논•문

자바 프로그램에서 예외 사용에 대한 실험

(Experiments on Exception Use in Java Programs)

이 지연, 창 병모
숙명여자대학교 정보과학부
{jylee,chang}@cs.sookmyung.ac.kr

요 약

프로그램 수행 전에 예외 사용에 대한 정적 분석을 통해 프로그램 개발자에게 예외와 그 사용에 대한 좀더 정확한 정보를 제공함을 목표로 한다. 이를 위해 본 논문에서는 프로그램에서의 예외 사용에 대한 정보를 분석하기 위하여 예외 관련 구문들의 사용 빈도 및 사용 형태를 실험을 통해서 알아본다. 또한 예외 관련 클래스들의 계층구조를 보여줌으로써 예외 구조에 대한 보다 효과적인 이해를 돕는다. 또한 소스 코드에서 예외 관련 문장들만을 추출하여, 프로그램의 대략적인 구조와 예외의 사용을 한눈에 볼 수 있도록 한다.

1. 서론

최근 들어 가장 대표적인 객체 지향 언어로 사용되는 자바는 소프트웨어 공학의 관점으로 볼 때, 수정이 용이하고 적은 비용으로 유지보수가 가능하며 재사용성과 확장성을 가지게 하는 언어이다[AW98]. 자바는 객체 지향에 충실한 언어로, 철저하게 모듈화가 되어 있어, 개발자에게 편리함을 주고 있으나, 거대하고 견고한 소프트웨어를 개발함에 있어서 정상적인 명령의 수행을 방해하는 여러 가지 다양한 상황들에 부딪칠 수 있는데, 이러한 비정상적인 상황을 예외(exception)라고 한다. 자바는 이러한 예외를 발생시키고 이를 감시하고 발생된 예외 처리(exception handling)를 위한 기능을 제공하고 있다. 이러한 예외 처리 메카니즘은 개발자들에게 예외로 인한 비정상적인 프로그램 수행을 피하도록 프로그램을 설계할 수 있는

방법과 기회를 제공한다.

본 논문에서는 프로그램 수행 전에 예외 사용에 대한 정적 분석을 통해 프로그램 개발자에게 예외와 그 사용에 대한 좀더 정확한 정보를 제공함을 목표로 한다. 이를 위해 본 논문에서는 다음과 같은 분석과 실험을 하였다.

첫째, 프로그램에서의 예외 사용에 대한 정보를 분석하기 위하여 예외 관련 구문들의 사용 빈도 및 사용 형태를 알아본다.

둘째, 클래스들의 계층 구조를 보여줌으로써, 예외 사용과 그 구조의 보다 효과적인 이해를 돕는다.

마지막으로, 프로그램 슬라이싱을 이용하여, 소스 코드에서 예외 관련 문장들만을 추출하여, 프로그램의 대략적인 구조와 예외의 사용을 한눈에 볼 수 있도록 한다.

본 논문의 구성은 다음과 같다. 2장에서는 관련 연구들과 본 실험의 기반이 되고 있는 도구에 대해서 알아보고, 3장에서는 자바에서의 예외 및 예외 처리에 대한 개요를 설명한다. 4장에서는 시스템의 설계와 구현에 대해서 설명하고, 5장에서는 실험 결과에 대하여 설명한다. 6장에서는 결론 및 향후 연구 방향을 제시한다.

2. 관련연구

2.1 예외의 흐름 분석[RM99, JEX]

1999년 British Columbia 대학의 Robillard와 Murphy에 의해 제안된 Jex는 자바 프로그램 내에서 예외의 흐름에 대한 정보를 제공하기 위한 분석 도구로, 소스 파일 내에서 메소드마다 각각 발생하는 예외의 리스트와 예외가 발생하는 위치 정보를 제공한다. Jex는 자바 소스를 입력받아 각 메소드에 대한 .jex파일을 생성한다. 이 .jex 파일은 예외의 흐름에 관한 정보를 포함하고 있는 텍스트 파일로서, 메소드의 헤더와 예외에 대한 명세로 이루어져 있다. 특히 프로그램 내에서 처리되지 않는 예외(uncaught exception)에 대한 정보를 보여준다.

2.2 자바 예외의 정적 분석[JESP]

Rutgers 대학의 Ryder 등에 의해 제안된 도구인 JESP를 이용한 이 연구는 자바 코드 내에서 발생하는 예외 관련문장들(try-catch, finally, throw)과 같은 일반적인 문장들을 구분하려는 시도였다. 이 연구에서는 JavaSpec을 포함한 31개의 자바로 이루어진 응용 프로그램들을 선정하여, 프로그램 자체의 크기에 비례하여 예외 관련 문장들을 포함하는 메소드의 비율과 예외 관련 문장이 사용 비율에 관해 조사하였다.

2.3 Java Compiler Compiler(JavaCC)

metamata에서 개발된 도구로서, Purdue 대학의 Palsburg에 의해 개발된 트리 생성기인 JTB와, 자바 컴파일러 컴파일러인 JavaCC로 구성된다. JavaCC는 자바로 작성된 자바 파서 생성기로서 자바 문법 명세

(grammar specification)를 읽어서, 그것을 자바 프로그램으로 변환시킨다. JavaCC는 어휘 분석기 명세와 파서 명세를 하나의 파일에 작성하고, 이를 이용하여 파서를 생성한다. 또한 Visitor 패턴을 제공하여, 깊이 우선 탐색(Depth-First search)을 지원하는 클래스들을 제공한다[JTB].

2.4 Barat

Barat은 자바로 작성된 자바 컴파일러 전단부(front-end)로 JavaCC를 이용하여 구현되었다[BD98]. Barat은 자바 파일이나 자바 클래스 파일로부터 이름과 타입 정보를 포함하는 AST(Abstract syntax tree)를 구성한다. 또한 Visitor를 이용하여 AST를 탐색할 수 있다.

Barat의 구성은 크게 barat이라는 하나의 패키지로 이루어져 있으며 이 barat은 Barat, Visitor, Node등의 파일들과 barat.codegen, barat.collections, barat.parser, barat.reflect 이렇게 4개의 Package로 이루어진다.

Barat의 특징 중 하나는 visitor 디자인 패턴[GHJW95]을 기반으로 한 구조를 제공하고 있다는 것이다. visitor 패턴이란 트리의 각 노드들을 방문하면서 수행하는 연산을 정의할 수 있다. 각 노드들에 대한 정의는 바꾸지 않고 수행할 연산만을 새로 정의함으로써 다양한 연산을 수행할 수 있도록 visitor를 작성할 수 있다.

3. 자바 언어에서의 예외

자바에서의 예외는 클래스로 표현된다. 또한 자바의 모든 예외 타입(exception type)은 Throwable 클래스나 이것의 서브 클래스(subclass)로부터 상속받아야 한다. 예외는 검사 예외(checked excpetion)와 비검사 예외(unchecked exception)로 구분할 수 있는데 검사 예외는 컴파일러가 예외가 처리 혹은 명세 되었는지를 검사하게 된다. 비검사 예외는 보통 라이브러리 루틴으로의 호출이나 런타임 환경(runtime environment)에 의해서 발생되는 RuntimeException 클래스의 하위 클래스로 컴파일러가 이런 검사를 하지 않는다 [GJS96].

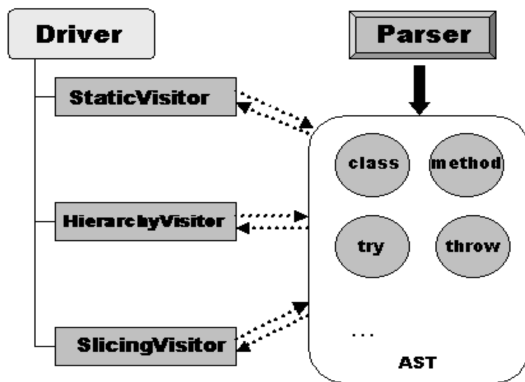
자바에서의 예외 처리에는 *try-catch* 문이 사용된다. 먼저 *try* 문은 예외가 발생하는지를 감시해서 예외가 발생되면 그 예외의 타입에 해당되는 *catch* 블록 안에서 예외가 처리된다. 물론 해당하는 예외의 타입을 가지는 *catch* 블록이 없으면 처리될 수 없다. 처리되지 않는 예외는 호출된 메소드로 계속해서 전파(propagation)되어 처리기가 있는 부분에서 처리되며, 이렇게 계속 전파(propagation)되어도 예외 처리기가 존재하지 않는다면, 프로그램은 오류 메시지를 출력하며 종료된다.

4. 시스템 설계 및 구현

본 논문에서 구현하고자 하는 시스템

은, 자바 언어의 정적분석을 위한 도구로서, 특히 예외 사용에 대한 분석에 초점을 맞추고 있다. 이를 위해 예외가 발생하는 위치와 예외 관련 문장들의 사용 형태를 분석하고 이를 통계로 제공한다. 예외 관련 클래스들의 계층구조를 분석함으로써 예외 구조에 대한 정보를 제공한다. 또한 예외 사용과 관련된 문장들만을 추출하여 보여줌으로써 예외 사용 형태를 보다 쉽게 파악할 수 있도록 한다.

본 논문에서는 Barat 시스템을 기반으로 하여 하나의 드라이버(Driver)와 세 개의 visitor를 설계 구현하였다. 각 visitor들은 visitor 디자인 패턴를 이용하여 AST 노드를 깊이-우선-탐색(depth first traversal)한다. visitor들의 실행 과정은 [그림 1]과 같다.



[그림 1] 시스템의 구조

[표 1]은 본 논문에서 구현한 각각의 visitor의 동작을 구분하여 설명한 것이다.

visitor	기능 및 제공하는 정보
Static Visitor	- 프로그램의 크기와 예외 관련 문장들에 대한 통계
Hierarchy Visitor	- 예외관련 클래스들의 계층구조 - 입력 프로그램의 클래스 이름은 모아 예외 클래스들의 계층 구조를 보임
Slicing Visitor	- 프로그램 슬라이싱 - 프로그램에서 예외 관련 문장 추출 - 예외 문장의 위치와 전체 구조 제공

[표 1] visitor의 종류와 그 기능

예외 관련 문장의 분석 및 통계를 위해 StaticVisitor를 구현하였다. StaticVisitor는 Barat의 DescendingVisitor로부터 유도하여 구현하였고, AST에서 클래스나 메소드, 예외 관련 문장들에 해당하는 노드를 방문하면서 예외 관련 통계 정보를 모은다. 파일에서의 파일 내 하나의 클래스 노드에 대한 방문이 끝날 때마다 통계에 관한 정보를 CompilationUnit에서 수합하게 된다. 반면, 패키지 단위에서의 분석은 클래스 노드들을 방문할 때마다 클래스 각각에 대해서 visitor를 생성해서 해당 클래스을 분석하고 각 클래스의 예외 관련 통계 정보들은 드라이버에서 수합하도록 구현하였다.

HierarchyVisitor에서는 클래스의 계층구조를 알기 위해서, 클래스에 해당하는 노드를 방문하면서 현재 클래스와 상위 클래스들의 이름에 대한 정보를 차례로 모으고, 클래스가 예외 클래스인 Throwable의 하위 클래

스인지를 확인하여 예외 클래스의 계층구조를 현재 클래스로부터 가장 상위 클래스인 Object까지 차례로 보여지도록 구현하였다.

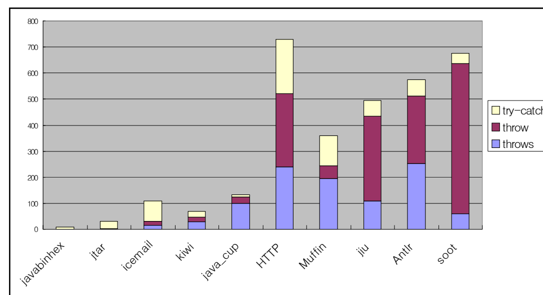
SlicingVisitor는 프로그램 소스코드 내에서 예외관련 문장들을 추출해내기 위해서 Barat에서 제공하는 OutputVisitor를 상속받아 구현하였다. OutputVisitor는 프로그램 소스를 입력받아 이를 그대로 다시 출력한다. SlicingVisitor는 예외에 관한 문장들과 더불어 클래스 구조를 알기 위해서 클래스와 메소드 헤더과 메소드 호출문 들도 함께 출력하도록 구현하였다.

5. 실험

실험은 소스 코드가 공개되고 자주 사용되는 자바 프로그램을 대상으로 하였다 (http://www.jars.com,http://www.gamelan.com). 대상 프로그램들의 간략한 설명과 특징은 아래 [표 2]와 같다.

(1) 실험1 : 예외 관련 문장에 대한 통계

프로그램 내에서 예외 관련 문장의 수를 조사하기 위한 실험에서는 [그림 2]와 같이 예외 관련문장의 수를 조사한 결과 프로그램의 크기와 비례하여 나타났다. 특이한 점으로는 icemail과 HTTPClient와 같은 통신 관련 프로그램에서 프로그램의 크기에 비해 예외 관련 문장들이 많이 쓰임을 알 수 있었다.



[그림 2] 예외 관련 문장의 통계

Benchmark	프로그램 설명	크기(#line)	#class	#method
javabinhex	Binhex(.hqx)의 압축 복원	300	1	4
jtar	GZIP의 compression/decompression	5248	22	123
icemail	Java email client	2598	14	136
kiwi	Complement the Java Foundation Classes(JFC)	9567	62	357
java_cup	java의 LALR parser generator	10337	32	321
HTTPClient	HTTP client library	10677	73	368
muffin	Filtering proxy server for WWW	19771	117	1058
jiu	Java Imaging Utilities	22334	110	988
antlr	Framework for compiler construction	41085	147	1730
soot	Java bytecode analyzer	96144	1012	8457

[표 2] 예제 프로그램의 특징

Benchmark	예외 관련 문장의 통계				메소드(m)에 대한 비율			
	#throws	#throw	#try	Total	throws/m	throw/m	try/m	Total
javabinhex	0	0	8	8	0	0	2	2
jtar	2	0	29	31	0.016	0	0.235	0.252
icemail	15	16	77	108	0.110	0.117	0.566	0.794
kiwi	29	17	22	68	0.081	0.048	0.062	0.190
java_cup	99	26	8	133	0.308	0.081	0.022	0.411
HTTPClient	240	281	208	729	0.313	0.366	0.271	0.949
muffin	194	49	116	359	0.183	0.046	0.110	0.339
jiu	326	60	63	449	0.110	0.340	0.061	0.481
antlr	253	258	63	574	0.146	0.164	0.036	0.347
soot	60	575	40	675	0.007	0.065	0.004	0.077

[표 3] 예외 관련 문장의 수와 비율에 대한 통계

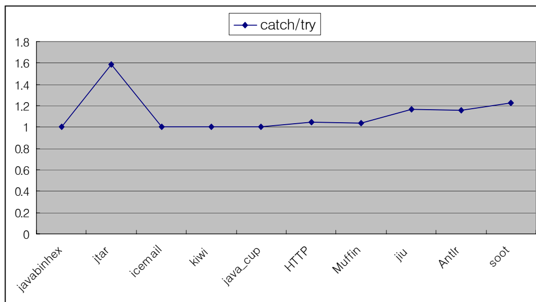
[표 3]은 프로그램 내에서 예외 관련 문장들의 수, 즉 예외 관련 문장들의 통계와 메소드 하나가 포함하는 예외 관련 문장의 비율을 나타낸 표이다. [표 3]의 javabinhex는 try 블록의 개수가 무려 메소드의 두배로 나타난 점이 특이한 사항이고, 역시 다른 프로그램에서는 메소드가 포함하고 있는 예외 관련 문장의 수는, 메소드 하나당 0.077 ~ 0.949개로 나타남을 알 수 있다.

다음의 [표 4]에서는 예외 관련 문장들의 각각 대한 통계이다. 실험은 try-catch문, throw문 throws문의 사용에 대한 통계를 조사하였다.

Benchmark	try				throw의 식			throws exceptions(E)		
	#try	#catch	#finally	catch/try	#throw new	#throw m()	#throw x	#throws	#E	E/throws
javabinhex	8	8	0	1	0	0	0	0	0	
jtar	29	46	0	1.59	0	0	0	2	4	2
icemail	77	77	0	1	16	0	0	15	15	1
kiwi	22	22	0	1	1	0	16	29	30	1.03
java_cup	8	8	0	1	25	0	1	99	101	1.02
HTTPClient	208	217	9	1.04	240	1	40	240	344	1.43
muffin	116	120	5	1.03	42	0	4	194	203	1.05
jiu	60	70	0	1.16	326	0	0	109	212	1.695
antlr	63	73	1	1.58	275	0	10	253	498	1.97
soot	40	49	0	1.23	573	0	2	60	70	1.17

[표 4] 예외 관련 문장의 사용에 대한 통계

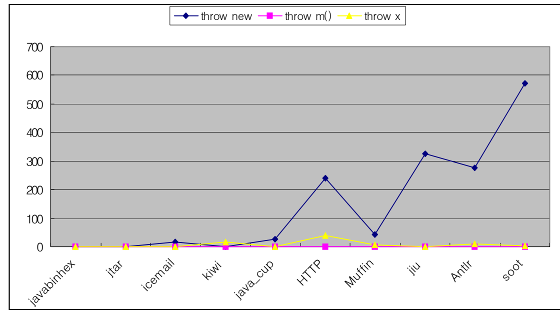
우선 try-catch 블록에서, try 하나당 catch는 하나 이상이 올 수 있다. 그러나 [그림 4]에서와 같이 대부분의 프로그램에서는 try 하나당 catch 하나, 또는 두 개 정도를 사용하고 대부분은 catch 하나만 사용하는 것을 알 수 있었다. 또한 finally는 거의 사용하지 않는다.



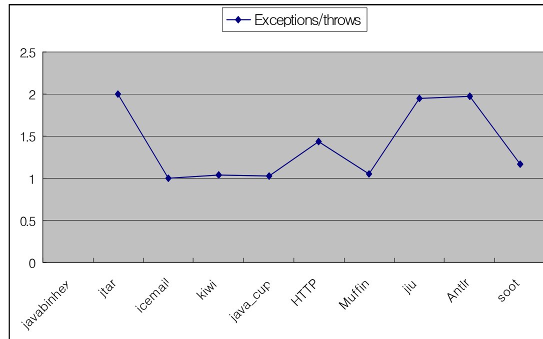
[그림 3] try 문에 대한 catch문의 비율

throw 문은 식(expression)과 함께 사용되며, throw문이 가지는 식의 형태는 new 연산자와 메소드 호출, 또는 변수를 사용할 수 있다. 이 실험에서는 [표 4]와 같이, throw문이 식으로서 new 연산자, 변수, 메소드 호출 순으로 사용하고 있다는것을 알 수 있었고, 메소드 호출은 거의 사용하지 않음을 알 수 있었다.

throws 문에서는 하나 이상의 예외를 명세할 수 있다. 이 실험을 통해서는 throws 문 하나당 명세되는 예외의 수는 대부분 1~2개임을 알 수 있었다.

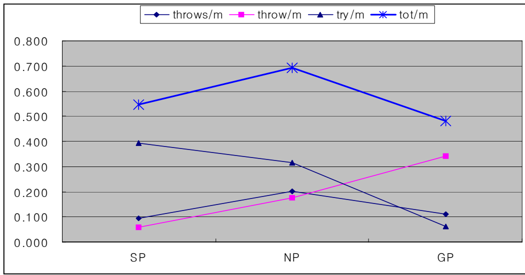


[그림 4] throw문의 사용에 관한 통계



[그림 5] throws문의 사용에 관한 통계

[그림 6]은 프로그램 그룹에 따른 예외 사용 비율을 알아보기 위한 실험이다. 우선 예제 프로그램들을 각각 시스템 프로그램(SP), 그래픽 관련 프로그램(GP), 네트워크 관련 프로그램(NP)으로 구분하여, 각 프로그램 그룹에 대해서 예외 관련 문장의 사용 비율을 조사한다



[그림 6] 프로그램 그룹에 따른 예외 사용 비율 비교

이 실험에서는 네트워크 관련 프로그램(NP) 그룹에서, 다른 그룹보다 예외 관련문장의 사용 비율이 높은 것과 try-catch문의 사용 비율이 시스템 프로그램(SP) 그룹에서는 높게 나타나고, 그래픽 관련 프로그램(GP)에서는 현저하게 낮게 나타남을 알 수 있었다.

(2) 실험2 ; 예외 관련 클래스 계층 구조

다음은 Benchmark 프로그램 중 antlr의 예외 관련 클래스의 계층구조를 보여주고 있다.

```
ANTLRError => Error => Throwable
ANTLRException => Exception => Throwable
CharStreamException => ANTLRException => Exception
        => Throwable
CharStreamIOException => CharStreamException =>
        ANTLRException => Exception =>Throwable
FileCopyException => IOException => Exception =>
        Throwable
MismatchedCharException => RecognitionException =>
        ANTLRException => Exception => Throwable
MismatchedTokenException => RecognitionException =>
        ANTLRException => Exception => Throwable
NoViableAltException => RecognitionException =>
        ANTLRException => Exception => Throwable
TokenStreamRetryException => TokenStreamException
        => ANTLRException => Exception=>Throwable
. . .
```

(3) 실험3: 프로그램 슬라이싱

간단한 예제 프로그램에 대해서 예외 관련 문장들만을 추출한 결과는 [그림 7]과 같다. 자바 프로그램의 클래스 헤더와 메소드, 그리고 예외 관련 클래스들이 차례로 출력된다. 이 프로그램 내에는 Propagation_Demo와 Exception_scope 이렇게 두 개의 클래스가 있고, Propagation_Demo 클래스는 Propagation_Demo()와 main() 메소드를 갖고 있으며, Exception_scope은 level3(), level2(), level1() 이라는 메소드를 갖는다. 또한 try-catch 블록이 level1()이라는 메소드 내에 있다. 이렇게 슬라이싱을 이용한 실험에서는 예외와 관련된 프로그램의 구조를 한눈에 알 수 있도록 시각적인 정보를 제공한다.

```
public class Propagation_Demo {
  public Propagation_Demo() {
      super();
    }
  public static void main(String[] args) {
  }
}
class Exception_Scope {
  Exception_Scope() {
      super();
  }
  public void level3(int adjustment) {
  }
  public void level2() {
  }
  public void level1() {
    try {
        this.level2();
      } catch (ArithmeticException problem){
        java.lang.System.out.println(
        problem.getMessage());
        problem.printStackTrace();
      }
    }
}
```

[그림 7] 예외 관련 문장 추출 예

6. 결론 및 향후 연구 방향

자바는 객체 지향에 충실한 언어로 개발자에게 편리함을 주고 있으나, 거대하고 견고한 소프트웨어를 개발함에 있어서 정상적인 명령의 수행을 방해하는 여러 가지 다양한 예외가 발생할 수 있다.

본 논문에서는 프로그램 개발자에게 예외와 그 사용에 대한 좀더 정확한 정보를 제공하기 위해 프로그램 수행 전에 예외 관련 구문들의 사용 형태를 다양한 베치마크 프로그램에 대해서 분석하고 이를 통계 형태로 제공하였다. 또한 예외 클래스들의 계층구조를 보여주고 있으며 예외 관련 문장들을 추출하여 보여주고 있다. 이러한 정보들은 개발자로 하여금 처리되지 않은 예외에 대한 처리기를 다시 설계할 수 있고, 예외를 발생시키는 코드 부분의 디버깅 역시 지원하게 된다.

그러나 예외 처리에 대한 정보로서, 예외가 발생되어 전파되는 과정과 같은 보다 자세한 정보들은 제공하지 못하고 있다. 이러한 과정은 보다 세밀한 정적 분석을 통해서 이루어질 수 있을 것이다. 이를 위한 연구를 현재 진행하고 있다.

참고문헌

[GJS96] J. Gosling, B. Joy, and G. Steele, *The Java Language Specification,* Addison-Wesley, Reading, May, 1996

[BD98] B.Bokowski, M.Dahm, *Poor Man's Generity for Java*, Proceeding of JIT'98, Springer Verlag, 1998

[AW98] David N. Arnow and Gerald Weiss, *Introduction to Programming Using Java*, Addison-Wesley, 1998

[GHJV95] E.Gamma, R.Helm, R.Johnson, J.Vlissides, *Design Patterns Elements of Reuseable Object_Oriented Software*, Addison-Wesly, 1995

[RM99] M. Robillard and G. Murphy,

"Analyzing Exception Flow in Java Programs" in *Proc. of ESEC/FSE '99 Seventh European Sortw. Eng. Conf. and Seventh ACM SIGSOFT Symp. on the Found. of Softw.Eng.* September 1999.

[MT97] Robert Miller and Anand Tripathi, "Issues with excpetion handling in Object-Oriented Systems", *In Proceeding of the 11th European conference on Object-Oriented Programming*, vol.1241 of *Lecture Notes in Computer Science*, pp.85-103, Springer-Verlag, June 1997

[JESP] Rabara G.Ryder, Donald Smith, Ulrich Kremer, Micheal Gordon, and Nirav Shah, "A Static Study of Java Exception using JESP", 9th International Conference on Compiler Construction, Berlin, Germany, March 25-April 2, 2000

[BARAT] The Barat homepage http://www.inf.fu-berin.de/~bokowski/barat

[JTB] K.Tao, J.Palsberg, The Java Tree Builder,http://www.cs.purdue.edu/homes/taokr/jtb/index.html

[JEX] The Jex Static Analysis Tool, http://www.cs.ubs.ca/spider/mrobilla/jex.