Static Analysis to Support the Evolution of Exception Structure in Object-Oriented Systems

MARTIN P. ROBILLARD and GAIL C. MURPHY
University of British Columbia

Exception-handling mechanisms in modern programming languages provide a means to help software developers build robust applications by separating the normal control flow of a program from the control flow of the program under exceptional situations. Separating the exceptional structure from the code associated with normal operations bears some consequences. One consequence is that developers wishing to improve the robustness of a program must figure out which exceptions, if any, can flow to a point in the program. Unfortunately, in large programs, this exceptional control flow can be difficult, if not impossible, to determine.

In this article, we present a model that encapsulates the minimal concepts necessary for a developer to determine exception flow for object-oriented languages that define exceptions as objects. Using these concepts, we describe why exception-flow information is needed to build and evolve robust programs. We then describe Jex, a static analysis tool we have developed to provide exception-flow information for Java systems based on this model. The Jex tool provides a view of the actual exception types that might arise at different program points and of the handlers that are present. Use of this tool on a collection of Java library and application source code demonstrates that the approach can be helpful to support both local and global improvements to the exception-handling structure of a system.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging—*Error handling and recovery*; D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*Object-oriented programming*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*; D.3.2 [**Programming Languages**]: Language Classifications—*Object-oriented languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms: Languages, Reliability, Verification

Additional Key Words and Phrases: Error handling, exception handling, exception structure, exception flow, program evolution, static analysis

A preliminary version of this article appeared in *Proceedings of the seventh European Software Engineering Conference and seventh ACM SIGSOFT Symposium on the Foundations of Software Engineering*. Lecture Notes in Computer Science (Sept.), 1999, Vol. 1687, 322–337. Springer-Verlag.
This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).
Authors' address: Department of Computer Science, University of British Columbia, 2366 Main Mall, Vancouver, BC, Canada V6T 1Z4; email: {mrobilla,murphy}@cs.ubc.ca.

1. INTRODUCTION

Most modern programming languages provide an exception-handling mechanism. Syntactically, an exception-handling mechanism provides a means to raise an exceptional condition explicitly, and a means to express a block of code to handle one or more exceptional conditions [Goodenough 1975]. By enabling software developers to separate source code that deals with unusual situations from code that supports normal processing, exception-handling mechanisms are intended to make it easier for developers to conceptualize and build robust software systems.

Separating the exceptional structure from the code associated with normal operation has consequences. One consequence is that developers wishing to improve the robustness of a program must figure out which exceptions, if any, can flow to a point in the program. Some exception-handling mechanisms help a developer in this reasoning process. The mechanisms for Java [Gosling et al. 1996] and CLU [Liskov and Snyder 1979], 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 an unchecked exception by the compiler. Object-oriented languages, which typically support the classification of exceptions into exception-type hierarchies, further complicate the reasoning about exception structure because a handler that explicitly names one exception type may implicitly catch a set of more specific exception types. Such implicit catching of exception types can complicate the development and evolution of robust classes [Miller and Tripathi 1997; Robillard and Murphy 2000].

In this article, we present a model that encapsulates the minimal concepts necessary for a developer to reason about exception flow for the purpose of software evolution (Section 2). Using these concepts, we describe why exception-flow information is needed to build and evolve robust programs (Section 3). We then describe Jex, a static analysis tool we have developed to provide exception-flow information for Java systems (Section 4). The Jex tool extracts information about the flow of exceptions in Java programs, providing a view of the actual exception types that might arise at different program points and of the handlers that are present. In Section 5, we describe how we have used this tool on a collection of Java library and application source code to demonstrate that the approach can be helpful to support both local and global improvements to the exception handling structure of a system. In Section 6, we compare our approach to other exception-flow analysis tools and techniques. Finally, Section 7 summarizes and concludes the article.

2. EXCEPTION FLOW

Goodenough [1975] introduced the exception-handling concepts in use today. To provide a common basis for discussion, we begin with a brief review of these concepts and the related terminology.

An *exception* corresponds to an abnormal state in the execution of a program. An exception is *raised* when such a state is detected. An exception *handler* is a lexical region of code that is executed in response to an exception occurrence. Different programming languages have different rules for matching an exception occurrence to a specific handler. An exception is *handled* when the execution of the handler is complete [Miller and Tripathi 1997]. The control flow of a program after a handler is executed is determined by an exception-handling *model* [Yemini and Berry 1985].

Three exception-handling models are commonly referred to in the literature [Buhr and Mok 2000; Miller and Tripathi 1997; Yemini and Berry 1985]. In the *termination* model, the lexical scope raising an exception is destroyed, and, if a handler is found and executed, control resumes at the first syntactic unit following this handler. In the *resumption* model, once an exception is handled, control continues where the exception was raised. Finally, in the *retry* model, when an exception is handled, the syntactic block raising the exception is terminated and then retried.

There are a number of variants of exception-handling mechanisms: many variants can be distinguished by the exception model supported, and by the rules used to bind a handler to an exception occurrence. In this article, we focus on *class-based* [Abadi and Cardelli 1996] object-oriented languages that implement the termination model of exception handling, and in which handler selection is based on object types [Dony 1990]. Two common programming languages that fit this description are Java and C++ [Stroustrup 1991].

We present an overview of the challenges involved with reasoning about exceptions in these languages, present a generalized model to support reasoning about exception flow, and describe exception handling in Java with respect to this model.

2.1 Challenges

In the exception-handling mechanism we consider, an exception is represented as an object of a class. This representation allows type subsumption to occur. Abadi and Cardelli [1996, p. 18] define subsumption as follows: "...*subsumption*, is the characteristic property of subtype relations. By subsumption, a value of type A can be viewed as a value of a supertype B. We say that the value is subsumed from type A to type B."

A developer who is trying to determine the flow of exceptions must consider the effects of type subsumption in two places. First, type subsumption affects the selection of a handler. A handler is selected based on the run-time type of the exception object. Following the type compatibility rules of object-oriented languages [Halbert and O'Brien 1987; Liskov and Wing 1994], a handler for an exception type E can catch any exception object that is of a subtype of E.

Second, subsumption also plays a role in defining exceptions that may propagate from a method. Several languages, including Java, support the declaration of a list of the exception types, the *exception interface*, that may propagate out of the method [Scott 2000]. Again following type compatibility rules, exception types declared in the exception interface subsume any of their subtypes.

As a result, a method declaring to propagate an exception of type E can also propagate exceptions of any subtype of E.

2.2 Exception Flow Model

To support reasoning about exception flow in the context of program evolution, we present a general model of the exception-handling structures and algorithms that can influence exception flow in object-oriented languages that define exceptions as objects. This model is adapted from the work of Schaefer and Bundy [1993] on Ada systems, and, wherever possible, we use consistent terminology. The goal of the model is to provide a unified basis for discussing problems related to the design, implementation, and maintenance of exception-handling structures, and of the analyses that can help alleviate these problems. The focus of the model is on the description of possible exception flows in a program.

In our model, a *scope* $s = (I, G)$ represents an atomic step of control flow for exceptions. Any exception encountered in a scope flows to the boundary of the scope without further modifying the control flow of the program. A scope consists of a set I of instructions and a set G of *guarded scopes* defined immediately within the scope.

A guarded scope, $g = (s, C)$, is defined as a scope that can prevent, or *catch*, certain exceptions from propagating to the enclosing scope. A guarded scope consists of a scope, s, that is guarded, and a sequence C of `catch` clauses. Each `catch` clause $c \in C$ is a type of exception that can be caught if encountered in the guarded scope. In languages with a singly-rooted exception type hierarchy (e.g., Java), the root of the exception type hierarchy can be used in a `catch` clause to catch all types of exceptions. In languages that do not have a singly rooted exception type hierarchy (e.g., C++), a special keyword can be used for this purpose.

For a sequence C of `catch` clauses, it is important to note that the model does not explicitly include a notion of handler. In the termination exception model, exceptions raised in a handler cannot propagate back into the guarded scope. The handlers that would typically be attached to `catch` clauses are thus considered to be a part of the enclosing scope, since any exception they can raise would propagate to the next enclosing scope.

To support reasoning about exception flow, we define three functions:

$$\begin{aligned} \mathit{encounters}(s) &\rightarrow E \\ \mathit{catches}(g, c) &\rightarrow E \\ \mathit{uncaught}(g) &\rightarrow E \end{aligned}$$

where s is a scope, $g = (s_1, C)$ is a guarded scope where s_1 is a scope and C is a sequence of `catch` clauses, $c \in C$ is a `catch` clause, and E is a set of exception types. The *encounters* function returns the set of exception types that a scope can encounter. The *catches* function returns the set of exception types encountered in a scope that would be caught by a particular `catch` clause. The *uncaught* function returns the set of exception types that are not caught by any `catch` clause associated with the given guarded scope. We now define each of these functions.

Encounters. The exception types that a scope can encounter consist of all of the exception types that may occur during the execution of instructions associated with that scope and that are not caught during the execution of any associated guarded scope. For a scope $s = (I, G)$, we have

$$\text{encounters}(s) \equiv \text{generates}(I) \cup \text{propagates}(I) \cup \text{raises}(I) \cup \text{uncaught}(G)$$

The *generates*$(i) \rightarrow E$ function returns the set of exception types that can be raised as the result of a system operation in the instruction i, such as a division by zero or a null pointer dereference. The definition of this function depends on the semantics of the programming language that is being analyzed. For convenience, we define

$$\text{generates}(I) \equiv \bigcup_{\forall i \in I} \text{generates}(i)$$

Similar definitions apply to *propagates*(I) and *raises*(I).

The *propagates*$(i) \rightarrow E$ function returns the set of exception types that can be propagated by a method call at instruction i. We model the definition of a method as a scope. Let S_m be the set of all scopes corresponding to the definition of methods that can be called as the result of executing an instruction i according to the semantics of the programming language.[1] We then have

$$\text{propagate}(i) \equiv \text{encounters}(S_m)$$

The *raises*$(i) \rightarrow E$ function returns the set of exception types, if any, that are explicitly raised by the instruction i, according to the semantics of the language. For example, in Java, an instruction involving the keyword `throw` explicitly raises an exception.

Catches. The set of exception types that a `catch` clause c can catch consists of the exception types that the corresponding guarded scope can encounter, that match c according to typing rules, and that are not caught by any `catch` clause defined lexically before c for a given guarded scope. In other words, `catch` clauses are evaluated in order and the first matching one is selected. We represent a class c_1 to be *assignable* to a variable of class type c_2 according to a language's type system as $c_1 < c_2$.

For a guarded scope $g = (s, C)$, where $C = (c_1, ..., c_n)$, we have:

$$\text{catches}(c_i) \equiv \{e \mid e \in \text{encounters}(s) \wedge e \notin \bigcup_{j=1..i-1} \text{catches}(c_j) \wedge e < c_i\}.$$

We also define the *catches* function on all the `catch` clauses of a single guarded scope:

$$\text{catches}(C) \equiv \bigcup_{\forall c \in C} \text{catches}(c).$$

[1]For an object-oriented language, this usually includes a set of dynamically bound methods.

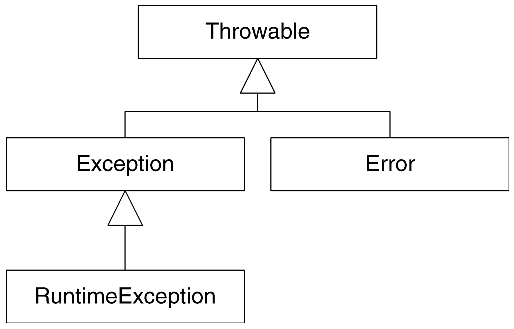

Fig. 1. Exception type hierarchy in Java.

Uncaught. The set of uncaught exception types for a guarded scope consists of all of the exception types that the guarded scope can encounter and that would not be caught by any `catch` clause. For a guarded scope $g = (s, C)$, we have

$$\text{uncaught}(g) \equiv \text{encounters}(s) - \text{catches}(C).$$

Similar to other exception flow functions, we define *uncaught* over a set of guarded scopes:

$$\text{uncaught}(G) \equiv \bigcup_{\forall g \in G} \text{uncaught}(g).$$

2.3 Exception Handling in Java

To illustrate how the exception-handling mechanism of a real language can be mapped onto the general model, and to describe the exception-handling mechanism that is used in the rest of the article, we describe the syntax and semantics of exception handling in Java in terms of the model. Mapping Java to our model requires that we make conservative assumptions to handle language-specific considerations, such as `finally` blocks. Mappings for other languages are outside the scope of this article.

In Java, exceptions are objects of a type that are inherited from a special `Throwable` class. The exception type hierarchy defines three different groups of exceptions: *errors*, *run-time* exceptions, and *checked* exceptions (see Figure 1). Errors and run-time exceptions are unchecked. Unchecked exceptions can be raised at any point in a program and, if uncaught, may transparently 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. User-defined errors and run-time exceptions are created by extending the `Error` and `RuntimeException` classes, respectively. Checked exceptions can also be raised at any point in a program. However, Java requires that checked exceptions that may be raised in the body of a method be declared as part of the method signature. The compiler can then check that a caller of the method either handles the exception or declares the exception in its interface. User-defined checked exceptions are created by extending the `Exception` class.

In our model, a Java method corresponds to a scope $s = (I, G)$. The set of instructions I comprises all of the instructions in the method that are not within a guarded scope. The set G comprises all of the guarded scopes declared in the method scope, but does not include any of the nested guarded scopes; these are declared in the appropriate parent scope.

In Java, a guarded scope $g = (s, C)$ is defined with a `try` block. A `try` block can have zero or more `catch` clauses. Each `catch` clause declares an exception type and a parameter name for the exception object that is caught. A handler block follows each `catch` clause. Handlers can contain any Java code, including other `try` blocks. Any code within the handlers that is not in a guarded scope belongs to the outer scope. Optionally, a programmer can attach a `finally` block to a `try` block. The `finally` block is executed whether or not an exception is raised in the `try` block. For a guarded scope, the inner scope s is the set of instructions and guarded blocks defined within the lexical scope of the `try` block. The list C of `catch` clauses consists of the ordered list of exception types declared in `catch` clauses that are attached to the `try` block. When a `try` block has a `finally` block but no `catch` clause, we chose not to model the `try` block as a guarded scope because it is only in special cases that the `finally` block can decrease the number of exception types flowing in a program. Our approach is thus conservative. We postpone a detailed discussion of the exception-handling behavior of `finally` blocks in Java programs to the end of this section.

For any instruction $i \in I$ within a scope, the *generates*(i) set corresponds to exceptions that can be raised by the Java run-time environment. Section 15.6 of the Java Language Specifications [Gosling et al. 1996] summarizes the exceptions that can be raised at run-time as the result of a basic operation.

The *propagates*(i) set is determined by exceptions propagated as the result of method calls. Java requires that checked exceptions that may propagate from a method be declared as a part of the method signature. The language also requires *exception conformance* [Miller and Tripathi 1997], 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. Because of unchecked exceptions and the fact that exception interfaces can subsume exception subtypes, the *propagates*(i) set cannot be completely determined by inspecting exception interfaces, and must be computed using the formulas of the previous section. For instructions that do not contain any method call, *propagates*$(i) = \emptyset$.

Exceptions in Java can be explicitly raised using the keyword `throw` followed by an expression that can be statically resolved, at compile-time, to a type $t <$ `Throwable`. For such instructions, *raises*(i) comprises all of the types that the raised expression can resolve to at run-time. If the instruction i is not a `throw`, then *raises*$(i) = \emptyset$.

Figure 2 illustrates the main exception-handling features in Java. In this example, `method1` contains two `try` blocks: one is top-level (lines 4–15) and the other is nested in the top-level block (lines 6–9). The `catch` clause at line 10 declares the supertype of all exception types and thus no exception should remain uncaught from line 8. Presuming an exception of type `ExceptionType2` can be raised at lines 12 or 14, this exception is caught by the handler at line 16. In the handler, a new exception of type `ExceptionType1` is raised

```
1:  public void method1() throws ExceptionType1
2:  {
3:     // instructions                                ⇑
4:     try                                            ‖
5:     {                                              ‖
6:        try                                      ⇑  ‖
7:        {                                   s3   ‖  ‖
8:           // instructions                  ⇕    ‖  ‖
9:        }                                        ‖  ‖
10:       catch( Throwable e )                     s2 ‖
11:       {                                        ‖  ‖
12:          // instructions                       ‖  ‖
13:       }                                        ‖  s1
14:       // instructions                          ⇓  ‖
15:    }                                              ‖
16:    catch( ExceptionType2 e )                      ‖
17:    {                                              ‖
18:       throw new ExceptionType1();                 ‖
19:    }                                              ‖
20:    finally                                        ‖
21:    {                                              ‖
22:       // instructions                             ‖
23:    }                                              ‖
24:    // instructions                                ⇓
25:}
```

Fig. 2. An example of exception handling in Java.

(line 18). For the sake of our example, this exception is a checked exception; since it can propagate out of the method, it is declared as part of the method signature.

The example is modeled as follows: (bars on the right of the figure indicate the range of each scope). The method is a scope $s_1 = (I_1, G_1)$ where I_1 consists of the instructions in lines 3, 18, 22, and 24, and G_1 consists of the single outer `try` block (lines 4–15) and associated `catch` clause (line 16). This `try` block defines an inner scope $s_2 = (I_2, G_2)$ where $I_2 =$ line 12 and 14 and G_2 is the innermost `try` block (lines 6–9) and associated `catch` clause (line 10). The innermost `try` block defines scope $s_3 =$ line 8. It is important to note that the code for the handlers and `finally` block is part of the scope *enclosing* their corresponding guarded scope.

Conservative Assumptions for `finally` *Blocks.* In a Java program, the execution of a `finally` block can alter the flow of exceptions in two basic cases.

(1) An exception raised in a `finally` block overrides an exception raised in the `try` block or in any handler for the `try` block (Java language specifications [Gosling et al. 1996, Sec. 14.19.2]).
(2) A `return` statement in a `finally` block, or a `break` or `continue` statement in a `finally` block nested in a loop results in a pending exception being discarded (Java language specifications [Gosling et al. 1996, Sec. 14.16, 14.14, and 14.15, respectively]).

```
public void crash( boolean pCrash )
{
    try
    {
        throw new Exception1Type1();
    }
    finally
    {
        if( pCrash )
            throw new ExceptionType2();
    }
}
```

Fig. 3. Overriding an exception in a `finally` block.

```
public void crash( boolean pCrash )
{
    while( true )
    {
        try
        {
            throw new ExceptionType1();
        }
        catch( ExceptionType2 e )
        {}
        finally
        {
            if( !pCrash )
                break;
        }
    }
    System.out.println( "Completed normally!" );
}
```

Fig. 4. Discarding an exception in a `finally` block.

Figure 3 illustrates the first case. In this example, if the input to the `crash` method is `true`, the pending exception of type `ExceptionType1` raised in the `try` block will be discarded and the method will propagate an exception of type `ExceptionType2` raised in the `finally` block. According to the mapping described in this section, the `try` block is not a guarded scope because it does not have any `catch` clauses. The instructions in both the `try` and `finally` block are mapped to the top-level method scope, and the *encounters* function for the scope representing the `crash` method returns the types `ExceptionType1` and `ExceptionType2`, the two types of exceptions that can be propagated by the method. Even though, in practice, the `finally` block can have an effect on the run-time exception flow, this does not influence the values computed for the *encounters* function in the model, since it is defined based on the conservative assumption that the value of `pCrash` can be either `true` or `false`.

Figure 4 illustrates the second case. In this example, execution of the `try` block results in an exception of type `ExceptionType1` being raised. Assuming that this exception is not caught by the `catch` clause, the `finally` block is executed with the exception pending. However, if the value of `pCrash` is false, the finally block will break out of the loop, the exception will be discarded, and the method will complete normally. Again, even though the finally block in this case also has an effect on the run-time exception flow, it does not influence the value of the *uncaught* set for the guarded scope (or try block). In all cases, the *uncaught* set for the try block includes Exception1, since the conservative assumption in this case is that pCrash can take both values.

In both cases, the exception flow-altering statement in the finally block is conditional upon the run-time value of the pCrash parameter. However, in the case where the statement is unconditional, the value of the *encounters* function calculated for the scope comprising the finally block will be overly conservative. For example, in Figure 3, if we remove the statement if(pCrash), the exception of type ExceptionType2 raised in the finally block will always override the exception of type ExceptionType1 raised in the try block. As such, the precise *encounters* set for the modified crash method should not include type ExceptionType1.

To summarize, our model is not intended to support reasoning about intrascope control and data flow; instead it produces a conservative estimate of the types of exceptions a scope can encounter. This ensures that the model is robust in the face of constructs that can alter the intrascope flow of exceptions, such as the finally block. The tradeoff for this characteristic is potential imprecision in the presence of unconditional return or throw statements, or unconditional break or continue statements for a loop that encloses a finally block.

3. REASONING ABOUT EXCEPTION FLOW

Implementing a robust software system in a programming language that supports an exception-handling mechanism requires a software developer to figure out the exceptions that a particular scope can encounter. When improving the robustness of object-oriented systems, we have found it useful to think about exception flow both locally, within a guarded scope, and globally, in terms of flow across methods in a system.

3.1 Local Exception Flow

Local exception flow refers to exceptional control flow that is relevant to a single guarded scope. Even though such reasoning does require global exception-flow knowledge, we refer to it as local since its focus is on the analysis of the relationship between the *encounters* and *catches* sets for a single scope. When evolving a system, two localized cases of interest are unused catch clauses and the unanticipated handling of unexpected exceptions.

Unused catch clauses can occur for different reasons. For instance, the conditions for which a scope was guarded can be removed, but the handler for these conditions may be left in place. This dead code may be left for safety reasons, but often, it may be more appropriate to remove it. Figure 5 illustrates this problem. In this example, a method executes two different operations on a field, each of which takes an argument arg that has been passed in to the method. If a developer updates method1 to remove the calls to method2, the developer needs to determine if the guarded scope can be safely removed without changing the

```
public void method1( Object arg )
{
   try
   {
      field1.method2( arg );   // expected IllegalArgumentException
      field1.method3( arg );
      field1.method2( arg );   // expected IllegalArgumentException
   }
   catch( IllegalArgumentException e )
   {
      field1.rollback();
   }
}
```

Fig. 5. An example of unused `catch` clause.

behavior of the program. The documentation for `method2` may show that it can raise an `IllegalArgumentException`. However, the documentation for `method3` may not include any exception information. Since this exception is unchecked, the developer cannot determine whether the guarded block was meant to apply only to the calls to `method2` or also to the call to `method3`.

This situation can be prevented, to a certain degree, with exception interfaces since the compiler can determine if a checked exception will be raised in a scope. However, compilers do not perform this check either for unchecked exceptions or for subtypes of checked exceptions. In the latter case, if a method called in a guarded scope declares to be propagating an exception of type e_1, then either a handler for this exception type must be provided or the exception must be declared in the header of the method. However, if a handler for e_1 is provided, then any number of handlers for types $e < e_1$ can also be provided, even if exceptions of these types can never be propagated in practice. The compiler will not be able to determine whether these handlers are used or not. These excess handlers also represent dead code.

The second problem is the unanticipated handling of unexpected exceptions. This situation can occur when a `catch` clause is defined for a type that is the supertype of many exception types. Although this default catching of exceptions is considered bad practice in some communities, such as the Ada community [Romanovsky and Sandén 2001], it is common practice in Java where types are naturally organized in a hierarchy. Moreover, it can be a succinct way of expressing recovery code when various sources of failure call for a similar handling behavior. For example, in crash situations, it may not matter whether a system crashes for one reason or another, so a general handler can be used.

However, catching exceptions by subsumption is a dangerous practice because of the difficulty of determining all of the reasons for which an exception may be raised. A handler may end up executing code in response to an event that was not anticipated and that does not conform to the semantics for which the handler was designed. For example, in Figure 6, the handler is designed to catch both a `NullPointerException` and an `ArrayIndexOutOfBoundsException` originating from the array access statements. The general exception type, `Exception`, will catch both types of exceptions. The danger here is that the coding style assumes that the instructions in statement 1 cannot raise any

```
try
{
   anObject.anArray[index];
   // statement 1
   anObject.anArray[index];
}
catch( Exception e )
{
   // recover from array access problem
}
```

Fig. 6. Catching exceptions by subsumption.

exceptions that are unrelated to array access problems. If this assumption is wrong, or becomes false as the code evolves, it might introduce subtle bugs that make the program less robust.

3.2 Global Exception Flow

Global exception flow refers to the flow of exceptions across methods. A robust system will have one or more exception-handling policies whose goals are to avoid crashes by limiting the flow of unexpected exceptions, and to enable better recovery by describing exceptional conditions in a meaningful way to client modules.

A developer evolving an object-oriented system can make the conditions under which an exception is propagated explicit to a client either through exception type names or through documentation. Using exception types implies that a developer can leverage some support from the type system of the programming language at the cost of creating and maintaining a new class definition for each exception type. Documenting the cause of exceptions in comments may be cheaper, but only minimal support is available to help developers ensure the documentation is kept up-to-date.

Choosing the right alternative for making exceptions explicit and determining a good location for recovery of an exception is difficult because developers must reason about how an exception arises. Specifically, a developer must understand the propagation paths for the exceptions that can arise. In practice, we have found it useful to reason about the global flow of exceptions in a system from both the breadth and depth perspectives.

3.2.1 *Breadth.* The breadth perspective involves examining the set of different exception types a scope can encounter, and the causes for these exception types. This knowledge enables a developer to design suitable recovery, if recovery is applicable, or to design appropriate reporting of the failure to clients of the method.

Unfortunately, it is difficult for a developer to perform this reasoning manually because it is hard to determine the causes for the exception types defined by the *generates*, *propagates*, and *raises* functions for a method. Anticipating environment-generated exceptions requires an in-depth knowledge of the programming language semantics. Determining the exceptions propagated requires reasoning about the possible run-time bindings for a method call and about the exceptions types that each of the potential targets can encounter.

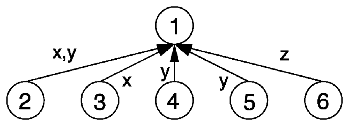

Fig. 7. Analyzing exception-flow breadth.

Compilers provide only limited support for reporting on the propagation of checked exception types, and no support for unchecked exception types. Indeed, for a scope, a compiler will usually only indicate that a certain type of exception must be handled. If a handler is present, the compiler does not report all of the possible causes of the exception type that might be encountered. It can even be hard to identify explicitly raised exceptions (the *raises* set) if the `throw` statements are located in deeply-nested, guarded blocks.

An example of the difficulty of analyzing exception flow breadth is shown in Figure 7. In this figure, circles represent methods and arrows represent the flow of exception types between methods. The figure documents the various exception types that method 1 can encounter, and the origins of exceptions of those types. The exception types x and y in this example can arise from a variety of sources. In one case, the same source, method 2, may result in either exception type x or y. This information can help a software developer determine if it is possible to recover from one or more of these exception types in method 1. If recovery is not possible, the information can help the developer determine how the method should report the various problems. For example, a developer might choose to propagate some exception types as received; others may be remapped to new exception types [Robillard and Murphy 2000].

3.2.2 *Depth.* The depth perspective involves determining, for a specific type of exception, the propagation paths that can result in the exception type being propagated to a specific method. This depth perspective is helpful in two ways: top-down, to help reason out the original cause for an exception type at a program point, and bottom-up, to help reason out the impact the propagation of an exception can have on the behavior of the system. Reasoning about the depth of exception flow is difficult because it typically requires determining whether the *propagates* set for each method call in the method of interest propagates a specific exception type, and, when it does, it requires iterating through each method until the *raise* or *generate* point for an exception is reached.

An example of analyzing exception-flow depth is shown in Figure 8. In this figure, all of the arrows represent a single exception type. Four separate paths exist that may cause the exception type to propagate from method 8 to method 1. Understanding these separate paths, or at least their existence, can help a developer evaluate the cost of declaring or renaming an exception type, and can provide more insight into the causes for its propagation. When evolving a system, the depth of propagation paths might cause a developer to represent an exception as an unchecked exception rather than a checked exception because of the cost of modifying the method interfaces to declare the exception.

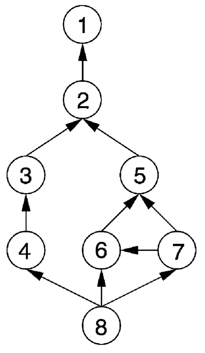

Fig. 8. Analyzing exception-flow depth.

4. THE JEX STATIC ANALYSIS TOOL

To help developers address the kinds of problems described in the previous section, we have developed a static analysis tool, called Jex, that computes exception flow for Java systems. We describe the functionality of Jex, explain its implementation, and discuss how a user interacts with the tool.

4.1 Functionality

An exception-flow tool must calculate the *generates*, *propagates*, and *raises* functions for any set of instructions I in a scope $s = (I, G)$. For a developer to use the information calculated, the tool must present the information in a format that supports understanding exception flow both locally, to address scope-level issues, and globally, to address system-level issues. To support local reasoning, Jex displays the exception types calculated for the *generates*, *propagates*, and *raises* functions in the context of the exception-handling structure—the `try` blocks, `catch` clauses, and `finally` blocks—in the source code. To support global reasoning, Jex displays all possible origins of each exception type reported.

Specifically, the Jex tool marks exception types reported by the *generates* function with a special `*environment*` string. Exception types reported by the *propagates* function are followed by the name and signature of the method that raises the exception (prefixed by the fully qualified name of the class declaring the method). Finally, exception types reported by the *raises* function are prefixed by the `throws` keyword. All possible origins are listed for each exception type.

The information provided by Jex conforms to the model of Section 2.2. In addition, Jex presents the origin of the various exception types for a scope, and distinguishes the lexical context for handlers and `finally` blocks from the rest of the enclosing scope.

We illustrate this view of exception structure using code from one of the constructors of the class `java.io.FileOutputStream` from the JDK 1.1.3 API. Figure 9 shows the code for the constructor; Figure 10 shows the exception

```
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. 9. The source code for the constructor of class `FileOutputStream`.

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

Fig. 10. Exception flow and structure for the constructor of class `FileOutputStream`.

structure extracted according to our 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 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 signature of the constructor.

[2]Figure 10 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 and exception information that was redundant for the purpose of our example.

In this example, the retention of the basic method structure shows the declared IOException, the top-level try block, and the single catch clause. The analysis shows that the top-level scope (lines 3, 4, and 12) can encounter a SecurityException that may be propagated by a method call (line 3), a NullPointerException that may be generated by the environment (line 4),[3] and a FileNotFoundException that may be raised explicitly (line 12). The scope of the try block (lines 7 and 8) can encounter an IOException from two possible origins, the two methods called in the block. These IOExceptions, if raised, will be caught by the catch clause. Using the algorithms of Section 2.2, the *uncaught* set for the try block is empty, and the *encounters* set for the constructor comprises the FileNotFoundException, NullPointerException, and SecurityException exception types. In this case, the exception interface is valid, since NullPointerException and SecurityException are unchecked exceptions and need not be declared, and FileNotFoundException < IOException.

The information produced by Jex makes it possible for a developer to understand the flow of exception types, not exception objects. As such, exception-flow information produced by Jex cannot be used to determine, for example, if an exception raised in a handler is the same exception object as the one that was caught by the catch clause. Furthermore, if different methods propagate the same exception type through a call chain, it should not be assumed that it is the *same* exception that is propagated. An exception could be caught, and a new exception of the same type could be raised. We have left out the exception values from our analysis because our goal was to support reasoning about the control flow caused by exceptions; determining the flow of exception values is thus outside the scope of the work. In the infrequent cases where a developer would need to examine the data flow for a specific exception object, traditional data-flow analysis [Shelekhov and Kuksenko 1999] can provide this support.

4.2 The Architecture and Implementation of Jex

Jex takes as input a set of Java source files and a configuration file (described in Section 4.4), and produces, for each Java class, a human-readable *Jex* file. A Jex file contains, for each method in the associated Java class, a view of the exception flow formatted as shown in Figure 10. A Jex file is both a result of the exception-flow analysis, and an input to the analysis process, as we describe below. Jex files are organized in a hierarchical directory structure based on package names.

Jex (version 1.2.1) is implemented as an extension to the code of Kjc (www.dms.at/kopi), an open-source Java compiler. The architecture of Jex is organized around four central functions: parsing and typechecking Java files, extracting exception flow, loading Jex files for querying, and performing class-hierarchy analysis to determine potential dynamic bindings for method calls.

[3]The syntactic analysis performed by Jex cannot determine that the local variable security is never null at that point. Section 4.3 discusses how additional semantic analysis can help eliminate false positives in the results of the *generates* function.

The parsing and typechecking of Java source files is performed by the Kjc compiler. This compiler operates like the standard Sun Javac compiler, taking as input a series of Java source files. The Kjc compiler parses the Java files and produces an abstract syntax tree (AST) for each file. The Kjc compiler also typechecks the code, which evaluates the type of all expressions in the AST. The Kjc compiler provides access to the AST through an implementation of the Visitor design pattern [Gamma et al. 1995].

We implemented the extraction of exception-flow information as a pass through the AST. The AST visitor we implemented performs two tasks: it detects and stores the `try` blocks and their corresponding `catch` clause and `finally` blocks, and it computes the sets of exception types from the *generates*, *propagates*, and *raises* functions for all scopes. The first task requires a simple recursive descent through the AST. The second task involves examining each statement, and computing the type and origins of exceptions that could result from the statement according to the three functions.

For a statement, Jex reports environment-generated exception types based on determining, according to the conditions summarized in Section 15.6 of the Java Language Specifications [Gosling et al. 1996], whether the statement being examined can generate exceptions.

To determine the exceptions propagated by a method call, Jex first determines all possible dynamic bindings for the call. Jex determines all potential targets of virtual method calls by performing *class hierarchy analysis* (CHA) [Dean and Chambers 1995] over all of the classes in a set of packages specified by the user. Next, Jex determines the exception types in the *propagates* set for each target. Jex determines this information by accessing and parsing the Jex file for each method implementation. Two cases require special processing: methods for which the source code is not available, and methods for which the exception flow has not yet been computed. It is not possible to handle the second case purely by ordering the computation because the presence of cycles in the control flow of the program will prevent the determination of a partial order on the execution of methods. Jex handles these two cases by initially creating a stub for any method of interest it encounters, using the exception interface contained in the byte code for the methods. Jex relies on this stub if further analysis is not possible. Jex iterates until a global fixed point is reached for all methods in the system; as Jex iterates, a stub in the system can be expanded into full Jex information based on the analysis of a method body.

To calculate the *raises* set, Jex uses the static type of the expression of the Java `throw` statement as the type of an exception that is explicitly raised.

4.3 Tradeoffs

To strike a balance between the usefulness and precision of the information provided to a developer, and to manage the cost of the exception-flow analysis, we have made a number of implementation tradeoffs in Jex.

Source Code Analysis. Java programs are compiled to byte code, an intermediate platform-independent format. Exception-flow information can be calculated from either the byte code or the source code. We have chosen source code because we wanted to present a view of the exception flow in the context of the source code in which the flow is relevant. As such, we wanted to present the complete exception-handling structures, including `catch` handlers and `finally` blocks. It can be difficult, and in some cases impossible, to obtain the lexical scopes for handlers and `finally` blocks from byte code. Furthermore, optimizations such as method inlining also create differences between byte code instructions and the corresponding source code.

Initialization Code. Java supports two kinds of initialization code. Class-scoped (static) fields can have assignments as part of their declaration. Class initializer blocks can be defined that are executed when the class is loaded. Each of these kinds of initialization code can raise exceptions when a class is loaded, and these exceptions are reported through an `ExceptionInInitializerError`. Practically, it is not possible to determine when a class will first be loaded. As a result, Jex does not provide exception-flow information for static initialization code.

Asynchronous Exceptions. Java supports *asynchronous* exceptions. 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 any scope can potentially encounter these exceptions, there is no value in reporting this information to users, and Jex does not report it.

Class-Hierarchy Analysis. The algorithm implemented in Jex to determine the targets of a virtual method call relies on the definition, by the user, of a set of packages to consider in the analysis. If a user fails to specify a relevant package, Jex may not report some exception types that may arise through a virtual method call. The tool can, in some cases, warn a user about this situation (see Section 4.4). This approach for determining the value of the *propagates* function can also be overly conservative. For example, some possible targets of a virtual method call may never be bound at that program point. More precision might be achieved using type-inferencing algorithms [Bacon 1998; Palsberg and Schwartzbach 1991; Plevyak and Chien 1994]. Further investigation is needed to determine whether the additional precision would benefit users and be cost effective.

Semantic Analysis. The tool calculates the *generates* function based solely on the type of input statement. For example, an integer division will consistently generate an `ArithmeticException`. The results produced are thus conservative, and do not depend on the semantics of the program. In some cases, semantic analyses, such as constant propagation [Callahan et al. 1986], could help make the result of the *generates* function more precise. For example, an integer division instruction should not generate an `ArithmeticException` if it can be determined with certainty that the divisor is nonzero. Similarly, array-access instructions should not generate an `ArrayIndexOutOfBoundsException` if it can be determined with certainty that the array index is within the array bounds.

Type inference for `throw` *Statements.* The tool calculates the *raises* function using the static type of the expression following a `throw` keyword in a scope. Technically, this information is incomplete since any subtypes of the static type could also be raised by a `throw` statement. Type-inferencing algorithms could potentially help determine a more precise set for the value of the type raised. However, in practice, this additional analysis might not be cost effective. For instance, other investigators have reported that "an overwhelming majority of `throw` statements are new-instance expressions, and, therefore, require no type-inference analysis" [Sinha and Harrold 2000, p. 860].

4.4 Using Jex

To run Jex, a user must provide a configuration file that describes the root of the path for Jex files, the packages for which stubs must be generated, and the packages to use for class-hierarchy analysis. The root path for the Jex files indicates where the hierarchical directory structure containing Jex files should be placed. As Jex runs, it expects a Jex file for any class it encounters. Stub generation ensures that Jex files for all classes that need to be queried for exception-flow information are present. The packages for class hierarchy analysis list the classes that are to be considered for determining the method implementations that can be involved at a method call.

Given this configuration information and the source files to analyze, Jex iterates over the source, producing a Jex file for each class analyzed, until a fixed point is reached. When the tool concludes, Jex produces a report with warnings and errors. Warnings are issued when a call is made to a method that is not defined by any class specified for class-hierarchy analysis. Such a problem can be corrected, if desired, by adding the package in which the method is defined. Errors are generated when a Jex file for a class is not found, or when a source file cannot be compiled correctly.

After running Jex, a developer can inspect Jex files of interest to determine the exception flow to a particular point in the program. This raw exception-flow data can also be filtered or visualized to ease the developer's investigation. For instance, one filtering tool we have written automatically detects `catch` clauses that catch exception types by subsumption; another filtering tool detects unused `catch` clauses. We have also built a tool to support the visualization of exception flow in terms of various structural entities, including packages, classes, methods, and components (a component is specified by the user as a collection of packages, classes, and/or methods). Figure 11 shows a screen snapshot of our visualizer, displaying part of the exception flow in a system we analyzed. For this view, the developer selected two classes of interest, resulting in the display (in boxes) of methods of the selected classes, as well as methods from other classes that may cause exceptions to flow into methods of the selected classes. The arcs between the boxes show the flow of exceptions from one method to another. Hovering over an arc in the visualizer displays the exception flow. For example, the view shows that the `BB_Member.init` method potentially propagates exceptions of eight different types to the `BB_Method.init`

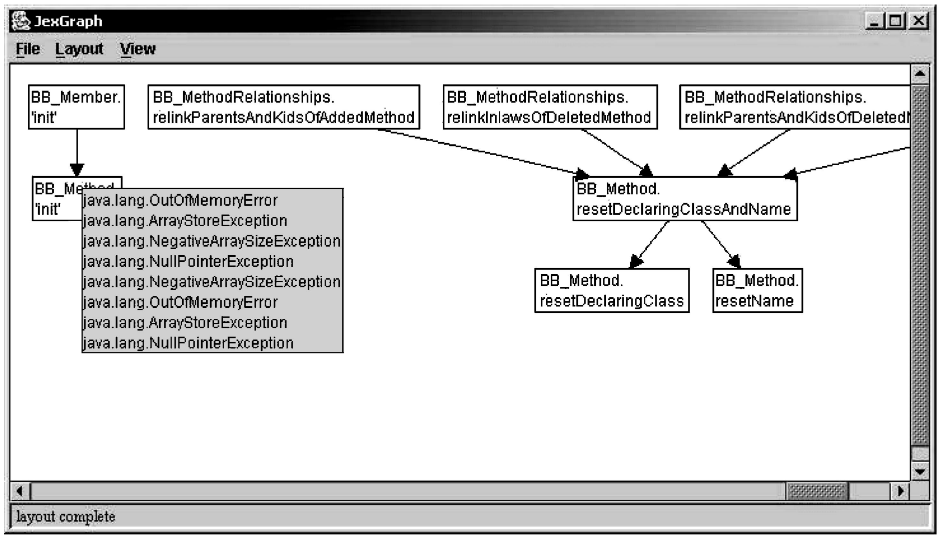
JexGraph
File
Layout
View
BB_Member. 'init'
BB_MethodRelationships. relinkParentsAndKidsOfAddedMethod
BB_MethodRelationships. relinkInlawsOfDeletedMethod
BB_MethodRelationships. relinkParentsAndKidsOfDeleted
BB_Method. 'init'
java.lang.OutOfMemoryError
java.lang.ArrayStoreException
java.lang.NegativeArraySizeException
java.lang.NullPointerException
java.lang.NegativeArraySizeException
java.lang.OutOfMemoryError
java.lang.ArrayStoreException
java.lang.NullPointerException
BB_Method. resetDeclaringClassAndName
BB_Method. resetDeclaringClass
BB_Method. resetName
layout complete

Fig. 11. The exception flow vizualization tool.

method.[4] The various structural entities we support in the visualizer allow a developer to view exception flow at a coarse structural level, such as across packages, and then to focus on the particular parts of the program of interest, such as specific classes.

5. USING EXCEPTION FLOW

To evaluate the practical benefits of extracting exception flow for programs, we ran our static analysis tool on three systems from different domains: the classes of the javax.swing package from the Java Development Kit version 1.3; Bobby,[5] a Java class file manipulation library; Mesquite [Maddison and Maddison 2001], a molecular biology application for drawing and analyzing phylogenetic trees. We chose to analyze both the libraries and applications in order to cover a range of exception-handling behavior. Requirements for error handling in libraries often differ from those in applications, as libraries can assume errors will be handled by client code whereas robust applications must perform all of the recovery.

Table I summarizes the size and characteristics of the systems analyzed. This table provides some traditional size metrics, such as the total number of packages, classes, fully defined (nonabstract) methods, and lines of source code (LOC).[6] The table also shows the total number of `try` blocks in each system. This figure gives an indication of the amount of exception handling in each system.

[4]In this case, all of the exceptions are unchecked; uncaught checked exceptions are reported similarly.
[5]Now distributed as the Jikes Bytecode Toolkit (www.alphaworks.ibm.com/tech/jikesbt).
[6]All the LOC figures in this article indicate the number of lines of source code, not including comments or blank lines.

Table I. Systems Analyzed with Jex

System	Packages	Classes	Methods	LOC	`try` blocks
Java swing classes	1	331	4097	33903	124
Bobby	1	117	1293	11134	42
Mesquite	25	460	5307	54487	107

We analyzed the exception-flow information produced by Jex for each system both locally, at the guarded scope level, and globally, at the system level. The guarded scope analysis allowed us to find places within the current exception-handling structure that could be improved. The system-level analysis allowed us to explore ways of integrating new exception-handling structures to improve the overall robustness of a system.

5.1 Local Analysis

In Section 3.1, we discussed two challenges facing developers interested in improving the exception-handling structure of a software system: detecting potentially unused `catch` clauses, and detecting and indentifying potential cases of improper handling of unexpected exceptions.

We used exception-flow information produced by Jex to address these concerns. To find these cases in each system, we developed a small tool that analyzes a Jex file and that returns, for each `try` block, the number of unused `catch` clauses, the number of uncaught exceptions, the number of exceptions caught by their direct type, and the number of exceptions caught by subsumption.

In the Mesquite code, over a total of 107 `try` blocks (all with `catch` clauses), we found seven unused handlers. Upon inspection, we found that five of these handlers were for an unchecked exception, `NumberFormatException`, that was not raised in the guarded block, and two of the handlers were for `FileNotFoundException`. Although these two handlers involved a checked exception type, they were not flagged as unreachable by the Java compiler because a method in the guarded scope declared to raise an `IOException`, a supertype of `FileNotFoundException`. Based on our analysis, we determined that all of the `catch` clauses in the Swing and Bobby systems apparently have a purpose.

Cases of potential improper handling of unexpected exceptions are more subjective. Such cases can occur when a `catch` clause catches exception types by subsumption. Figure 12 shows the significant percentage of exception types for each system we analyzed that may be caught by subsumption. In each case, the ratio represents the total number of different exception types encountered in `try` blocks that are caught by subsumption to the total number of such exception types that are caught, as calculated over all of the `try` blocks. The number of uncaught exception types is not represented.

We used the exception-flow information produced by Jex to point us to potential cases of improper handling. We looked at all of the `try` blocks in a system for which at least one exception type was caught by subsumption, and we inspected the code of the handler to determine whether it was specific to only a subset of the exception types subsumed. The analysis resulted in one potential trouble spot in the code of Bobby, one in the code of Mesquite, and five in Swing.

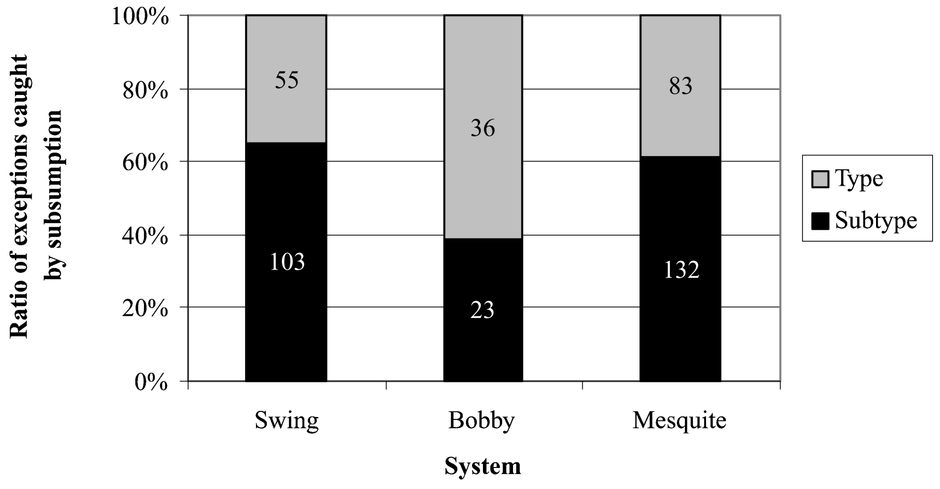

Fig. 12. Percentage of caught exceptions that are caught by subsumption.

In Bobby, one `try` block in method `loadClassFromClassPath` of class `BB_Class` declares a `catch` clause for the general `Exception` type. The `catch` clause is commented as follows, "assume all exceptions are due to unexpected class formats". However, the Jex exception-flow analysis for the corresponding `try` block shows that eight different exception types from eight different sources can be caught by this handler. Thus, there is a possibility that the assumption might not hold.

In Mesquite, a `catch` clause in method `setCurrentTool` of class `MesquiteWindow` declares type `Throwable`, the root of the Java exception type hierarchy, in an attempt to catch some exception types that would indicate that a particular version of the Java class libraries is unavailable. Since the handler assumes that a particular library is not available, the handler code simply uses a default value for a tool object. Even though this assumption might be true most of the time, our analysis showed that the handler can catch eight different exception types from 12 different sources. For the program to be robust, all of these exceptions would have to correspond to an unavailable version of the library, which is not the case.

For Swing, the information produced by Jex helped to point us to five potential trouble spots.

(1) `JComponent.readObject`—A handler assumes the failure of the `ReadObjectCallback` method that declares an `IOException`. However, other exceptions can be generated by the environment, or by the call to `UIDefault.put`.
(2) `JEditorPane$HeaderParser.findInt`—A `catch` clause declares the type `Throwable` but recovers with default behavior. This `catch` clause can catch many fatal errors.
(3) `JEditorPane.read`—A handler performs recovery based on the assumption that a `NullPointerException` caught is propagated by a call to `HeaderParser.findValue`. However, `NullPointerException` can also be propagated by a call to `putClientProperty`, which does not correspond to the semantics of the handler.

(4) `KeyStroke.getKeyStroke`—An exception might cause this method to return null according to the wrong semantics. This problem is documented in the comments for the method.
(5) `JTable$GenericEditor.getTableCellEditorComponent`—The method returns a null component under three exceptional conditions, one of which is a potential null pointer dereference, which could make the system unstable.

In addition to helping in the detection of such potential trouble spots, exception-flow information provides insight into how to improve the exception structure in these cases by making explicit the types of exceptions that may cause the handlers to be reached.

5.2 Global Analysis

We analyzed the exception-flow information produced by Jex for each system from both a depth and a breadth perspective as described in Section 3.2. These perspectives helped us determine ways in which we could improve the exception handling in the Mesquite application and the Bobby library.

5.2.1 *Global Breadth Analysis.* Mesquite was designed using a "pluggable module" paradigm, in which different combinations of modules can be loaded at run-time and interact. To support this paradigm, the behavior of some of the classes of the system is performed through the execution of a `doCommand` method, defined in a `Commandable` interface. In total, 43 classes implement the `doCommand` method. Many of these implementations invoke the `doCommand` method of other classes.

Currently, `doCommand` methods are not guarded for exceptions. As a result, various unexpected failures in commands can propagate to other commands. Additionally, the last statement in most `doCommand` methods is a *super* call to `doCommand`, that is, any command not recognized by a class is passed to the superclass for interpretation. The lack of structure for dealing with command failures can lead to the propagation of a number of uninformative exceptions. For example, the `doCommand` method of class `ManageForeignBlocks` can encounter nine different types of exceptions originating from 26 different sources—25 methods calls and the environment. Because the exceptions raised in `doCommand` do not have any strong meaning in the context to which they are propagated, the only possible recovery for clients is to catch all exceptions and to announce a crash. Indeed most of the handlers in Mesquite only perform basic crash recovery.

A possible first step to improving the robustness of Mesquite is to implement handlers around the `doCommand` method that catch all exceptions. The handlers would ensure that a `doCommand` method has only two modes of returning, either normally or exceptionally, by raising a `CommandFailedException`. Exception-flow information could then be used to implement the handlers. By inspecting the various causes of failure within a `doCommand` method, it is possible to figure out whether all of the exceptions in the method should simply be remapped to `CommandFailedException`, or whether specific causes of failures should be reported separately, through a subtype of `CommandFailedException`. Learning

```
catch( Exception e )
{
   started = false;
   startupTime.end();
   Debugg.printStackTrace(e);
   zeroMenuResetSuppression();
   module.alert("There has been a problem starting up a module. " +
                 "This may be the result of an old, incompatible module being used. " +
                 " (module: " + mb.getName() + "; EXCEPTION: " + e.getMessage() + ")");
}
```

Fig. 13. Handler for `Exception` in `EmployerEmployee.startupEmployee`.

about the various ways in which a method can fail can also enable better failure messages to be encapsulated in an exception object.

Another example from Mesquite is the case of method `startupEmployee` of class `EmployerEmployee`. This method implements the behavior to start a worker module to perform some task. Currently, the handler performs basic recovery by announcing to users of Mesquite that some problem has arisen. Figure 13 shows the code for the handler. Clearly, only basic recovery is provided, and the cause for the crash provided to clients is hypothetical. In this case, exception-flow analysis in the guarded block could help improve this handler by providing specific origins for failures in a robust fashion.

5.2.2 *Global Depth Analysis.* Bobby is a library of classes to load and manipulate Java byte code. In the version of Bobby analyzed, many problems were signaled as `InternalErrors`. In Java, `InternalErrors` are system-defined, potentially asynchronous, exceptions that signal a problem with the Java virtual machine [Gosling et al. 1996]. From the point of view of a library, this strategy has drawbacks which were first identified in previous work [Robillard and Murphy 2000]. For instance, a variety of problems, including bugs, inconsistencies, user errors, invalid input, and locking problems are all signaled through one exception type. As another example, the use of `InternalError` prevents client code from catching errors in the Bobby class library independently from errors raised in the application code.[7] A possible improvement in this case is to convert the `InternalErrors` to user-defined exceptions describing a problem with the library. In doing so, one must decide how these errors should be reported. Factors influencing this choice include the value of documenting an exception and the cost of documenting it. The most inexpensive alternative, to make all exceptions unchecked and undocumented, prevents any meaningful client-side recovery. An intermediate alternative, documented unchecked exceptions, allows, but does not ensure, client-side awareness at the cost of documenting all methods raising the exception with a comment (e.g., the Javadoc `@exception` comment). Finally, reporting the problems as checked exception ensures client-side awareness at the cost of declaring the exception in the interface of each method that can transitively propagate it. This also involves a cost for clients modules which are then forced to handle or declare the exception. The value

[7]These problems were corrected in a later version of Bobby.

of documenting an exception is determined by outside factors. The cost can be determined by analyzing exception flow. Basically, exceptions with complex propagation graph are expensive to document while ones with small graphs are cheap.

In Bobby, method `write(DataOutputStream)` of class `BB_Class` throws an `InternalError` when an attempt is made to write out a class marked as a special system class. An analysis of the depth of the exception flow shows that the `InternalError` raised in `write` can only be propagated to three other methods. These methods, entry points to the library, do not propagate the exception further within the library. The visibility for this exception is thus limited to a total of four methods. The effort involved in documenting this case is to create a new exception class, for example, `IllegalClassWriteException`, raised in `BB_Class.write`, and to document the interfaces of the four methods propagating it. Whether or not to make this exception a checked exception depends on the frequency with which the case can occur, and on how critical it is for clients to be aware of this case.

6. RELATED WORK

Various static analysis tools and techniques have been proposed to address problems related to exception handling in different programming languages. We distinguish between two broad classes of tools and techniques for exception handling based on the underlying goal: human-oriented approaches, meant to help developers build robust programs, and machine-oriented approaches, meant to be integrated into compilers or other static analysis tools (e.g., dynamic optimization of exception handling [Ogasawara et al. 2001], or control- and data-flow analysis in the presence of exceptions [Choi et al. 1999; Sinha and Harrold 2000]). This section presents an overview of human-oriented approaches, as machine-oriented approaches are outside the scope of our work.

Human-oriented static analysis for exception handling has been proposed for three languages with large user-bases: ML [Milner et al. 1990], a functional language; Ada-83 [ANSI 1983], an imperative language; Java, an object-oriented language.

The ML language represents exceptions as singular values that are not organized into a hierarchy and does not support exception interfaces [Lang and Stewart 1998]. These two characteristics make it difficult for programmers to ensure that all exceptions are caught. Pessaux and Leroy [1999] report that uncaught exceptions are the most frequent mode of failure in large ML applications. The goal of the analyses for ML is thus to help programmers identify the points in a program where different exceptions can be raised.

Guzmán and Suárez [1994] have proposed an extension of the ML type system by which it is possible to estimate all uncaught exceptions that can be raised. Their type system is limited in that it does not handle exceptions carrying arguments.

A different approach has been adopted by Yi [1998], who developed an exception analyzer based on abstract interpretation techniques [Cousot and Cousot 1977]. Since this analyzer suffered from performance problems, Yi and Ryu [1997] developed a more efficient one using control-flow analysis and a set-constraints framework [Heintze 1992].

Fähndrich et al. [1998] have built an Exception Analysis Tool (EAT) that allows the programmer to display uncaught exceptions in various scopes while browsing the corresponding source code. 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 [1999] propose a type-based analysis of uncaught exceptions in ML that offers different speed and precision tradeoffs than the previous constraint-based approaches. Most of the work on exception analysis for ML focuses on the technical difficulties and tradeoffs involved in the analysis, with little or no discussion of how the results can be applied in a software engineering context.

Ada exceptions share some of the characteristics of ML exceptions. In particular, they are defined according to a flat structure [Gauthier 1995], and the language does not support exception interfaces. Tools proposed for Ada include the work of Schaefer and Bundy [1993], and of Brennan [1993]. Similar to Jex, these tools calculate the *encounters* function for all scopes in a program. However, since Ada-83 is not an object-oriented language and since Ada exceptions are not organized hierarchically, their analysis does not need to consider dynamic method bindings or subsumption for handler selection. Furthermore, neither of these approaches provides complete information about the flow of predefined Ada exceptions. However, exceptions generated by language operations can play an important part in the behavior of a program under exceptional circumstance: Jex includes the propagation paths for predefined exceptions in its analysis. Additionally, the main focus of the work on Ada exception analysis was to detect specific defects in the code, such as unused handlers. Such detection relates to local reasoning in our terminology. The tool described by Brennan was originally meant to help systematically implement fault-tolerant Ada programs following a compartmenting approach proposed by Litke [1990]. We have not found any report of the application of this technique in Ada, but in earlier work, we reported on the use of Jex to evaluate the feasibility of the approach for improving the design of exception structure in Java systems [Robillard and Murphy 2000]. In comparison to the approaches for Ada, our work considers support for both local and system-wide reasoning about exceptions in object-oriented systems.

Tools and algorithms have also been developed to analyze various aspects of exception handling in Java. The first version of Jex [Robillard and Murphy 1999] was limited to version 1.0 of the Java language, and provided only a subset of the *generates* function for any scope. These shortcomings have been addressed in the version described in this article. Sinha and Harrold [1999] have described a framework and methodology for selecting unit and integration test cases for Java programs that contain exception-handling constructs. Ryder et al. [1999] have built the Java Exception Static Profiler (JESP), a suite of tools to extract statistical information about the frequency of the occurrence of exception-handling structures in Java programs. Ryu and Yi [2001] have proposed an algorithm for the exception analysis of an extension of the Java language that supports the propagation of exceptions across threads. Their language extension involves a new keyword for raising an exception in a different thread. The analysis involves first determining the expressions of a program that can be executed concurrently, and then analyzing the flow of exceptions between concurrent expressions. Their analysis of exception flow is based on the resolution of constraints on the type of expressions in a Java program. No implementation of this analysis was described in the work cited above, so the practical feasibility of the approach remains to be evaluated. In our work, because we support the Java language without special extensions, we do not distinguish thread-related exceptions from other asynchronous exceptions, which can theoretically be raised at any program point. For that reason, we do not consider asynchronous exceptions in our analysis.

Chang et al. [2001] have proposed an interprocedural exception analysis also based on the set-constraints framework. Their exception-flow analysis offers a refinement over the one performed by the Java compiler. This analysis determines more precisely the types of exceptions that can be propagated by a method, based on an analysis of the types of expressions in `throw` statements. The analysis, however, does not include the flow of unchecked exceptions, including exceptions generated by the Java virtual machine at run-time. This precludes examining many of the problems we discussed in this article, such as the unanticipated handling of unexpected exceptions. In contrast, the model we propose is intended to address all synchronous exceptions that can be raised in a Java program, and, to this effect, the Jex tool supports the analysis of both checked and unchecked exceptions.

Chang et al. [2002] have also reported on a visualizer they have created to show exception propagation in Java programs. Their visualizer consists of textual views in an integrated development environment that allow a developer to view, for a selected method, a list of uncaught checked exceptions for that method, and to see the propagation path of those exceptions from their origin. In contrast, our visualizer provides graphical views of the exception flow at different levels of structural detail, from methods, to packages, and user-specified components. Our intent is to allow a developer to investigate the overall exception flow in a system.

7. CONCLUSIONS

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 programming languages provide a means to enable software developers to build applications that avoid these problems. Building and evolving applications with appropriate error-handling strategies, though, requires support above and beyond that provided by a language's compiler or linker. To implement an appropriate strategy, a developer requires knowledge about how exceptions might flow through a system. Unfortunately, exception-handling mechanisms introduce a different control flow that can be difficult, if not impossible, to assess. In object-oriented languages where exceptions are defined as objects, the control flow of a program under exceptional circumstances is determined by the type of the exception objects that are raised.

In this article, we have shown the need for a developer to reason about exception flow when producing robust systems, and we have presented a model that encapsulates the minimal concepts necessary for a developer to understand exception flow for object-oriented languages that define exceptions as objects. Researchers interested in the use of exception-flow information for the purpose of software engineering tasks can build on this minimal model to describe techniques and methodologies that use exception-flow information.

Using the concepts of this exception-flow model, we have described why exception flow information is necessary to build and evolve robust programs. Exception-flow information can be used both at the level of guarded scopes, to detect and correct problems such as unused handlers, and at the global level, to improve exception-handling policies based on the knowledge of how certain exceptions can propagate through the system.

We have also described Jex, a static analysis tool we have developed to provide exception-flow information for Java systems. The Jex tool extracts information about the structure of exceptions in Java programs, providing a view of the actual exception types that might arise at different program points and of the handlers that are present. Use of this tool on a collection of Java library and application source code demonstrates that the approach can be helpful to support both local and global improvements to the exception-handling structure of a system.

The model of exception flow described in this article, and the corresponding challenges a developer faces when trying to assess exception flow, are based only on the characteristics of the general exception-handling mechanism targeted by our work, and do not integrate any specific programming language semantics. For this reason, both the model presented, the problems discussed, and the functionality for the exception-flow analysis tool described in this article are in theory applicable to any object-oriented programming language that define exceptions as objects. Furthermore, even if the implementation of an exception-flow analysis tool is inevitably language-specific, many of the challenges we faced in the implementation of Jex will have to be addressed for other languages. Researchers and practitioners wishing to develop exception-flow analysis technologies for other languages can thus build on our work to elicit the tool requirements, plan the design and implementation of the tool, and evaluate the results.

ACKNOWLEDGMENTS

We thank Lindsay Mason, who was instrumental in the development of the exception-flow visualizer, and Wayne Maddison, who generously contributed help and feedback during the analysis of the Mesquite system. We also thank Jonathan Sillito and the anonymous referees, for their thorough and insightful comments on this article.

REFERENCES

ABADI, M. AND CARDELLI, L. 1996. *A Theory of Objects*. Monographs in Computer Science, Springer-Verlag, New York, NY.

BACON, D. F. 1998. Fast and effective optimization of statically typed object-oriented languages. Ph.D. Thesis CSD-98-1017, (Oct.), University of California, Berkeley.

BRENNAN, P. T. 1993. Observations on program-wide Ada exception propagation. In *Proceedings of the Conference on TRI-Ada '93* (Sept.). ACM, 189–195.

BUHR, P. A. AND MOK, W. R. 2000. Advanced exception handling mechanisms. *IEEE Trans. Softw. Eng. 26*, 9 (Sept.), 820–836.

CALLAHAN, D., COOPER, K. D., KENNEDY, K., AND TORCZON, L. 1986. Interprocedural constant propagation. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction* (June). ACM, 152–161.

CHANG, B.-M., JO, J.-W., AND HER, S. H. 2002. Visualization of exception propagation for Java using static analysis. In *Proceedings of the Second International Workshop on Source Code Analysis and Manipulation* (Oct.). IEEE , CA, 173–182.

CHANG, B.-M., JO, J.-W., YI, K., AND CHOE, K.-M. 2001. Interprocedural exception analysis for Java. In *Proceedings of the 2001 ACM Symposium on Applied Computing* (March). ACM, 620–625.

CHOI, J.-D., GROVE, D., HIND, M., AND SARKAR, V. 1999. Efficient and precise modeling of exceptions for the analysis of Java programs. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (Sept.). ACM, 21–31.

COUSOT, P. AND COUSOT, R. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Jan.). ACM, 238–252.

DEAN, J., GROVE, D., AND CHAMBERS, C. 1995. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the Ninth European Conference on Object-oriented Programming*. Lecture Notes in Computer Science (Aug.), vol. 952. Springer-Verlag, 77–101.

DONY, C. 1990. Exception handling and object-oriented programming: towards a synthesis. In *Proceedings of the Conference on Object-oriented Programming Systems, Languages, and Applications, and of the European Conference on Object-oriented programming* (Oct.). ACM, 322–330.

FAHNDRICH, M., FOSTER, J., CU, J., AND AIKEN, A. 1998. Tracking down exceptions in standard ML programs. Tech. Rep. CSD-98-996, Feb. University of California, Berkeley.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design Patterns—Elements of Reusable Object-Oriented Software*. Addison Wesley Longman, Inc., Reading, MA.

GAUTHIER, M. 1995. Exception handling in Ada-94: Initial users' requests and final features. *ACM Ada Letters*, *XV*, 1 (Jan./Feb.), 70–82.

GOODENOUGH, J. B. 1975. Exception handling: Issues and proposed notation. *Commun. ACM*, *18*, 12 (Dec.), 683–696.

GOSLING, J., JOY, B., AND STEELE, G. 1996. *The Java Language Specification*. Addison Wesley. Longman, Inc., Reading, MA.

GUZMÁN, J. C. AND SUÁREZ, A. 1994. A type system for exceptions. In *Proceedings of the 1994 ACM SIGPLAN Workshop on ML and Its Applications* (June). ACM, 127–135.

HALBERT, D. C. AND O'BRIEN, P. D. 1987. Using types and inheritance in object-oriented programming. *IEEE Softw. 4*, 5 (Sep.), 71–79.

HEINTZE, N. 1992. Set-based program analysis. Ph.D. Thesis (Oct.), Carnegie-Mellon University, Pittsburgh, PA.

LANG, J. AND STEWART, D. B. 1998. A study of the applicability of existing exception-handling techniques to component-based real-time software technology. *ACM Trans. Program. Lang. Syst. 20*, 2 (Mar.), 274–301.

LISKOV, B. H. AND SNYDER, A. 1979. Exception handling in CLU. *IEEE Trans. Softw. Eng. 5*, 6 (Nov.), 546–558.

Liskov, B. H. and Wing, J. M. 1994. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst. 16*, 6 (Nov.), 1811–1841.

Litke, J. D. 1990. A systematic approach for implementing fault tolerant software designs in Ada. In *Proceedings of the Conference on TRI-ADA '90* (Dec.). ACM, 403–408.

Maddison, W. and Maddison, D. 2001. Mesquite: A modular system for evolutionary analysis. `http://mesquiteproject.org`.

Miller, R. and Tripathi, A. 1997. Issues with exception handling in object-oriented systems. In *Proceedings of the 11th European Conference on Object-Oriented Programming*. Lecture Notes in Computer Science (June), vol. 1241. Springer-Verlag, New York, NY, 85–103.

Milner, R., Tofte, M., and Harper, R. W. 1990. *The Definition of Standard ML*. MIT Press, Cambridge, MA.

Ogasawara, T., Komatsu, H., and Nakatani, T. 2001. A study of exception handling and its dynamic optimization in Java. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications* (Oct.). ACM, 83–95.

Palsberg, J. and Schwartzbach, M. I. 1991. Object-oriented type inference. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications* (Oct.). ACM, 146–161.

Pessaux, F. and Leroy, X. 1999. Type-based analysis of uncaught exceptions. In *Proceedings of the 26th Symposium on the Principles of Programming Languages* (Jan.). ACM, 276–290.

Plevyak, J. and Chien, A. A. 1994. Precise concrete type inference for object-oriented languages. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications* (Oct.). ACM, 324–340.

Robillard, M. P. and Murphy, G. C. 1999. Analyzing exception flow in Java programs. In *Proceedings of the Seventh European Software Engineering Conference and Seventh ACM SIGSOFT Symposium on the Foundations of Software Engineering*. Lecture Notes in Computer Science (Sept.) vol. 1687. Springer-Verlag, New York, NY, 322–337.

Robillard, M. P. and Murphy, G. C. 2000. Designing robust Java programs with exceptions. In *Proceedings of the 8th ACM SIGSOFT International Symposium on the Foundations of Software Engineering* (Nov.). ACM Press, New York, NY, 2–10.

Romanovsky, A. and Sandén, B. 2001. Except for exception handling. *Ada Letters 21*, 3 (Sept.), 19–25 (*Proceedings of the Workshop on Exception Handling for a 21st Century Programming Language*).

Ryder, B. G., Smith, D., Kremer, U., Gordon, M., and Shah, N. 1999. A static study of Java exceptions using JESP. Tech. Rep. DSC-TR-406 (Oct.). Department of Computer Science, Rutgers University.

Ryu, S. and Yi, K. 2001. Exception analysis for multithreaded Java programs. In *Proceedings of the Second Asia-Pacific Conference on Quality Software* (Dec.). IEEE Computer Society Press, Los Alamitos, CA, 23–30.

Schaefer, C. F. and Bundy, G. N. 1993. Static analysis of exception handling in Ada. *Softw. Pract. Exper. 23*, 10 (Oct.), 1157–1174.

Scott, M. L. 2000. *Programming Language Pragmatics*. Morgan Kaufmann Publishers, San Francisco, CA.

Shelekhov, V. I. and Kuksenko, S. V. 1999. Data flow analysis of Java programs in the presence of exceptions. In *Proceedings of the third International Andrei Ershov Memorial Conference on Perspectives of System Informatics*. Lecture Notes in Computer Science (July), vol. 1755. Springer-Verlag, New York, NY, 389–396.

Sinha, S. and Harrold, M. J. 1999. Criteria for testing exception-handling constructs in Java programs. In *Proceedings of the International Conference on Software Maintenance* (Sept.). IEEE Computer Society Press, Los Alamitos, CA, 265–274.

Sinha, S. and Harrold, M. J. 2000. Analysis and testing of programs with exception handling constructs. *IEEE Trans. on Softw. Eng. 26*, 9 (Sept.), 849–871.

Stroustrup, B. 1991. *The C++ Programming Language*, 2nd ed. Addison Wesley Longman, Inc., Reading, MA.

U.S. Department of Defense 1983. *Reference Manual for the Ada Programming Language, ANSI/Military Standard MIL-STD-1815A-1983*. U.S. Department of Defense.

YEMINI, S. AND BERRY, D. M. 1985. A modular verifiable exception-handling mechanism. *ACM Trans. Program. Lang. Syst. 7*, 2 (Apr.), 214–243.

YI, K. 1998. An abstract interpretation for estimating uncaught exceptions in standard ML programs. *Sci. Comp. Program. 31*, 1 (May), 147–173.

YI, K. AND RYU, S. 1997. Towards a cost-effective estimation of uncaught exceptions in SML programs. In *Proceedings of the 4th International Static Analysis Symposium*. Lecture Notes in Computer Science (Sept.), vol. 1302. Springer-Verlag, 98–113.

Received April 2002; revised November 2002; accepted July 2003