Interprocedural Exception Analysis for Java *

Byeong-Mo Chang
Sookmyung Women's University
Seoul, Korea
chang@cs.sookmyung.ac.kr

Jang-Wu Jo
Pusan University of Foreign Studies
Pusan, Korea
jjw@saejong.pufs.ac.kr

Kwangkeun Yi Kwang-Moo Choe
Korea Advanced Inst. of Science & Technology
Taejon, Korea
{kwang,choe}@cs.kaist.ac.kr

Keywords: Java, class analysis, uncaught exception analysis, set-based analysis

ABSTRACT

Current JDK Java compiler relies too much on programmer's specification for checking against uncaught exceptions of the input program. It is not elaborate enough to remove programmer's unnecessary handlers (when programmer's specifications are too many) nor suggest to programmers for specialized handlings (when programmer's specifications are too general). We propose a static analysis of Java programs that estimates their exception flows independently of the programmer's specifications. This analysis is designed and implemented based on set-constraint 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 exception analysis can effectively detect uncaught exceptions for realistic Java programs.

1. INTRODUCTION

Exception facilities in Java allow the programmer to define, throw and catch exceptional conditions. Exceptional conditions are brought (by a `throw` expression) to the attention of another expression where the thrown exceptions may be handled. Because unhandled 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* analysis by relying on the programmer's specifications to check that the input program will have no uncaught exceptions at run-time. Programmers have to declare in a method definition any exception class whose exceptions may escape from its body.

The problem is that the current compiler is not elaborate enough to do "better" than as specified by the programmers. It cannot avoid programmer's unnecessary handlers nor suggest to programmers for specialized handlings. It is foreseeable for careless (or inconfident) programmers to excessively declare at every method that some exceptions can be uncaught. Similarly, programmers can specify exceptions in too a broad sense. Programmers can declare that a method throws exceptions of the most general class `Exception` even if the actual exceptions are of much lower, specific classes. Then its handler cannot offer proper treatments specific to the exact classes of actual exceptions.

We propose an *interprocedural* static analysis of Java programs that estimates their exception flows independently of the programmer's specifications. Our exception analysis is designed based on set-based framework and needs class (or type) information $Class(e)$ for expression e. Class information can be obtained by type inference or class analysis as in [3, 4, 9, 5]. The classes of uncaught exceptions from a method call $e_1.m(e_2)$ is then the classes of exceptions that can be raised *and* unhandled during the execution of m's body for every class c in $Class(e_1)$.

We first design an exception analysis at expression-level and then design a sparse exception analysis at method-level for cost-effectiveness. We show theoretically that the sparse exception analysis gives the same exception information for *every method* as the expression-level analysis. In addition, we show, through implementation and experiments, that our sparse analysis can detect effectively uncaught exceptions for realistic Java programs.

2. SOURCE LANGUAGE

For presentation brevity we consider an imaginary core of Java with its exception constructs. Its abstract syntax is in Figure 1. 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

`try` e_0 `catch (`c x e_1`)`

*This work was supported by grant No. 2000-1-30300-009-2 from the Basic Research Program of the Korea Science & Engineering Foundation and by Creative Research Initiatives of the Korean Ministry of Science and Technology.

P	::=	C^*	program
C	::=	`class` c `ext` 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

Figure 1: Abstract Syntax of a Core of Java

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 evaluation chain until 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 in a method definition any exception class whose exceptions 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.

For exception analysis, every expression e of the program has a constraint: $\mathcal{X}_e \supseteq se$. The $\mathcal{X}_e$ is for the exception classes that the expression e's uncaught exception belongs to. The meaning of a set constraint $\mathcal{X} \supseteq se$ is intuitive: set $\mathcal{X}$ contains the set represented by set expression se. Multiple constraints are conjunctions. We write $\mathcal{C}$ for such conjunctive set 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 [8]. We write $lm(\mathcal{C})$ for the least model of a collection $\mathcal{C}$ of constraints.

Set-based analysis consists of two phases [8]: collecting set constraints and solving them. The first phase constructs constraints by the derivation 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. A solution is a table from set variables in the constraints to the finite descriptions of such sets of values.

Our implementation computes the solution by the conventional iterative fixpoint method because our solution space is finite: exception classes in the program. Correctness proofs

```
class Demo1 {
  public static void main(String args[]) {
      try {
          demoproc();
      } catch (Exception e) {  ; }
    }

    void demoproc() throws Exception {
      try {
          throw new IOException("demo");
      } catch (Exception e) {
          throw e
      }
    }
}
```

Figure 2: The source code for a broad specification

```
class Demo2 {
    public static void main(String args[]) {
      try {
          demoproc();
      } catch (Exception e) {  ; }
    }

    void demoproc() throws Exception {
      try {
          throw new IOException("demo");
      } catch (IOException e) { ; }
    }
}
```

Figure 3: The source code for a unnecessary specification

are done by the fixpoint induction over the continuous functions that are derived [2] from our constraint system.

3. MOTIVATION

Because unhandled exceptions will abort the program's execution, the current JDK Java compiler does an intraprocedural exception analysis relying on the programmer's specifications to check that the input program will have no uncaught exceptions at run-time.

However, programmers can declare that a method throws exceptions of the most general class `Exception` even if the actual exceptions are of much more specific classes. Then its handler cannot offer proper treatments specific to the actual exceptions. As well as that, programmers can also write too broad `catch` clauses. These two cases are shown in the source code in Figure 2.

Careless (or inconfident) programmers may also declare with unnecessary specifications at method definition that some exceptions can be uncaught.

The problem is that the current JDK compiler is not elaborate enough to do "better" than as specified by the programmers. This is mainly due to the intraprocedural exception analysis of JDK compiler relying on programmers' specification.

We will devise an interprocedural exception analysis so that it can report programmer's unnecessary handlers or

suggest to programmers for specialized handlings.

4. UNCAUGHT EXCEPTION ANALYSIS

We present our exception analysis based on the set-constraint framework [8]. We assume class information $Class(e)$ is already available for every expression e in the exception analysis. There are several choices for class information. First, we can approximate it using type information, since Java is shown to be type sound [4, 9, 5]. Second, we can utilize information from class analysis [3, 10]. 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. A set-based class analysis for Java is shown in [1].

In Section 4.1 we 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 in Section 4.2 we present a sparse constraint system that analyzes uncaught exceptions at a larger granularity than at every expression. Similar technique of enlarging constraint granularity has already been successfully used in ML [11]'s exception analysis [16]. Our analysis result is the solution of this sparse constraints.

4.1 Exception Analysis at Expression-Level

Figure 4 has the rules to generate set constraints for the object classes of *every* expression. For exception analysis, every expression e of the program has a constraint: $\mathcal{X}_e \supseteq se$. The $\mathcal{X}_e$ is a set-variable for the exception classes that the expression e's uncaught exception belongs to. The subscript e of set variables $\mathcal{X}_e$ denotes the current expression to which the rule applies. The relation "$\triangleright\ e\ : \mathcal{C}$" is read "constraints $\mathcal{C}$ are generated from expression e."

Consider the rule for `throw` expression:

$$[\text{Throw}]\quad \frac{\triangleright\ e_1 : \mathcal{C}_1}{\triangleright\ \texttt{throw}\ e_1 : \{\mathcal{X}_e \supseteq Class(e_1) \cup \mathcal{X}_{e_1}\} \cup \mathcal{C}_1}$$

It throws exceptions e_1 or, prior to throwing, it can have uncaught exceptions from inside e_1 too.

Consider the rule for `try` expression:

$$[\text{Try}]\quad \frac{\triangleright e_0 : \mathcal{C}_0 \quad \triangleright\ e_1 : \mathcal{C}_1}{\begin{array}{c}\triangleright \texttt{try}\ e_g\ \texttt{catch}(c_1\ x_1\ e_1) : \\ \{\mathcal{X}_e \supseteq (\mathcal{X}_{e_g} - \{c_1\}^*) \cup \mathcal{X}_{e_1}\} \cup \mathcal{C}_0 \cup \mathcal{C}_1\end{array}}$$

Raised exceptions from e_0 can be caught by x_1 only when their classes are covered by c_1. After this catching, exceptions can also be raised during the handling inside e_1. Hence, $\mathcal{X}_e \supseteq (\mathcal{X}_{e_0} - \{c_1\}^*) \cup \mathcal{X}_{e_1}$, where $\{c\}^*$ represents all the subclasses of a class c.

Consider the rule for method call:

$$[\text{MethCall}]\quad \frac{\triangleright e_1 : \mathcal{C}_1 \quad \triangleright\ e_2 : \mathcal{C}_2}{\begin{array}{c}\triangleright e_1\,.\,m(e_2)\ : \\ \{\mathcal{X}_e \supseteq \mathcal{X}_{c.m} | c \in Class(e_1), m(x) = e_m \in c\} \\ \cup \{\mathcal{X}_e \supseteq \mathcal{X}_{e_1} \cup \mathcal{X}_{e_2}\} \cup \mathcal{C}_1 \cup \mathcal{C}_2\end{array}}$$

Uncaught exceptions from the call expression first include those from the subexpressions e_1 and e_2 : $\mathcal{X}_e \supseteq \mathcal{X}_{e_1} \cup \mathcal{X}_{e_2}$. The method $m(x)$ `=` e_m is the one defined inside the classes

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

$$[\text{FieldAss}]\quad \frac{\triangleright\ e_1 : \mathcal{C}_1}{\triangleright\ id.x := e_1 : \{\mathcal{X}_e \supseteq \mathcal{X}_{e_1}\} \cup \mathcal{C}_1}$$

$$[\text{ParamAss}]\quad \frac{\triangleright\ e_1 : \mathcal{C}_1}{\triangleright\ x\ \texttt{:=}\ e_1 : \{\mathcal{X}_e \supseteq \mathcal{X}_{e_1}\} \cup \mathcal{C}_1}$$

$$[\text{Seq}]\quad \frac{\triangleright\ e_1 : \mathcal{C}_1 \quad \triangleright\ e_2 : \mathcal{C}_2}{\triangleright\ e_1\,;e_2 : \{\mathcal{X}_e \supseteq \mathcal{X}_{e_1} \cup \mathcal{X}_{e_2}\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

$$[\text{Cond}]\quad \frac{\triangleright e_0 : \mathcal{C}_0 \quad \triangleright\ e_1 : \mathcal{C}_1 \quad \triangleright\ e_2 : \mathcal{C}_2}{\begin{array}{c}\triangleright \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2\ : \\ \{\mathcal{X}_e \supseteq \mathcal{X}_{e_0} \cup \mathcal{X}_{e_1} \cup \mathcal{X}_{e_2}\} \cup \mathcal{C}_0 \cup \mathcal{C}_1 \cup \mathcal{C}_2\end{array}}$$

$$[\text{FieldVar}]\quad \frac{\triangleright\ id : \mathcal{C}_{id}}{\triangleright\ id.x : \mathcal{C}_{id}}$$

$$[\text{Throw}]\quad \frac{\triangleright\ e_1 : \mathcal{C}_1}{\triangleright\ \texttt{throw}\ e_1 : \{\mathcal{X}_e \supseteq Class(e_1) \cup \mathcal{X}_{e_1}\} \cup \mathcal{C}_1}$$

$$[\text{Try}]\quad \frac{\triangleright e_0 : \mathcal{C}_0 \quad \triangleright\ e_1 : \mathcal{C}_1}{\begin{array}{c}\triangleright \texttt{try}\ e_0\ \texttt{catch}(c_1\ x_1\ e_1)\ : \\ \{\mathcal{X}_e \supseteq (\mathcal{X}_{e_0} - \{c_1\}^*) \cup \mathcal{X}_{e_1}\} \cup \mathcal{C}_0 \cup \mathcal{C}_1\end{array}}$$

$$[\text{MethCall}]\quad \frac{\triangleright e_1 : \mathcal{C}_1 \quad \triangleright\ e_2 : \mathcal{C}_2}{\begin{array}{c}\triangleright e_1\,.\,m(e_2)\ : \\ \{\mathcal{X}_e \supseteq \mathcal{X}_{c.m} | c \in Class(e_1), m(x) = e_m \in c\} \\ \cup \{\mathcal{X}_e \supseteq \mathcal{X}_{e_1} \cup \mathcal{X}_{e_2}\} \cup \mathcal{C}_1 \cup \mathcal{C}_2\end{array}}$$

$$[\text{MethDef}]\quad \frac{\triangleright\ e_m : \mathcal{C}}{\triangleright\ m(x)\ \texttt{=}\ e_m : \{\mathcal{X}_{c.m} \supseteq \mathcal{X}_{e_m}\} \cup \mathcal{C}}$$

$$[\text{ClassDef}]\quad \frac{m_i : \mathcal{C}_i,\ i = 1, \cdots, n}{\begin{array}{c}\texttt{class}\ c = \{\text{var}\ x_1, \cdots, x_k, m_1, \cdots, m_n\}\ : \\ \mathcal{C}_1 \cup \cdots \cup \mathcal{C}_n\end{array}}$$

$$[\text{Program}]\quad \frac{\triangleright\ C_i : \mathcal{C}_i,\ i = 1, \cdots, n}{\triangleright\ C_1, \cdots, C_n : \mathcal{C}_1 \cup \cdots \cup \mathcal{C}_n}$$

Figure 4: Exception Analysis at Expression-Level

$c \in Class(e_1)$ of e_1's objects. Hence, $\mathcal{X}_e \supseteq \mathcal{X}_{c.m}$ for uncaught exceptions. (The subscript $c.m$ indicates the index for the body expression of class c's method m.)

4.2 Exception Analysis at Method-Level

In our new, sparse constraint system, only two groups of set variables are considered: set variables for class' 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, set variable $\mathcal{X}_m$ is a set-variable for the classes of uncaught exceptions during the call to m. The try-block e_g in `try` e_g `catch` $(c\ x\ e)$ also has a set variable $\mathcal{X}_g$, which is for uncaught exception classes in e_g.

Figure 5 shows this new constraint system. The left-hand-side m in relation $m \triangleright e\ : \mathcal{C}$ indicates that the expression e is a sub-expression of method m (or try-block g).

Consider the rule for `throw` expression:

$$[\text{Throw}]_m\quad \frac{m \triangleright\ e_1 : \mathcal{C}_1}{m \triangleright\ \texttt{throw}\ e_1 : \{\mathcal{X}_m \supseteq Class(e_1)\} \cup \mathcal{C}_1}$$

The classes $\mathcal{X}_m$ of uncaught exceptions from method m include the exception classes of the expression e_1.

$$[\text{New}]_m \quad m \rhd \texttt{new}\ c : \emptyset$$

$$[\text{FieldAss}]_m \quad \frac{m \rhd\ e_1 : \mathcal{C}_1}{m \rhd\ id.x := e_1 : \mathcal{C}_1}$$

$$[\text{ParamAss}]_m \quad \frac{m \rhd\ e_1 : \mathcal{C}_1}{m \rhd\ x\ \texttt{:=}\ e_1 : \mathcal{C}_1}$$

$$[\text{Seq}]_m \quad \frac{m \rhd\ e_1 : \mathcal{C}_1\ \ m \rhd e_2 : \mathcal{C}_2}{m \rhd\ e_1\texttt{;}e_2 : \mathcal{C}_1 \cup \mathcal{C}_2}$$

$$[\text{Cond}]_m \quad \frac{m \rhd e_0 : \mathcal{C}_0\ \ m \rhd e_1 : \mathcal{C}_1\ \ m \rhd e_2 : \mathcal{C}_2}{m \rhd \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \mathcal{C}_0 \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

$$[\text{FieldVar}]_m \quad \frac{m \rhd\ id : \mathcal{C}_{id}}{m \rhd\ id.x : \mathcal{C}_{id}}$$

$$[\text{Throw}]_m \quad \frac{m \rhd\ e_1 : \mathcal{C}_1}{m \rhd\ \texttt{throw}\ e_1 : \{\mathcal{X}_m \supseteq Class(e_1)\} \cup \mathcal{C}_1}$$

$$[\text{Try}]_m \quad \frac{m \rhd e_g : \mathcal{C}_g\ \ m \rhd e_1 : \mathcal{C}_1}{\begin{array}{l} m \rhd \texttt{try}\ e_g\ \texttt{catch}(c_1\ x_1\ e_1) : \\ \{\mathcal{X}_m \supseteq (\mathcal{X}_g - \{c_1\}^*)\} \cup \mathcal{C}_g \cup \mathcal{C}_1 \end{array}}$$

$$[\text{MethCall}]_m \quad \frac{m \rhd e_1 : \mathcal{C}_1\ \ m \rhd e_2 : \mathcal{C}_2}{\begin{array}{l} m \rhd e_1\texttt{.}m'\texttt{(}e_2\texttt{)} : \\ \{\mathcal{X}_m \supseteq \mathcal{X}_{c.m'} | c \in Class(e_1), m'(x) = e_{m'} \in c\} \\ \cup \mathcal{C}_1 \cup \mathcal{C}_2 \end{array}}$$

$$[\text{MethDef}]_m \quad \frac{m \rhd\ e_m : \mathcal{C}_m}{m \rhd\ m\texttt{(}x\texttt{)}\ \texttt{=}\ e_m : \mathcal{C}_m}$$

$$[\text{ClassDef}]_m \quad \frac{m_i : \mathcal{C}_i,\ i = 1, \cdots, n}{\begin{array}{l} \texttt{class}\ c = \{\text{var}\ x_1, \cdots, x_k, m_1, \cdots, m_n\} : \\ \mathcal{C}_1 \cup \cdots \cup \mathcal{C}_n \end{array}}$$

$$[\text{Program}]_m \quad \frac{\rhd C_i : \mathcal{C}_i,\ i = 1, \cdots, n}{\rhd C_1, \cdots, C_n : \mathcal{C}_1 \cup \cdots \cup \mathcal{C}_n}$$

Figure 5: Exception Analysis at Method-Level

Consider the rule for `try` expression:

$$[\text{Try}]_m \quad \frac{m \rhd e_g : \mathcal{C}_g\ \ m \rhd e_1 : \mathcal{C}_1}{\begin{array}{l} \rhd\texttt{try}\ e_g\ \texttt{catch}(c_1\ x_1\ e_1) : \\ \{\mathcal{X}_m \supseteq (\mathcal{X}_g - \{c_1\}^*)\} \cup \mathcal{C}_g \cup \mathcal{C}_1 \end{array}}$$

Some of the uncaught exceptions $\mathcal{X}_g$ from e_g can be caught and handled, if the exception's classes are covered by c. Hence the uncaught exceptions from this expression includes the uncovered ones.

Consider the rule for method-call expression:

$$[\text{MethCall}]_m \quad \frac{m \rhd e_1 : \mathcal{C}_1\ \ m \rhd e_2 : \mathcal{C}_2}{\begin{array}{l} m \rhd e_1\texttt{.}m'\texttt{(}e_2\texttt{)} : \\ \{\mathcal{X}_m \supseteq \mathcal{X}_{c.m'} | c \in Class(e_1), m'(x) = e_{m'} \in c\} \\ \cup \mathcal{C}_1 \cup \mathcal{C}_2 \end{array}}$$

Thus, if m's body has a method call $e_1\texttt{.}m'\texttt{(}e_2\texttt{)}$, raised exceptions' classes $\mathcal{X}_m$ include those $\mathcal{X}_{c.m'}$ uncaught 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 5, except that m is replaced by g.

The least model of the sparse constraints $\mathcal{C}$, which are derived ($\rhd pgm : \mathcal{C}$) from an input program *pgm* is our analysis result. The solutions for $\mathcal{X}_m$ has the exception classes whose exceptions might be thrown from m's execution.

5. SOUNDNESS AND COMPLETENESS

We have designed a method-level exception analysis from the expression-level analysis in Figure 4. In order to prove the soundness and completeness of the method-level analysis, 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 the partitioning function $\pi : Expr \rightarrow Expr \cup Method$ as follows :

$$\begin{array}{lll} \pi(e) & = g & \text{if } e \text{ is a sub-expression of } e_g \\ & & \text{in an expression } \texttt{try}\ e_g\ \texttt{catch}(c_1\ x_1\ e_1) \\ \pi(e) & = m & \text{if } e \text{ is a sub-expression of a method } m, \\ & & \text{which is not of a try-block.} \end{array}$$

This partitioning function specifies that there is one set variable $\mathcal{X}_g$ for all sub-expressions of a try-block e_g, and one set-variable $\mathcal{X}_m$ for all sub-expressions of a method m, not of a try-block.

In the following, we assume that $\mathcal{C}$ is the collection of set constraints for a program *pgm* constructed by the rules in Figure 4, and $\mathcal{C}_\pi$ is the collection of set constraints for the same program *pgm* constructed by the rules in Figure 5.

The least model of the method-level constraints $\mathcal{C}_\pi$ is a sound approximation of that of the original constraints $\mathcal{C}$. The proof is based on the observation in [2] that the least model $lm(\mathcal{C})$ is equivalent to the least fixpoint of the continuous function $\mathcal{F}$ derived from $\mathcal{C}$.

THEOREM 1. *$lm(\mathcal{C}_\pi)(\pi(\mathcal{X})) \supseteq lm(\mathcal{C})(\mathcal{X})$ for every set variable $\mathcal{X}$ in $\mathcal{C}$.*

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 $\mathcal{C}_\pi$ is *equivalent* to $\mathcal{C}$ with respect to every method and try-block : if $lm(\mathcal{C}_\pi)(\mathcal{X}_f) = lm(\mathcal{C})(\mathcal{X}_f)$ for every method and try-block f.

THEOREM 2. *$lm(\mathcal{C}_\pi)(\mathcal{X}_f) = lm(\mathcal{C})(\mathcal{X}_f)$ for every method and try-block f.*

Proof. *See Appendix A.* □

6. EXPERIMENTAL RESULTS

This section evaluates exception analysis on realistic Java programs. We have implemented the method-level exception analyzer. The analyzer is implemented in C by two passes for setting up constraints and solving constraints, respectively. Our testbed consists of UltraSPARC Enterprise 450 running Sun Solaris.

We have selected a set of 6 medium-sized benchmarks described in Table 2 for our experiments.

No.	Programs	Descriptions
1	Statistician	methods statistics for a class
2	JavaBinHex	BinHex(.hqx) decompressor
3	JHLZIP	ZIP compressor
4	JHLUNZIP	ZIP uncompressor
5	com.ice.tar	UNIX Tar Archive
6	Jess-Rete	Reasoning engine of Jess

Table 1: Benchmarks programs

No	Total classes	Total methods	Lines of Code
1	1	1	387
2	1	3	300
3	2	11	425
4	1	3	187
5	10	141	4045
6	1	98	1667

Table 2: Benchmarks programs

No	Kinds of exceptions	throws spec	catch block	Uncaught exceptions
1	3	1	24	1
2	2	0	8	0
3	3	4	8	4
4	2	1	3	1
5	6	40	21	41
6	7	38	10	33

Table 3: Uncaught exceptions

A prototype's preliminary performance on the numbers of total uncaught exceptions from all methods is shown in Table 3. With the analysis result of every method, we have counted the numbers of `throws`'s and `catch`'s which are unnecessary, and similarly the numbers of `throws`'s and `catch`'s which are broader than raised exceptions. They are shown in Table 4. Our analysis has detected meaningful amount of unnecessary and broader specifications and `catch`'s in real Java programs. Execution times for constraints set-up and solving are sufficiently fast and shown in Table 5.

	Unnessesary		Broader	
No	throws	catch	throws	catch
1	0	1	0	0
2	0	0	0	1
3	0	0	2	1
4	0	0	1	1
5	4	0	3	4
6	6	0	1	3

Table 4: Analysis result

No	Set-up time(sec)	Solving time(sec)
1	0.02	0.01
2	0.02	0.01
3	0.02	0.01
4	0.01	0.01
5	0.18	0.03
6	0.05	0.02

Table 5: Analysis times

7. RELATED WORKS

In ML, exceptions are first class values that can be declared, assigned and passed as parameters. These values can be raised at any point once they are declared. Several exception analyses have been introduced to trace uncaught exceptions in ML [13, 16, 17, 7]. Yi first designed an exception analysis by abstract interpreation [13], which was too slow, and then redesigned it based on set-based framework and showed better speed. Fähndrich and Aiken [6] 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 [7].

In Java, the JDK compiler ensures by an intraprocedural analysis, with programmers's specifications of uncaught exceptions of each method, that raised exceptions are caught or specified. In [12], they developed Jex tool based on JDK's approach, analyzed exception matching in catch clauses, and showed a ratio of several classified exception matchings in catch clauses. In [15], first exception analysis for Java was designed and presented, but without experimental data about its cost-effectiveness.

8. CONCLUSIONS

We have presented an exception analysis for Java, that estimates their exception flows independently of the programmer's specifications. 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 exception flows. By an implementation and its experiments, we have shown that our exception analysis can effectively detect uncaught exceptions for realistic Java programs.

Our exception analysis' hint about Java program's exception flows help the programmers or the compilers to efficiently handle exceptions in the source programs or in the compiled codes.

9. 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] 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.

[3] 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.

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

[5] 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.

[6] 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.

[7] X. Leroy and F. Pessaux, Type-based analysis of uncaught exceptions. *ACM Transactions on Programming Languages and Systems*, Vol.22, No. 2, pp. 340-377, March 2000

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

[9] Java is type safe-definitely, *Proceedings of 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*,January 1998.

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

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

[12] M. P. Robillard and G. C. Murphy, Analyzing exception flow in Java programs. *Proceedings of 7th European Software Engineering Conference and ACM SIGSOFT Symposium on Foundations of Software Engineering*, 1999.

[13] 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.

[14] Kwangkeun Yi. An abstract interpretation for estimating uncaught exception in Standard ML programs. *Science of Computer Programming*, 31(1):147–173, 1998.

[15] Kwangkeun Yi and Byeong-Mo Chang. Exception analysis for java. In A. Moreira and D. Demeyer, editors, *Object-Oriented Technology. ECOOP'99 Workshop Reader (Formal Techniques for Java Programs)*, volume 1743 of *Lecture Notes in Computer Science*, pages 111–112. Springer-Verlag, June 1999. Extended version of this paper is available from `ropas.kaist.ac.kr/paper/99-ecoop-yich.ps.gz`.

[16] 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.

[17] Kwangkeun Yi and Sukyoung Ryu. A cost-effective estimation of uncaught exceptions in Standard ML programs. *Theoretical Computer Science*, 273(1), 2001.

Appendix A. Proof

Theorem 2 *Proof.* As in [2], the continuous function $\mathcal{F}$ can be defined from $\mathcal{C}$, and $\mathcal{F}_\pi$ can also be defined from $\mathcal{C}_\pi$ likewise. We prove this theorem by showing that $lfp(\mathcal{F}_\pi)(\mathcal{X}_f) = lfp(\mathcal{F})(\mathcal{X}_f)$ for every method and try-block f. By the soundness theorem, $lfp(\mathcal{F}_\pi)(\mathcal{X}_f) \supseteq lfp(\mathcal{F})(\mathcal{X}_f)$. So, we just prove that $lfp(\mathcal{F}_\pi)(\mathcal{X}_f) \subseteq lfp(\mathcal{F})(\mathcal{X}_f)$ for every method and try-block f.

The proof is by induction on the number of iterations in computing $lfp(\mathcal{F}_\pi)$.

Induction step : Suppose $\mathcal{I}_\pi(\mathcal{X}_f) \subseteq \mathcal{I}(\mathcal{X}_f)$ for every method and try-block f. Let $\mathcal{I}'_\pi = \mathcal{F}_\pi(\mathcal{I}_\pi)$. Then there exists $\mathcal{I}'$ such that $\mathcal{I}' = \mathcal{F}^i(\mathcal{I})$ for some i and $\mathcal{I}'_\pi(\mathcal{X}_f) \subseteq \mathcal{I}'(\mathcal{X}_f)$ for every method and try-block f.

(1) For every set variable $\mathcal{X}_f$, suppose $\mathcal{I}'_\pi(\mathcal{X}_f) = \mathcal{I}_\pi(\mathcal{X}_f) \cup \alpha$.

(2) Then, α must be added by some of the rules $[\text{Throw}]_m$, $[\text{Try}]_m$, and $[\text{MethodCall}]_m$ in Figure 5.

(3) There must be the corresponding rules [Throw], [Try], and [MethodCall] in Figure 4.

(4) By (3) and induction hypothesis, there must be $\mathcal{X}_e$ such that $\mathcal{F}(\mathcal{I})(\mathcal{X}_e) \supseteq \alpha$, which will be eventually included in $\mathcal{X}_f$ in some more iterations $\mathcal{F}^i(\mathcal{I})$ by the rules in Figure 4, because e is in f. □