# A Review on Exception Analysis and its Applications (Extended Abstract)

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

## ABSTRACT

Exception handling has become popular in most major languages, including C++, Java, Ada, and ML. 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. This paper aims to summarize works so far on exception analyses and their applications. We first review several exception analyses including exception usage analysis, uncaught exception analysis and exception propagation analysis. We also review applications of exception analyses including control-flow graph representation, software development tools, and visualization.

## 1. INTRODUCTION

Since exception handling was pioneered by the language PL/I, it has become popular in most major languages, including C++, Java, Ada, and ML. An exception is any unexpected or unusual event, such as input failure or a timeout. An exception handler is a code sequence that is designed to catch and handle a particular exception, when it is raised or thrown. Exception facilities, for example, in Java allow the programmer to define, throw and catch exceptional conditions [9]. Exceptions and exception handling aim to support the development of robust programs with reliable error detection, and fast error handling.

In Java, there are two kinds of exceptions: *checked* and *unchecked* exceptions [9]. Checked exceptions must be specified at a method definition, if they are not caught. Unchecked exceptions are exempt from the requirement of being specified. Java compiler checks whether checked exceptions are caught or specified. 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.

There are several research directions for exception handling and analysis.

(1) The first one is to approximate dynamic behavior of thrown exceptions by static analysis, which includes analysis of uncaught exception and of exception propagation flow [9, 23, 1, 16, 3]. This kind of analysis information can be used to facilitate understanding of the exception behavior and to make better use of the exception mechanism.

(2) The second one is to model program execution flow based on exception analysis so as to incorporate normal and exceptional control flows. This flow information is usually represented as control-flow graph(CFG) which can be used to perform various kinds of analysis, slicing and structural testing.

(3) The third one is to develop applications of exception analyses, which include compiler optimizations, software development tools and visualization.

This paper aims to summarize works on exception analyses and their applications. We first review several exception analyses including exception usage analysis, uncaught exception analysis and exception propagation analysis. We will focus on exception analyses for Java, even though we review all exception analysis proposed so far. We then review applications of exception analyses, which include control-flow graph representation, software development tools, and visualization.

Section 2 presents a background for exception handling of Java, and for constraint-based analysis. Section 3 presents a number of exception analyses proposed so far. Section 4 presents applications of exception analyses. Section 5 concludes this paper.

## 2. BACKGROUND

We first introduce exception facilities in Java, which allow the programmer to define, throw and catch exceptional conditions [9]. Programmers can define exceptions as first-class objects in Java. Like normal objects, they can be defined by classes, instantiated, assigned to variables, passes as parameters, etc.

The exception object $e_0$ is raised or thrown by throw statement `throw` $e_0$.

The `try-catch` statement

$$\texttt{try } S_0 \texttt{ catch } (c\ x)\ S_1$$

evaluates $S_0$ first. If the statement don't raise exception, the execution is normal and continue to execute the next statement. If an exception is raised from $S_0$ and its class is covered by $c$ then the handler statement $S_1$ 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

This work was supported by grant No. (R01-2002-00363-0) from the Basic Research Program of the Korea Science & Engineering Foundation. This work has been done while the author is visiting the University of Pennsylvania as a visiting scholar.

chain until it meets another handler.

In Java, the programmers have to declare, using `throws` clause in a method definition, any exception classes whose exceptions may escape from its body without being caught. The current JDK compiler checks whether thrown exceptions are caught or specified by `throws` clause.

In [6, 14], they have shown the type soundness of a subset of Java. It was extended so as to incorporate exception constructs of Java in [7].

For exception analysis, we construct a constraint: $\mathcal{X}_S \supseteq se$ for every statement $S$ of the program. The $\mathcal{X}_S$ is for the exception classes that the statement $S$'s uncaught exceptions belong to. The meaning of a set constraint $\mathcal{X}_S \supseteq se$ is intuitive: set $\mathcal{X}_S$ 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 [11]. We write $lm(\mathcal{C})$ for the least model of a collection $\mathcal{C}$ of constraints.

# 3. RESEARCH ON EXCEPTION ANALYSIS

## 3.1 Exception Usage Analysis

Ryder and colleagues [17] developed a tool called JESP to conduct a study of the usage patterns of exception-handling constructs such as `try, catch, finally,` and `throw` in Java programs. Their study shows that substantial percentage of methods (on average 16%) contains exception constructs. Exception constructs are sparsely used and almost all `try`s have one `catch` clause. `Finally`s are rarely used and thrown exceptions are usually not caught in the same method. User-defined exception hierarchies are shallow as expected.

Sinha and Harrold [18] present a new control-flow representation to model exception flow for data-flow and control-dependence analyses. They also conducted a static study of seven Java programs, and found that on average 23 % of their methods contained a `try` or a `catch`. This observation, shows that analysis algorithms will have to take account of exceptions. Their study offers an evidence to support the belief that exception-handling constructs are used frequently in Java programs.

## 3.2 Uncaught Exception Analysis

Because unhandled exceptions will abort the program's execution, it is important to check at compile-time whether thrown exceptions are caught or not. This is called uncaught exception analysis.

Historically, uncaught exception analysis was first introduced for ML based on abstract interpretation [22], which is shown to be very slow. So, the analysis was redesigned based on set-constraint framework to improve analysis speed [24], and compared with [8]. Their recent implementation was integrated in SML/NJ compiler to give programmers information on potential uncaught exceptions [25].

Fähndrich and Aiken [8] 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 designed an exception analysis for OCaml based on type and effect systems, and it provides good performance for real OCaml programs [15].

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

In [23, 1], an efficient interprocedural exception analysis was proposed by applying the idea in [24] to Java so as to estimate uncaught exceptions independently of programmers's specified exceptions. The analysis is designed based on set-constraint analysis framework in [11].

They make one set variable for every statement and construct set-constraint for every statement. For example, let's consider the rule for `try` statement
$S = \texttt{try}\ S_0\ \texttt{catch}\ (c_1\ x_1)\ S_1$

$$\frac{S_0 \rhd \mathcal{C}_0 \quad S_1 \rhd \mathcal{C}_1}{\texttt{try}\ S_0\ \texttt{catch}(c_1\ x_1)\ S_1 \rhd \{\mathcal{X}_S \supseteq (\mathcal{X}_{S_0} - \{c_1\}^*) \cup \mathcal{X}_{S_1}\} \cup \mathcal{C}_0 \cup \mathcal{C}_1}$$

Raised exceptions from $S_0$ 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 $S_1$. Hence, $\mathcal{X}_S \supseteq (\mathcal{X}_{S_0} - \{c_1\}^*) \cup \mathcal{X}_{S_1}$, where $\{c_1\}^*$ represents all the subclasses of a class $c_1$.

They first designed an exception analysis at statement-level and then designed a sparse exception analysis at method-level for cost-effectiveness. It is shown theoretically and experimentally that the sparse exception analysis gives the same exception information for *every method* as the expression-level analysis.

They implemented the analysis on top of Barat [26], which is a front end of Java compiler. They also compared it with JDK-style analysis by experiments on realistic Java programs. They also have shown that the analysis is able to detect uncaught exceptions, unnecessary `catch` and `throws` clauses effectively.

Chang et al. generalized the idea of design of sparse analysis by a transformational approach , which was proposed to design more efficient constraint-based analyses for Java at a coarser granularity [2]. In this approach, a less or equally precise but more efficient version of an original analysis can be designed by transforming the original constraint construction rules.

Robillard and Murphy [16] have developed a similar tool called Jex for analyzing uncaught exceptions in Java. They also take account of some unchecked exceptions. They first designed intraprocedual analysis and then improved it to interprocedual analysis. Jex extracts the uncaught exceptions in a Java program, and also generates views of the exception structure.

## 3.3 Exception Propagation Analysis

Exception analyses usually estimate uncaught exceptions only by their names and structure, so that they cannot provide information on dynamic behavior of thrown exceptions such as their *propagation paths*, which are necessary to construct interprocedural control flow graph, visualize exception propagation, and slice exception-related parts of programs [18].

In [3], Chang et al. present a static analysis by extending the work in [1] so as to estimate exception propagation paths of thrown exceptions in Java programs. An exception propagation graph is constructed from the static analysis information, which includes the origin, handlers, and propagation paths of thrown exceptions.

This analysis is designed by extending the uncaught exception analysis in [1] so that it can include propagation paths of thrown exceptions. For example, let's consider the rule for the `try-catch` statement $S$ with a label $\ell$ :

$$\frac{S_0 \rhd \mathcal{C}_0 \quad S_1 \rhd \mathcal{C}_1}{\begin{array}{c}\ell : \texttt{try}\ S_0\ \texttt{catch}(c_1\ x_1)\ S_1 \rhd \{\mathcal{X}_S \supseteq ((\mathcal{X}_{S_0} - \{c_1\}^*) \cup \mathcal{X}_{S_1}) \cdot \ell\} \\ \cup \mathcal{C}_0 \cup \mathcal{C}_1\end{array}}$$

Thrown exceptions from $S_0$ 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 $S_1$. Uncaught exceptions from this expression are followed by the label $\ell$ to record the exception propagation path. Hence, $\mathcal{X}_S \supseteq ((\mathcal{X}_{S_0} - \{c_1\}^*) \cup \mathcal{X}_{S_1}) \cdot \ell$.

They implemented the exception propagation analysis by extending the uncaught exception analysis in [1]. Exception propagation graph can be constructed from the analysis results.

# 4. APPLICATIONS

## 4.1 Control-flow Representation

The *control-flow graph*(CFG) is a representation of control flow relation that exists in a program. Many program-analysis techniques, such as data-flow and control-dependence analysis depend on control-flow information. For these analyses to be safe and useful, the control-flow representation should incorporate the exception-induced control flow.

Recently, several researchers have constructed control-flow representation for exception-related constructs based on exception analysis [5, 18], and they considered the effects of exception-induced control flow on various types of analyses. Sinha and Harold present an algorithm which constructs control-flow representations for programs with exception-handling constructs by analyzing exceptional control-flow together with normal control-flow analysis [18]. They also discuss the effects of exception-handling constructs on several static analyses such as control-flow, data-flow, and control dependence analysis. Choi and colleagues [5] construct control-flow representation called the factored control-flow graph (FCFG) for exception-handling constructs, and use the representation to perform data-flow analyses. Chatterjee and Ryder [4] describe an approach to performing points-to analysis that incorporates exceptional control flow. They also provide an algorithm for computing definition-use pairs that arise because of exception variables, and along exceptional control-flow paths.

Control-flow and control-dependence analysis are useful for software engineering and maintenance tasks. The control-flow representation can be applied to performing slicing and testing of the programs with exception constructs. There are two alternative approaches to computing slices, that either propagate solutions of data-flow equations using a control-flow representation [21, 10], or perform graph reachability on system dependence graphs[12, 20]. Using the interprocedural control-flow representation [20], the slicing technique in [10] can be extended to take into consideration the effects of exception-handling constructs.

Structural testing develops test cases to cover structural elements of a program. Control-based structural testing criteria use the flow of control in a program to guide the selection test cases or to access the adequacy of a test suit. For example, in branch testing, test cases are developed by considering inputs that cause certain branches to be executed.

Exception-handling constructs introduce new structural elements, such as exceptional control-flow paths, that should be considered for coverage by structural test techniques. In [19], they have developed a family of exception testing criteria to adequately test the behavior of exception-handling constructs. These criteria subsume the all-throw and all-catch criteria, and test exception-handling constructs with various degree of thoroughness.

## 4.2 Development tools

In [8], they also developed a program analysis mode for EMACS editor, providing a textual point-and-click interface for displaying the results of an exception analysis of ML programs. This system can give the list of declared exceptions, a list of handlers, a list of function declarations and a list of potentially uncaught exceptions. By using this system, it is easy to start at the body of the main function of a program and follow the uncaught exceptions backwards to see where they were raised.

A similar tool Jex is also developed for Java in [16], even though it is not integrated into an editor. Jex provides information about the exceptions that can be raised in a Java program. For each method of each class, Jex outputs a description of all exceptions that can be raised in the method. The description shows the origin of each exception.

Here is an example describing the flow of exceptions for an hypothetical constructor, in the Jex format, which is from [**?**].

```
<init>(java.lang.String,boolean) throws IOException {
   java.lang.SecurityException:SecurityManager.
   checkWrite(java.lang.String);
   try
   {
      java.io.IOException:java.io.FileOutputStream.
      openAppend(java.lang.String);
      java.io.IOException:java.io.FileOutputStream.
      open(java.lang.String);
      java.lang.NullPointerException:*environment*;
   }
   catch( java.lang.IOException )
   {
      throws java.io.FileNotFoundException;
   }
}
```

This example shows that an `SecurityException` can be raised as a result of the call to method `checkWrite` of class `SecurityManager`. Furthermore, in the try block, `IOExceptions` can result from the calls to methods `openAppend` and open of class `FileOutputStream`, and a `NullPointerException` can be raised by the run-time environment. The view of exception flow produced by Jex is more precise and more complete than the information available through exception interfaces. Jex also reports the origin of exceptions

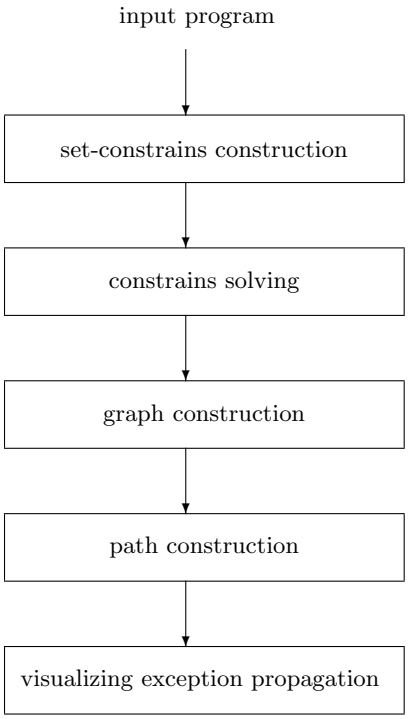


**Figure 1: Visualization system architecture**

stemming from polymorphic calls by using a conservative class hierarchy analysis algorithm.

## 4.3 Visualization

The exception propagation analysis information can be used to visualize exception propagation. This can include the origin of exceptions, handler of exceptions, and propagation paths of exceptions.

In [3], they also implemented a visualization tool on top of an open source IDE called Jipe [27], which can visualize interactively propagation paths of exceptions using the exception propagation graph.

As in Figure 1, the visualization 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.

When programmers selects a method, it shows all uncaught exceptions from that method. If programmers choose one of them, then it shows where it is propagated to or from. This process can be continued until it is caught or get to the main method.

As shown in Figure 2, when users select a method `getStatement`, it first displays its all uncaught exceptions. When users select the uncaught exception `EmptyException`, it then displays the exception propagation path of that exception, which originates from the line 81, passes through the methods `checkEmpty` and `getWord` and arrive to the selected method `getStatement`.

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. The contribution of this work is that it can estimate exception propagation paths, while others estimate a set of uncaught exceptions rather than their propagation paths.

# 5. CONCLUSION

We have given an overview of exception handling and various exception analyses. We have also addressed the impact of them by presenting their applications. Applications such as slicing, structural testing and compiler optimizations are not still mature and need to be explored in more detail.

# 6. REFERENCES

[1] B.-M. Chang, J. Jo, K. Yi, and K. Choe, Interprocedural Exception Analysis for Java, *Proceedings of ACM Symposium on Applied Computing*, pp 620-625, Mar. 2001 .

[2] B.-M. Chang and J. Jo, Granularity of Constraint-based Analysis for Java, *ACM Conference on Principles and Practice of Declarative Programming*, September 2001.

[3] B.-M. Chang, J. Jo, and S. Her, Visualization of Exception Propagation for Java using Static Analysis, *Proceedings of IEEE Workshop on Source Code Analysis and Manipulation*, Oct. 2002.

[4] 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.

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

[6] S. Drossopoulou, and S. Eisenbach, Java is type safe-probably, *Proceedings of 97 European Conference on Object-Oriented Programming*, 1997

[7] 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.

[8] 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.

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

[10] M. Harrold and N. Ci, Reuse Driven Interprocedural Slicing, *Proceedings of the International Conference on Software Engineering,* April 1998.

[11] N. Heintze, Set-based program analysis. Ph.D thesis, Carnegi e Mellon University, October 1992.

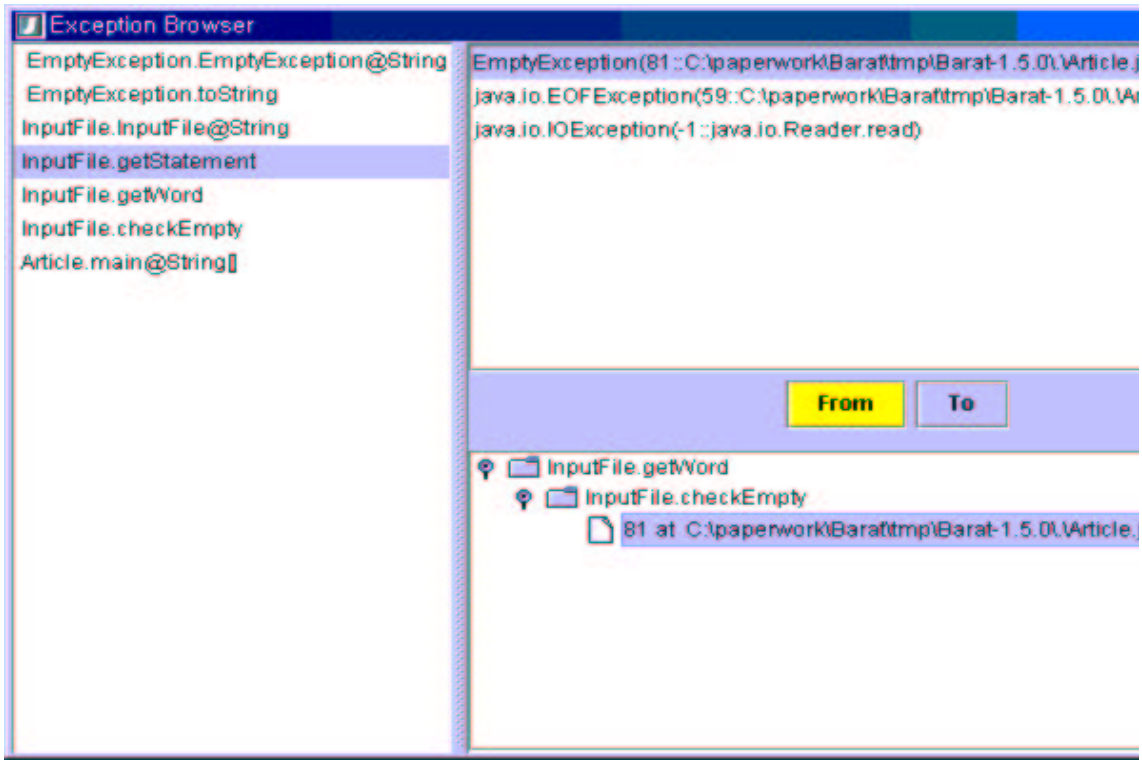


**Figure 2: Visualization of exception propagation**


[12] S. Horwitz, T. Reps, and D. Binkley, Interprocedural slicing using dependence graphs, *ACM Transactions on Programming Languages and Systems*, 11(3), pp 345-387, July 1989.

[13] Jipe, http://jipe.sourceforge.net.

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

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

[16] M. P. Robillard and G. C. Murphy, Analyzing exception flow in Java programs, in *Proc. of '99 European Software Engineering Conference and ACM SIGSOFT Symposium on Foundations of Software Engineering*, pp. 322-337, Springer-Verlag.

[17] 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.

[18] S. Sinha and M. Harrold, Analysis and testing of programs with exception-handling constructs, *IEEE Transations on Software Engineering* 26(9) (2000).

[19] S. Sinha and M. Harrold, Criteria for testing exception-handling constructs in Java programs, In *Proc. of the Int. Confenerce on Software Maintenance*, September 1999, pp. 265-274.

[20] S. Sinha, M. Harrold, and G. Rothermel, System-Dependence-Graph-Based Slicing of Programs with Arbitrary Inte rprocedural Control Flow, *Proceedings of the International Conference on Software Engineering,* May 1999, pp. 432-441.

[21] M. Weiser, Program Slicing, *IEEE Transations on Software Engineering,* 10(4), pp 352-357, July 1984.

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

[23] K. Yi and B.-M. Chang Exception analysis for Java, ECOOP Workshop on Formal Techniques for Java Programs , June 1999, Lisbon, Portugal.

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

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

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

[27] http://jipe.sourceforge.net.