Visualization of Exception Propagation for Java using Static Analysis [*]

Byeong-Mo Chang
Department of Computer Science
Sookmyung Women's University
Seoul 140-742, Korea
chang@sookmyung.ac.kr

Jang-Wu Jo
Department of Computer Engineering
Pusan University of Foreign Studies
Pusan 608-738, Korea
jjw@taejo.pufs.ac.kr

Soon Hee Her
Department of Computer Science
Sookmyung Women's University
Seoul 140-742, Korea
hsh@cs.sookmyung.ac.kr

Abstract

In this paper, we first present a static analysis based on set-based framework, which estimates exception propagation paths of Java programs. We construct an exception propagation graph from the static analysis information, which includes the origin of exceptions, handler of exceptions, and propagation paths of exceptions. We have implemented the exception propagation analysis and a visualization tool which visualizes propagation paths of exceptions using the exception propagation graph. This propagation information can guide programmers to detect uncaught exceptions, handle exceptions more specifically, and put exception handlers at appropriate places by tracing exception propagation.

Keywords: Java, exception propagation, exception analysis, set-based analysis.

1 Introduction

Exception facilities in Java allow the programmer to define, throw and catch exceptional conditions. Java distinguishes between *checked* and *unchecked* exceptions. Unchecked exceptions are exempt from the requirement of being declared. Java compiler checks whether checked exceptions are caught or declared, so checked exception must be declared if they are not caught.

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 (checked) exceptions which are uncaught at run-time. There have been several uncaught exception analyses, that estimate uncaught exceptions [10, 24, 1, 18].

However, they estimate uncaught exceptions only by their names, so that they cannot provide information on the *propagation paths* of thrown exceptions, which is necessary to construct interprocedural control flow graph [20], visualize exception propagation, and slice exception-related parts of programs.

In this paper, we first present a static analysis to safely approximate propagation paths of thrown exceptions and then present a visualization tool to show exception propagation paths.

We design an exception propagation analysis based on set-based framework [12]. We first design set-constraint construction rules to safely approximate propagation paths of thrown exceptions. We then design constraint solving rules. We compute the solution of the constraints in finite time by applying the solving rules.

Our visualization tool displays exception propagation information using the static analysis information. If users select a method, the visualization tool first displays all uncaught exceptions from that method. If one of the uncaught exceptions is selected, its exception propagation paths are visualized. This visualization tool can guide programmers to detect uncaught exceptions, handle exceptions more specifically and declare exceptions more exactly. Moreover, this information

---

[*] This work was supported in part by grant No. 2000-1-30300-009-2 from the Basic Research Program of the Korea Science & Engineering Foundation.

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

Figure 1. Abstract Syntax of a Core of Java

can guide programmers to put exception handlers at appropriate places by tracing exception propagation.

We have implemented the exception propagation analysis on top of Barat [26], which is a front-end of Java compiler, and have also implemented the visualization tool on top of Jipe [27] using the static analysis information.

The next section describes preliminaries including the core of Java, on which our presentation is based. Section 3 describes the exception propagation analysis. Section 4 describes implementation of the exception propagation analysis and visualization tool. Section 5 discusses related works and Section 6 concludes this paper.

2 Preliminaries

For presentation brevity of our static analysis, we consider an imaginary core of Java with its exception constructs [24]. 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. An assignment expression returns the object of its righthand side expression. A 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

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

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 thrown 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 thrown exception is not covered by class c then the thrown exception continues to propagate back along the evaluation chain until it meets another handler. Multiple handlers for a single expression e_0 can be expressed by a nested `try` expression:

$$\texttt{try (try}\ e_0\ \texttt{catch (}c_1\, x_1\texttt{)}\, e_1\texttt{)}\ \texttt{catch (}c_2\, x_2\texttt{)}\, e_2.$$

The exception object e_0 is thrown by $\texttt{throw}\ e_0$. The programmers have to declare in a method definition any exception class whose exceptions may escape from its body.

Like normal objects, exceptions can be defined by classes, instantiated, assigned to variables, passed as parameters, etc. Exception facilities in Java allow the programmer to define, throw and catch exceptional conditions.

The formal semantics of Java was proposed in [7] with exception throwing, propagation and handling taken into consideration.

Let's consider a simple example in Java which shows exception propagation. The thrown exception `E1` from the method `m2` is propagated through `m2` and `m1`, and caught by the `try-catch` in the main method. The exception `E2` may be thrown from the method `m3`. If it is thrown, then it is propagated until the `main` method

```
class Demo{
  public static void main(String[] args ) throws E2
  {
     try {
         m1( );
     } catch (E1 x) {  ; }
     . . .
     m3( );
  }

  void m1( ) throws E1{
     m2( );
  }

  void m2( ) throws E1{
     throw new E1();
  }

  void m3( ) throws E2 {
     if (...) throw new E2();
     if (...) m3( );
  }
}
```

Figure 2. An example program for exception propagation

and not caught. The method `m3` also has a recursive call to itself, so that the thrown exception `E2` may propagated back through the recursive calls.

3 Exception Propagation Analysis

Our analysis is based on the set-based framework [12]. Set-based analysis consists of two phases: collecting set constraints and solving them. The first phase constructs set-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. A solution is a table or mapping from set variables in the constraints to the finite descriptions of such sets of values.

We shall first describe the notion of set constraints and then present a constraint system that estimates traces of thrown exceptions from every expression of the input program.

3.1 Set Constraints

Each set constraint is of the form $\mathcal{X} \supseteq se$ where $\mathcal{X}$ is a set variable and se is a set expression. 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.

In case of our analysis, the set expression is of this form:

$$\begin{array}{rlll}
se \rightarrow & & \langle c^{\ell}, \ell \rangle & \text{thrown exception from } \ell \\
& | & \mathcal{X} & \text{set varaible} \\
& | & se \cup se & \text{union} \\
& | & se - \{c_1, ..., c_n\} & \text{catching exceptions} \\
& | & se \cdot \ell & \text{exception propagation}
\end{array}$$

The thrown exception from a `throw` expression labeled with ℓ is represented by $\langle c^{\ell}, \ell \rangle$ where c is the name or class of the exception and l is the location or label of the `throw` expression. We call c^{ℓ} the *unique identifier* of the thrown exception in this paper.

The set expression $se - \{c_1, ..., c_n\}$ is for catching exceptions. The set expression $se \cdot \ell$ records exception propagation paths by appending a label ℓ to se.

The semantics of set expressions naturally follows from their corresponding language constructs. The formal semantics of set expressions is defined by an interpretation $\mathcal{I}$ that maps from set expressions to sets of values in

$$V = Exception \times Trace$$

where $Exception = ExnName \times Label$ where $ExnName$ is the set of exception names, and $Trace = Label^*$. A trace $\tau \in Trace$ is a sequence of labels in $Label$, which is an exception propagation path. For example, $\mathcal{I}(se \cdot \ell') = \mathcal{I}(se) \cdot \ell'$ where $\mathcal{I}(se) \cdot \ell' = \{\langle c^{\ell}, \ell_1 \cdots \ell_n \ell' \rangle | \langle c^{\ell}, \ell_1 \cdots \ell_n \rangle \in \mathcal{I}(se)\}$. We call an interpretation $\mathcal{I}$ a *model* (a solution) of a conjunction $\mathcal{C}$ of constraints if, for each constraint $\mathcal{X} \supseteq se$ in $\mathcal{C}$, $\mathcal{I}(\mathcal{X}) \supseteq \mathcal{I}(se)$.

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 [12]. We write $lm(\mathcal{C})$ for the least model of a collection $\mathcal{C}$ of constraints.

3.2 Set-constraint Construction

For simple presentation, our analysis traces exception propagation paths by recording the labels of just exception-related constructs such as `throw`, `try-catch`, and method declarations. We assume this kind of expressions e has a label ℓ, denoted by $\ell : e$. If more detailed trace information is necessary, it is possible to record other expressions such as method calls and `try`-blocks.

Figure 3 has the rules to generate set-constraints for every expression. For our analysis, every expression e of the program has a constraint: $\mathcal{X}_e \supseteq se$. The $\mathcal{X}_e$ is a set-variable for collecting the propagation paths of thrown exceptions inside e. The subscript e of $\mathcal{X}_e$ denotes the current expression to which the rule applies. The relation "$e \rhd \mathcal{C}$" is read "constraints $\mathcal{C}$ are generated from expression e."

We assume that class information $class(e)$ is already available for every expression e in the analysis. There are several choices for obtaining class information. First, we can approximate it using type information [6, 15, 7]. Second, we can utilize information from class analysis [5, 16], which estimates for each expression e the classes (including exception classes) that the expression e's object belongs to.

Consider the rule for the `throw` expression with a label ℓ:

$$\frac{e_1 \rhd \mathcal{C}_1}{\ell : \texttt{throw}\, e_1 \rhd \{\mathcal{X}_e \supseteq \langle c^\ell, \ell\rangle \cup \mathcal{X}_{e_1}\} \cup \mathcal{C}_1} \; c = class(e_1)$$

It throws an exception e_1, which is represented by $\langle c^\ell, \ell\rangle$ where $c = class(e_1)$ is the name or class of the exception and l is the label of the `throw` statement, an origin of the exception. Prior to the throwing, it can have uncaught exceptions from sub-expressions inside e_1 too.

Consider the rule for the `try` expression with a label ℓ' :

$$\frac{e_0 \rhd \mathcal{C}_0 \quad e_1 \rhd \mathcal{C}_1}{\ell' : \texttt{try}\, e_0\, \texttt{catch}(c_1\, x_1)\, e_1 \rhd \quad \begin{array}{l}\{\mathcal{X}_e \supseteq ((\mathcal{X}_{e_0} - \{c_1\}^*) \cup \mathcal{X}_{e_1}) \cdot \ell'\} \\ \cup \mathcal{C}_0 \cup \mathcal{C}_1\end{array}}$$

Thrown 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 thrown during the handling inside e_1. Uncaught exceptions from this expression are followed by the label ℓ' to record the exception propagation path. Hence, $\mathcal{X}_e \supseteq ((\mathcal{X}_{e_0} - \{c_1\}^*) \cup \mathcal{X}_{e_1}) \cdot \ell'$, where $\{c\}^*$ represents all the descendant classes of a class c including itself.

Consider the rule for the method call:

$$\frac{e_1 \rhd \mathcal{C}_1 \quad e_2 \rhd \mathcal{C}_2}{\mathrm{e}_1.m(e_2) \rhd \quad \begin{array}{l}\{\mathcal{X}_e \supseteq \mathcal{X}_{c.m} | c \in class(e_1), m(x) = e_m \in c\} \\ \cup\{\mathcal{X}_e \supseteq \mathcal{X}_{e_1} \cup \mathcal{X}_{e_2}\} \cup \mathcal{C}_1 \cup \mathcal{C}_2\end{array}}$$

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 $c \in class(e_1)$ 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.)

Consider the rule for the method definition with a label ℓ':

$$\frac{e_m \rhd \mathcal{C}}{\ell' : m(x) \;=\; e_m \;\; \rhd \{\mathcal{X}_{c.m} \supseteq \mathcal{X}_{e_m} \cdot \ell'\} \cup \mathcal{C}} \; m \in c$$

$$\begin{array}{lll}
\mathcal{X}_{main} & \supseteq & \mathcal{X}_{try-catch} \cdot main \\
\mathcal{X}_{main} & \supseteq & \mathcal{X}_{m3} \cdot main \\
\mathcal{X}_{try-catch} & \supseteq & (\mathcal{X}_{m1} - \{Exception\}^*) \cdot try - catch \\
\mathcal{X}_{m1} & \supseteq & \mathcal{X}_{m2} \cdot m1 \\
\mathcal{X}_{m2} & \supseteq & \mathcal{X}_{throwE1} \cdot m2 \\
\mathcal{X}_{throwE1} & \supseteq & \langle E1, throwE1\rangle \\
\mathcal{X}_{m3} & \supseteq & \mathcal{X}_{throwE2} \cdot m3 \\
\mathcal{X}_{throwE2} & \supseteq & \langle E2, throwE2\rangle \\
\mathcal{X}_{m3} & \supseteq & \mathcal{X}_{m3} \cdot m3
\end{array}$$

Figure 4. Set-constraints

$$\frac{\mathcal{X} \supseteq \mathcal{X}_1 \cup \mathcal{X}_2}{\mathcal{X} \supseteq \mathcal{X}_1} \qquad \frac{\mathcal{X} \supseteq \mathcal{X}_1 \cup \mathcal{X}_2}{\mathcal{X} \supseteq \mathcal{X}_2} \qquad \frac{\mathcal{X} \supseteq \mathcal{Y} \quad \mathcal{Y} \supseteq \langle c^\ell, \tau\rangle}{\mathcal{X} \supseteq \langle c^\ell, \tau\rangle}$$

$$\frac{\mathcal{X} \supseteq \mathcal{X}_1 \cdot \ell' \quad \mathcal{X}_1 \supseteq \langle c^\ell, \tau\rangle}{\mathcal{X} \supseteq \langle c^\ell, \tau \cdot \ell'\rangle}$$

$$\frac{\mathcal{X} \supseteq \mathcal{X}_1 - \{c_1, \cdots, c_k\} \quad \mathcal{X}_1 \supseteq \langle c^\ell, \tau\rangle \quad c \notin \{c_1, \cdots, c_k\}}{\mathcal{X} \supseteq \langle c^\ell, \tau\rangle}$$

Figure 5. Rules S for solving set constraints

Uncaught exceptions from the this method m include those from the method body e_m, which are followed by the label ℓ' to record exception propagation path.

We can construct the set-constrains in Figure 4 by applying the construction rules to the example program in Figure 2. After identifying the set-constraints, we use the statements with some simplification instead of labels for better understanding.

3.3 Solving the set-constraints

We first design naive constraint solving rules S. We can compute the possibly infinite solution $lm_S(\mathcal{C})$ of the constraints $\mathcal{C}$ by applying the naive solving rules S. This solution can be infinite due to recursive calls in the input program.

The naive solving phase closes the initial constraint set $\mathcal{C}$ under the rules S in Figure 5. Intuitively, the rules propagate values along all the possible data flow paths in the program. Each propagation rule decomposes compound set constraints into smaller ones, which approximates the steps of the value flows between expressions.

Consider the rule for tracing exception propagation path :

$$\frac{\mathcal{X} \supseteq \mathcal{X}_1 \cdot \ell' \quad \mathcal{X}_1 \supseteq \langle c^\ell, \tau\rangle}{\mathcal{X} \supseteq \langle c^\ell, \tau \cdot \ell'\rangle}$$

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

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

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

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

[Cond]
$$\frac{e_0 \rhd \mathcal{C}_0 \quad e_1 \rhd \mathcal{C}_1 \quad e_2 \rhd \mathcal{C}_2}{\texttt{if}\, e_0 \,\texttt{then}\, e_1 \,\texttt{else}\, e_2 \rhd \{\mathcal{X}_e \supseteq \mathcal{X}_{e_0} \cup \mathcal{X}_{e_1} \cup \mathcal{X}_{e_2}\} \cup \mathcal{C}_0 \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

[FieldVar]
$$\frac{id \rhd \mathcal{C}_{id}}{id.x \rhd \mathcal{C}_{id}}$$

[Throw]
$$\frac{e_1 \rhd \mathcal{C}_1}{\ell : \texttt{throw}\, e_1 \rhd \{\mathcal{X}_e \supseteq \langle c^{\ell}, \ell\rangle \cup \mathcal{X}_{e_1}\} \cup \mathcal{C}_1} \quad c = class(e_1)$$

[Try]
$$\frac{e_0 \rhd \mathcal{C}_0 \quad e_1 \rhd \mathcal{C}_1}{\ell' : \texttt{try}\, e_0 \,\texttt{catch}(c_1\, x_1)\, e_1 \rhd \{\mathcal{X}_e \supseteq ((\mathcal{X}_{e_0} - \{c_1\}^*) \cup \mathcal{X}_{e_1}) \cdot \ell'\} \cup \mathcal{C}_0 \cup \mathcal{C}_1}$$

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

[MethDef]
$$\frac{e_m \rhd \mathcal{C}}{\ell' : m\texttt{(}x\texttt{)} \;\texttt{=}\; e_m \rhd \{\mathcal{X}_{c.m} \supseteq \mathcal{X}_{e_m} \cdot \ell'\} \cup \mathcal{C}} \quad m \in c$$

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

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

Figure 3. Set-constraint construction rules

This rule simulates the propagation path of the thrown exception by appending the label ℓ' to the exception trace τ in $\mathcal{X}_1$. Other rules are similarly straightforward from the semantics of corresponding set expressions.

We can compute the solution $lm_S(\mathcal{C})$ of set-constraints $\mathcal{C}$ by applying the rules S in Figure 3. We can sketch the soundness of the solution as follows:

Theorem 1 *Let P be a program and $\mathcal{C}$ be the set-constraints constructed by the rules in Figure 3. Every exception trace of P is included in the solution $lm_S(\mathcal{C})$.*

We can compute the infinite solution for the set-constraints $\mathcal{C}$ in Figure 4 by applying the rule S. Possible solutions are the following:

$$\begin{array}{lll}
lm_S(\mathcal{C})(\mathcal{X}_{m1}) \supseteq & \{ & \langle E1, throwE1 \cdot m2 \cdot m1 \rangle \quad \} \\
lm_S(\mathcal{C})(\mathcal{X}_{m2}) \supseteq & \{ & \langle E1, throwE1 \cdot m2 \rangle \quad \} \\
lm_S(\mathcal{C})(\mathcal{X}_{m3}) \supseteq & \{ & \langle E2, throwE2 \cdot m3 \rangle, \\
 & & \langle E2, throwE2 \cdot m3 \cdot m3 \rangle, \\
 & & \langle E2, throwE2 \cdot m3 \cdot m3 \cdot m3 \rangle, \\
 & & \ldots \\
 & \} & \\
lm_S(\mathcal{C})(\mathcal{X}_{main}) \supseteq & \{ & \langle E2, throwE2 \cdot m3 \cdot main \rangle \\
 & & \langle E2, throwE2 \cdot m3 \cdot m3 \cdot main \rangle, \\
 & & \langle E2, throwE2 \cdot m3 \cdot m3 \cdot m3 \cdot main \rangle, \\
 & & \ldots \\
 & \} &
\end{array}$$

The solution can be infinite in case there are recursive methods, which contain uncaught exception(s).We need to find a finite representation for the possibly infinite solution.

So, we design the new solving rules S' for finite solution by modifying the exception propagation rule in S. The main idea is to represent an exception propagation path, that is a trace, by the edges constituting the path and the unique identifer of the thrown exception. They are finite because the number of exception names and labels is finite.

To do this, at every step of exception propagation, we record the last two labels (that is an edge) together with the unique identifer of the thrown exception. We modify the rule for tracing exception propagation as follows :

$$\frac{\mathcal{X} \supseteq \mathcal{X}_1 \cdot \ell' \quad \mathcal{X}_1 \supseteq \langle c^{\ell}, \tau \rangle}{\mathcal{X} \supseteq \langle c^{\ell}, \lfloor \tau \cdot \ell' \rfloor_2 \rangle}$$

where

$$\lfloor \ell_1 \cdots \ell_n \rfloor_2 \quad = \quad \ell_{n-1}\ell_n \quad \text{when } n \geq 2$$

This rule simulates the propagation of thrown exceptions, by recording the last two labels together with the thrown exception's unique identifier c^{ℓ}. Because this is done at every step of exception propagation, the dropped information has already been included into the solution together with the unique identifier c^{ℓ}.

In the following, S' denotes the solving rules S with the propagation rule being replaced by the new one. Our analysis computes the least model $lm_{S'}(\mathcal{C})$ of set-constraints $\mathcal{C}$ by applying the new solving rules S'. We can compute the solution for the set-constraints $\mathcal{C}$ in Figure 4 by applying the new rule S'. Possible solutions are as follows:

$$\begin{array}{lll}
lm_{S'}(\mathcal{C})(\mathcal{X}_{m1}) \supseteq & \{ & \langle E1, m2 \cdot m1 \rangle \} \\
lm_{S'}(\mathcal{C})(\mathcal{X}_{m2}) \supseteq & \{ & \langle E1, throwE1 \cdot m2 \rangle \} \\
lm_{S'}(\mathcal{C})(\mathcal{X}_{m3}) \supseteq & \{ & \langle E2, throwE2 \cdot m3 \rangle, \langle E2, m3 \cdot m3 \rangle \} \\
lm_{S'}(\mathcal{C})(\mathcal{X}_{main}) \supseteq & \{ & \langle E2, m3 \cdot main \rangle \}
\end{array}$$

We can see exception propagation paths by defining the *exception propagation graph* of the solution $lm_{S'}(\mathcal{C})$.

Definition 1 Let $\mathcal{C}$ be the set-constraints constructed for a program P. *Exception propagation graph* of the solution $lm_{S'}(\mathcal{C})$ is defined to be a graph $\langle V, E \rangle$ where V is the set of labels in P and $E = \{\ell_1 \rightarrow^{c^{\ell}} \ell_2 | \langle c^{\ell}, \ell_1\ell_2 \rangle \in lm_{S'}(\mathcal{C})(\mathcal{X})$ for a set variable $\mathcal{X}$ in $\mathcal{C}\}$ where $\ell_1 \rightarrow^{c^{\ell}} \ell_2$ denotes an edge from ℓ_1 to ℓ_2 labeled with c^{ℓ}. □

We can easily draw the exception propagation graph for the finite solution by making the following labeled edges :

$$throwE1 \rightarrow^{E1} m2 \quad m2 \rightarrow^{E1} m1 \quad throwE2 \rightarrow^{E2} m3$$

$$m3 \rightarrow^{E2} main \quad m3 \rightarrow^{E2} m3$$

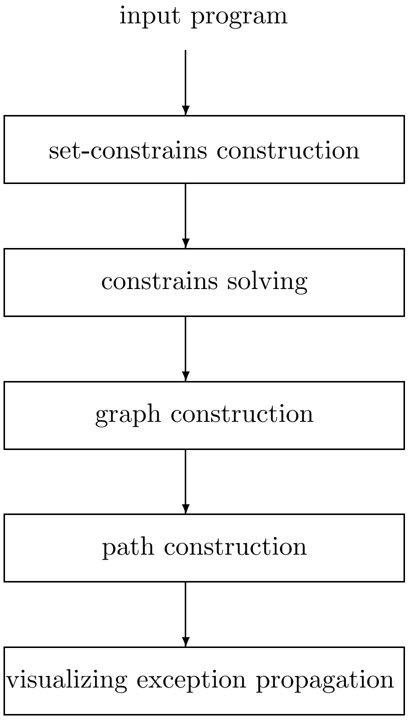


Figure 6. System architecture

We can show the soundness of the finite solution by finding a path in the exception propagation graph for every trace in the possibly infinite solution.

Theorem 2 *Let $lm_S(\mathcal{C})$ and $lm_{S'}(\mathcal{C})$ be the solutions of set-constraints $\mathcal{C}$ by applying the solving rules S and S' respectively. For every exception trace $\langle c^{\ell}, \tau \rangle$ in $lm_S(\mathcal{C})$, there is a path for τ with every edge labeled c^{ℓ} in the exception propagation graph of $lm_{S'}(\mathcal{C})$.*

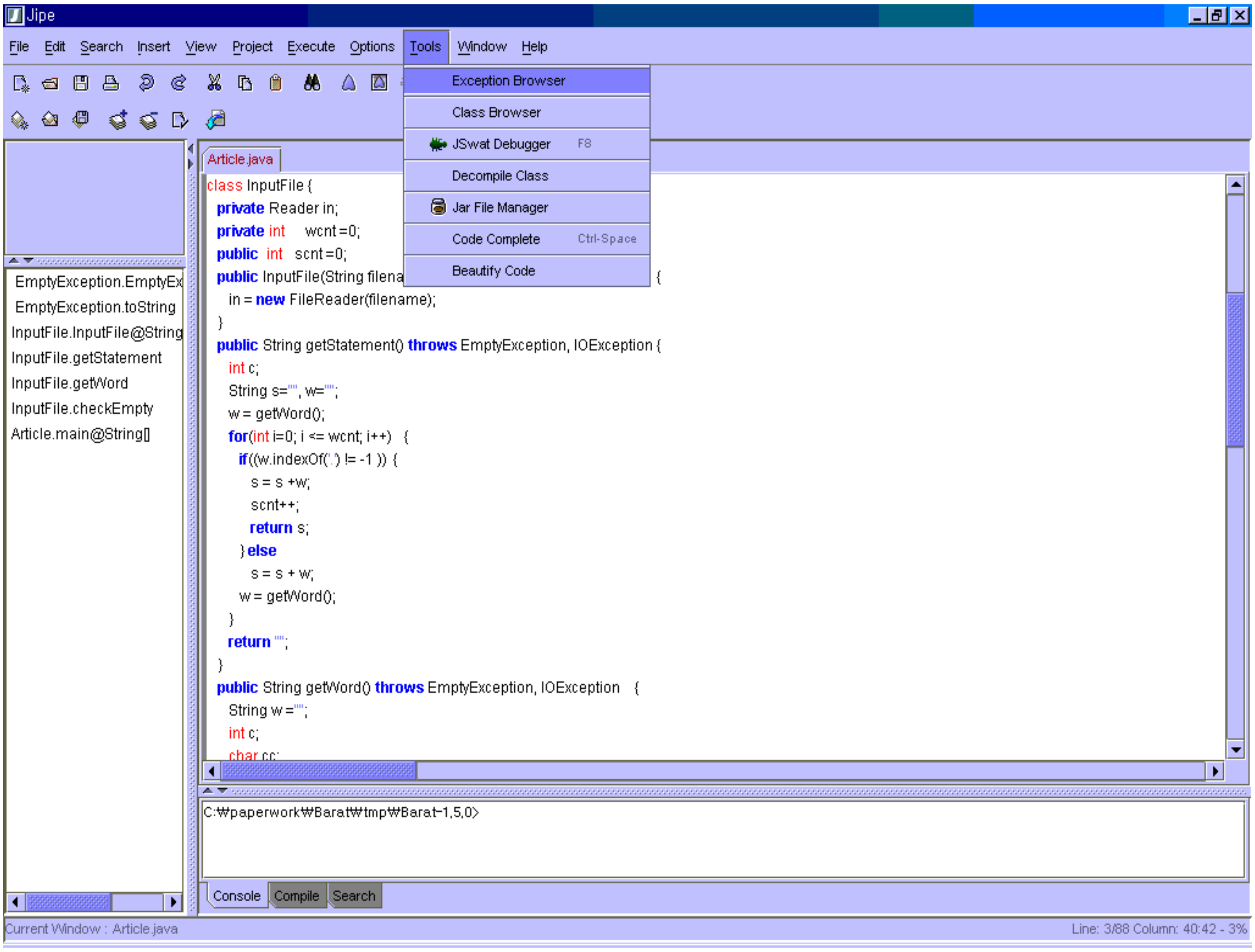
4 Implementation

We first implemented the exception propagation analysis and then a tool to visualize exception propagation paths using the static analysis information.

We take the followings into consideration in the implementation :

1. Even if we present our analysis for a core Java in Figure 1, we considered the full Java in the implementation, which includes object-allocations, explicit constructor calls, interfaces, abstract methods, and nested classes.

2. We considered checked exceptions only, because including unchecked exceptions can generate too



```
class InputFile {
  private Reader in;
  private int    wcnt=0;
  public  int   scnt=0;
  public InputFile(String filena                                  {
    in = new FileReader(filename);
  }
  public String getStatement() throws EmptyException, IOException {
    int c;
    String s="", w="";
    w = getWord();
    for(int i=0; i <= wcnt; i++)   {
      if((w.indexOf('.') != -1 )) {
        s = s +w;
        scnt++;
        return s;
      } else
        s = s + w;
      w = getWord();
    }
    return "";
  }
  public String getWord() throws EmptyException, IOException   {
    String w ="";
    int c;
    char cc;
```

```
C:₩paperwork₩Barat₩tmp₩Barat-1,5,0>
```

Figure 7. Source program and a menu for the visualization tool

much information, which affects the usability of our analysis.

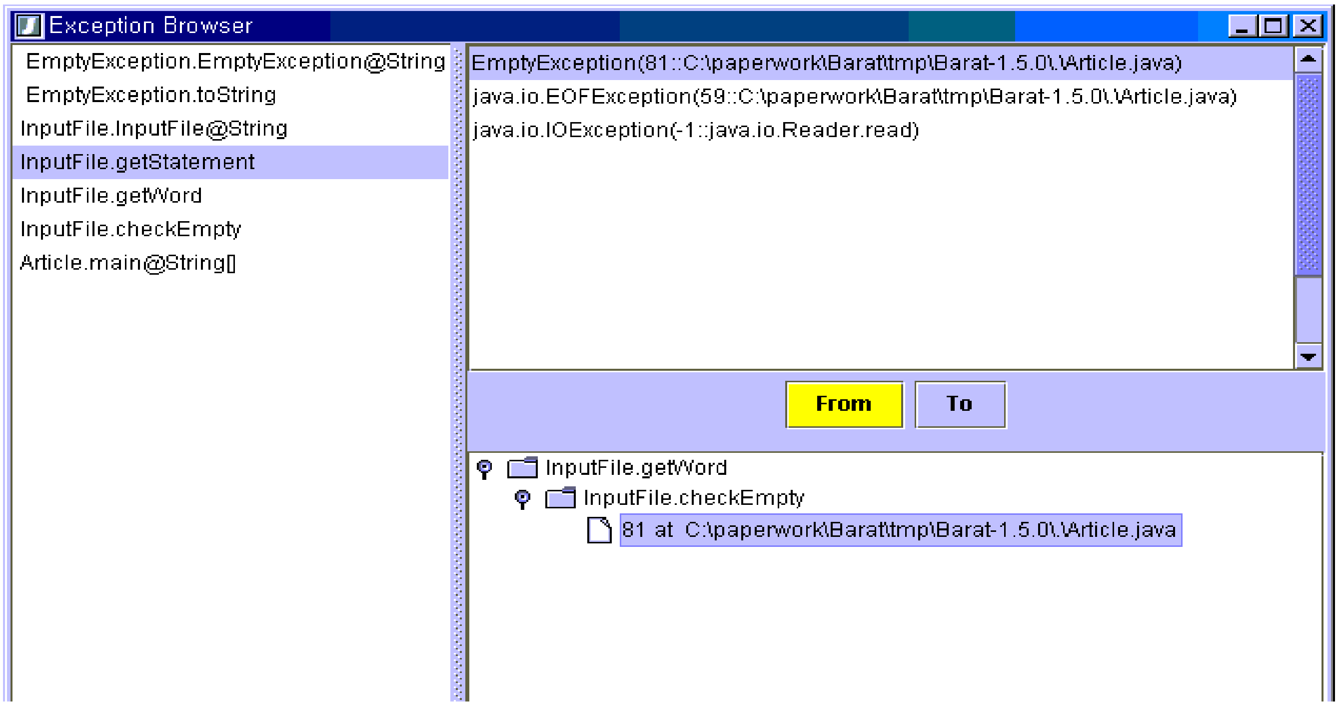
3. Java programs use libraries, which may have no source code. In case no source code is available, our analyzer also depends on the `throws` declarations as in JDK compiler.

As in Figure 6, our system consists of five subsystems:

1. set-constrains construction, which constructs set-constraints for a Java input program
2. constraint solving, which solves the set-constraints
3. graph construction, which constructs exception propagation graphs from the solution
4. path construction, which constructs propagation paths for a thrown exception, and
5. visualization, which visualizes exception propagation paths.

Our exception propagation analysis is implemented in Java on top of Barat framework [26], which is a front-end for a Java compiler. Barat builds an abstract syntax tree for an input Java program and enriches it with type and name analysis information. It also provides interfaces for traversing abstract syntax trees, based on visitor design pattern in [9].

The implementation of our analysis consists of two passes. The first pass sets up set-constraints by traversing the input Java program. The first pass is implemented by writing visitors so as to construct set-constraints. In constructing set-constraints, we need a naming convention for the indices of set variables. Instead of simply naming by number, for example $\mathcal{X}_1, \mathcal{X}_2, ..., \mathcal{X}_n$, we use source code information such as package name, class name, method name, and try-



```
EmptyException.EmptyException@String
EmptyException.toString
InputFile.InputFile@String
InputFile.getStatement
InputFile.getWord
InputFile.checkEmpty
Article.main@String[]
```

```
EmptyException(81::C:\paperwork\Barat\tmp\Barat-1.5.0\.\Article.java)
java.io.EOFException(59::C:\paperwork\Barat\tmp\Barat-1.5.0\.\Article.java)
java.io.IOException(-1::java.io.Reader.read)
```

```
InputFile.getWord
    InputFile.checkEmpty
        81 at C:\paperwork\Barat\tmp\Barat-1.5.0\.\Article.java
```