Analyzing Exception Flow in Java™ Programs

by

Martin P. Robillard

B.Eng. (Computer Engineering)

École Polytechnique de Montréal

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

we accept this thesis as conforming
to the required standard

The University of British Columbia

September 1999

Abstract

Exception handling mechanisms provided by programming languages are intended to ease the difficulty of developing robust software systems. Using these mechanisms, a software developer can describe the exceptional conditions a module might raise, and the response of the module to exceptional conditions that may occur as it is executing. Creating a robust system from such a localized view requires a developer to reason about the flow of exceptions across modules. The use of unchecked exceptions, and in object-oriented languages, subsumption, makes it difficult for a software developer to perform this reasoning manually. In this thesis, I describe an approach for analyzing the flow of exceptions in Java source code to produce views of the exception structure. The approach is supported by a tool called Jex. I demonstrate how Jex can help a developer identify program points where exceptions are caught accidentally, where there is an opportunity to add finer-grain recovery code, and where error handling policies are not being followed.

Contents

List of Tables

List of Figures

Acknowledgements

This Master's project would not have been possible without the generous help and encouragement of my supervisor, Gail Murphy. I would also like to thank my friend and colleague Stéphane Durocher, and Prof. Alan Wagner, for patiently reading through this thesis, providing me with very helpful comments.

MARTIN P. ROBILLARD

The University of British Columbia

September 1999

Chapter 1

Introduction

From the early days of computing, software developers have recognized that programs can encounter various situations that prevent a correct continuance of the sequence of instructions. Conditions such as an arithmetic logical unit reporting a division by zero, a system running out of memory, or an error reported by input/output devices can prevent programs from producing meaningful results. Detecting and reporting such situations presents an opportunity to recover from, or to provide more details about these kinds of problems.

These disruptions of normal program execution are called *exceptions*, and the mechanism to detect and react to these exceptions is called *exception handling*. Syntactically, an exception handling mechanism consists of a means to explicitly raise an exceptional condition at a program point, and a means of expressing a block of code to handle one or more exceptional conditions. Software exceptions can be supported at the operating system level (e.g., Mach [3], Windows NT [9]), or at the programming language level (e.g. Ada95 [1], C++ [26], Java [12], Modula-3 [5]).

One of the main goals of exception handling is to separate code dealing with unusual situations from the code supporting normal processing. This usually leads to cleaner, more understandable programs.

Unfortunately, since exceptions can propagate in a program, local reasoning about the code is not generally sufficient to develop a module that will react appropriately to all unexpected situations. This thesis shows that the lack of information describing the global flow of exceptions, especially in object-oriented languages, can prevent developers from fully leveraging the power of exception handling mechanisms. For example, a lack of information about which exceptions can arise at a particular program point can make it difficult for developers to effectively implement error handling policies. To compensate for this information gap, I introduce an approach to provide information about the flow of exceptions. The approach has been implemented in a tool that performs the extraction of exception information from programs written in Java.

The remainder of this chapter presents the general concepts of exception handling.

1.1 Exception Handling Concepts and Terminology

1.1.1 Defining Exceptions

Although the implementation of exception systems varies in different programming languages and operating systems, there seems to be a consensus on the more general concept of an exception [8, 11, 19, 20, 21]. An *exception* can be defined as "an abnormal computation state" [21, p. 86]. This definition applies both to *programming-language-based* and *operating-system-based* exceptions [19]. This thesis, however, focuses exclusively on programming-language-based exceptions.

Exceptions can be classified as either *pre-defined* or *user-defined* [8, 11, 19]. Pre-defined exceptions are generally associated with conditions that are detected by the system. User-defined exceptions are defined and detected at the application level [19].

An exception is *raised*, or *signaled*, when the corresponding abnormal state is detected. Exceptions can be raised *implicitly* or *explicitly* [11]. Exceptions potentially raised by function calls or language-defined operators (e.g., arithmetic operators) are said to be raised implicitly, while exceptions raised deterministically using a command or language-defined keyword, like `raise` or `signal`, are said to be raised explicitly.

1.1.2 Exception Handling Mechanisms

Exceptions would not be useful if they always resulted in the abnormal termination of a program. For this reason, programming languages supporting exceptions also provide *exception handling mechanisms*. Exception handling mechanisms allow programmers to define code to be executed when certain exceptions are detected, and to link the occurrence of exceptions to the corresponding code. An *exception handler* is the code executed in response to an exception. In most languages, when an exception is raised, the system halts the execution of the program and searches for a handler for the exception. The search starts in the *target*, usually the enclosing syntactic scope. An exception is said to be *handled* when a handler for it is found and the corresponding code has been executed. If the handler is found directly in the target, the exception is said to be *masked*. Otherwise, the exception is *propagated* to an enclosing scope selected according to the implementation of the exception handling mechanism. Exceptions can be propagated *implicitly* (automatic propagation), or *explicitly.* In systems supporting automatic propagation, if no handler for an exception is found in a target, the exception is automatically *re-raised.* With explicit propagation, in order to be propagated, an exception has to be explicitly re-raised in a handler. There exists different models describing the control flow between signalers, targets, and handlers. The most common models are described in the next section.

1.1.3 Exception Models

This section describes three well-known exception models: the *termination* model, the *resumption* model, and the *retry* model. The descriptions of the models are adapted from [21].

Termination Model In the termination model, the scope of the signaler is destroyed, and, if a handler is found, control resumes at the first syntactic unit following this handler.

Resumption Model In the resumption model, once an exception is handled, computation continues from the point where the exception was originally raised.

Retry Model In the retry model, when the exception is handled, the signaler's block is terminated and then retried.

Figure 1.1 illustrates the three models. In every schema, the topmost block represents the syntactic unit where the exception is raised. The shaded block represent the exception handling code, the bottommost block represents the next logical control block, and the arrows represent control flow.

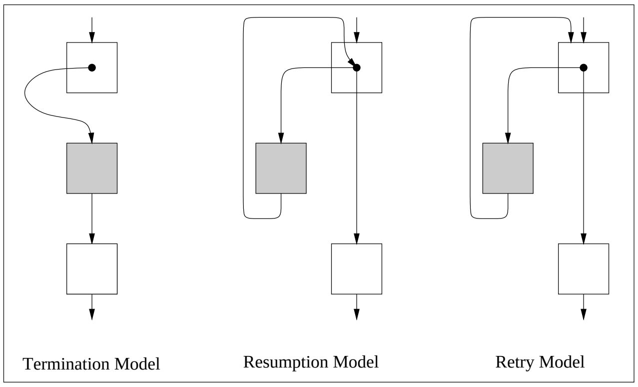

Figure 1.1: Exception Models

1.1.4 Hierarchies of Exceptions

Exceptions are represented differently in different languages. They can be defined as various program entities, such as data types, procedures, values, or programming language primitives. Depending on their representation, we can classify exceptions as having either a *singular* or *hierarchical* structure [19].[1] Exceptions implementing the singular structure are unrelated and there is no way to group them together. Examples of programming languages implementing the singular exception structure include CLU and Ada.

In languages implementing exceptions in a hierarchical structure, an exception can have several sub-exceptions (and, in some cases, an exception can have several parents). This is the de facto standard for object-oriented languages representing exceptions as objects (e.g., Java, C++). There are obvious advantages to representing exceptions in a hierarchy. First, the hierarchy provides a semantic organization for exceptions. For example, an exception representing a file not found

[1] For the purpose of this work, I do not retain the nuance between the *hierarchical* and *object* structures as described by Lang and Stewart [19]. Both type of structures are considered hierarchical.

and an exception representing the end of a file being read are both IO exceptions. A second advantage, for languages that map the exception structure to their type system, is that the handler for a parent exception can naturally handle all its child exceptions [19]. As we will see in section 1.2, there is unfortunately a cost associated with this property.

1.2 The Flow of Exceptions

1.2.1 Exception Interfaces

The propagation of exceptions introduces the possibility of non-local control flow. If the caller of a module ignores the exceptions that can cross the module's boundary, the caller cannot adequately prepare for these exceptions, and robustness problems may arise. For this reason, many programming languages support *exception interfaces* [19] (or *exception specifications* [21]). An exception interface "is the part in a module interface that explicitly specifies the exceptions that might be raised by the module" [19, p. 295]. Usually the system prevents exceptions that are not declared in the interface to propagate outside of the module boundaries. C++, CLU, and Java are examples of languages supporting exception interfaces (see section 2.1). In addition to providing information to users of a module, exception interfaces can be statically checked and enforced, thus providing an additional level of reliability.

The problem with exception interfaces is that, in practice, they cannot be exhaustive. Indeed, it would be prohibitive for a programmer to have to both figure out and to declare the complete set of exceptions that a module can raise, mostly because of the high frequency of redundant system exceptions, like arithmetic, null pointer, or memory-related exceptions. To address this issue, languages enforcing exception interfaces typically provide a means to bypass the checking mechanism. This is done, for example, by providing mechanisms for specifying exceptions that do not have to be checked (see section 2.1). In object-oriented languages representing exceptions as objects, there exists a second way of limiting the precision of exception interfaces. Since exceptions are organized in a hierarchy, declaring a general supertype of some exceptions automatically declares, by extension, all of its more specialized subtypes.

1.2.2 A Metric of Exception Flow

Keeping in mind the exception interfaces between modules, we can imagine a simple metric of exception flow between module boundaries. The metric describes the precision and completeness of information that is available to programmers (and to static checkers). This metric can be described as a simple two-dimensional space. The first dimension quantifies the *completeness* of the exception interface, and the second dimension quantifies the *granularity* (or *precision*) of the interface. Completeness designates the relative fraction of different exceptions crossing the module boundary that are actually declared in the interface. Let C_m be the completeness of the exception interface for a given module m, let E be the complete set of exceptions that can cross a module's boundary, and let E_d be the set of exceptions declared in m's exception interface. To take into account any hierarchical properties of exceptions that may exist in a system, E_c is defined as the set of exceptions in E that are either elements of E_d, or that can be functionally considered as a child of any exception in E_d.

Completeness can then be defined as

$$C_m \equiv \frac{\mid E_c \mid}{\mid E \mid} \tag{1.1}$$

The granularity G_m of an exception interface for a given module m, cannot be defined as intuitively as the completeness. For the purpose of this discussion, I shall define the granularity of an exception interface as the fraction of elements in E_c that are explicitly declared in the interface. Formally, we have

$$G_m \equiv \frac{\mid E_d \mid}{\mid E_c \cup E_d \mid} \tag{1.2}$$

I chose to include the set of declared exceptions (E_d) in the denominator both to bound the granularity metric to one, and to avoid the situation where an exception $e \in E_d$, $e \notin E_c$ cancels the effect of another exception in E_c.

We see that in the case where $E_d = E_c$, we can assume a very fine granularity, as every exception that crosses the interface and that can actually match it is precisely known. Thus finer granularity is represented by a G_m closer to one, while a coarser granularity is represented by a value closer to zero.

Figure 1.2 shows the basic representation of the two-dimensional quantification of exception flow through exception interfaces. The points noted m_1 through m_4 represent four particular cases of exception interfaces. The interface corresponding to module m_1 has a high granularity but low completeness. This means that exceptions crossing the module's boundary, if they are in E_c, will tend to correspond exactly to what is declared in the interface, and not to some subtype. However, many exception are not in E_c. The interface corresponding to module m_2 has both a high granularity and completeness. Exceptions crossing m_2's boundaries are precisely and exhaustively defined. Module m_3 is an example of a very complete but coarse specification. This kind of specification can be found in practice when a module simply declares to be raising a very general type of exception, and all the more specialized children are thus implicitly declared. Finally, module m_4 has a very weak exception interface, in the sense that it is both incomplete and coarse-grain. Very little information can be obtained from such an interface.

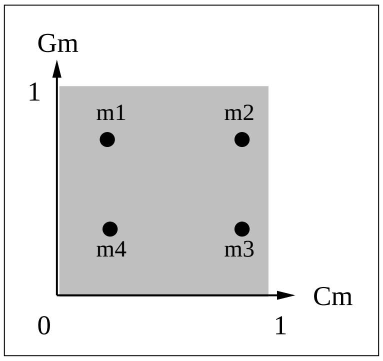

Figure 1.2: A Metric for Quantifying Exception Interfaces

In the general sense we can interpret the metric presented in this section the following way: the closer a point is to the upper-right corner, the more descriptive the corresponding interface. Conversely, the closer a point is to the origin, the more loosely defined the corresponding interface.

The C_m–G_m metric has some obvious limitations. First, it can only quantify module interfaces actually declaring exceptions, even though not declaring any exceptions does not imply that no exception will be raised. Second, the value of its interpretation decreases with the values of $\mid E \mid$, $\mid E_d \mid$, and $\mid E_c \mid$. However, we can see that, over a large number of modules, the metric is sufficient to describe the quantity of information provided by the use of exception specifications. Reference to this simple metric will be used later to describe various exception handling approaches (section 4.3).

1.3 Exception Handling in Java

Since the work described in this thesis applies to the Java language, it is necessary to describe how the exception system is implemented in this language. This section describes how the concepts presented previously are implemented in Java.

In Java, exceptions are represented as first-class objects. As such, they can be instantiated, assigned to variables, passed as parameters, etc. An exception is explicitly signaled using a `throw` statement. Code can be guarded for exceptions within a `try` block. A `try` block is basically a syntactic scope defining the target. Exceptions signaled through execution of code within a `try` block may be caught in one or more `catch` clauses declared immediately following the `try` block. Optionally, a programmer can provide a `finally` block that is executed independently of what happens in the `try` block. Exceptions thrown in the `finally` block mask any exception that would have been thrown in the `try` block. Figure 1.3 illustrates Java's basic exception handling structures.

```
try
{
   // Code potentially raising IOException
}
catch( IOException e )
{
   // Code recovering from an IOException
}
finally
{
   // Some finalization code
}
```

Figure 1.3: Exception Handling Structures in Java

Java supports the termination model (see section 1.1.3) with automatic propagation. Exceptions not caught in any `catch` block are propagated back to the next level of `try` block scope, possibly in the caller module.

Like all other objects in Java, exceptions are typed. Exceptions are thus organized in a hierarchy corresponding to their type.

What distinguishes exceptions from other objects is that all exceptions inherit from the class type `java.lang.Throwable`. The exception type hierarchy defines three semantically and functionally different groups of exception types: *errors*, *runtime* exceptions, and *checked* exceptions (Figure 1.4).

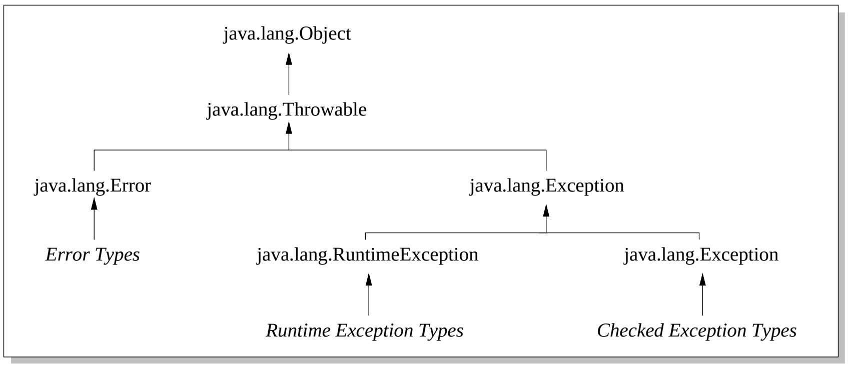

Figure 1.4: The Java Exception Type Hierarchy

Java enforces a partial exception interface. Errors and runtime exceptions are unchecked by the compiler and do not have to be declared in the method headers. Unchecked exceptions can be thrown at any point in a program and, if uncaught, may propagate back to the program entry point, causing the Java Virtual Machine to terminate abnormally. Errors represent unrecoverable conditions and are typically raised by the virtual machine. As opposed to unchecked exceptions, checked exceptions that can potentially be raised in a method and that are not masked need to be declared in the header of the corresponding method. The language also requires exception conformance [21], so a method M' overriding the method M of a supertype must not declare any exception type that is not the same type or a subtype of the exception types declared by M. Exceptions types present in the exception interface thus vary covariantly with the method type.

The ability to declare exceptions within a hierarchy also means that an exception may be cast back implicitly to one of its supertypes when a widening conversion requires it. For example, this conversion occurs when the assignment of an object of a subtype is made to a variable declared to be of its supertype. This property is called *subsumption* [2]; a subtype is said to be *subsumed* to the parent type. When looking for a handler, exceptions can be subsumed into the type of the target `catch` clause if the type associated with the `catch` clause is a supertype of the exception type. Similarly, a method declaring an exception type E can throw any of the subtypes of E without having to explicitly declare them.

As we will see in more detail later, Java's support for unchecked exceptions and subsumption means that it is difficult for a software developer to know the actual set of exceptions that may cross a method's boundaries. The following section describes the information that is necessary to gain this knowledge.

1.4 Motivation and Thesis Statement

To design and implement a robust and reliable system, local reasoning about the code is generally insufficient. In some applications, such as games, it may be sufficient to trap an unexpected condition, write a generic error message, and terminate. In many other applications, it is preferable to either recover silently, or at least provide a meaningful error message. For example, a user of a word-processor trying to open a file may want to know that a file sharing violation has occurred and be allowed to correct the problem, rather than just being told there was a file problem.

Fine-grain reactions to exceptions require a software engineer to reason about the code on which the module being constructed depends. This includes being able to reason about the exceptions that might flow out of a module.

In section 1.2.2, we have seen that the precision of exception interfaces—represented as the combination of granularity and completeness—can basically vary between all extremes. This variability is a factor of three things:

- the characteristics of the language of implementation;
- the actual interface specification; and
- the implementation of the module.

Since Java supports both exception type hierarchies and unchecked exceptions, we can assume that exception interfaces in that language can take any value on the granularity-completeness grid. Hence, it is possible for developers to specify exception interfaces that carry very little information about the flow of exceptions across module boundaries. The rationale behind such a design is not necessary negligence. It can stem from a lack of information about the modules being used. With languages actually enforcing interfaces at compile time, the choice to broadly define the specification of an interface might be additionally driven by the concern for greater compatibility with other components, or simply by convenience for users of the module.

Whatever the reasons, it is not practical to expect an exception interface to completely and unambiguously specify every single type of exception that can cross a method's boundary. The hypothesis underlying the work described in this thesis is that, to produce quality code, developers need to have access to more complete and precise exception flow information than what typical exception interfaces can provide them. In an attempt to provide this information to software developers, I propose an approach, based on static analysis techniques, to extract exception flow information from source code.

1.5 Overview

This chapter presented the use of exceptions, provided general background on exception handling, and motivated the purpose of the research described in this thesis. The remaining chapters of the thesis are organized as follows. Chapter 2 covers the related work. Chapter 3 describes the approach chosen to address the problem of exception handling in Java. Chapter 4 shows the results that can be obtained using the approach and the corresponding tool. Finally, chapter 5 discusses the applicability and general future directions of the work.

Chapter 2

Related Work

There exists two basic ways to obtain information about various aspects of a program. One is to refer to information that a software developer has explicitly provided, either in syntactic declarations or in the source code documentation. Another approach is to extract this information from unannotated program representations.

This chapter describes how both approaches can provide information about the flow of exceptions. Section 2.1 presents an overview of the mechanisms used to specify and check exception declarations in languages supporting exception handling. Section 2.2 discusses typical program analysis techniques. Unfortunately, to this day, most program analysis tools and techniques typically overlook exceptions. The section also provides an overview of analysis techniques that integrate exception flow analysis. Finally, section 2.3 describes some tools that were specifically built to extract information about the flow of exceptions, and explains how they differ from the work described in this thesis.

2.1 Specifying and Verifying Exception Interfaces

This section describes and compares six relatively modern programming languages directly supporting exceptions: Ada [1], C++ [26], CLU [20], Java [12], ML [14], and Modula-3 [5].

ML, Ada95 and Modula-3 do not support specification of exceptions in function declarations. ML is a functional language in which exceptions are values that can be declared anywhere in a program. These values can be signaled at any point following their declaration. Similarly, in Ada95, exceptions are simple name declarations. In Modula-3, exceptions are also names, but they can optionally be associated with a data type. In all three, ML, Ada95, and Modula-3, the representation of exceptions follows the singular structure (see section 1.1.4).

CLU, C++ and Java, on the other hand, support exception specifications. In CLU, like in Modula-3, exceptions are represented by a name to which is associated zero or more values. However, in CLU, for every exception signaled in a routine, the compiler ensures that there is a corresponding exception present in the routine declaration. Since exceptions have a singular representation, modules do not suffer from the granularity problem presented in section 1.1.4. However, completeness can be a problem because of a special exception called *failure*. This exception is implied in every exception interface (i.e., it does not have to be declared), and can represent any type of failure. Since a failure exception can describe all the different types of failures, this special exception reduces the descriptive power of the exception specification. In comparison to Java, the use of `failure` in CLU is roughly equivalent to using the Java exception type `RuntimeException` without specializing it. Nevertheless, according to the criteria of section 1.1.4, CLU is one of the languages enforcing the highest level of exception flow information in its module interfaces.

Table 2.1: Exception Handling Characteristics of Some Programming Languages

Language	Representation	Category	Exception Interface
ML	Named value	singular	No
Ada95	Name	singular	No
Modula-3	Named value	singular	No
CLU	Named set of values	singular	Yes (checked)
C++	Object	hierarchical	Yes (unchecked)
Java	Object	hierarchical	Yes (checked)

The C++ language specification ensures that a method can only raise exceptions it declares. If a method signature does not include the declaration of exceptions, it is assumed that all types of exceptions may be raised. However, C++ adopts a different strategy to enforce interfaces. It does not check clients of a function to make sure that declared exceptions are either masked or re-declared. Instead, any exception raised within the method that is not declared is re-mapped to a special `unexpected` exception. The developer of a client is not informed of missing handlers. Furthermore, in C++, exceptions are typed objects, and thus are subject to granularity degradation (see section 1.2.2).

Finally, as we have seen in section 1.3, exceptions in Java are typed objects. Java supports exception interfaces, and checks these interfaces only for a subset of all exceptions (checked exceptions). The combination of the type hierarchy in Java, and the fact that not all exceptions have to be declared in exception interfaces means that developers do not have precise and complete information about the number and type of exceptions potentially crossing a method's boundary.

Table 2.1 summarizes the characteristics of the exception systems of the languages discussed in this section.

2.2 Analysis Tools and Techniques

Many approaches and tools exist to help software engineers obtain information about properties of programs. These approaches are usually based on some form of intermediate program representation, such as system dependence graphs [16] or abstract syntax trees (AST). Historically, these approaches have mostly been used for languages without exceptions. In the case of languages that do support exceptions, the integration of exception-related structures complicates the intermediate representations of programs and is not relevant for most tasks, such as compiling. These reasons can serve to explain why most program analysis techniques typically avoid the consideration of exceptions.

Slicing [28] is a technique used to identify the flow of information responsible for a specific value at a program point. It is based on data-flow and control-flow analysis of programs, and has many software engineering applications, such as debugging and testing [25]. Recent work is beginning to incorporate exception information into data-flow and control-flow representations of programs. Sinha and Harrold describe techniques to model control-flow in the presence of exceptions [24, 25]. Choi et al. [7] describe a representation to improve procedural optimizations in the presence of exceptions. These efforts differ from my work in that their focus is on modeling program execution rather than on enabling a developer to make better use of exception mechanisms.

Program databases, like the C Information Abstraction System (CIA) [6], are tools that allow users to retrieve various types of information about a program, and to perform some analysis tasks, like call-graph generation and subsystem extraction. Most of the implementations of such systems were designed for the C language, which does not support exceptions. In the case of CIA, there exists a version for

C++ called ACACIA, but it does not currently support the querying of exception information.[1]

Finally, there exists many other systems that perform various analyses on Java programs specifically. Recent examples include CoffeeStrainer [4], an AST-based framework for the static checking of constraints on Java programs, and the Womble tool [17], built to extract object models from Java bytecode. Although such tools have access to much of the exception handling information, they typically ignore this information, mostly because the tasks they are intended to support do not involve exceptions.

2.3 Exception Analysis Approaches

Many exception analysis tools have been developed for ML, a functional language that represents exceptions as singular values and that does not support exception interfaces (see section 2.1). Theses two characteristics make it difficult for programmers to ensure that all exceptions are caught. Pessaux and Leroy report that uncaught exceptions are the most frequent mode of failure in large ML applications [22]. The goal of the tools is thus obvious: to help programmers identify the points in a program where different exceptions can be thrown.

Guzmán and Suárez have proposed an extension of the ML type system by which it is possible to estimate all uncaught exceptions that can be raised [13]. Their type system is limited in that it does not handle exceptions carrying arguments.

A different approach has been adopted by Yi, who developed an exception analyzer based on abstract interpretation techniques [29]. Since this analyzer suffered from severe performance problems, Yi and Ryu developed a more efficient

[1] Emden R. Gansner. AT&T Labs-Research. May 1999. Personal Communication.

one [31], using control-flow analysis and a set-constraints framework [15].

Fähndrich et al. [10] have built an Exception Analysis Tool (EAT) "that allows the programmer to display uncaught exceptions at certain program points while browsing the code" [10, p. 1]. EAT is based on BANE, a general framework for implementing constraint-based program analyses. Yi's tool is more precise than EAT, but EAT, which uses a more conservative approach, is more scalable. The EAT tool also provides support for visualizing the declaration and handling of exceptions at different points in the program. Pessaux and Leroy [22] propose a type-based analysis of uncaught exceptions in ML that offers different speed/precision tradeoffs than the previous constraint-based approaches.

The analysis tools proposed for ML have basically the same goal as the work described in this thesis: to provide programmers with information about the flow of exceptions, in order to allow them to design more robust code. However, there are many differences between functional and object-oriented languages in general, and between ML and Java in particular. Exception interfaces and hierarchies of exceptions, two concepts present in Java and absent in ML, introduce subtleties and tradeoffs that programmers must take into account, and that an analysis tool must consider. The focus of this work is on these latter concepts.

Finally, Yi and Chang [30] have sketched an approach within the set-constraint framework that would provide an exception flow analysis for Java similar to that implemented by the tool presented in this thesis. It is unclear whether formalization in the set-constraint framework will cause them to make different trade-offs between precision and scalability than have been made in the approach I propose.

Chapter 3

The Jex Approach and Tool

In the previous chapters, I have justified the need for providing information about the flow of exceptions in object-oriented programs. There remains the questions of *what* information should be extracted, and of *how* it should be presented to a user. Extracting information flow is not a well-defined problem and, as such, there exists numerous possible solutions. The specification and design of my approach is based on two considerations of a practical nature, namely, accessibility and usefulness of the information. In other words, the initial goal of the Jex approach is that the information produced has to be easy to interpret and useful.

This chapter describes the approach taken to extract exception information from Java programs (section 3.1), and discusses the details of the tool used to realize the extraction (section 3.2).

3.1 Extracting Exception Structure

Understanding and evaluating how exceptions are handled within a method requires reasoning about which exceptions might arise as a method is executing, which exceptions are handled and where, and which exceptions are passed on.

Manually extracting this information from source code is a tedious task for all but the simplest programs. In the case of an object-oriented program, a developer must consider how variables bind to different parts of the type hierarchy, the methods that might be invoked as a result of the binding, and so on. For this reason, I have built the Jex tool, that automates this task for Java programs.

To retain meaning for a developer, I wanted to present a view of the exception flow within the context of the structure of the existing program. The Jex tool thus extracts, synthesizes, and formats only the information that is pertinent to the task. In the case of Java, for each method, our tool extracts, the nested `try` block structures, including the guarded block, the `catch` clauses, and the `finally` block. Within each of these structures, Jex displays the precise type of exceptions that might arise from operations, along with the possible origins of each exception type. If an exception originates from a method call, the name of the class and method raising the exception are identified. If an exception originates from the run-time environment, the qualifier `environment` is used. This information is placed within a Jex file corresponding to the analyzed class.

We can illustrate this exception structure using code from one of the constructors of the class `java.io.FileOutputStream` from the JDK 1.1.3 API.[1] Figure 3.1 shows the code for the constructor; Figure 3.2 shows the exception structure extracted according to the proposed technique.[2] The extracted structure shows that the code preceding the explicit `try` block may raise a `SecurityException`, and that the code inside the `try` block may result in an `IOException` being raised by the call

[1]Source code for the JDK 1.1.3 API is publicly available. This source code can be used to determine the exceptions possibly thrown by the various methods.

[2]Figure 3.2 is a simplified view of the information generated by Jex. Specifically, for clarity in presentation, the full qualification of Java names that is usually shown was removed.

to `openAppend` or to `open` on an object of type `FileOutputStream`. The `catch` clause indicates that any `IOException` raised during the execution of the code in the `try` block may result in a `FileNotFoundException` being raised. `FileNotFoundException` is a subtype of `IOException`, the exception declared in the constructor's signature.

```
public FileOutputStream(String name, boolean append)
   throws IOException
{
   SecurityManager security = System.getSecurityManager();
   if (security != null) {
      security.checkWrite(name);
   }
   try {
      fd = new FileDescriptor();
      if(append)
         openAppend(name);
      else
         open(name);
   } catch (IOException e) {
      throw new FileNotFoundException(name);
     }
}
```

Figure 3.1: The Source Code for the Constructor of `FileOutputStream`

```
FileOutputStream(String,boolean) throws IOException
{
   SecurityException:SecurityManager.checkWrite(String);
   try {
      IOException:FileOutputStream.openAppend(String);
      IOException:FileOutputStream.open(String);
   }
   catch ( IOException ) {
      throws FileNotFoundException;
   }
}
```

Figure 3.2: The Structure of Exceptions for the Constructor of `FileOutputStream`

This analysis provides two useful kinds of information to a software developer implementing or maintaining this constructor. First, the developer can see that the constructor may signal an unchecked `SecurityException` that originates from a `checkWrite` operation; a comment to this effect may be added to the constructor's header for the use of clients. Second, the developer can determine that the exceptions that may be raised within the scope of the `try` block are actually of type `IOException` and not some more specialized subtype; thus, finer-grained handling of the exception is not possible and should not be attempted. Neither of these cases would be detectable based on an inspection of the constructor's source code alone.

The analysis can also benefit a client of the constructor. Consider the code for the `doSomething` method in Figure 3.3. This code will pass the checking of the Java compiler as there is a handler for the declared exception, `IOException`. Applying the exception extraction technique to this code returns the information that the invocation of the `FileOutputStream` constructor might actually result in the more specialized `FileNotFoundException` or an unchecked `SecurityException`.

Knowing the details about the exceptions flowing out of the constructor allows the developer of the client code to introduce additional handling. Figure 3.4 shows an enhanced version of the `doSomething` client code. A handler has been introduced to catch `SecurityException`. This handler warns the user that permission to modify the file is missing. A handler is also introduced to provide a specialized error message for the case when a `FileNotFoundException` occurs.

To conform to the constructor's interface, it is also necessary to provide a handler for `IOException`: this handler serves to protect the client from future modifications of the constructor, which may result in the throwing of an IO exception different from `FileNotfoundException`.

```
public void doSomething( String pFile )
{
  try {
    FileOutputStream lOutput = new FileOutputStream( pFile, true );
  }
  catch( IOException e ) {
    System.out.println( "Unexpected exception." );
  }
}
```

Figure 3.3: An Example of Code not Using Jex Information

```
public void doSomething( String pFile )
{
   try{
      FileOutputStream lOutput = new FileOutputStream( pFile, true );
      // Various stream operations
   } catch( SecurityException e ) {
      System.out.println( "No permission to write to file " + pFile );
   } catch( FileNotFoundException e ) {
      System.out.println( "File " + pFile + " not found" );
   } catch( IOException e ) {
      System.out.println( "Unexpected exception" );
   }
}
```

Figure 3.4: An Example of Code Making Use of Jex Information

3.2 The Architecture and Implementation of Jex

As described in the previous section, Jex is a tool for extracting exception information from Java source files. It is entirely developed in Java and comprises 23 000 lines of commented Java source code spread over 138 classes and interfaces. It consists of two command-line applications: `jex.Analyzer` (or Jex), the main source code analyzer, and `jex.subsumption.Analyzer` (or JSA), a higher-order analysis tool used to perform analysis on the information extracted by Jex.

The architecture of Jex consists of five components: the application controller, the parser, the abstract syntax tree (AST), the type system, and the Jex loader (Figure 3.5).

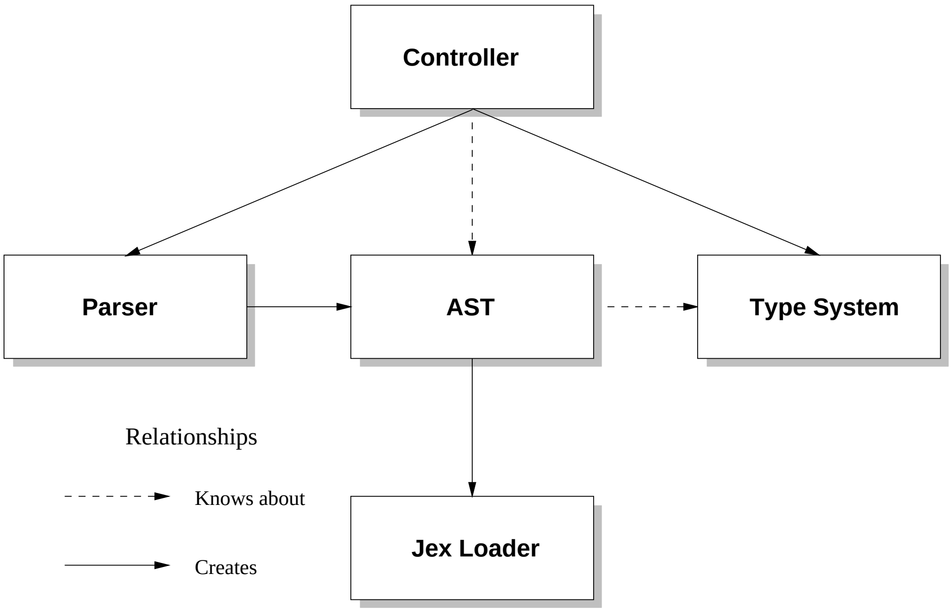

Figure 3.5: The Simplified Architecture of Jex.

3.2.1 The Application Controller

The application controller is the entry point to Jex. It processes the command-line arguments, loads the type system (section 3.2.4), and parses the input file (section 3.2.2), which returns a reference to a corresponding AST (section 3.2.3). It then passes the type system to the AST component, and requests the AST to perform the exception analysis.

The application controller is implemented as a standard Java program entry point, that is, a static `main` method.

3.2.2 The Parser

The parser loads and parses a Java source file specified as input. The parser also contains actions to build an abstract syntax tree representation of the program analyzed (section 3.2.3). The parser was constructed using version 0.8pre1 of the Java Compiler Compiler™ (JavaCC) [27]. The current implementation of the tool supports the Java 1.0 language specification, which does not include the support for inner classes and initializer blocks [12].

3.2.3 The Abstract Syntax Tree

The AST is built by parser actions. It contains the functionality to extract exception information from the representation of the program and to generate a Jex file containing this information. A Jex file is a normal text file containing a description of the exception flow for all the methods of a class, following the technique highlighted in section 3.1 and illustrated in Figure 3.2. Appendix A contains the formal description of Jex files.

From the original version of the AST created by the parser, it takes four steps to generate a Jex file:

1. *Type analysis.* The AST resolves the type of every expression in the source code. The type system is not used at this preliminary step, because the types of program expressions can all be derived from declared types.

2. *Control-flow analysis.* The AST performs a total ordering of the methods to analyze in the input source code file. The order is based on the call hierarchy within the methods of a class, so that a method A called by a method B would be analyzed first. The current version of Jex does not support cycles in the call graph.

3. *Jex analysis.* Every operation (i.e., operator, method call) that can generate an exception is examined, and the exceptions that can be thrown by that

operation are mapped to a dynamically-allocated data structure representing the exception handling constructs.[3] For every method call, a type analysis using the type system component (section 3.2.4) is performed to determine all of the possible implementations of the method being called. For all of the implementations identified, the Jex loader (section 3.2.5) is used to obtain the list of exception types that can be thrown by the method. In the case of operations that are built in the language (such as multiplicative operators and array accesses), the exceptions generated are based on the Java language specifications [12]. The list of environment-related exceptions generated by Jex can be found in appendix B.

4. *Generation of the Jex files.* Finally, the data structure representing the exception information is written to Jex files corresponding to every class in the input source.

Steps 1 to 3 correspond to a traversal of the AST, while step 4 corresponds to a traversal of the exception data structure. The AST was built using the JJTree preprocessor distributed with JavaCC.

3.2.4 The Type System

The AST relies on the type system to return a list of all types that override or implement a particular method. Ensuring all possible types are considered in such an operation would require global analysis of all Java classes reachable through the Java class path. This approach has the disadvantage of being overly conservative because unrelated classes may be considered. For example, the method `toString`

[3] The exception to this statement is that exceptions potentially thrown as a consequence of the initialization of static variables are not considered because it is not always possible to statically identify the program points where a class is first loaded.

of class `Object` is often redefined by application classes. Two classes, both in the class path but from two unrelated applications, might each redefine `toString`. If a method in a class of the first application makes a call to `toString`, it is reasonable to assume that the method `toString` implemented by the second class will not be invoked. To prevent this, we restrict the analysis to a set of packages defined by the user. The normal Java method conformance rules are taken into account in establishing the potential overriding relationships between methods.

The type system is implemented as a 3-level hash table. This hash table maps package names to class names, to method signatures, to a list of types possibly implementing the methods. Loading the type system component is done in two steps. A first step loads all the classes of all the packages specified in the data structure. The second step establishes the implementation relationships between all the methods in the system, using introspection to extract type hierarchies.

3.2.5 The Jex Loader

To determine the actual exceptions thrown by a Java method call, the AST component relies on the Jex loader. Given a fully qualified Java type name, the Jex loader locates the Jex file describing that type. The AST component can then query the Jex loader to return the exceptions that might arise from a method conforming to a particular method signature for that type. The Jex files for a Java type are stored in a directory structure that parallels the directory structure of the Java source files. It is necessary to have a different directory structure for Jex files because some class files might not be in writable directories. The Jex files thus serve both to provide a view of the exception structure for the user, and as an intermediate representation for the Jex system.

The Jex loader is implemented as a parser for Jex files, enhanced with actions to recognize method matching according to the conformance rules of Java, and to extract uncaught exceptions.

3.2.6 Using Jex

To use Jex, a user must specify a list of packages, a path to search for Jex files, and a Java source code file. The list of packages and the path for Jex files are specified in a resource file named `.include`. The first line of the file must be a valid directory pointing to the root of the Jex file hierarchy. The subsequent lines are a list of packages to include in the type system. The Java source file to analyze is specified as a command-line argument.

Currently, the Jex system requires that all necessary Jex files to analyze a source code file be available. Since an analysis will terminate abnormally if a such a Jex file is missing, the user must make sure that all necessary Jex files are available before launching an analysis. For this reason, Jex supports the command-line option `-d`, that will simply produce a list of all the Jex files necessary for the analysis of the input file specified.

3.2.7 The Subsumption Analysis Tool

Jex files provide two different kinds of detailed information to users: types of exceptions that can be raised, and the program structures in place to handle them. To make efficient use of this information, it is sometimes necessary to reason about the interaction between the two types of information. This requires knowing which types of exception can be caught by which types of `catch` clauses. Because of subsumption, this is sometimes difficult, because developers have to keep in mind the complete type hierarchy of exceptions. For example, Figure 3.6 shows Jex information for a `print` method.[4]

```
print()
{
   try
   {
      java.lang.NullPointerException:environment;
      java.lang.OutOfMemoryError:environment;
      java.io.IOException:FileWriter.<C>FileWriter(String);
      java.io.FileNotFoundException:FileReader.<C>FileReader(String);
      java.io.IOException:BufferedReader.readLine();
      java.io.IOException:LineNumberReader.readLine();
   }
   catch ( java.io.IOException )
   {
   }
}
```

Figure 3.6: An Example of Jex Information

```
print()
{
   try
   {
      java.lang.NullPointerException  ->  *UNCAUGHT*
      java.lang.OutOfMemoryError  ->  *UNCAUGHT*
      java.io.IOException  ->  java.io.IOException
      java.io.FileNotFoundException  ->  java.io.IOException
   }
   catch ( java.io.IOException )
   {
   }
}
```

Figure 3.7: The Result of Applying JSA to the Code of Figure 3.6

Without knowing how the exception types appearing in the `try` block relate to `IOException`, it is not possible to infer what exception will get caught. To help developers perform this task, Jex includes an accessory tool called `jex.subsumption.Analyzer`, or JSA. JSA takes a Jex file as input, and produces a reduced, annotated version

[4]For clarity in presentation, the full qualification of origin type names was removed.

of the Jex file. The information produced by JSA consists of, for every block, a set of unique exception types (without origin information), and an indication of which type is used to catch it. If the exception is not caught by any `catch` clause attached to the try block, the exception type is simply identified as "uncaught". Figure 3.7 gives the JSA information corresponding to the Jex information of Figure 3.6. It is now clear that `NullPointerException` and `OutOfMemoryError` remain uncaught, and that both `IOException` and `FileNotFoundException` are caught by the `catch` clause declaring the type `IOException`. Of course, this is a trivial example; in cases where `try` blocks contain numerous exception types and have more than one `catch` clause, JSA greatly simplifies the task of inferring exception flow.

When Jex files contain nested try blocks, an important issue to consider is whether to make a global or local analysis of the exceptions escaping a `try` block. In the strictly local view, exceptions escaping an inner `try` block are not propagated to the outer `try` block. In the global view, these exceptions are propagated. Both views are supported by JSA. The default is the local view, and the global view can be selected using the `-propagate` command-line argument.

JSA is implemented as a Jex parser with actions. Like the Jex loader, JSA parses a Jex file and stores the information in a dynamic structure. The structure is then written to a file according to the choice of view determined by the user.

Chapter 4

Validating the Jex Approach

As mentioned in the introduction, the hypothesis of this thesis is that to produce quality code, developers need to have access to more complete and precise exception flow information than what typical exception interfaces can provide them. This initial hypothesis can be separated into two clauses: *(a)* in practice, programming languages cannot provide complete exception flow information, and *(b)* information about the flow of exceptions is necessary to build robust programs. The first clause is based on a survey of programming languages supporting exception handling (see section 2.1), while the second clause is based on observations made while programming in Java. Particularly in the initial construction of a method, it is often tempting, for expediency, to insert a `catch` clause that will simply handle all exception types. A developer might choose this course of action not as the result of negligence, but rather because of a lack of access to information that allows an appropriate decision to be made. As an example, a developer may not have suitable information about the recovery possible for a particular kind of exception in the absence of knowledge about the application as a whole. Introducing these generalized handlers causes exceptions to be caught through subsumption. Although such short-cuts should be refined as development proceeds, some occurrences may evade detection.

To validate my initial hypothesis, I was interested in determining how often cases of exception subsumption and uncaught exceptions occur in released code, and in quantifying how well Java formally describes the flow of exceptions across a method's boundary (clause a). I was also interested in determining how knowing about the flow of exceptions could suggest ways in which programs could be made more robust (clause b).

The last chapter presented an approach to provide information about the flow of exceptions, and a tool, Jex, implementing the approach. This chapter uses results obtained from applying Jex to demonstrate the relevance of the initial hypothesis, and the contribution of the approach to providing information about exception flow. Section 4.1 describes the code that was analyzed with Jex and the conditions in which the analysis was performed. Sections 4.2 and 4.3 provide arguments demonstrating that, in practice, Java cannot provide complete and precise exception flow information. Section 4.4 is a qualitative analysis of the results obtained with Jex and discusses how information about the flow of exceptions is necessary to build robust programs. Finally, section 4.5 summarizes the conclusions of the experiments.

4.1 Methodology

To investigate the various factors mentioned in the previous section, a variety of source code was analyzed using Jex:

- JTar, a command-line utility for the extraction of `tar` files;[1]

[1] Package `net.vtic.tar`, developed by J. Marconi and available from the Giant Java Tree, `http://www.gjt.org`.

- A GNU regular expression package;[2]
- the `java.util.Vector` and `java.io.FileOutputStream` classes from the Sun™ Java Development Kit version JDK 1.1.3;
- a command-line rule parser; [3]
- four database and networking packages from the Atlas web course server project [18]: `userDatabase`, `userData`, `userManager`, and `userInfoContainers`;
- the code of Jex itself.[4]

Together, these packages comprise roughly 32 000 commented lines of code, including input/output, networking, and parsing operations.

To perform an analysis on a source file, Jex requires a Jex information file corresponding to every class referenced in the source file, and to all subclasses indicated by the Jex type system (see sections 3.2.3 and 3.2.4). These Jex files contain information about the exceptions potentially raised by the various methods called by the code being analyzed. This implies that Jex files for most of the JDK API be available. Since performing Jex analyses on the bulk of the JDK class libraries would not have been a cost effective measure at this stage of the experimentation with Jex, I decided not perform the Jex analysis on the classes comprising the JDK API (except for the two mentioned in the list above). Instead, a Jex file for each of the relevant API classes was generated using a script that extracts the information

[2]Packages `gnu.getopt` and `gnu.regexp`, also available from the Giant Java Tree

[3]Available from a compiler course web page of the School of Computing, National University of Singapore (`http://dkiong.comp.nus.edu.sg/compilers/a/`).

[4]The code of Jex was analyzed both to test the Jex tool itself and to provide insight into the usefulness of the approach. It was not, however, used in the statistical compilations of sections 4.2 and 4.3. Because Jex was designed with exception policies in mind, I have chosen to leave Jex out of the statistical analyses so it would not bias the results. The statistics of sections 4.2 and 4.3 thus only represent independent code.

from corresponding HTML files produced by Javadoc. Javadoc is a tool that automatically converts Java source code files containing special markup comments into HTML documentation. The Jex files produced from these scripts simply consist of a list of exception types potentially thrown by each method of the class. The list consists of a union of the exception types declared in the method's signature with the exception types annotated in the special markup comments. The exception types annotated in the comments for a class may include both checked and runtime exception types.

4.2 Analysis of Subsumption in `try` Blocks

A first aspect investigated was whether or not subsumption in `try` blocks actually occurred in practice. By "subsumption in `try` block", I mean the situation in which a `catch` clause declaring to catch an exception type T can also catch exceptions that are subtypes of T. A first experiment was performed with a version of Jex that did not include runtime exceptions generated by the environment [23]. A subsequent experiment was done including such exceptions. Figures 4.1 and 4.2 show a comparison of the data, both with and without runtime exceptions, respectively.

The graph in Figure 4.1 shows a breakdown of exceptions and their associated handling in the analyzed code. It represents information from the packages that contain at least one `try` block. Each bar in the graph shows the number of occurrences of different *levels of subsumption* in handlers. The level of subsumption between the type T of an exception potentially raised in a `try` block and the type T' declared in a `catch` clause is the difference in depth in the type hierarchy between T' and T. Levels zero and one are labeled by their semantic equivalent: "same type" and "supertype", respectively. Exception types raised in a `try` block that cannot be subsumed to any of the types declared in the `catch` clauses remain uncaught by the `try` block. In all but one case, the Rule Parser, some exceptions in `try` blocks remain uncaught. All but one of the packages, the Java JDK code, contain exception handlers that catch exceptions through subsumption.

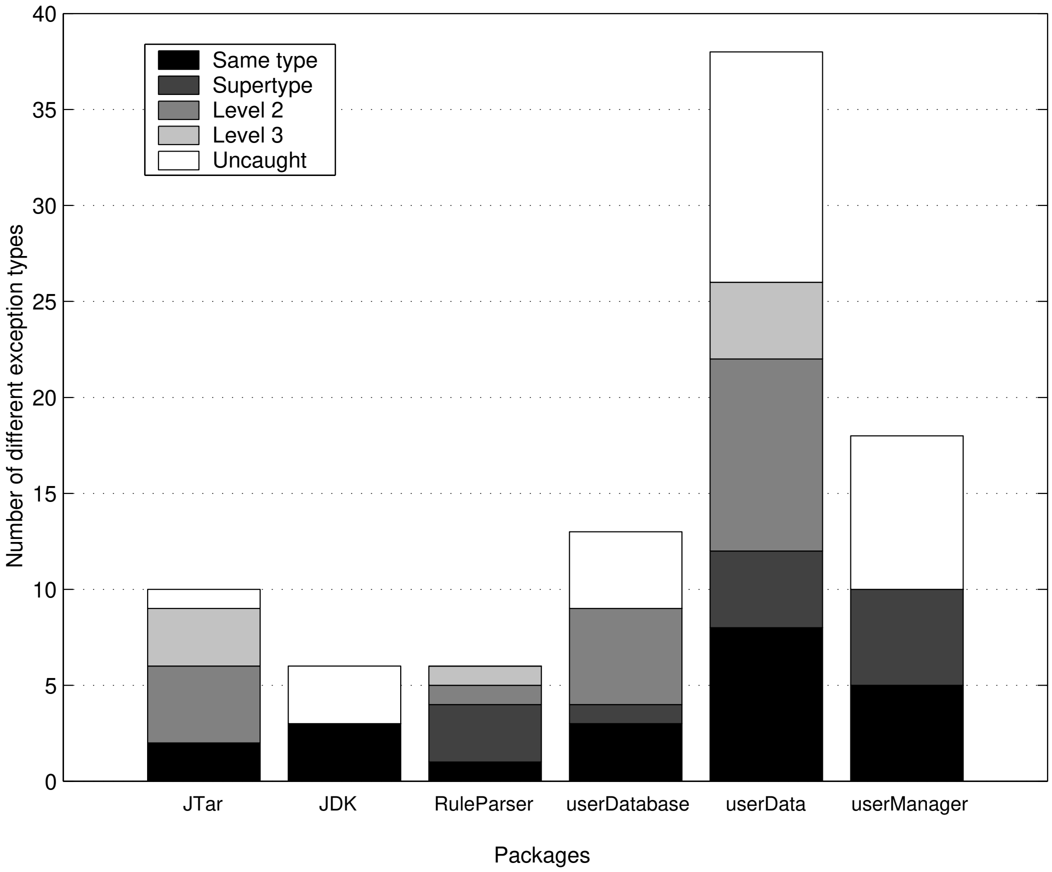

Figure 4.1: Exception Matching in `catch` Clauses, no Environment-Related Exceptions

Because of its similarity to Figure 4.1, Figure 4.2 shows that the presence of runtime exceptions generated by the runtime environment does not significantly influence the distribution of subsumption in `catch` clauses. We can notice two minor effects: an increase in the number of uncaught exceptions and an increase in subsumption levels greater than or equal to one. Both of these effects can be explained by the observation that environment-related runtime exceptions are usually not considered by the user and thus no `catch` clause is inserted to explicitly catch these types of exceptions. These types of exceptions either remain uncaught, or are caught by subsumption by more generalized types, such as `Exception`.

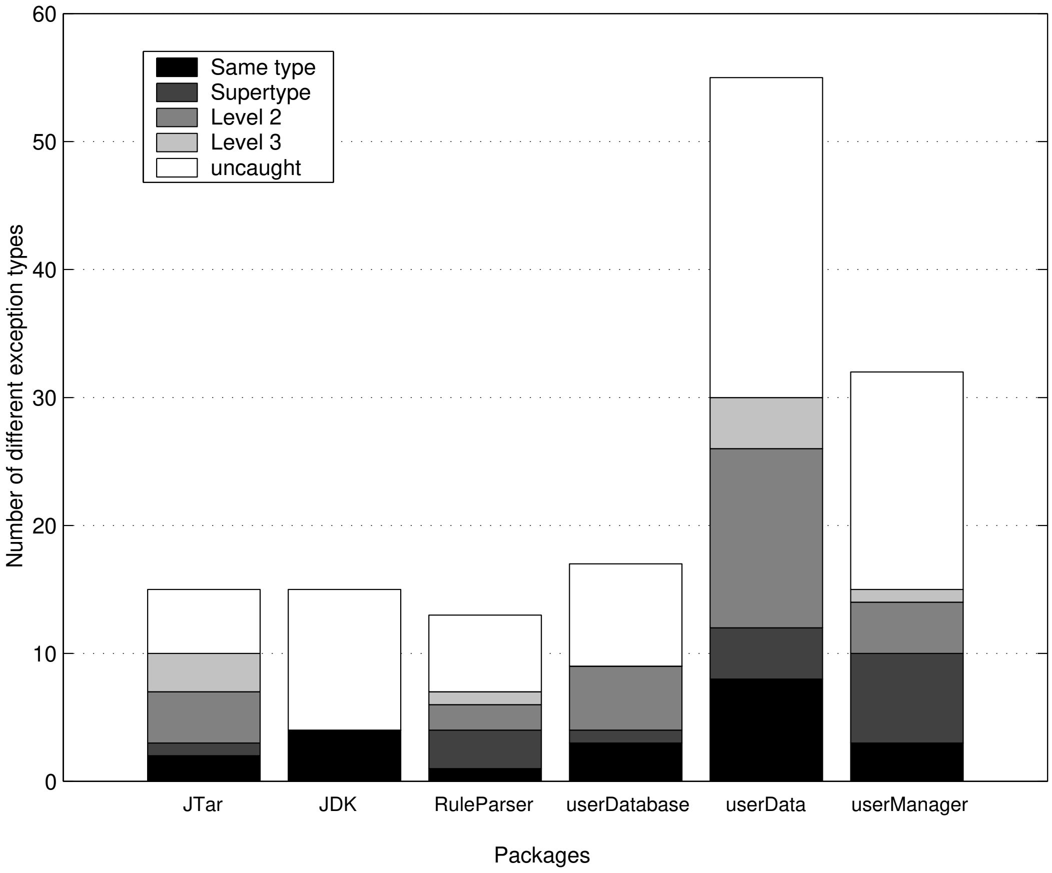

Figure 4.2: Exception Matching in `catch` Clauses, with Environment-Related Exceptions

Table 4.1: Levels of Subsumption Required to Catch an Exception

Level of subsumption	Frequency	
	No Environment	Environment
Same type	24 %	14 %
Supertype	14 %	11 %
2 Levels	22%	20 %
3 Levels	9 %	6 %
Uncaught	31 %	49 %

Table 4.1 provides a different view of the data. This view illustrates that, for the analysis including environment-related exceptions, 49% of the different exception types present in `try` blocks remain uncaught in the target. In 37% of the cases, exceptions are not caught with the most precise type available (i.e., some degree of subsumption occurs), and in only 14 % of the cases the exceptions potentially raised in a `try` block are caught by their exact type (i.e., no subsumption occurs).

This data lends evidence to support the claims that exception subsumption and unhandled exceptions are prevalent in Java source code. However, this quantitative data does not indicate whether the quality of the code could be improved through the use of Jex-produced information. Quantitative aspects are discussed in section 4.4.

4.3 Analysis of Exception Specifications

A second aspect investigated was how much information about the flow of exceptions is actually present in formally-declared method headers. This information was favored as opposed to program documentation because formally-declared exceptions are less likely to be subject to errors and inconsistencies than program documentation. Furthermore, the formal syntax allows a more conservative verification.

To investigate how methods in Java typically describe localized exception flow, I used the metric described in section 1.2.2. I applied this metric to all packages analyzed by Jex, except for Jex itself. Of the 375 methods present in the 10 packages considered, only 55 (15%) specified exceptions. For these 55 methods, the completeness (C_m) and granularity (G_m) of the interfaces was calculated. Figure 4.3 shows the distribution of the interfaces in the completeness-granularity graph (some data points overlap).

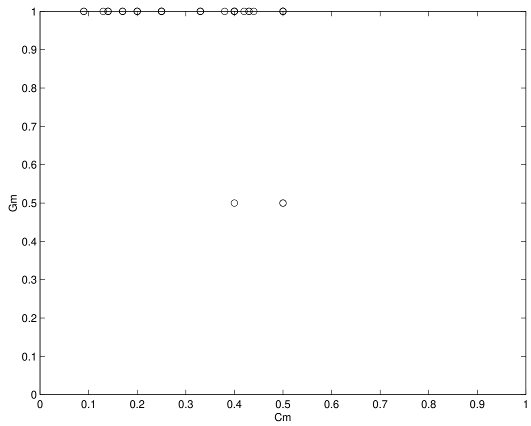

Figure 4.3: C_m–G_m Distribution for all Packages Except the Jex Packages

In analyzing the completeness and granularity of exception specifications, several observations were made. First, in the packages analyzed, exception interfaces are never used to specify runtime exceptions. Indeed, there is not much incentive for a Java programmer to specify runtime exceptions since these are not checked by the compiler. As a result, completeness of the exception specification in Java is strongly related to the number of different runtime exception types that can be raised by the method. As expected, the number is usually high, leading to low C_m values.

A second observation is that the completeness and granularity metrics are non-orthogonal. The relationship between the two aspects is subsumption. Declaring a general supertype will result in low granularity, but on the other hand will tend to encompass a bigger set of the exceptions potentially thrown by the method. Finally, I observed that, in the packages analyzed, granularity usually has a value of one for user-defined exceptions, which allows one to think that when a developer defines an exception, it is used with precision.

The distribution of exception specifications on the C_m–G_m graph, shown in Figure 4.3, is somewhat extreme in that most of the specifications represented have a granularity of 1.00. As a comparison, Figure 4.4 shows the distribution of a low-level package of the Jex application, `jex.util`. We can notice that the distribution is more scattered than the one represented in Figure 4.3, with some interfaces having a granularity of 0.5 and lower. A possible explanation for this is that `jex.util` deals with components involving more hierarchical exception types, such as `IOException`.

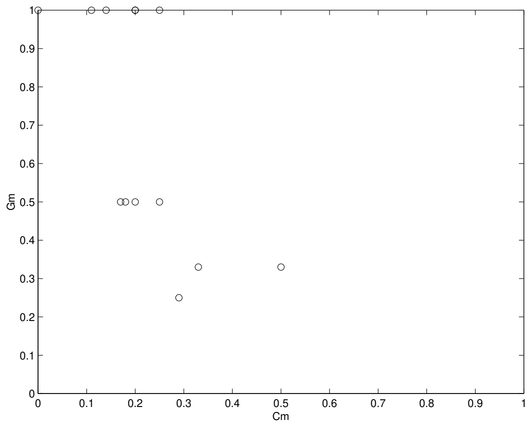

Figure 4.4: C_m–G_m Distribution for the Package `jex.util`

4.4 Quantitative Considerations

The first part of this chapter showed that, in practice, not all exceptions flowing out of `try` blocks and methods are specified by the programming language constructs. As a result, developers do not always leverage the full expressiveness of an exception system. In section 3.1, I have shown the approach by which Jex fills in these information gaps. Based on experience, this section illustrates how Jex can be used to improve code, and thus provides evidence for the clause `b` of the hypothesis.

To investigate the usefulness of the Jex information, I performed an after-the-fact manual inspection of the source code. I focused this inspection on cases of subsumption since the benefits of identifying uncaught exceptions are straightforward and are discussed in greater depth elsewhere [10, 29, 31].

The investigation of the cases of exception subsumption found several instances in which knowledge of the subsumption could be used to improve the code. In the RuleParser application, for instance, the body of a method reading a line from an input buffer is guarded against all exceptions using the `Exception` type. This type is a supertype to much of the exception hierarchy. Expecting input problems, the code produces a message about a source input exception. However, Jex analysis reveals that two other types of runtime exception may also arise: `StringIndexOutOfBoundsException` and `SecurityException`. These two unchecked exceptions will be caught within the `Exception` handler, producing an inappropriate error message. More specific exception handlers could be added to improve the coherence of termination messages, or to implement recovery actions.

Cases of subsumption were also useful in pointing out program points at which exception handling code did not conform to the strategy established by the developer. For example, in one of the Atlas classes, an exception was *explicitly* thrown in a `try` block, caught in a `catch` clause corresponding to the same `try` block, and re-thrown. In another case, two similar accessor methods displayed different exception handling strategies: one masked all exceptions; the other one masked only two specific exceptions. A discussion with the developer of Atlas allowed the irregular exception handling strategies to be traced to unstable or unfinished code. The abstract view of the exception flow provided by Jex made it easy to identify these suspicious cases.

I found other uses of subsumption in the Atlas packages. For example, in a database query, exceptions signaled by reading from a stream are all caught by a generalized `catch` clause which generates a generic "read error" message and which re-throws a user-defined exception. However, the exceptions thrown in the `try` block include such specialized types as `StreamCorruptedException`, `InvalidClassException`, `OptionalDataException`, and `FileNotFoundException`. It may be advantageous to catch these exceptions explicitly, producing a more descriptive error message when one of the exceptions occurs.

As I have already pointed out, the use of Jex is not only beneficial to sloppy programs, or to programs written without a general exception handling strategy. It can also help fine-tune programs designed with exception handling in mind. As an example, we can take the code of Jex itself. In the code of Jex, knowing about the precise local flow of exceptions allowed the identification of methods declaring exceptions that were never thrown. The cause of this incoherence is one of evolution: a method called another method that threw an `IOException`. The called method then evolved to perform its own handling of the exception.

Other high-level improvements stemming from such detailed knowledge of how exceptions flow include limiting the scope of some exception types. For example, a specification of the Jex type system was that all `ClassCastException`s had to be caught in the type system component, and only rethrown as `TypeException`s when necessary. Since `ClassCastException` is a type of runtime exception, and thus not checked, it was very difficult to verify that the specification was respected. With Jex, this turned into a trivial task.

4.5 Summary

In brief, experimentation with Jex allowed to validate the hypothesis of this thesis, namely that, to produce quality code, developers need to have access to more complete and precise exception flow information than what typical exception interfaces can provide them. An analysis of cases of subsumption and uncaught exceptions in Java code (section 4.2) showed that developers do not always use the most precise type available to catch exceptions, and do not catch all exceptions. An analysis of exception specifications (section 4.3) showed that the Java language structures in place to provide information about the flow of exception do not provide complete and precise information. Finally a qualitative study of the code analyzed (section 4.4) showed that knowing about the flow of exceptions could help developers build more robust and reliable code.

Chapter 5

Conclusions

During the initial conception of the Jex approach, and all through the development of the corresponding tool, many tradeoffs and decisions have been made. Most of them were engineering tradeoffs that have no impact on the use of the approach, and as such are of limited interest to readers. A few aspects, however, inevitably have had an influence on the use of Jex and on the quality of the information that can be obtained from it. These aspects include the information included in Jex files, the expressiveness of the information, and the level of conservatism of the information. In this chapter, I discuss how these decisions influence the performance and usefulness of Jex, and compare my technique to other potential approaches. I then present the future avenues for Jex and discuss how it generalizes to object-oriented languages other than Java. Finally, I summarize the thesis.

5.1 Discussion

5.1.1 White-box Exception Information

By expressing the actual exceptions that may flow out of a method invocation, we expose knowledge about the internals of a supplier method to a client. If a software developer relied upon this knowledge of a supplier's implementation rather than on the supplier's declared interface, unintended dependencies could be introduced, potentially limiting the evolution of the client.

For instance, consider the case for Atlas described in section 4.4, in which the developer learned that a particular method could receive a number of specialized exception types, such as `StreamCorruptedException` and `InvalidClassException`. Assume that the methods that can raise these exceptions declare more general exception types as part of their interfaces. If the developer introduced handlers only for each of the specialized types that could actually occur, the code might break if a method evolved to signal a different specialized exception type. In the case of Java, this situation cannot arise because the compiler forces the presence of handlers for the exception types declared by supplier operations. If the language environment did not provide this enforcement, the Jex approach would have to be extended to ensure that the use of white-box information did not complicate evolution.

5.1.2 Alternative Approaches

Increasing the robustness and recovery granularity of applications does not require a static analysis tool. One alternative currently in use is to document the precise types of exceptions that a method may throw in comments about the method. With this approach, a developer can retain flexibility in a method interface, but still provide additional information to clients wishing to perform finer-grained recovery. A disadvantage of this approach is that it forces the developer to maintain consistency between the program code and the documentation, an often arduous task. Moreover, this approach assumes that a developer knows all of the exception types that might be raised within the body of the method being developed; the presence of runtime exceptions makes it difficult for a developer to provide complete documentation.

Another course of action available is for a software developer to inspect the exception type hierarchy, and to provide handlers for all subtypes of a declared exception type. It is unlikely that in most situations the extra cost of producing and debugging these handlers is warranted. Furthermore, this solution is not robust since subtypes can be added to the exception hierarchy at any point in the development. The Jex approach provides a means of determining cost-effectively which of the many possible handlers might be warranted at any particular source code point.

5.1.3 The Descriptive Power of the Current Exception Structure

The current exception structure extracted for source files enables a developer to determine the exceptions that can be signaled at any point in the program, along with the origin of these exceptions. The former information allows a developer to determine the actual exceptions that can cross a module boundary. The latter information allows a developer to trace exceptions to their source, enabling a more thorough inspection.

One aspect missing from the information currently produced by Jex is a link to the particular statements that can produce an exception. As a result, it is not possible to trace actual instances of exceptions. For example, when an exception is explicitly thrown, it is not possible to determine, only from Jex information, if it is a new exception or if an existing exception instance is being re-thrown. Information about exception instances would allow developers to reason about how specific exceptional conditions circulate in a program. However, it is unclear whether the additional benefits that could be obtained from the more specific origin information outweigh the possible disadvantage of reducing the clarity and succinctness of the exception structure.

5.1.4 The Precision of Jex Information

There are three cases in which the Jex tool may not return conservative information. First, Jex uses the packages specified by the user as the "world" in which to search for all possible implementations of a particular method. If a user fails to specify a relevant package, Jex may not report certain exceptions. If, in specifying the packages, the user fails to include a package defining a type being analyzed, Jex can issue a warning message. If the user fails to specify packages that extend types that are already defined, then Jex is unable to warn the user.

Second, Jex relies on a model of the language environment to determine the exceptions that might arise from basic operations, such as an add operation, and the exceptions that might arise from native methods. Although the model of the environment used when applying Jex to the code described in chapter 4 was partial, Jex still returned information useful to a developer.

Third, Jex does not report asynchronous exceptions [12]. An asynchronous exception may arise from a virtual machine error, such as running out of memory, or when the `stop` method of a thread object is invoked. Since these exceptions can arise at virtually any program point, one can assume a user of Jex will find it easier to use the output of the tool if it is not cluttered with this information. However, it may prove useful, once more experience is gained with Jex, to introduce an option into the tool to output such exceptions as a means of reminding the user.

If Jex returned information that was too conservative, the usability of our approach would likely be impacted. With Jex, this situation can arise when reporting all possible runtime exceptions because there are many points in the code that can raise exceptions such as `NullPointerException`. This situation can be managed by providing a means of eliding this information when desired. To make this possible, the structure of the files generated by Jex is designed to make it possible to remove certain types of exceptions by using the UNIX command `grep -v <exception type>` on a Jex file. This command has the effect of producing a new Jex file where information about the exception type specified does not appear.

Another source of imprecision in Jex arises from the assumption that a call to a method made through a variable might end up binding to any conforming implementation on any subtype of the variable's type. In some cases, it may be possible to use type inference to limit the subtypes that are considered. However, with my current experience with Jex, I have not found that this assumption greatly increases the exception information returned.

5.2 Future Work

This thesis addressed the problem of providing information about the flow of exceptions in a localized way. It described a mean to analyze how exceptions flow in and out of `try` blocks and methods. Although this solution provides useful information to developers, there remains a need for a more global method of representing the flow of exceptions. Coming up with a practical method of representing the flow of exceptions in software applications would enable software engineers to better design and test applications. For example, a method for representing the global flow of exceptions would allow developers to trace exceptions to their origins, or to better understand how exception evolve in programs.

A second aspect of this research that appears worthy of further investigation is the generalization of the Jex technique to other object-oriented languages. Even though most object-oriented languages supporting exceptions display the same fundamental concepts justifying Jex, their exception handling systems differs from Java. It would prove useful to explore how useful and effective the Jex approach can prove for such languages, such as C++.

Finally, in order to be able to address a wider base of source code and a larger pool of user, the Jex tool necessitates some improvements. The most important of these improvements include supporting the Java 1.1 language specification, and resolving cycles in method and class dependencies. Supporting Java 1.1 basically means supporting the concept of inner classes. This introduces difficulties in the organization of Jex information, and requires a more complex analysis at the AST level. It is nevertheless feasible. For Jex to be able to resolve cycles in method and class dependencies, although strictly a usability improvement, would allow developers to analyze source code files in batches, thus greatly reducing the level of intervention currently required of Jex users.

5.3 Summary

It is not uncommon for users of software applications to become frustrated by misleading error messages or program failures. Exception handling mechanisms present in modern languages provide a means to enable software developers to build applications that avoid these problems. Building applications with appropriate error handling strategies, though, requires support above and beyond that provided by a language's compiler or linker. To encode an appropriate strategy, a developer requires some knowledge of how exceptions might flow through the system.

In this thesis, I have described an approach to help developers access this information. The approach, based on static analysis techniques, is supported by a tool named Jex. The Jex tool extracts information about the structure of exceptions in Java programs, providing a view of the actual exceptions that might arise at different points and of the handlers that are present. Use of this tool on a collection of Java library and application-oriented source code demonstrates that the approach can help detect both uncaught exceptions, and uses of subsumption to catch exceptions.

The view of exception flow synthesized and reported by Jex can provide several benefits to a developer. First, a developer can introduce handlers for uncaught exceptions to increase the robustness of code. Second, a developer can determine cases in which unanticipated exceptions are accidentally handled; refining handlers for these cases may also increase code robustness. Third, inspection of subsumption cases may indicate points where the addition of finer-grain recovery code could improve the usability of a system. Finally, the abstract view of the exception structure can help a developer detect potentially problematic or irregular error handling code. The approach described in the thesis and the benefits possible are not limited to Java, but can also apply to other object-oriented languages.

Bibliography

[1] *Ada 95 Reference Manual: Language and Standard Libraries, version 6.0*, December 1994. Revised international standard ISO/IEC 8652:1995.

[2] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer, 1996.

[3] David L. Black, David B. Golub, Richard F. Rashid, Jr. Avadis Tevanian, and Michael W. Young. The Mach exception handling facility. Technical Report CMU-CS-88-129, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pa., USA, April 1988.

[4] Boris Bokowski. CoffeeStrainer: Statically-checked constraints on the definition and use of types in Java. In *Proceedings of the Joint 7th European Software Engineering Conference and 7th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, September 1999. To appear.

[5] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 report (revised). Technical Report 52, Digital Systems Research Center, November 1989.

[6] Yih-Farn Chen, Michael Y. Nishimoto, and C.V. Ramamoorthy. The C information abstraction system. *IEEE Transactions on Software Engineering*, 16(3):325–333, March 1990.

[7] Jong-Deok Choi, David Grove, Michael Hind, and Vivek Sarkar. Efficient and precise modeling of exceptions for the analysis of Java programs. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'99)*, September 1999. To appear.

[8] Flaviu Christian. Exception handling and software fault tolerance. *IEEE Transactions on Computers*, 31(6):531–540, June 1982.

[9] Helen Custer. *Inside Windows NT*. Microsoft Press, Bellevue, Wa., USA, 1993.

[10] Manuel Fahndrich, Jeffrey Foster, Jason Cu, and Alexander Aiken. Tracking down exceptions in standard ML programs. Technical Report CSD-98-996, University of California, Berkeley, February 1998.

[11] John B. Goodenough. Exception handling: Issues and proposed notation. *Communications of the ACM*, 18(12):683–696, December 1975.

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

[13] Juan Carlos Guzmán and Ascánder Suárez. A type system for exceptions. In *Proceedings of the 1994 ACM SIGPLAN Workshop on ML and its applications*, pages 127–135, June 1994. Research report 2265, INRIA.

[14] Robert Harper, Robin Milner, and Mads Tofte. The Definition of Standard ML: Version 3. Technical Report ECS-LFCS-89-81, Laboratory for the Foundations of Computer Science, University of Edinburgh, May 1989.

[15] Nevin Heintze. *Set-Based Program Analysis*. PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, USA, October 1992.

[16] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.

[17] Daniel Jackson and Allison Waingold. Lightweight extraction of object models from bytecode. In *Proceedings of the 21st International Conference on Software Engineering*, pages 194–201, May 1999.

[18] Mik A. Kersten and Gail C. Murphy. Atlas: A case study in building a web-based learning environment using aspect-oriented programming. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1999. To appear.

[19] Jun Lang and David B. Stewart. A study of the applicability of existing exception-handling techniques to component-based real-time software technology. *ACM transactions on Programming Languages and Systems*, 20(2):274–301, March 1998.

[20] Barbara H. Liskov and Alan Snyder. Exception handling in CLU. *IEEE Transactions on Software Engineering*, 5(6):546–558, November 1979.

[21] Robert Miller and Anand Tripathi. Issues with exception handling in object-oriented systems. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 85–103. Springer-Verlag, June 1997.

[22] François Pessaux and Xavier Leroy. Type-based analysis of uncaught exceptions. In *Proceedings of the 26th Symposium on the Principles of Programming Languages*, pages 276–290, January 1999.

[23] Martin P. Robillard and Gail C. Murphy. Analyzing exception flow in Java programs. In *Proceedings of the Joint 7th European Software Engineering Conference and 7th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, September 1999. To appear.

[24] Saurabh Sinha and Mary Jean Harrold. Analysis of programs with exception-handling constructs. In *Proceedings of the International Conference on Software Maintenance*, pages 348–357, November 1998.

[25] Saurabh Sinha, Mary Jean Harrold, and Gregg Rothermel. System-dependence-graph-based slicing of programs with arbitrary control flow. In *Proceedings of the 21st International Conference on Software Engineering*, pages 432–441, May 1999.

[26] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 2nd edition, 1991.

[27] Sun Microsystems, Inc. *The Java Parser Generator*. http://www.suntest.com/JavaCC/.

[28] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.

[29] Kwangkeun Yi. An abstract interpretation for estimating uncaught exceptions in standard ML programs. *Science of Computer Programming*, 31:147–173, 1998.

[30] Kwangkeun Yi and Byeong-Mo Chang. Exception analysis for Java. In *ECOOP'99 Workshop on Formal Techniques for Java Programs*, June 1999. To appear.

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

Appendix A

Grammar for Jex Files

A.1 Notation

In the production rules, the string representation of terminal symbols are displayed in **boldface** font and non-terminal symbols are shown in *italic* font. Special token parsed during the lexical analysis are underlined. The definition of a non-terminal symbol is introduced by the name of the symbol being defined followed by a colon. Symbols inclosed in square brackets ([]) are optional. Parenthesis can be used to indicate sets of related symbols. An asterisk (*) indicates zero or more repetitions of a symbol or set of symbols. A plus sign (+) indicates one or more repetitions of a symbol,or set of symbols. A vertical bar (|) indicates different possibilities among a set. The epsilon (ϵ) letter represents the null symbol.

A.2 Grammar

JexFile: (_MethodDeclarator_ _Block_)* EOF

MethodDeclarator: _SimpleMethodName_ _FormalParameters_ [**throws** _NameList_]

SimpleMethodName: [<**C**>] IDENTIFIER

NameList: _Name_ (**,** _Name_)*

FormalParameters: **(** [_Type_ (**,** _Type_)*] **)**

Block: **{** (_ExceptionGet_ **;** | _ExceptionThrow_ **;** | _TryCatch_)* **}**

ExceptionGet: _Name_ **:** (**environment** | _MethodName_ _FormalParameters_)

ExceptionThrow: **throws** _Name_

TryCatch: **try** _Block_ (**catch** **(** _Name_ **)** _Block_)+ [**finally** _Block_]

Name: IDENTIFIER (**.** IDENTIFIER)*

MethodName: (_Name_ | ϵ) _SimpleMethodName_

Type: (_PrimitiveType_ | _Name_ | _ArrayType_)

ArrayType: ([)+ ((**B** | **C** | **D** | **F** | **I** | **J** | **S** | **Z**) | (**L** _Name_ **;**))

PrimitiveType: (**boolean** | **char** | **byte** | **short** | **int** | **long** | **float** | **double**)

Appendix B

Environment-Generated Exceptions

The following is a list of all environment-generated exceptions supported by Jex. Every type of exception is followed by an enumeration of all the environment operations potentially raising the exception. The list is based on the Java language specifications [12].

- `java.lang.ArithmeticException` (multiplicative expression);
- `java.lang.ArrayStoreException` (Assignment to an array element);
- `java.lang.ArrayIndexOutOfBoundsException` (array access);
- `java.lang.ClassCastException` (cast expression);
- `java.lang.NegativeArraySizeException` (array allocation);
- `java.lang.NullPointerException` (field access, method invocation, array access);
- `java.lang.OutOfMemoryError` (additive expression with string arguments, allocation expression);