_{e_1}\} \cup C_0 \cup C_1}$$

식 e_0 에서 발생하는 예외들 중에 클래스 c_1 이거나 이의 하위 클래스인 경우에는 예외 처리가 된다. 그리고 예외 처리기 e_1이 실행될 때도 예외가 발생할 수 있다. 그러므로 try-식의 처리되지 않는 예외 클래스들은 식 e_0 에서 발생하는 예외들 중에서 처리되는 예외를 제외한 예외 클래스들과 예외 처리기 e_1 에서 발생하는 예외 클래스들이다.

[MethCall] $$\frac{> e_1 : C_1 \quad > e_2 : C_2}{> e_1.m(e_2) : \{\chi_e \supseteq \chi_{c.m} \mid c \in Class(e_1), m(x) = e_m \in c\} \; Y \{\chi_e \supseteq \chi_{e_1} Y \chi_{e_2}\} Y C_1 Y C_2}$$

집합 변수 $X_{c.m}$ 는 클래스 c 의 메소드 m에서 발생되었으나 처리되지 않는 예외들을 의미한다. 메소드 호출-식에서 처리되지 않는 예외는 식 e_0, e_1 에서 처리되지 않는 예외들과 식 e_0 의 결과가 나타내는 객체의 클래스의 메소드에서 처리되지 않는 예외들이다.

[그림3]의 생성 규칙을 이용하여 입력 자바 프로그램에 대한 집합 관계식을 구성하고 구성된 집합-관계식들의 해를 구하면 각 수식에서 처리되지 않는 예외들을 구할 수 있다. 이 정보를 이용해서 throws 구문에 광범위한 선언과 필요하지 않은 선언을 검증할 수 있고 또한 광범위한 예외 처리기와 불필요한 예외 처리기를 검증할 수 있다.

3.3 프로시저-내 예외 상황 분석

자바 컴파일러의 예외 상황 분석은 프로그래머의 선언에 의존하는 프로시저-내 분석이다. 프로시져-내 분석에서 메소드 호출문 $e_0.m(e_1)$ 의 처리되지 않는 예외는 호출된 메소드 $m(e_1)$ 의 **throws** 구문에 선언된 예외이다. 그러므로 프로시저-내 예외 상황 분석의 설계는 [그림3] 의 생성 규칙 중 [MethDef], [MethCall] 생성 규칙을 다음과 같이 변경하면 된다.

[MethDef] $$\frac{> e_m : C}{> m(x) = e_m \text{ throws } T^*_{cm} : \{T_{cm} \supseteq \chi_{e_m}\} Y C}$$

[MethCall] $$\frac{> e_1 : C_1 \quad > e_2 : C_2}{> e_1.m(e_2) : \{\chi_e \supseteq T_{cm} \mid c \in Class(e_1), m(x) = e_m \in c\} Y \{\chi_e \supseteq \chi_{e_1} Y \chi_{e_2}\} Y C_1 Y C_2}$$

4. 분석 속도 향상을 위한 최적화

식-단위의 예외 상황 분석은 명료한 설명을 위해 이론적인 설계에서 일반적으로 사용되고 있으나 모든 식에 대해서 집합-변수를 구성하여 분석하는 것은 수천 혹은 수만 줄의 자바 프로그램에 대해서는 그 분석 속도가 대단히 느리므로 현실성이 부족하다. 따라서 본 연구에서는 분석 속도 면에서 실용적인 분석을 위해서 예외 관련 식에 대해서만 집합-변수를 정의함으로써 집합-변수들의 수를 줄이고 이를 통해 속도를 향상시킨다.

4.1 메소드 단위의 예외 상황 분석

식-단위의 예외 상황 분석은 이론적으로는 정확한 분석 결과를 제공하지만 집합 변수의 수가 식의 수 만큼 필요하므로 분석 속도 면에서 실용적이지 못하다. 집합-기반 분석의 시간 복잡도는 집합 변수의 수가 N 이라고 할 때 N^3 이므로 실제적인 자바 프로그램의 분석에는 적절하지 않다. 그러므로 각각의 식이 아닌 각 메소드와 try 블록에 집합 변수를 정의하여 집합 변수의 수를 줄임으로서 분석 속도 면에서 실용적인 예외 상황 분석기를 설계한다. 메소드 단위의 시간 복잡도는 식-수준 분석과 같이 집합 변수의 수가 N 일 때 N^3 이지만 N 의 수를 줄이는 것이다. 이와 같이 분석의 단위를 조절하는 기법은 [2,5]의 예외 상황 분석에 성공적으로 적용된 바 있다. [그림4]는 메소드 단위의 집합-관계식을 생성하는 생성 규칙이다.

[New]m $$f > \text{new } c : \phi$$

[FieldAss]m $$\frac{f > e_1 : C_1}{f > \text{id.x} := e_1 : C_1}$$

[Paramss]m $$\frac{f > e_1 : C_1}{f > x := e_1 : C_1}$$

[Seq]m $$\frac{f > e_1 : C_1 \quad f > e_2 : C_2}{f > e_1 ; e_2 : C_1 Y C_2}$$

[Cond]m $$\frac{f > e_0 : C_0 \quad f > e_1 : C_1 \quad f > e_2 : C_2}{f > \text{if } e_0 \text{ then } e_1 \text{ else } e_2 : C_0 Y C_1 Y C_2}$$

[FieldVar]m $$\frac{f > \text{id} : C_1}{f > \text{id.x} : C_1}$$

[Throw]m $$\frac{f > e_1 : C_1}{f > \text{throw } e_1 : \{\chi_f \supseteq Class(e_1)\} Y C_1}$$

[Try]m $$\frac{g > e_0 : C_0 \qquad f > e_1 : C_1}{f > \text{try } e_g \text{ catch}(c_1 \; x_1 \; e_1) : \{\chi_f \supseteq \chi_g - \{c_1\}^*\} Y C_g Y C_1}$$

[MethCall]m $$\frac{f > e_1 : C_1 \qquad f > e_2 : C_2}{f > e_1.m(e_2) : \{\chi_f \supseteq \chi_{c.m'} \mid c \in Class(e_1), m'(x) = e_{m'} \in c\} Y C_1 Y C_2}$$

[MethDef]m $$\frac{m > e_m : C_m}{> m(x) = e_m : C_m}$$

[ClassDef]m $$\frac{> M_i : C_i \; i = 1, K, m}{> \text{class } c = \{\text{var } x_1, K, x_k, M_1, K, M_n\} : C_1 Y \Lambda Y C_m}$$

[Program]m $$\frac{> c_i : C_i \; i = 1, K, n}{> c_1, K, c_n : C_1 Y \Lambda Y C_n}$$

[그림 4] 메소드-단위의 프로시저-간 예외상황 분석

[그림4]의 $f \triangleright e$ 는 식 e 가 메소드 f 내의 문장이란 의미이다. [그림4]의 생성 규칙의 의미는 다음과 같다.

[Throw]m $$\frac{f > e_1 : C_1}{f > \text{throw } e_1 : \{\chi_f \supseteq Class(e_1)\} Y C_1}$$

throw e 가 메소드 m 의 구문일 때 메소드

m 에 대한 집합 변수 χ_m 는 처리되지 않는 예외 클래스 Class(e_1)를 포함한다.

$$[\text{Try}]\text{m}\quad \frac{g > e_0 : C_0 \qquad f > e_1 : C_1}{f > \text{try}\, e_g\, \text{catch}\,(c_1\, x_1\, e_1) : \{\chi_f \supseteq \chi_g - \{c_1\}^*\}\ \ YC_g\, YC_1}$$

try 블록 e_g 에서 처리되지 않는 예외들은 집합 변수 X_g 가 가지고 있고 e_g 에서 발생한 예외들 중에서 catch 블록에서 처리될 수 있다. 그러므로 위 구문에서 처리되지 않은 예외들은 e_g 에서 발생한 예외들 중에서 클래스 c_1 의 하위클래스들을 제외한 예외들이다.

$$[\text{MethCall}]\text{m}\quad \frac{f > e_1 : C_1 \qquad f > e_2 : C_2}{f > e_1.m(e_2) : \{\chi_f \supseteq \chi_{c.m} \mid c \in \text{Class}(e_1), m(x) = e_m \in c\}\ \ YC_1\, YC_2}$$

메소드 호출 $e_1.m(e_2)$ 에서 처리되지 않은 예외들은 호출된 메소드 $m(e_2)$ 에서 처리되지 않은 예외들이다. 자바 컴파일러에서는 호출된 메소드에서 처리되지 않은 예외는 호출 된 메소드의 throws 구문에 선언된 예외들을 나타내고 프로시저-간 분석 방법에서는 호출된 메소드에서 처리되지 않은 예외 $\chi_{c.m}$ 은 실제 발생 가능한 예외들 중에서 처리되지 않는 예외들을 나타낸다.

5. 구현 및 실험

5.1 구현

4장에서 설계된 메소드 단위의 예외 상황 분석기는 2-패스로 구현된다. 첫 번째 패스에서는 입력 프로그램에 대해 집합-관계식을 구성한다. lex&yacc을 이용하여 예외 상황 분석을 위한 집합-관계식 구성 규칙을 yacc 의 semantic action 형태로 기술한다. 두 번째 패스에서는 구성된 집합-관계식의 해를 계산한다. 집합-관계식의 해는 집합-관계식의 iterative-fixed point 를 구하는 과정으로 계산된다. 이 과정은 프로그램에 존재하는 예외 클래스의 수가 유한하므로 반드시 종료하게 된다.

5.2 대상 프로그램

실험은 소스 코드가 공개되고 자주 사용되는 자바 응용 프로그램과 자바 애플릿을 대상으로 했다. 여기서 사용된 자바 프로그램들은 [7,8]에서 구할 수 있다. [표1]과 [표2]는 대상 프로그램에 대한 간략한 설명이다.

표 1 대상 프로그램

Programs	Description	Total Classes	Total methods	Lines of code
Statistician	methods statistics for a class	1	1	387
JavaBinHex	BinHex(.hqx) decompressor	1	3	300
JHLZIP	ZIP compressor	2	11	425
JHLUNZIP	ZIP uncompressor	1	3	287
com.ice.tar	Unix Tar Archive	10	141	4045
Jess-Rete	Reasoning Engine of Jess	1	98	1667

표 2 대상 프로그램

Programs	Kinds of exception	Throws spec	catch block
Statistician	3	1	24
JavaBinHex	2	0	8
JHLZIP	3	4	8
JHLUNZIP	2	1	3
com.ice.tar	6	40	21
Jess-Rete	7	38	10

5.3 실험 결과

5.2절의 대상 프로그램에 대해 프로시저-내와 프로시저-간 예외 상황 분석을 적용한 결과는 [표3]과 [표 4]와 같다.

[표 3] 광범위한 throws 구문과 불필요한 throws 구문

Programs	broader throws		unnecessary throws	
	interprocedural	intraprocedural	interprocedural	intraprocedural
Statistician	0	0	0	0
JavaBinHex	0	0	0	0
JHLZIP	2	1	0	0
JHLUNZIP	1	0	0	0
com.ice.tar	3	1	4	0
Jess-Rete	1	1	6	0

[표3]에서 unnecessary throws 는 발생하지 않는 예외들을 throws 구문에 선언한 경우이고 broader throws 는 발생 가능한 예외 클래스보다 상위의 클래스들을 선언한 경우이다. 이와 같은 경우 자바 컴파일러와 같이 프로시저-내 분석을 하면 불필요하고 광범위한 throws 구문을 검증하지 못해 적절한 예외처리가 어려운

경우가 발생하는 것을 [표 4]에서 알 수 있다.

[표 4] 광범위한 예외처리기와 불필요한 예외처리기

Programs	broader catch		unnecessary catch	
	interprocedural	intraprocedural	interprocedural	intraprocedural
Statistician	0	0	1	1
JavaBinHex	1	1	0	0
JHLZIP	1	1	0	0
JHLUNZIP	1	0	0	0
com.ice.tar	4	2	0	0
Jess-Rete	3	2	0	0

6. 결론

자바 컴파일러의 예외 상황 분석 기법은 프로그래머가 선언한 throws 구문에 의존하는 프로시저-내 분석이다. 따라서 프로그래머가 throws 구문에 실행 시 발생 가능한 예외보다 상위의 예외를 선언하거나 실제 발생하지 않는 예외들을 선언하는 경우 정확한 분석을 할 수 없다. 따라서 본 연구에서는 프로그래머의 선언과 무관한 프로시저-간(interprocedural) 예외 상황 분석을 설계하고 이를 구현하였다. 또한 두 분석 방법의 비교를 위해 프로그래머의 선언에 의존하는 자바 컴파일러 방식의 예외 상황 분석도 설계 구현하였으며 실험을 통해서 실제 프로그램에 적용시켜 봄으로써 두 분석 방법을 비교하였다. 본 연구에서 제안한 분석은 선언을 이용하지 않고 예외 상황을 분석함으로써 보다 정확한 처리되지 않는 예외 상황 정보를 줄 수 있을 뿐만 아니라 프로그래머가 선언한 예외 상황 명세를 보다 정확히 검증할 수 있다.

7. 참고 문헌

[1] K. Arnold and J. Gosling, "The Java Programming Languages, Second Edition", Addison-Wesley, 1997

[2] B.-M. Chang, K. Yi, and J. Jo, "Constraint-based analysis for Java", SSGRR2000 Computer and e-business Conference, August 2000, L'Aquila, Italy.

[3] G. Defouw, D. Grove, and C. Chambers, "Fast interprocedural class analysis", in Proceedings of 25th ACM SIGPLAN Symposium on Principles of Programming Languages, pages 222-236, Jan. 1998.

[4] N. Heintze, "Set-based program analysis", Ph.D thesis, Carnegie Mellon University, Oct, 1992.

[5] K. Yi and S. Ryu, "Towards a cost-effective estimation of uncaught exceptions in SML programs", In Lecture Note in Computer Science, volume 1302, pages 98-113, Springer-Verlag, Proceedings of 4th Static Analysis Symposium, Sep, 1997.

[6] K. Yi and B.-M. Chang, "Exception analysis for Java", '99 ECOOP Workshop on Formal Techniques for Java Programs, June 1999.

[7] http://www.jars.com

[8] http://www.gamelan.com