Analyzing Exception Flow in Java™ Programs*

Martin P. Robillard and Gail C. Murphy

Department of Computer Science
University of British Columbia,
Vancouver, B.C., Canada V6T 1Z4
{mrobilla,murphy}@cs.ubc.ca

Technical Report Number TR-99-02

University of British Columbia

March 2, 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 paper, we describe a tool called Jex that analyzes the flow of exceptions in Java code to produce views of the exception structure. We demonstrate how Jex can help a developer identify program points where exceptions are caught accidentally, where there is an opportunity to add finer-grained recovery code, and where error-handling policies are not being followed.

Keywords: Exception Handling, Software Analysis, Object-Oriented Programming Languages, Software Engineering Tools.

1 Introduction

To ease the difficulty of developing robust software systems, most modern programming languages incorporate explicit mechanisms for exception handling. Syntactically, these mechanisms consist 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. In essence, these mechanisms provide a way for software developers to separate code that deals with unusual situations

* This research project is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

from code that supports normal processing. This separation helps a developer structure and reason about the code within a module.

Unfortunately, local reasoning about the code is not generally sufficient to develop a module that will react appropriately to all unexpected situations. 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 instance, a user of a word processing application 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. Finer-grained reactions to exceptions require a software engineer to reason about code on which the module being constructed depends.

The exception handling mechanisms provided in some programming languages help a software developer perform this reasoning. Java [GJS96] and CLU [LS79], for instance, both support the declaration of exceptions in module interfaces; the compiler can then check that appropriate handlers are provided in a client module. However, this support is only partial because each of these languages also provides a form of unchecked exceptions. The developer of a client module is not warned of the possibility of these exceptions by a compiler. Furthermore, object-oriented languages typically support the classification of exceptions into exception type hierarchies. These hierarchies introduce the possibility of writing general handlers that may implicitly catch a subset of more specific exceptions. This implicit catching of exceptions can complicate the development and evolution of robust classes [MT97].

To investigate whether information about the flow of exceptions might help a developer fill in these gaps, we have built a tool, called Jex, to analyze exception flow in Java source code. When applied to a Java class, Jex determines the precise types of exceptions that may be raised at each program point, and then presents this information in the context of exception-handling constructs in the source. Using this abstracted view, a developer can reason about unhandled exceptions, and can identify and reason about exceptions handled through subsumption.

In this paper, we describe the Jex tool and present the results of applying the tool to both Java library code and several sample Java applications. The analysis of this code with Jex indicated a number of occurrences of unhandled exceptions and a number of occurrences of exceptions handled implicitly. A qualitative investigation of these occurrences suggested places where finer-grained recovery code could be usefully added, and identified program points at which exception policies intended by a developer were not being followed. The ease by which these code locations could be found using Jex suggests that a flow-oriented view of exceptions could help developers improve the quality of their code economically.

We begin, in Section 2, with an overview of the terminology of exception handling and of previous work involving the analysis of exceptions. In Section 3, we detail the basic exception handling mechanism in Java. Section 4 describes the view of the exception structure extracted by Jex, and the means by which the view is produced. We describe the use of Jex on sample Java code in Section 5, and discuss issues related to the use and generality of our approach in Section 6. Section 7 summarizes the paper.

2 Previous Work

Goodenough [Goo75] introduced the exception-handling concepts in common use today. To provide a common basis for discussion, we begin with a brief review of these concepts and the related terminology as defined by Miller and Tripathi [MT97].

> An *exception* is an abnormal computation state. An *exception occurrence* is an instance of an exception. ...
>
> An exception is *raised* when the corresponding abnormal state is detected. *Signaling* an exception by an operation (or a syntactic entity such as a statement or block) is the communication of an exception occurrence to its invoker. The recipient of the originator of an exception is a syntactic entity, called the *exception target* (or *target*); the originator of an exception is the *signaler*. The target is determined either by static scope rules or by dynamic invocation chain.
>
> An *exception handler* (or *handler*) is the code invoked in response to an exception occurrence. It is assumed that the handler's code is separate from the non-exception (or *normal*) control path in the program. Searching for eligible handlers begins with the target (i.e. the search starts with the handlers associated with the target). An exception is considered *handled* when the handler's execution has completed and control flow resumes in the normal path. An exception handled by the target *masks* the exception from the target's invokers.
> [MT97, pp. 86–87]

Variants of Goodenough's basic models have since been realized in many programming languages.

In ML [HMT89], a functional language, exceptions are values that can be declared anywhere in a program. These values can be signaled at any point following their declaration. Because it is difficult for programmers to ensure that all exceptions are caught, several static analyzers have been developed to track down unhandled exceptions in ML, including one by K. Yi [Yi94, YR97, Yi98], and a second, EAT[1], developed at the University of California at Berkeley [FFCA98]. These tools differ in the precision of the uncaught exceptions reported and in the form in which the information is reported. 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.

Other, primarily imperative, languages support the declaration of exception types that may arise through execution of a module. In these languages it is

[1] Exception Analysis Tool

possible to specify, in the signature of a method, a list of exception types that can be signaled by that method. Languages differ in the kinds of checking that are provided concerning declared exceptions.

In C++ [Str91], the language specification ensures that a method can only raise exceptions it declares. If a method signature does not include a declaration of exception, it is assumed that all types of exceptions may be raised. 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.

In contrast, in Java [GJS96] and CLU [LS79], the compiler ensures that clients of a function either handle the exceptions declared by that function, or explicitly declare to signal them. In addition to these checked exceptions, Java and CLU also support *unchecked* exceptions which do not place such constraints on a client. We describe the exception handling mechanism of Java and the problems it raises in further detail in the next section.

3 Exception Handling in Java

In Java, exceptions are first-class objects. These objects can be instantiated, assigned to variables, passed as parameters, etc. An exception is signaled using a `throw` statement. Code can be guarded for exceptions within a `try` block. 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 not caught in any `catch` block are propagated back to the next level of `try` block scope, possibly in the caller module.

Similar to other Java objects, exceptions are instances of a type, and types are organized into a hierarchy. What distinguishes exceptions from other objects is that all exceptions inherit from the type `java.lang.Throwable`. The exception type hierarchy defines three different groups of exception types: *errors*, *runtime* exceptions, and *checked* exceptions. Errors and runtime exceptions are unchecked. 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. By convention, errors represent unrecoverable conditions, such as virtual machine problems.

Java requires that checked exceptions which may be thrown from the body of a method be declared as a part of the method signature. The language also requires exception conformance [MT97], 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.

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 an assignment of an object of a subtype is made to a variable declared to be of its supertype.

This property is called *subsumption* [AC96]; a subtype is said to be *subsumed* in the parent type. When looking for a target, 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.

Java's support for unchecked exceptions and subsumption means that it is impossible 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.

4 Jex: A Tool for Producing a View of the Exception Flow

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 would be a tedious task for all but the simplest programs. In the case of an object-oriented program, a developer would have to 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, we have built the Jex tool, that automates this task for Java programs. In Sect. 4.1, we describe the view of exception flow and structure produced by Jex. In Sect. 4.2, we describe the implementation of the Jex tool.

4.1 Extracting Exception Structure

To retain meaning for a developer, we wanted to present a view of the exception flow within the context of the structure of the existing program. Our Jex tool thus extracts, synthesizes, and formats only the information that is pertinent to the task. In the case of Java, our tool extracts, for each method, 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, the class name and method signature raising that exception is 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 illustrate this exception structure using code from one of the constructors of the class `java.io.FileOutputStream` from the JDK 1.1.3 API. Figure 1 shows the code for the constructor; Fig. 2 shows the exception structure extracted according to our technique.[2] The extracted structure shows that the code preceding

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

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 to `openAppend` or `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);
     }
}
```

Fig. 1. The source code for the constructor of class `FileOutputStream`

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

Fig. 2. The structure of exceptions for a constructor of class `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 in Fig. 3: This code will pass the checking of the Java compiler as there is

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

Fig. 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" );
   }
}
```

Fig. 4. An example of code making use of Jex information

a handler for the declared exception, `IOException`. Applying our 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 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`: the presence of 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`.

4.2 The Architecture of Jex

Jex, which comprises roughly 20 300 lines of commented Java source code spread over 131 classes, consists of four components: the parser, the Abstract Syntax Tree (AST), the type system, and the Jex loader. We constructed the parser using version 0.8pre1 of the the Java Compiler Compiler™ (JavaCC) [Sun]. The current implementation of the tool supports the Java 1.0 language specification. We built the AST using the JJTree preprocessor distributed with JavaCC. We use the AST to identify the structure of a class, to identify the structure of exceptions within a method or constructor, and to evaluate the operation invocations and expressions that may cause exceptions to be thrown.[3]

The AST relies on the type system to return a list of all types that override 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 are considered. For example, the method `toString` 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 `Object.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.

To determine the actual exceptions thrown by a Java statement, 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 serve both to provide a view of the exception structure for the user, and as an intermediate representation for the Jex system.

To use Jex, a user must specify a list of packages, a path to search for Jex files, and a Java source code file. Currently, the Jex system requires that all necessary Jex files to analyze that source code file are available. We plan to eliminate this restriction in a future version.

[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 difficult to identify the program points where a class is first loaded.

5 Evaluating Jex

Our original hypotheses about the usefulness of the Jex tool were 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. We were interested in determining how often cases of exception subsumption and uncaught exceptions occur in released code, as well as whether the detection of these cases could suggest ways in which the code could be made more robust.

To investigate these two factors, we analyzed a variety of source code using Jex:

- JTar, a command-line utility for the extraction of tar files,[4]
- the `java.util.Vector` and `java.io.FileOutputStream` classes from the Sun™ Java Development Kit version JDK 1.1.3,
- the Sun™ `javax.servlet` and `javax.servlet.http` packages, version 1.23,
- a command-line rule parser, [5] and
- four database and networking packages from the Atlas web course server project[6]: `userDatabase`, `userData`, `userManager`, and `userInfoContainers`.

Together, these packages comprise roughly 6500 commented lines of code, including input/output, networking, and parsing operations.

In applying Jex to these packages, we made several choices. First, we decided not to generate exceptions signaled by the environment, such as `ArithmeticException` and `ArrayIndexOutOfBoundsException`, to avoid unnecessarily cluttering of the results with highly redundant exception types. Second, we did not perform the Jex analysis on the classes comprising the JDK API. Instead, we generated a Jex file for each of the relevant API classes using a script that extracts the information 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

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

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

[6] Under development by M. Kersten at the University of British Columbia.

comments. The exception types annotated in the comments for a class may include both declared and run-time exception types.

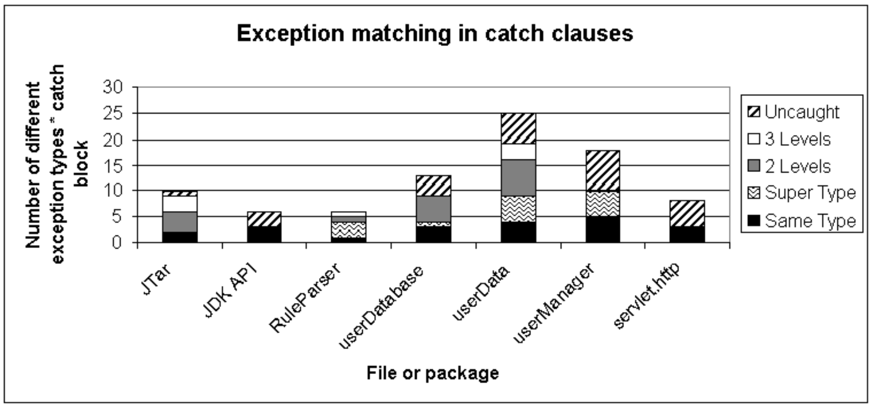

Fig. 5. Exception matching in `catch` clauses. The subsumption number indicates the level of inheritance that was necessary to match the exception type. For example supertype indicates that an exception was caught by declaring its supertype in the catch clause.

The graph in Fig. 5 shows a breakdown of exceptions and their associated handling in the analyzed code. It represents information from all the packages we analyzed that contained at least one `try` block. In all but one case, the Rule Parser, uncaught and undeclared exceptions might percolate from the subsumption used in exception handlers. Occurrences of "same type" refer to an exception handler that names the type of an actual exception that may occur; occurrences of "supertype" refer to an exception handler that names a superclass of an actual exception that may occur; and so on. All but two of the packages, namely the the Java JDK and servlet code, contain exception handlers that catch exceptions through subsumption.

Table 1 provides a different view of the data. This view illustrates that 32% of the different exception types present in `try` blocks remain uncaught in a target. In 44% of the cases, exceptions are not caught with the most precise type available.

This data lends evidence to support our 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. To investigate the usefulness of the information, we performed an after-the-fact manual inspection of the source code. We focused this inspection on cases of subsumption since the benefits of identifying uncaught exceptions are straightforward and are discussed in greater depth elsewhere [FFCA98, Yi94, YR97, Yi98].

Our 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. The addition of more specific exception handlers could improve the coherency of the recovery actions.

We 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 `StreamCorruptedException`, `InvalidClassException`, `OptionalDataException`, and `FileNotFoundException`. It may be advantageous to catch these exceptions directly and produce a more descriptive error message.

Cases of subsumption were also useful in pointing out source 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 hone in on these suspicious cases.

Table 1. Levels of subsumption required to catch an exception

Level of Subsumption	Frequency
Same type	24 %
Supertype	16 %
2 Levels	20%
3 Levels	8 %
Uncaught	32 %

6 Discussion

6.1 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 dependences could be introduced, potentially limiting the evolution of the client.

For instance, consider the case previously described for Atlas 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 operations 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 an operation 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. In cases in which the language environment does not provide this enforcement, our approach would have to be extended to ensure the use of white-box information does not complicate evolution.

6.2 Alternative Approaches

Increasing the robustness and recovery granularity of applications does not require a static analysis tool. One alternative that is currently in use is to document the precise types of exceptions that a method may throw in comments to 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. Our approach provides a means of cost-effectively determining which of the many possible handlers might be warranted at any particular development point.

6.3 The Expressiveness of the Current Exception Structure

The current exception structure extracted for source files enables a developer to determine unambiguously the exceptions that can be signaled at any point in the program, along with the origin of those exceptions. This former information allows a developer to determine the actual exceptions which 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 which can produce an exception. As a result, when an exception is explicitly thrown, it is not possible to determine if it is a new exception or if an existing exception is being re-thrown. This information could be determined by adding relationships to symbols (i.e. variable, parameter, and field names) defined in the programs. 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.

6.4 The Precision of Jex Information

There are two cases in which our 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 that might arise at runtime. 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 we used when applying Jex to the code described in Section 4 was partial, Jex still returned information useful to a developer. These two choices are however under a user's control: a Jex user who is concerned about finding all potential exceptions that might be raised must be careful about them.

The usability of our approach could also be impacted if a tool such as Jex returned information that was too conservative. With Jex, this situation can arise when reporting all possible runtime exceptions because there are many points in the code that can raise such exceptions as `ArithmeticException`. This situation can be managed by providing a means of eliding this information when desired. For the code we analyzed, we chose not to consider these runtime exceptions so as to more easily focus on application-related exception-handling problems.

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. Although our experience with Jex is limited, we have not found that this assumption greatly increases the exception information returned.

7 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 paper, we have described a static analysis tool we have built to help developers access this information. 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 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-grained 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 paper and the benefits possible are not limited to Java, but also apply to other object-oriented languages.

Acknowledgments

The authors are grateful to B. Speckmann for providing useful suggestions and comments, and to the authors of the code analyzed in this paper, for providing a basis for empirical testing of our ideas.

References

[AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer, 1996.

[FFCA98] 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.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley Longman, Inc., 1996.

[Goo75] John B. Goodenough. Exception handling: Issues and proposed notation. *Comunications of the ACM*, 18(12), December 1975.

[HMT89] 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.

[LS79] Barbara H. Liskov and Alan Snyder. Exception handling in CLU. *IEEE Transactions on Software Engineering*, 5(6), November 1979.

[MT97] Robert Miller and Anand Tripathi. Issues with exception handling in object-oriented systems. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97—Object-Oriented Programming, 11th European Conference*, volume 1241 of *Lecture Notes in Computer Science*, pages 85–103. Springer, June 1997.

[Str91] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 2nd edition, 1991.

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

[Yi94] Kwangkeun Yi. Compile-time detection of uncaught exceptions in standard ML programs. In Baudouin Le Charlier, editor, *Static Analysis Symposium*, volume 864 of *Lecture Notes in Computer Science*, pages 238–254. Springer-Verlag, September 1994.

[Yi98] Kwangkeun Yi. An abstract interpretation for estimating uncaught exceptions in standard ML programs. *Science of Computer Programming*, 31:147–173, 1998.

[YR97] Kwangkeun Yi and Sukyoung Ryu. Towards a cost-effective estimation of uncaught exceptions in SML programs. In *Static Analysis Symposium*, volume 1302 of *Lecture Notes in Computer Science*, pages 98–113, September 1997.