An Uncaught Exception Analysis for Java [⋆,⋆⋆]

Jang-Wu Jo [a,*], Byeong-Mo Chang [b], Kwangkeun Yi [c], Kwang-Moo Choe [c]

[a] *Department of Computer Engineering, Pusan University of Foreign Studies, 55-1, Uam-dong, Nam-gu, Pusan, 608-738, Korea*

[b] *Department of Information Science, Sookmyung Women's University, Seoul, Korea*

[c] *Department of Computer Science, Korea Advanced Institute of Science and Technology, Daejon, Korea*

Abstract

Current JDK Java compiler relies on programmer's declarations (by `throws` clauses) for checking against uncaught exceptions of the input program. It is not elaborate enough to remove programmer's unnecessary handlers nor suggest to programmers for specialized handlings (when programmer's declarations are too broad). We propose a static analysis of Java programs that estimates their uncaught exceptions independently of the programmer's declarations. This analysis is designed and implemented based on set-based framework. Its cost-effectiveness is suggested by sparsely analyzing the program at method-level (hence reducing the number of unknowns in the flow equations). We have shown that our interprocedural exception analysis is more precise than JDK-style intraprocedural analysis, and also that our analysis can effectively detect uncaught exceptions for realistic Java programs.

Key words: Java, uncaught exception analysis, class analysis, static analysis, set-based analysis

⋆ Expanded version of a talk presented at the ACM Symposium on Applied Computing (Las Vegas, March 2001).

⋆⋆This work was supported by grant No. 2001-30300-009-2 from the Basic Research Program of Korea Science & Engineering Foundation and by Creative Research Initiatives of the Korean Ministry of Sceience and Technology.

* Corresponding author.

Email addresses: `jjw@taejo.pufs.ac.kr` (Jang-Wu Jo), `chang@cs.sookmyung.ac.kr` (Byeong-Mo Chang), `kwang@cs.kaist.ac.kr` (Kwangkeun Yi), `choe@cs.kasit.ac.kr` (Kwang-Moo Choe).

1 Introduction

Exceptions and exception handling aim to support the development of robust programs with reliable error detection, and fast error handling [12]. Exception mechanism in Java allows the programmer to define, throw and catch exceptional conditions[11]. Programmers have to declare in a method definition any *checked* exception classes which may escape from its body.

Because uncaught exceptions will abort the program's execution, it is important to make sure at compile-time that the input program will have no uncaught exceptions at run-time. The current Java compiler does an *intraprocedural* exception analysis by relying on the declared exceptions of methods, to check that the input program will have no uncaught exceptions at run-time.

The problem is that the current compiler is not elaborate enough to do "better" than as declared by the programmers. It is foreseeable for careless (or inconfident) programmers to excessively declare at methods some exceptions that may not be thrown. In order to use such methods, programmers have to catch or redeclare such exceptions unnecessarily. Similarly programmers can declare exceptions in too broad a sense. Programmers can declare that a method throws exceptions of the general class `Exception` even if the actual exceptions are more specific ones. Then its handler can hardly offer proper treatments specific to the exact classes of actual exceptions.

We propose an *interprocedural* analysis of Java programs based on set-based framework, that estimates uncaught exceptions independently of the declared exceptions. We aim to develop an effective and accurate analysis. First, we design an expression-level analysis that analyzes uncaught exceptions at every expression of input programs. For enhancing cost-effectiveness of analysis, we design sparse analysis that analyzes uncaught exceptions at a larger granularity than at every expression. We prove the soundness and equivalence of accuracy between the two analyses. We implement our exception analysis and JDK-style exception analysis, and evaluate the two analyses by experiments on realistic Java programs. By the experiments, our analysis is shown to detect uncaught exceptions for realistic Java programs more precisely than JDK-style analysis.

The next section gives a motivation of this research using an example. In Section 3, we first define sub-language of Java with exception facilities, and basic concepts about set-based analysis. In Section 4, we design an effective and accurate exception analysis. Section 5 describes our implementation and presents experimental results. We discuss related works in Section 6 and then conclude in Section 7.

```
public class Demo {
1   void proc1() throws IOException {
2     proc2();
3   }
4   void proc2() throws IOException {
5     try {
6       proc3()
7     } catch (IOException e) {
8       ...
9     }
10   }
11   final void proc3() throws IOException {
12     ...
13     throw new FileNotFoundException();
14   }
}
```

```
public class TestExn {
15   public static void main(String args[ ]) {
16     Demo x =3D new Demo();
17     try {
18       x.proc1();
19     } catch (IOException e) {
20       ...
21     }
22   }
}
```

Fig. 1. Java code for broad and unnecessary declarations and catches

2 Motivation

Java programmers must declare uncaught exceptions from each method at its `throws` clause, which might escape from execution of its body. Programmers may declare with broad or unnecessary declaration for *flexibility*. *Careless* (or *inconfident*) programmers may also declare with unnecessary or broad declarations. Because exceptions propagated from called methods must be either caught or declared by a calling method, such a broad or unnecessary declaration forces the calling method to contain broad or unnecessary declaration or `catch` clause. So the method that calls broadly or unnecessarily declared methods has a difficulty to handle exceptions specifically.

The example program in Figure 1 illustrates such a situation. In this program, the method `proc3()` is declared to throw `IOException`, which is broader than the actual exception `FileNotFoundException`. Then its handlers in line 7 of the method `proc2()` have also to be written broadly, so it might not offer proper treatments specific to the actual exceptions. The method `proc2()` contains an unnecessary declaration in line 11, which causes `proc1()`, a caller of `proc2()`, to have an unnecessary declaration in line 1. The unnecessary declaration of `proc1()` also causes `main()` to have an unnecessary `catch` clause in line 19.

The problem is that the current JDK compiler is not elaborate enough to report programmers unnecessary or broad declarations "better" than as declared by the programmers. It is also difficult to suggest to programmers for specialized handlings of exceptions. This is mainly due to the intraprocedural exception analysis of JDK compiler relying on programmers' declaration.

To solve this problem, we will devise interprocedural exception analysis independent of programmer's declaration. So it can not only produce more accurate information on uncaught exceptions but also verify the usages of exceptions more accurately.

For example, our interprocedural analysis can report for the program in Figure 1, that `FileNotFoundException` is uncaught from `proc3()`, and that `proc1()`, `proc2()` and `main()` have no uncaught exception. So, it can inform that `FileNotFoundException` is thrown from `try`-block of `proc2()` in line 5-7 and it is caught broadly in the `catch` clause. It can also report that `throws` declarations of `proc1()` and `proc2()` are unnecessary, and that `catch` clause of `main()` in line 19 is unnecessary as well.

JDK-style analysis can report that the uncaught exception of `proc3()` is `FileNotFoundException`, and that `throws` declaration of `proc2()` is unnecessary. This is the same as our analysis. But it report that `throws` declarations of `proc1()` is necessary and exact(not broad), and that `catch` clause of `main()` in line 19 is also necessary and exact.

3 Preliminaries

3.1 Source Language

For presentation brevity we consider a sub-language of Java with its exception constructs. Its abstract syntax is in Figure 2. A program is a sequence of class definitions. Class bodies consist of field variable declarations and method definitions. A method definition consists of the method name, its parameter, and its body expression. Every expression's result is an object. Assignment expression returns the object of its right hand side expression. Sequence expression returns the object of the last expression in the sequence. A method call returns the object from the method body. The `try` expression

$$\texttt{try}\ e_0\ \texttt{catch}\ (c\ x\ e_1)$$

evaluates e_0 first. If the expression returns a normal object then this object is the result of the `try` expression. If an exception is raised from e_0 and its class is covered by c then the handler expression e_1 is evaluated with the exception object bound to x. If the raised exception is not covered by class c then the raised exception continues to propagate back along the method call chain until

P	::=3D* C	program
C	::=3D `class` c `ext` c `{var`* xM^*`}`	class definition
M	::=3D $m(x)$ =3D e [`thr`*`o`]`ws` c	method definition
e	::=3D 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	::=3D x	method parameter
	| id`.`x	field variable
c		class name
m		method name
x		variable name

Fig. 2. Abstract Syntax of sub-Java

it meets another handler. Note that nested `try` expression can express multiple handlers for a single expression e_0 :

`try (try` e_0 `catch (`$c_1\, x_1\, e_1$`)) catch (`$c_2\, x_2\, e_2$`).`

The exception object e_0 is raised by `throw` e_0. The programmers have to declare at a `throws` clause exceptions which may escape from its body.

Note that exceptions are first-class objects in Java. Like normal objects, they can be defined by classes, instantiated, assigned to variables, passes as parameters, etc. Exception facilities in Java allow the programmer to define, throw and catch exceptional conditions.

We omit the formal semantics of the core language. Its operational semantics should be straightforward, not much different from existing works [8,17].

3.2 Set-Based Analysis

Set-based analysis consists of two phases [13]: collecting set constraints and solving them. The first phase constructs constraints by the construction rules, that describe the data flows between the expressions of the analyzed program. The second phase finds the sets of values that satisfy the constraints by the solving rule. A solution is an assignment from set variables in the constraints to the finite descriptions of such sets of values.

Each set constraint is of the form $X \supseteq se$ where X is a set variable and se is a set expression. The meaning of a set constraint $X \supseteq se$ is intuitive: set X contains the set represented by set expression se. Multiple constraints are conjunctions. We write C for a finite collection of set constraints.

In case of exception analysis, every expression e of the program has a constraint: $X_e \supseteq se$ where $se \rightarrow c \mid X \mid se \cup se \mid se - \{c_1, ..., c_n\}$ where $c_{1 \leq i \leq n}$ are exception class names. The set variable X_e is for incaught exceptions from expression e.

The formal semantics of set expressions is defined by an interpretation I that maps from set expressions to sets of values. We call an interpretation I a *model* (a solution) of a conjunction C of constraints if, for each constraint $X \supseteq se$ in C, $I(X) \supseteq I(se)$.

Our exception analysis is defined to be the least model of constraints. Collected constraints for a program guarantee the existence of its least solution (model) because every operator is monotonic (in terms of set-inclusion) and each constraint's left-hand-side is a single variable [13]. We write $lm(C)$ for the least model of a collection C of constraints.

Our implementation computes the solution by the conventional iterative fixpoint method. Note that the iteration always terminates because our solution space is finite: exception classes in the program. Correctness proofs are done by the fixpoint induction over the continuous functions that are derived from our constraint system [5].

4 Uncaught Exception Analysis

We present our exception analysis based on the set-based framework [13]. We assume class information $Class(e)$ is already available for every expression e. There are several choices for class information. First, we can approximate it using type information, since Java is shown to be type sound [8,17,9]. Second, we can utilize information from class analysis [6,18]. The class analysis estimates for each expression e the classes (including exception classes) that the expression e's normal object belongs to. Note that exception classes are normal classes in Java until they are thrown.

We first present a constraint system that analyzes uncaught exceptions from *every* expression of the input program. Because exception-related expressions are sparse in programs, generating constraints for every expression is wasteful. The analysis cost-effectiveness need to be addressed by enlarging the analysis granularity. Hence we present a sparse constraint system that analyzes uncaught exceptions at a larger granularity. Our analysis result is the solution of this sparse constraints. We show the soundness of the method-level analysis and equivalence of accuracy between the two analyses. We discuss the complexity of exceptions analysis.

4.1 Exception Analysis at Expression-Level

Figure 3 has the rules to generate set constraints for the uncaught exceptions from *every* expression. For exception analysis, every expression e of the program has a constraint: $X_e \supseteq se$. The X_e is a set-variable for the uncaught exceptions from expression e. The subscript e of set variables X_e denotes the current expression to which the rule applies. The relation "$\triangleright e : C$" is read "constraints C are generated from expression e."

Consider the rule for `throw` expression:

$$[\text{Throw}]\ \frac{\triangleright e_1 : C_1}{\triangleright\, \texttt{throw}\, e_1 : \{X_e \supseteq Class(e_1) \cup X_{e_1}\} \cup C_1}$$

It throws an exception $Class(e_1)$ or, prior to throwing, it can have uncaught exceptions from inside e_1 too.

Consider the rule for `try` expression:

$$[\text{Try}]\ \frac{\triangleright e_0 : C_0 \quad \triangleright e_1 : C_1}{\triangleright\, \texttt{try}\, e_0\, \texttt{catch}\ (c_1\, x_1\, e_1) : \{X_e \supseteq (X_{e_0} - \{c_1\}^*) \cup X_{e_1}\} \cup C_0 \cup C_1}$$

Thrown exceptions from e_0 can be caught only when they are covered by a class c_1. After this catching, exceptions can also be raised during the handling inside e_1. Hence, $X_e \supseteq (X_{e_0} - \{c_1\}^*) \cup X_{e_1}$, where $\{c\}^*$ represents the class c and all the subclasses of it.

[New] $$\triangleright \texttt{new}\, c : \emptyset$$

[FieldAss] $$\frac{\triangleright e_1 : C_1}{\triangleright id.x :=3D\ e_1 \{X_e \supseteq X_{e_1}\} \cup C_1}$$

[ParamAss] $$\frac{\triangleright e_1 : C_1}{\triangleright x\ \texttt{:=((}_1 e \{X_e \supseteq X_{e_1}\} \cup C_1}$$

[Seq] $$\frac{\triangleright e_1 : C_1 \quad \triangleright e_2 : C_2}{\triangleright e_1 \texttt{;} e_2 : \{X_e \supseteq X_{e_1} \cup X_{e_2}\} \cup C_1 \cup C_2}$$

[Cond] $$\frac{\triangleright e_0 : C_0 \quad \triangleright e_1 : C_1 \quad \triangleright e_2 : C_2}{\triangleright \texttt{if}\, e_0\, \texttt{then}\, e_1\, \texttt{else}\, e_2 : \{X_e \supseteq X_{e_0} \cup X_{e_1} \cup X_{e_2}\} \cup C_0 \cup C_1 \cup C_2}$$

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

[Throw] $$\frac{\triangleright e_1 : C_1}{\triangleright \texttt{throw}\, e_1 : \{X_e \supseteq Class(e_1) \cup X_{e_1}\} \cup C_1}$$

[Try] $$\frac{\triangleright e_0 : C_0 \quad \triangleright e_1 : C_1}{\triangleright \texttt{try}\, e_0\, \texttt{catch}\ (c_1\, x_1\, e_1) : \{X_e \supseteq (X_{e_0} - \{c_1\}^*) \cup X_{e_1}\} \cup C_0 \cup C_1}$$

[MethCall] $$\frac{\triangleright e_1 : C_1 \quad \triangleright e_2 : C_2}{\begin{array}{c}\triangleright\ e_1 \texttt{.} m \texttt{(} e_2 \texttt{)} : \{X_e \supseteq X_{c.m} | c \in Class(e_1), m(x)\ =3D_m e \in c\} \\ \cup \{X_e \supseteq X_{e_1} \cup X_{e_2}\} \cup C_1 \cup C_2\end{array}}$$

[MethDef] $$\frac{\triangleright e_m : C}{\triangleright m \texttt{(} x \texttt{)}\ \texttt{=((}_m e\texttt{:} \{X_{c.m} \supseteq X_{e_m}\} \cup C}$$

[ClassDef] $$\frac{\triangleright m_i : C_i,\ i =3D\ 1, \cdots, n}{\triangleright \texttt{class}\ c\ \texttt{=((}\ \texttt{\{var}_1 x \cdots, x_k, m_1, \cdots, m_n\texttt{\}} : C_1 \cup \cdots \cup C_n}$$

[Program] $$\frac{\triangleright C_i : C_i,\ i =3D\ 1, \cdots, n}{\triangleright C_1, \cdots, C_n : C_1 \cup \cdots \cup C_n}$$

Fig. 3. Exception Analysis at Expression-Level

Consider the rule for method call:

[MethCall] $$\frac{\triangleright e_1 : C_1 \quad \triangleright e_2 : C_2}{\begin{array}{c}\triangleright\ e_1 \texttt{.} m \texttt{(} e_2 \texttt{)} : \{X_e \supseteq X_{c.m} | c \in Class(e_1), m(x)\ \texttt{=3(}_m e \in c\} \\ \cup \{X_e \supseteq X_{e_1} \cup X_{e_2}\} \cup C_1 \cup C_2\end{array}}$$

Uncaught exceptions from the call expression first include those from the subexpressions e_1 and e_2 : $X_e \supseteq X_{e_1} \cup X_{e_2}$. The method $m(x)\ \texttt{=3(}_m e$is the one defined inside the classes $c \in Class(e_1)$ of e_1's objects. Hence, $X_e \supseteq X_{c.m}$ for uncaught exceptions. (The subscript $c.m$ indicates the index for the body expression of class c's method m.)

$[\mathrm{New}]_m$ $$m \rhd \texttt{new}\, c : \emptyset$$

$[\mathrm{FieldAss}]_m$ $$\frac{m \rhd e_1 : C_1}{m \rhd id\,.\,x \;:=3\mathrm{D}\; e_1 : C_1}$$

$[\mathrm{ParamAss}]_m$ $$\frac{m \rhd e_1 : C_1}{m \rhd x \;\texttt{:=((}\; e_1 : C_1}$$

$[\mathrm{Seq}]_m$ $$\frac{m \rhd e_1 : C_1 \quad m \rhd e_2 : C_2}{m \rhd e_1\,;e_2 : C_1 \cup C_2}$$

$[\mathrm{Cond}]_m$ $$\frac{m \rhd e_0 : C_0 \quad m \rhd e_1 : C_1 \quad m \rhd e_2 : C_2}{m \rhd \texttt{if}\, e_0\, \texttt{then}\, e_1\, \texttt{else}\, e_2 : C_0 \cup C_1 \cup C_2}$$

$[\mathrm{FieldVar}]_m$ $$\frac{m \rhd id : C_{id}}{m \rhd id.x : C_{id}}$$

$[\mathrm{Throw}]_m$ $$\frac{m \rhd e_1 : C_1}{m \rhd \texttt{throw}\, e_1 : \{X_m \supseteq Class(e_1)\} \cup C_1}$$

$[\mathrm{Try}]_m$ $$\frac{m \rhd e_g : C_g \quad m \rhd e_1 : C_1}{m \rhd \texttt{try}\, e_g\, \texttt{catch}(c_1\; x_1\; e_1) : \{X_m \supseteq (X_g - \{c_1\}^*)\} \cup C_g \cup C_1}$$

$[\mathrm{MethCall}]_m$ $$\frac{m \rhd e_1 : C_1 \quad m \rhd e_2 : C_2}{\begin{array}{c} m \rhd e_1.m'(e_2) : \\ \{X_m \supseteq X_{c.m'} | c \in Class(e_1), m'(x) =3\mathrm{D}\; e_{m'} \in c\} \cup C_1 \cup C_2 \end{array}}$$

$[\mathrm{MethDef}]_m$ $$\frac{m \rhd e_m : C_m}{m \rhd m(x) =3\mathrm{D}\; e_m : C_m}$$

$[\mathrm{ClassDef}]_m$ $$\frac{m_i : C_i,\ i =3\mathrm{D}\ 1, \cdots, n}{\begin{array}{c} \texttt{class}\ c =3\mathrm{D}\ \{\ \mathrm{var}\ x_1, \cdots, x_k, m_1, \cdots, m_n\ \} : \\ C_1 \cup \cdots \cup C_n \end{array}}$$

$[\mathrm{Program}]_m$ $$\frac{\rhd C_i : C_i,\ i =3\mathrm{D}\ 1, \cdots, n}{\rhd C_1, \cdots, C_n : C_1 \cup \cdots \cup C_n}$$

Fig. 4. Exception Analysis at Method-Level

4.2 Exception Analysis at Method-Level

In our sparse constraint system, only two groups of set variables are considered: set variables for methods and `try`-blocks. The number of unknowns is thus proportional only to the number of methods and `try` blocks, not to the total number of expressions. For each method m, X_m is a set-variable for uncaught exceptions from method m. The try-block e_g in `try` e_g `catch` $(c\ x\ e)$ also has a set variable X_g, which is for uncaught exceptions from e_g.

Figure 4 shows this new constraint system. The left-hand-side m in relation $m \rhd e\ : C$ indicates that the expression e is a sub-expression of a method m (or a try-block g).

Consider the rule for `throw` expression:

$$[\text{Throw}]_m \ \frac{m \ \rhd \ e_1 \ : C_1}{m \ \rhd \ \texttt{throw} \ e_1 \ : \{X_m \supseteq Class(e_1)\} \cup C_1}$$

The set variable X_m for uncaught exceptions from method m includes the thrown exception $Class(e_1)$.

Consider the rule for `try` expression:

$$[\text{Try}]_m \ \frac{g \ \rhd \ e_g \ : C_g \quad m \ \rhd \ e_1 \ : C_1}{m \ \rhd \ \texttt{try} \ e_g \ \texttt{catch} \ (c_1 \, x_1 \, e_1) \ : \{X_m \supseteq X_g - \{c_1\}^*\} \cup C_g \cup C_1}$$

Some of the uncaught exceptions X_g from e_g can be caught, if the exceptions are covered by a class c. Hence the uncaught exceptions from this expression includes the uncovered ones.

Consider the rule for method-call expression:

$$[\text{MethCall}]_m \ \frac{m \rhd e_1 : C_1 \quad m \rhd e_2 : C_2}{\begin{array}{c} m \rhd e_1.m'(e_2) : \\ \{X_m \supseteq X_{c.m'} | c \in Class(e_1), m'(x) =3\mathrm{D}_{m'} e \in c\} \cup C_1 \cup C_2 \end{array}}$$

Thus, if m's body has a method call e_1`.`m'`(`e_2`)`, uncaught exceptions X_m from the method m include $X_{c.m'}$ which is a set variable for uncaught exceptions from the called method $c.m'$.

It should be noted that the derivation rules for try-blocks, for example e_g, are the same as those in Figure 4, except that m is replaced by g.

The least model of the sparse constraints C, which are derived ($\rhd \, pgm : C$) from an input program pgm is our analysis result. The solutions for X_m has the exceptions which might be escape from m's execution.

4.3 Soundness and Equivalence

We have designed the method-level exception analysis from the expression-level analysis of Figure 3. In order to prove the soundness of the method-level analysis and the equivalence of accuracy between the two analyses, we first need to relate the expression-level analysis to the method-level analysis.

To relate the expression-level exception analysis to the method-level exception analysis, we can define an index determination function $\pi : Expr \rightarrow Expr \cup Method$ as follows :

$$\pi(e) = 3D \begin{cases} g, & \text{if } e \text{ is within a try-block } e_g \text{ in } \texttt{try}\ e_g\ \texttt{catch}(c_1\ x_1\ e_1) \\ m, & \text{if } e \text{ is within a method } m. \end{cases}$$

This index function specifies that there is one set variable X_g for all sub-expressions of a try-block e_g, and one set-variable X_m for all sub-expressions of a method m, not of a try-block.

In the following, we assume that C is the collection of set constraints for a program *pgm* constructed by the rules in Figure 3, and C_π is the collection of set constraints for the same program *pgm* constructed by the rules in Figure 4.

The least model of the method-level constraints C_π is a sound approximation of that of the original constraints C. The proof is based on the observation in [5] that the least model $lm(C)$ is equivalent to the least fixpoint of the continuous function F derived from C.

Theorem 1 (Soundness) *$lm(C_\pi)(\pi(X)) \supseteq lm(C)(X)$ for every set variable X in C.*
Proof. *See Appendix A.* □

We show that the method-level analysis gives, for every method and try-block, the same information on uncaught exceptions as the expression-level analysis. We call C_π is *equivalent* to C with respect to every method and try-block : if $lm(C_\pi)(X_f)$ =3D $lm(C)(X_f)$ for every method and try-block f.

Theorem 2 (Equivalence) *$lm(C_\pi)(X_f)$ =3D $lm(C)(X_f)$ for every method and try-block f.*
Proof. *See Appendix A.* □

Our exception analysis consists of two phases:collecting constraints and solving them. The complexity of the first phase is $O(n)$ where n is the number of expressions, because we make one set-constraint for every expression. The complexity of the second phase is $O(n * m^2)$ where n is the number of set variables and m is the number of methods. In case of the expression-level analysis, because it makes one set variable for every expression, n is the number of expressions. In case of the method-level analysis, because it makes one set variable for every method and `try`-block, n is the number of methods and `try`-blocks. The method-level analysis does not change the order of complexity itself, but the number n of set-variables is reduced to the number of methods and `try`-blocks, which is much smaller than the number of expressions.

5 Experiments

This section shows experiment numbers of our exception analysis and compares them against JDK-style exception analysis on realistic Java benchmark programs (Table 1). Our experiments are done in the following way. First, we implemented our analysis and JDK-style analysis. Second, we collected analysis information from the benchmark Java programs by applying each analysis. Third, with the collected analysis information, we compared the two analyses with respect to the precision of analysis.

5.1 Implementations

We implemented the two exception analyses; our method-level exception analysis and JDK-style exception analysis. The reason of implementing method-level exception analysis is that we have proven the equivalence of accuracy between the expression-level and method-level analyses. The JDK-style exception analysis is the same as ours in Figure 4 except for the method call case. Since JDK-style analysis depends on `throws` clauses, the rule for method call can be defined as follows:

$$[\text{MethCall}]_m \; \frac{m \triangleright e_1 : C_1 \quad m \triangleright e_2 : C_2}{\begin{array}{c} m \triangleright e_1.m'(e_2) : C_1 \cup C_2 \cup \\ \{X_m \supseteq T_{c.m'} \mid c \in Class(e_1), m'(x) =3D_{m'} e \in c\} \\ \text{where } T_{c.m'} \text{ means exceptions declared by the } \texttt{throws} \text{ clause} \\ \text{of the method } c.m. \end{array}}$$

Our implementation relies on the Barat framework [29] for the type information of $Class(e)$. Barat is a front-end for Java, which builds an abstract syntax tree from Java source files, enriched with type and name analysis information, and also provides the interfaces for traversing the abstract syntax trees.

We implemented the two analyses in Java. Each of them consists of two passes. The first pass traverses the input Java program and sets up set-constraints. The second pass solves the generated set-constraints by the conventional iterative fixpoint method.

In the implementation, we take the following into consideration.

Java language Even if we present our exception analysis for sub-Java programs in Figure 2, we considered the full Java in the implementation, which includes object-allocations, explicit constructor calls, interfaces, abstract methods, and nested classes.

programs	description
jasmin [30]	Java assembler interface
JavaSim [31]	Discrete event process-based simulation package
jb [32]	Java parser generator using the Gnu Bison
javacup [33]	LALR parser generator for Java
sablecc [34]	Framework for generating compilers and interpreters
jel [35]	Compiler for simple expressions into Java byte code
JFlex [36]	Lexical analyzer generator
antlr [37]	Framework for compiler construction
com.ice.tar [38]	Utility for UNIX tar archives
org.apache.tomcat [39]	Servlet container and JSP implementation
org.apache.jasper [39]	Component of Tomcat that compiles and executes JSP pages
jess [40]	Expert system shell based on NASA's CLIP
soot [41]	Framework for Java optimization

Table 1

List of benchmark Java programs

The exceptions analyzed Java distinguishes between *checked* and *unchecked* exceptions. The unchecked exceptions are subclasses of `RunTimeException` and `Error`, and are exempt from the requirement of being declared. We consider checked exceptions only, because including unchecked exceptions can generate too much information, which affects the usability of our analysis.

How to handle libraries Applications can also use libraries, which may have no source code. In case no source code is available, our analysis also depends on the `throws` delcarations as in JDK compiler, since we can assume that exceptions of libraries are usually well declared.

5.2 Experimental results

All experiments are conducted on a Compaq Armarda M700 (Pentium III 700 MHz uni-processor with 256MB of main memory), running Windows 2000 professional, and using the JVM of the SUN JDK1.3.0. The Java applications used in our experiments were obtained from several sources. Our collection of programs covers a wide range of application areas, including language processors, a compression utility, an aritificial intelligence system, a simulation utility, and a servlet container. Table 1 provides a brief description of each of the applications used in our experiments. Table 2 shows the number of classes, interfaces, methods for each benchmark, and the number of exception-related constructs including `try`, `catch`, and `throws`. In performing experiments, we have made no effort to modify the applications for experiments.

Our analysis and JDK-style analysis provide the same kind of information: uncaught exceptions from each method and each `try`-block. But these two

Programs	# classes	# interfaces	# methods	# `try`	# `catch`	# `throws`	Kbytes
jasmin	11	0	77	2	4	39	107.0
JavaSim	29	0	207	16	18	83	72.6
jb	52	0	515	19	21	116	154.3
javacup	45	0	333	9	9	115	315.7
sablecc	280	4	1,753	78	77	11	612.3
jel	47	11	385	100	107	57	341.2
JFlex	51	4	395	20	26	31	461.5
antlr	156	31	1,886	76	87	292	1,047.8
com.ice.tar	10	2	127	17	21	40	88.3
tomcat	189	15	2,033	209	237	353	1,232.0
jasper	98	16	703	80	91	309	591.9
jess	235	11	1,085	115	148	436	548.9
soot	1,063	182	8,923	33	39	61	3,746.1

Table 2
Syntactic properties of benchmark Java programs (the number of classes, interfaces, methods, and exception-related constructs)

analyses differ on the precision because ours is interprocedural and JDK-style analysis is intraprocedural. We have compared the two analyses by the benchmark programs in terms of the following measures:

- # exceptions declared by `throws`: the number of exceptions declared by `throws`
- # exceptions declared by `catch`: the number of exceptions declared by `catch`
- # exact `throws`: the number of exceptions declared by `throws`, which exactly correspond to the analysis result of its method.
- # exact `catch`: the number of exceptions declared by `catch`, which exactly correspond to the analysis result of its `try`-block.
- # broad `throws`: the number of exceptions whose higher classes are declared by `throws` than the analysis result of its method.
- # broad `catch`: the number of exceptions whose higher classes are declared by `catch` than the analysis result of its `try`-block.
- # unnecessary `throws`: the number of exceptions declared by `throws`, but not included in the analysis result of its method.
- # unnecessary `catch`: the number of exceptions declared by `catch`, but not included in the analysis result of its `try`-block.
- the analysis time

With the analysis result of each method, we have detected exact `throws`, broad `throws` and unnecessary `throws`, and compared our analysis with JDK-style analysis in Table 3.

Our analysis has detected more unnecessary `throws` than JDK-style analysis. This is thanks to the following reason. In case there is an unnecessary `throws`

Programs	# exceptions declared by `throws`	# exact `throws`		# broad `throws`		# unnecessary `throws`	
		JDK	Ours	JDK	Ours	JDK	Ours
jamin	42	38	38	1	1	3	3
JavaSim	119	84	80	0	0	35	39
jb	109	68	52	1	1	40	56
javacup	114	108	105	4	6	2	3
sablecc	18	17	16	0	0	1	2
jel	72	62	60	2	3	8	9
JFlex	28	24	21	0	3	4	4
antlr	576	446	139	64	306	66	131
com.ice.tar	40	35	34	0	0	5	6
tomcat	338	183	121	17	30	138	187
jasper	397	246	188	24	35	127	174
jess	432	395	378	1	1	36	53
soot	65	39	36	1	1	25	28

Table 3
Uncaught exceptions from methods and their declarations

at some method, the caller method should either catch or redeclare it. JDK-style analysis can not detect this unnecessary operation becasue it depends soley on programmer's `throws` declarations. On the other hand our analysis estimates program's flows and can detect the unnecessary situations. For example, `IOException` declared at the method `proc2()` in Figure 1 is unnecessary and so is `IOException` declared at the method `proc1()`. Our analysis can detect, with the uncaught exception information from the method `proc2()`, that `IOException` declared at the method `proc1()` is unnecessary, while JDK-style analysis can not detect this unnecessary declaration becasue it depends on `throws` declarations at the method `proc2()`.

Similarly our analysis detected more broad `throws` than JDK-style analysis did.

With the analysis result of each `try`-block, we have detected unnecessary `catch`, broad `catch`, and exact `catch`. Our analysis is compared with JDK-style analysis in Table 4.

Table 4 is similar to Table 3 except that it compares the analysis results of the two analyses on `try` blocks and `catch` clauses. Our analysis has detected more unnecessary `catch` and broad `catch` than JDK-style analysis did. These results come from the same reason as the experiment on `throws` declarations in Table 3.

These experiment results indicate that our analysis can guide programmers to declare or catch exceptions more precisely.

Table 5 compares the two analyses in terms of analysis time for each of the

Programs	# exceptions declared by `catch`	# exact `catch`		# broad `catch`		# unnecessary `catch`	
		JDK	Ours	JDK	Ours	JDK	Ours
jamin	4	3	3	0	0	1	1
JavaSim	18	11	11	1	1	6	6
jb	21	1	1	4	4	16	16
javacup	9	6	6	2	2	1	1
sablecc	77	69	68	2	2	6	7
jel	107	46	41	58	58	36	39
JFlex	26	7	7	1	1	18	18
antlr	87	68	40	11	34	8	13
com.ice.tar	21	16	16	2	2	3	3
tomcat	237	121	105	60	48	56	84
jasper	91	46	43	31	33	14	15
jess	148	100	99	18	18	30	31
soot	39	25	25	5	3	9	11

Table 4
uncaught exceptions from `try`-blocks and their handlings

benchmarks. We measured analysis time using `System.currentTimeMillis()`. The analysis time consists of constraint set-up and solving time. The analysis time of ours is comparable to that of JDK-style analysis. As for the constraint set up time, JDK-style approach is a little slower than ours. A possible explanation is that, for every method-call, JDK-style approach must traverse abstract syntax tree to visit `throws` clause of a called method, while our approach simply uses a new set variable without traversing abstract syntax tree. As for the solving time, however, JDK-style approach is up to 3 times faster than ours. This is because the right hand sides of set-constraints have no set variables except for that of catch clause, so the number of iterations for computing the solution does not exceed three. Ours, however in some cases, may iterate more than ten times corresponding to the depth of method call chains.

6 Related Works

Ryder and colleagues [21] and Sinha and Harrold [24] conducted a study of the usage patterns of exception-handling constructs in Java programs. Their study offers an evidence to support our belief that exception-handling constructs are used frequently in Java programs and more accurate exception information is necessary.

In Java[11], the JDK compiler ensures by an intraprocedural analysis that clients of a method either handle the exceptions declared by that method, or explicitly declare them. It is useful for enhancing modularity and code reuse to declare exceptions more broadly. However, it makes more specific

Programs	JDK			Ours		
	Setup	Solving	Total	Setup	Solving	Total
jamin	1.823	0.050	1.873	0.891	0.130	1.021
JavaSim	1.823	0.180	2.003	1.111	0.250	1.361
jb	2.353	0.040	2.393	1.572	0.100	1.672
javacup	2.273	0.050	2.323	1.062	0.33	1.392
sablecc	4.577	0.310	4.876	3.645	1.722	5.367
jel	5.608	0.471	6.079	3.234	2.413	5.647
JFlex	1.722	0.060	1.782	1.432	0.250	1.682
antlr	5.156	1.502	6.658	4.987	4.036	9.023
com.ice.tar	1.271	0.110	1.381	0.741	0.451	1.192
tomcat	11.456	2.223	13.679	8.803	14.171	22.974
jasper	10.214	1.131	11.345	5.718	3.775	9.493
jess	13.279	1.502	33.898	4.797	6.399	11.196
soot	26.698	0.200	26.898	18.076	0.732	18.808

Table 5
Analysis Time

handling of exceptions difficult. Our interprocedural analysis provides more exact uncaught exception information for every method and `try`-block. So, our analysis can guide programmers to handle exceptions more specifically and to declare them more exactly.

Robillard and Murphy [20] have developed Jex: A tool for analyzing exception flows in Java, which is similar to our work. They describe a tool that extracts the flow of exceptions in Java programs, and generates views of the exception structure. They include checked and unchecked exceptions. Jex is designed at expression-level because it aims for programmers to determine the uncaught exceptions at any point in the program. Because of these reasons, scalability of Jex is not certain.

We have designed analysis at method-level considering scalability, and shown its equivalence to the expression-level analysis. We have compared our analysis with JDK-style analysis by experiments on large realistic Java programs, and have shown that our analysis is able to detect uncaught exceptions, unnecessary `catch` and `throws` clauses effectively.

Several exception analyses have been introduced by Yi to trace uncaught exceptions in ML [25–28] based on abstract interpretation and set-based framework. Fähndrich and Aiken [10] have applied their BANE toolkit to the analysis of uncaught exceptions in SML. Their system is based on equality constraints to keep track of exception values. Fessaux and Leroy desiged an exception analysis for OCaml based on type and effect systems, and provides good performance for real OCaml programs [19].

Recently, several researchers have considered the effects of exception-related constructs on various types of analyses. Choi and colleagues [4] describe a representation called the factored control-flow graph (FCFG) to incorporate exceptional control flow, and use the representation to improve intraprocedural optimization in the presence of exceptions. Sinha and Harold [24] discusses the effects of exception-handling constructs on several analyses such as control-flow, data-flow, and control dependence analysis. They present techniques to construct representations for programs with checked exception and exception-handling constructs. Chatterjee and Ryder [2,3] describe an approach to performing points-to and data-flow analyses that incorporate exceptional control flow.

Failure to account for the effects of exception in performing analyses can result in incorrect analysis information. But these efforts differ from our work in that they focus on modelling program execution in the presence of exception rather than on enabling developers to make better use of exception mechanisms.

7 Conclusions

We have presented an exception analysis for Java, that estimates their uncaught exceptions independently of the programmer's declarations. We have designed two exception analyses at expression-level and at method-level, and have proven that the method-level exception analysis gives the same analysis result as the expression-level analysis. This situation is because we only consider uncaught exceptions from each method and `try`-block. By implementation and experiments, we have shown that our exception analysis is more precise than JDK-style analysis, and that our exception analysis can effectively detect uncaught exceptions for realistic Java programs.

Java programmers can apply our exception analysis to better use of exception mechanism. Our exception analysis provides uncaught exceptions for each method, and also provides uncaught exceptions for each `try`-block. In case of writing `catch` clauses, Java programmers can be guided to handle specific exceptions by using the uncaught exceptions for the corresponding `try`-block. The uncaught exceptions for each method also can be used for specifying `throws`-clauses.

References

[1] B.-M. Chang, K. Yi and J. Jo, Constraint-based analysis for Java, SSGRR 2000 Computer and e-Business Conference, August 2000, L'Aquila, Italy.

[2] R. K. Chatterjee, B. G. Ryder, and W. A. Landi, "Complexity of concrete type-inference in the presence of exceptions," *Lecture notes in Computer Science*, vol. 1381, pp. 57-74, Apr. 1998.

[3] R. K. Chatterjee and B. G. Ryder, "Data-flow-based testing of object-oriented libraries," Tech. Rep. DCS-TR-382, Rutgers University, Mar. 1999.

[4] J.-D. Choi, D. Grove, M. Hind, and V. Sarkar, "Efficient and precise modeling of exceptions for analysis of Java programs," in *Proceeding of* PASTE '99 ACM SIGPLAN-SIGSOFT *Workshop on Program Analysis for Software Tools and Engineering,* September 1999, pp. 21-31.

[5] Patrick Cousot and Radhia Cousot. Compositional and inductive semantic definitions in fixpoint, equational, constraint, closure-condition, rule-based and game-theoretic form. In *Lecture Notes in Computer Science*, volume 939, pages 293–308. Springer-Verlag, proceedings of the 7th international conference on computer-aided verification edition, 1995.

[6] G. DeFouw, D. Grove, and C. Chambers, Fast interprocedural class analysis, *Proceedings of 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* pages 222-236, Januaray 1998.

[7] C. Dony, Exception Handling and object-oriented programming: Towards a synthesis, *Proceedings of ACM Conference on OOPSLA*, pp. 322-330, 1990.

[8] S. Drossopoulou, and S. Eisenbach, Java is type safe-probably, Proceedings of 97 ECOOP, 1997

[9] S. Drossopoulou, and T. Valkevych, Java type soundness revisited. Techical Report, Imperial College, November 1999. Also available from: http://www-doc.ic.ac.uk/ scd.

[10] M. Fähndrich, J.S. Foster, A. Aiken,a nd J. Cu. Tracking down exceptions in Standard ML programs. Techical report, University of California at Berkeley, Computer Science Division, 1998.

[11] James Gosling, Bill Joy, and Guy Steele, *The Java Programming Language Specification*, Addison-Wesley Longman, 1996.

[12] J. Goodenough, Exception Handling: Issues and proposed notation. *Communications of the ACM*, 18(12), December 1975.

[13] N. Heintze, Set-based program analysis. Ph.D thesis, Carnegie Mellon University, October 1992.

[14] R. Miller and A. Tripathi, Issues with exception handling in object-oriented systems, In *Proceedings of 97 ECOOP*, 1997

[15] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1997.

[16] F. Nielson and H. R. Nielson. Infinitary Control Flow Analysis: A Collecting Semantics for Closure Analysis. *Proceedings of 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* , pages 332-345, January 1997.

[17] T. Nipkow, D. V. Oheimb Java is type safe-definitely, *Proceedings of 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*,January 1998.

[18] J. Palsberg and M. I. Schwarzbach, Object-oriented type inference, Proceedings of ACM Conference on OOPSLA, pp. 141-161, 1991.

[19] F. Pessaux and X. Leroy, Type-based analysis of uncaught exceptions. *Proceedings of 26th ACM Conference on Principles of Programming Languages*, January 1999.

[20] M. P. Robillard and G. C. Murphy, Analyzing exception flow in Java programs, in *Proc. of ESEC/FSE '99 Seventh European Softw. Eng. Conf. and Seventh* ACM SIGSOFT *Symp. on Notes in Computer Science*, pp. 322-337, Springer-Verlag.

[21] B. G. Ryder, D. Smith, U. Kremer, M. Gordon, and N. Shah, "A static study of Java exceptions using JESP," Tech. Rep. DCS-TR-403, Rutgers University, Nov. 1999.

[22] O. Shivers. Control-flow analysis in Scheme. *Proceedings of the SIGLAN '88 Conference on Programming Language Design and Implementation*, pages 164-174, June 1988

[23] O. Shivers. Control-flow analysis of higher-order languages. Ph.D thesis, Carnegie-Mellon University, May 1991

[24] S. Sinha and M. Harrold, Analysis and Testing of Programs With Exception-Handling Constructs, *IEEE Trans. SE.* **26-9.** (2000)

[25] Kwangkeun Yi. Compile-time detection of uncaught exceptions in standard ML programs. In *Lecture Notes in Computer Science*, volume 864, pages 238-254. Springer-Verlag, Proceedings of the 1st Static Analysis Symposium, September 1994.

[26] Kwangkeun Yi and Sukyoung Ryu. Towards a cost-effective estimation of uncaught exceptions in SML programs. In *Lecture Notes in Computer Science*, volume 1302, pages 98–113. Springer-Verlag, Proceedings of the 4th Static Analysis Symposium, September 1997.

[27] Kwangkeun Yi and Sukyoung Ryu. SML/NJ Exception Analysis version 0.98. `http://compiler.kaist.ac.kr/pub/exna/`, December 1998.

[28] Kwangkeun Yi and Sukyoung Ryu. A Cost-effective Estimation of Uncaught Exceptions in Standard ML Programs. *Theoretical Computer Science* **277-1.** (2002).

[29] http://www.sharemation.com/~bokowski/barat/index.html

[30] http://mrl.nyu.edu/~meyer/jvm/jasmin.html

[31] http://javasim.ncl.ac.uk/

[32] http://www.cs.colorado.edu/serl/misc/jb.html

[33] http://www.cs.princeton.edu/~appel/modern/java/CUP/

[34] http://www.sablecc.org/

[35] http://www.gnu.org/directory/jel.html

[36] http://jflex.sourceforge.net/

[37] http://www.antlr.org

[38] http://www.trustice.com/java/tar/

[39] http://jakarta.apache.org/tomcat/tomcat-3.2-doc/index.html

[40] http://herzberg.ca.sandia.gov/jess/

[41] http://www.sable.mcgill.ca/soot/

Appendix A. Proofs

Theorem 1 *Proof.* As in [5], the continuous function F can be defined from C, and F_π can also be defined from C_π likewise. So, we will prove this theorem by showing $\gamma \circ lfp(F_\pi) \supseteq lfp(F)$.

We can prove this by showing that :
(1) Galois insertion: Let Δ =3D $Vars(C)$ and Δ_π =3D $Vars(C_\pi)$. Let D =3D $\Delta \to \wp(Val)$ be the domain of interpretations I and D_π =3D $\Delta_\pi \to \wp(Val)$ be the domain of partitioned interpretations I_π. For every interpretation I, we define $\alpha(I)$ =3D I_π where $I_\pi : \Delta_\pi \to \wp(Val)$ is defined as $(I_\pi)(X_m)$ =3D $\sqcup_{e \in m} I(X_e)$ for every method $m \in \Delta_\pi$. We define $\gamma(I_\pi)$ =3D I' such that $I'(X_e)$ =3D $I_\pi(X_{\pi(e)})$ for every set variable $X_e \in \Delta$. Then, $(D, \alpha, D_\pi, \gamma)$ is a Galois insertion, since $\alpha(\gamma(I_\pi))$ =3D I_π.
(2) Soundness of the operation $\gamma \circ F_\pi(I_\pi) \supseteq F \circ \gamma(I_\pi)$:
For this proof, it should be noted as in [1] that the derivation rules in Figure 4 can be obtained by replacing every set variable X_e by $X_{\pi(e)}$ in the correspoinding rules in Figure 3. So, if there is a constraint $X_e \supseteq se$ constructed by the rules in Figure 3, then there must be a constraint $X_{\pi(e)} \supseteq se/\alpha D\pi$ constructed by the rules in Figure 4, where $se/\alpha D\pi$ denotes se with its set variable $X_{e'}$ replaced by $X_{\pi(e')}$.

Let the function F be defined as a collection of equations of the form : X_e =3D se for every $X_e \in \Delta$, and F_π as a collection of equations of the form : $X_{\pi(e)}$ =3D $se/\alpha D\pi$ for every $X_{\pi(e)} \in \pi(\Delta)$. Assume that, for each set variable $X_{e'}$ in se, $\gamma(I_\pi)(X_{e'})$ =3D S. Then $I_\pi(X_{\pi(e')})$ =3D S by the definition of γ. $X_{e'}$ is replaced by

$X_{\pi(e')}$ in $X_{\pi(e)}$ =3D $se/\alpha D\pi$, and F every set expression is monotone. Therefore, $F_\pi(I_\pi)(X_{\pi(e)}) \supseteq F \circ \gamma(I_\pi)(X_e)$ for every set variable X_e, and $\gamma \circ F_\pi(I_\pi) \supseteq F \circ \gamma(I_\pi)$ by the definition of γ. $\square$

Theorem 2 *Proof.* As in the soundness proof, the continuous function F and F_π can be defined. We prove this theorem by showing that $lfp(F_\pi)(X_f)$ =3D $lfp(F)(X_f)$ for every method and try-block f. By the soundness theorem, $lfp(F_\pi)(X_f) \supseteq lfp(F)(X_f)$. So, we just prove that $lfp(F_\pi)(X_f) \subseteq lfp(F)(X_f)$ for every method and try-block f.

The proof is by induction on the number of iterations in computing $lfp(F_\pi)$. *Induction step* : Suppose $I_\pi(X_f) \subseteq I(X_f)$ for every method and try-block f. Let I'_π =3D$F_\pi(I_\pi)$. Then there exists I' such that I' =3D$F^i(I)$ for some i and $I'_\pi(X_f) \subseteq I'(X_f)$ for every method and try-block f.
(1) For every set variable X_f, suppose $I'_\pi(X_f)$ =3D$I_\pi(X_f) \cup \alpha$.
(2) Then, α must be added by some of the rules $[\text{Throw}]_m$, $[\text{Try}]_m$, and $[\text{MethodCall}]_m$ in Figure 4.
(3) There must be the corresponding rules [Throw], [Try], and [MethodCall] in Figure 3.
(4) By (3) and induction hypothesis, there must be X_e such that $F(I)(X_e) \supseteq \alpha$, which will be eventually included in X_f in some more iterations $F^i(I)$ by the rules in Figure 3, because e is in f. $\square$