Hindawi Publishing Corporation
Advances in Software Engineering
Volume 2014, Article ID 197983, 16 pages
http://dx.doi.org/10.1155/2014/197983

Research Article

Mutation Analysis Approach to Develop Reliable Object-Oriented Software

Monalisa Sarma

Indian Institute of Technology Kharagpur, Kharagpur 721302, India

Correspondence should be addressed to Monalisa Sarma; monalisa@iitkgp.ac.in

Received 4 April 2014; Revised 19 August 2014; Accepted 22 September 2014; Published 25 December 2014

Academic Editor: Henry Muccini

In general, modern programs are large and complex and it is essential that they should be highly reliable in applications. In order to develop highly reliable software, Java programming language developer provides a rich set of exceptions and exception handling mechanisms. Exception handling mechanisms are intended to help developers build robust programs. Given a program with exception handling constructs, for an effective testing, we are to detect whether all possible exceptions are raised and caught or not. However, complex exception handling constructs make it tedious to trace which exceptions are handled and where and which exceptions are passed on. In this paper, we address this problem and propose a mutation analysis approach to develop reliable object-oriented programs. We have applied a number of mutation operators to create a large set of mutant programs with different type of faults. We then generate test cases and test data to uncover exception related faults. The test suite so obtained is applied to the mutant programs measuring the mutation score and hence verifying whether mutant programs are effective or not. We have tested our approach with a number of case studies to substantiate the efficacy of the proposed mutation analysis technique.

1. Introduction

Of late, computer applications have permeated heavily into every sphere of the daily life of an average person. Many of these applications are large, complex, and safety critical, requiring the software to be extremely reliable. An exception occurrence during a program execution is an abnormal computation state [1] and thus is a threat to the reliability of software. It is advocated that software should be tested adequately to trace if there is any exception in intended behavior [2]. To realize this software developers prefer exception handling mechanisms to be embedded inside code so that software can arrest and raise alarm whenever there is any exception and then cause of exception can be eliminated. Many reasons may be attributed to causes of exceptions such as erroneous input, hardware faults, logical errors in the code, semantic violation, linking errors, and limitation of system resources. It is reported that more than 50% of the operational failures that may occur in a system are due to faults in exception handling and recovery [3]. To cope with the system failures due to exceptions, most modern programming languages such as Ada, C++ and Java incorporate explicit mechanisms for exception handling. Syntactically, an exception handling mechanism consists of a means to explicitly raise an exceptional condition at a program point and a means of expressing a block of code to handle one or more exceptional conditions. In other words, an exception handling mechanism provides a way to separate code that deals with unusual situations and abnormal program termination.

Exception handling constructs are responsible for the detection and handling of system conditions that could potentially lead to failure. Recent studies of Java programs indicate that exception handling constructs occur frequently and approximately 15%–25% of the classes contained exception handling constructs [4, 5]. Despite the frequency of their occurrences, the behavior of exception handling constructs is often the least understood and poorly tested part of a program [6]. Testing strategy to test object-oriented programs, in general, and exception handling constructs, in particular, is scarcely reported [7]. For testing exception handling constructs, some works have been reported [4, 6, 8]. Sinha and Harrold [9] proposed a class of adequacy criteria that can be used to test the behavior of exception handling constructs. Sinha and Harrold [9] described a methodology for applying the criteria to unit and integration testing of programs that contain exception handling constructs. However, to the best of our literature survey, no approach has been reported so far to generate exception handling related faulty programs and then to test the adequacy of test suites. Adequacy testing is an important issue because a test suite may not locate all exception points. For example, in Java, an exception can be raised implicitly. In that case, it is not possible to draw proper control flow graph [4, 10, 11]. Further, even if all exception paths are known from the control flow graph [4] and satisfy exception-coverage criteria [9], it needs to design test data manually for raising all possible potential exceptions. This manual test case generation is not only tedious [6] but raising exceptions is not always guaranteed. As a consequence we need to determine the adequacy of test cases to raise exceptions. In general, determining the path of exceptions is a difficult problem and manual synthesis of test data may not be sufficient to raise all exceptions. In this context, efficiency of a test suite is necessary to be evaluated in order to assure the reliability of a system under test. In summary, given a program, we are to test whether the exception handling constructs are sufficient to throw and catch all possible exceptions or not.

Mutation analysis is a practical method to evaluate the quality of test suites [12–14]. In this work, we propose a mutation analysis method to evaluate test suites for testing exception handling constructs. We consider a set of mutation operators with which different types of faults can be injected to the original programs. The mutation operators are adopted from the work of Ji et al. [15]. The mutant programs are then analyzed with some test suites. We consider test suites generated according to the control flow based structural analysis approach proposed in [4, 9] for a given Java program. We also analyzed the program in JUnit 4.6 [16] and Jumble 1.0 [17] testing environment and their mutant programs. Our experimental results substantiate the efficiency of the proposed mutation analysis approach.

The rest of the paper is organized as follows. We discuss basic concepts, related terminologies, and tools in Section 2. In Section 3, we discuss the mutation operators followed in this work. Mutation analysis is discussed in Section 4. Our experimental results are presented in Section 5. Related work and comparison of our work with others are discussed in Section 6. Finally, Section 7 concludes the paper.

2. Basic Concepts

In this section, we discuss some basic concepts and terminologies which are referred to in our subsequent discussions. We also briefly introduce the tools used in our work.

2.1. Exception Handling Constructs in Java. Java [18] programming language is considered as a model language in this work. In this subsection, we provide an overview of exception handling constructs in Java. Details of exception handling constructs in Java can be found in [19, 20].

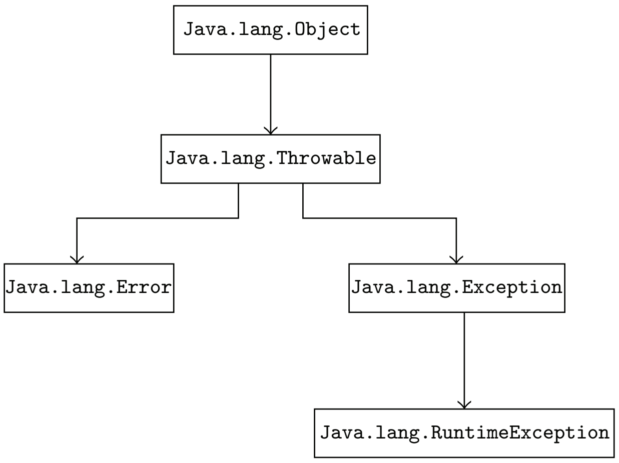
In Java, exceptions are first-class objects. Each exception is an instance of a class that is derived from the class `java.lang.Throwable`, which is defined in the standard Java API [19]. An exception can be raised at any point in the program through a `throw` statement. A `throw` statement is associated with an expression denoting an exception object, which may be a variable (e.g., `throw e`), a method call (e.g., `throw m()`) or a new instantiation (e.g., `throw new E()`), and so forth. A `throw` statement can appear anywhere in the program. Java exception handling construct includes three blocks: `try`, `catch` and `finally` (see Figure 1(a)). Code can be guarded for exceptions within a `try` block. Exceptions raised through execution of code (implicitly) or a `throw` statement (explicitly) within a `try` block may be caught in one or more `catch` clause(s) declared immediately following the `try` block. The `finally` block is executed independently of what happens in the `try` block. It may be noted that a `try` statement consists of a `try` block and optionally one or more `catch` blocks and `finally` block (see Figure 1(a)). The valid instances of a `try` statement are `try`{...} `catch` (...){...}[`catch` (...){...}] [...] [`finally`{...}], where the blocks in square brackets indicate optional. Here, a `try` block contains statement whose execution is monitored for exception occurrences. A `catch` block specifies the type of exception it handles and *exception handler*; that is, it contains a block of code that is executed when an exception of that type is raised in the associated `try` block. If a `finally` block is associated with a `try` block, then the `finally` block is always executed, irrespective of how control transfers out of the `try` block.

2.2. Java Exception Hierarchy. Similar to other Java objects, exceptions are typed, and types are organized into a hierarchy. All exceptions inherit from the class type `Throwable` defined in `java.lang API`. The exception type hierarchy defines three different groups of exceptions: *errors*, *runtime exceptions,* and *checked exception* (defined in `Java.lang.Exception`) (also see Figure 1(b)). Note that errors and runtime exceptions are unchecked exceptions. Unchecked exceptions can be thrown at any point in a Java program and, if uncaught, may propagate back to the program entry point, causing the Java Virtual Machine [20] to terminate. The class `Error` is responsible for giving errors in some catastrophic failures. The exceptions of type `RuntimeException` are typically created automatically during runtime in response to some execution errors. The exceptions of the type `Exception` are checked exceptions and are used for exceptional conditions that user programs can catch. A list of exceptions that a Java programmer can catch in programs is available in [19]. There are 11 runtime exceptions, 17 errors, and 7 checked exceptions defined in `Java.lang` API [19] as per the release of Java 6. Programmers can define and extend existing exception type in Java API to customize their needs. Some new exception types can also be defined [20].

2.3. Call Exception Hierarchy. In this subsection, we discuss the flow of exceptions within the context of the structure of a given program. This flow of exceptions, we specially term it as *call exception hierarchy (CEH),* is useful to understand and evaluate how exceptions are handled within a method, which exceptions might arise during the execution of a method,

```
try {
    ... //Guarded block,
    ... //where exception can
    ... //be raised
  }
  catch (Exception1 e1) {
     ... //Exception handler for
     ... //exception type Exception1
  }
  catch (Exception2 e2) {
     ... //Exception handler for
     ... //exception type Exception2
  }
  catch (Exceptionm em ) {
     ... //Exception handler for
     ... //exception type Exceptionm
  }
  finally {
     ... //Must executable code
  }
```

(a) Exception handling constructs in Java



(b) Exception class hierarchy in Java

Figure 1: Exception handling constructs and exceptions in Java [19].

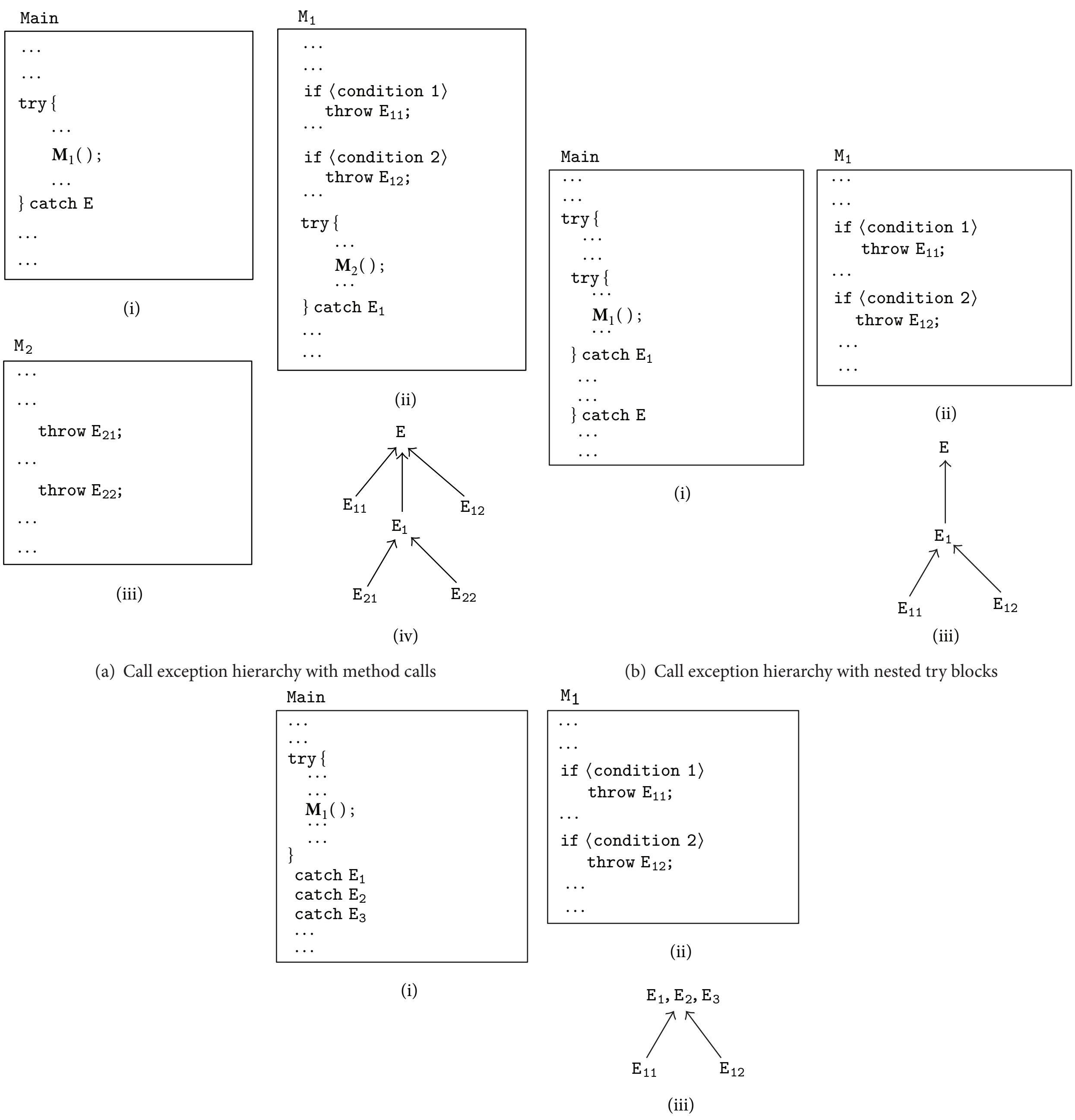
which exceptions are handled and where, and which exceptions are passed on. Alternatively, a call exception hierarchy specifies which exceptions may propagate to its caller. In essence, the call exception hierarchy is precisely specifying the behavior of the program as far as the exceptions and handling exceptions are concerned [21].

We identify three basic control flow constructs to obtain the call exception hierarchy in any program. These are method calls, nesting, and multicatch. Any complex and arbitrary call exception hierarchy can be resolved with the help of the above three basic constructs. We discuss the three basic constructs as follows.

2.3.1. Method Calls. A method say `Main` can call a method m_i, which in turn may call another method say m_j and so on. Any method can throw and catch exceptions. The call exception hierarchy, in this case, implies the parent-child relationship among exceptions. An exception E_2 is a child of another exception, say E_1, if E_2 occurs in a method that is called by the method where E_1 may occur. The call exception hierarchy with method calls is illustrated with an example in Figure 2(a). In Figure 2(a), the method `Main` calls the method $M_1()$ and is supposed to catch an exception E. Now, the method $M_1()$ calls method $M_2()$ and may throw two exceptions, namely, E_{11} and E_{12}. The method $M_2()$ in turns may throw exception E_{21} and E_{22}. In this case, the exception E should be at the root of the hierarchy because E is supposed to represent all instances of exceptions when the invocation of the method M_1 occurs. The exceptions E_{11} and E_{12} are E's children because they are instances, which may propagate to E. Similarly, E_1 is a child of E and E_{21}, E_{22} are the children of E_1. Note that here the exceptions E_{21} and E_{22} are supposed to be caught through the exception E_1, if it fails then through E. The call exception hierarchy for a situation of method calls in Figures 2(a)(i)–2(a)(iii) is shown in Figure 2(a)(iv).

2.3.2. Nested `try` Blocks. In Java, a `try` block can be nested within another `try` block and so on. In this case, the call exception hierarchy can be obtained as discussed below. All exceptions at outer `try` block are at the root. All exceptions at the next inner `try` blocks are the children of the root. Any exceptions in a `try` block at the next level are the children of exceptions at its upper `try` level and so on. We illustrate such a situation with an example. We consider a two-level nested `try` structure as shown in Figure 2(b). In Figure 2(b)(i) the method `Main is` supposed to catch the exception E, which is at the outermost `try` block. Now, `Main` calls method $M_1()$ at the innermost `try` block. The method $M_1()$ may raise two exceptions, namely, E_{11} and E_{12}. E_{11} and E_{12} would be the successor of E_1, which is supposed to catch these two exceptions. In turn E_1 would be the successor of E. The call exception hierarchy of the program structure of Figures 2(b)(i) and 2(b)(ii) is shown in Figure 2(b)(iii).

2.3.3. Multiple `catch` Blocks. There may be multiple `catch` blocks corresponding to a `try` block [19, 20]. In this case, all exceptions, which are supposed to be caught, would be placed as a single node and all exceptions, which are raised by a *called* method inside the `try` block, would be the children of this node. As an example, let us consider a simple `try` with multiple `catch` as shown in Figure 2(c). In Figure 2(c)(i), the method `Main` contains a `try` block and this `try` block is associated with three parallel `catch` blocks. These three `catch` blocks are corresponding to handle three different exceptions, namely, E_1, E_2, and E_3. Further, the method `Main()` calls the method $M_1()$, which is supposed to throw the exceptions E_{11} and E_{12} (see Figure 2(c)(ii)). In this case, the exceptions E_1, E_2, and E_3 of the *caller* method `Main` is at the root and the exceptions E_{11} and E_{12} of the *called* method $M_1()$ are the children of the exceptions E_1, E_2, and



(a) (i) Main
```
...
...
try {
    ...
    M1();
    ...
} catch E
...
...
```

(a) (ii) M1
```
...
...
if ⟨condition 1⟩
   throw E11;
...
if ⟨condition 2⟩
   throw E12;
...
try {
     ...
     M2();
     ...
} catch E1
...
...
```

(a) (iii) M2
```
...
...
  throw E21;
...
  throw E22;
...
...
```

(a) (iv)

$$\begin{tikzcd} & E & \\ E_{11} \arrow[ur] & E_1 \arrow[u] & E_{12} \arrow[ul] \\ E_{21} \arrow[ur] & & E_{22} \arrow[ul] \end{tikzcd}$$

(a) Call exception hierarchy with method calls

(b) (i) Main
```
...
...
try {
  ...
  ...
 try {
   ...
   M1();
   ...
 } catch E1
  ...
  ...
 } catch E
  ...
  ...
```

(b) (ii) M1
```
...
...
if ⟨condition 1⟩
   throw E11;
...
if ⟨condition 2⟩
   throw E12;
 ...
 ...
```

(b) (iii)

$$\begin{tikzcd} & E & \\ & E_1 \arrow[u] & \\ E_{11} \arrow[ur] & & E_{12} \arrow[ul] \end{tikzcd}$$

(b) Call exception hierarchy with nested try blocks

(c) (i) Main
```
...
...
try {
   ...
   ...
   M1();
   ...
   ...
}
 catch E1
 catch E2
 catch E3
 ...
 ...
```

(c) (ii) M1
```
...
...
if ⟨condition 1⟩
   throw E11;
...
if ⟨condition 2⟩
   throw E12;
 ...
 ...
```

(c) (iii)

$$\begin{tikzcd} & E_1, E_2, E_3 & \\ E_{11} \arrow[ur] & & E_{12} \arrow[ul] \end{tikzcd}$$

(c) Call exception hierarchy with multiple catch blocks

Figure 2: Basic constructs of call exception hierarchy.

E_3. The call exception hierarchy for this case (Figures 2(c)(i) and 2(c)(ii)) is shown in Figure 2(c)(iii).

Related to the call exception hierarchy, we define the following terminologies.

Caller Exception. An exception E in a CEH for a program P is called the caller exception if it is a parent of any other exception $E_i \in CEH$. In other words, all exceptions except at the leaf of call exception hierarchy are the *caller* exceptions.

Callee Exception. An exception E in a CEH for a program P is called the *callee* exception if it is a child of any other exception $E_i \in CEH$. In other words, all exceptions except the exception(s) at the root of a call exception hierarchy are *callee* exceptions.

Direct Caller Exception. In CEH, if an exception E_i is the immediate predecessor of another exception E_j, then the exception E_i is called the direct caller exception.

Direct Callee Exception. In CEH, if an exception E_i is the immediate successor of another exception E_j, then the exception E_i is called the direct callee exception.

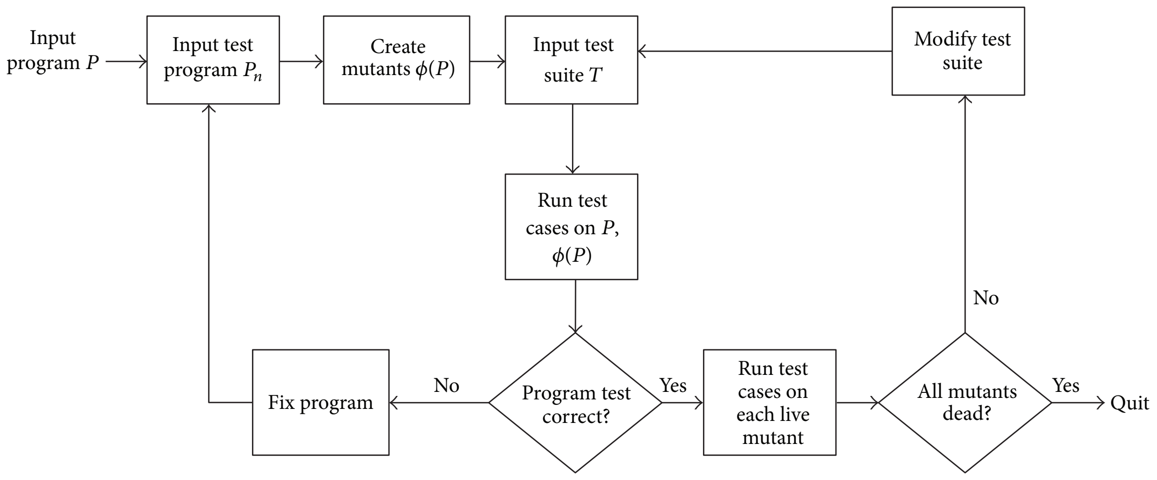


Figure 3: Mutation testing framework.

2.4. Mutation Testing. Mutation testing [22] is a widely accepted and practical method to evaluate the quality of a test suite to test programs. Mutation testing is a fault-based testing adequacy criterion proposed by DeMillo et al. [23] initially with the name *mutation analysis*. Given a program P, a set of alternative programs, say $\phi(P)$, is considered in order to measure the test adequacy of a test suite T. A test set T is adequate to P in relation to $\phi(P)$ if for each program $Q \in \phi(P)$, either Q is equivalent to P or Q differs from P on at least one test case $t \in T$ [24]. A mutation testing framework is shown in Figure 3. In the context of mutation analysis some commonly known terminologies are mentioned below.

2.4.1. Mutant. A set of alternative programs $\phi(P)$ is called mutants of P. A mutant program $Q \in \phi(P)$ is different from P only on some simple syntactic changes. This is also alternatively called fault injection to P.

2.4.2. Mutation Operator. Mutation operator is a set of rules to decide syntactic changes in the program P to obtain mutants $\phi(P)$.

2.4.3. Mutant Dead. To assess the adequacy of a test set T, each mutant $Q \in \phi(P)$, as well as the program P, has to be executed against the test cases in T. If the observed output of the mutant Q is the same as that of P for all test cases in T, then Q is considered alive; otherwise it is considered dead or distinguished.

2.4.4. Equivalent Mutant. Let D be the input domain of the test suite T. A mutant Q is called an equivalent mutant if $\forall x \in D$, P, and Q produce the same results. Note that an equivalent mutant cannot be distinguished and should be discarded from the mutant set as it does not contribute to the adequacy of T.

2.4.5. Mutation Score. Test adequacy is measured by the mutation score $\mu(P, T)$ as defined in

$$\mu(P, T) = \frac{\text{DM}}{\text{TM} - \text{EM}}, \qquad (1)$$

where DM denotes the number of dead mutants of P for a test suite T and TM and EM are the total number of mutants in $\phi(P)$ and equivalent mutants, respectively. In summary, a low value of the mutation score implies either test suite unable to cover sufficient faults or there is less faults and vice versa.

2.5. Java Program Testing Tools. Of late, Java programming has been extensively practiced in industries. A number of Java tools are available to support the Java practitioners. Two commonly used tools are JUnit [16, 25] and Jumble [17, 26]. These two tools are used in this work. We introduce these tools briefly in the following.

2.5.1. JUnit. JUnit [16] is an API framework and is used to automate unit and regression testing. Using JUnit, an entire class, part of a class, that is, a method or some interacting methods and interactions among several objects, can be tested. To test using JUnit, a tester class is required. The tester class contains more than one test case where each test case is written into one test method [25]. The tester class also includes a test executor to run the test cases and methods to set up the state(s) of object(s) before and update the state(s) after the execution of each test case.

The advantages with JUnit are that it allows assertions for testing expected results, it has test features for sharing common data, and it helps to isolate and localize errors. The limitation with JUnit is that developers have to write test methods themselves, which requires a substantial amount of coding effort. Further, JUnit is designed to call methods and compare the results they return against expected results. This works for methods that return some results, but many methods instead of returning values have side effects, such as displaying output, modifying states of constituent objects, and modifying attributes' values. Also, JUnit cannot directly test private methods and not suitable for system or integration testing.

2.5.2. Jumble. Jumble [17] is a mutation testing tool for Java programs, which interoperates with JUnit. Jumble takes a class to be mutation tested and a set of JUnit test classes as parameters. The output is the overall score of how many mutations are successfully caught as well as details about those mutations which are not. Jumble performs its mutations by directly modifying Java byte code using BCEL package [26]. This allows the mutation testing to be done with no compilations.

The advantages with Jumble are that it mutates a program at the byte code level; hence, Jumble is faster and it can be used to test the programs even when the source code is not available. Moreover, Jumble uses a heuristic to improve its performance. Using the heuristic, it determines the order in which to run the test cases so that a test fails at a faster rate. It ensures that the longer test cases run only when there is no shorter test case to cover mutations.

The major limitation with Jumble is that it only provides mutations operators for conditional, arithmetic, initialization and assignment, expressions, return values, and switch statements [26]. Jumble cannot discriminate `boolean`, `short`, and `char` data types because all these data types in byte code is represented as 32 bits `integer`. Further, Jumble uses strong mutation [26]. Weak mutation is not possible with Jumble because if JUnit test case uses a value that causes a mutated expression to return a different value, there is no guarantee that the JUnit test case will detect that different value.

3. Mutation Operators

We consider eight mutation operators to create mutant programs [15]. These mutation operators are listed in Table 1. There are two types of mutation operators: related to `catch` block and `try` block. These mutation operators are discussed in the following subsections. First, we discuss the proposed mutation operators to change in `catch` blocks. We then discuss the mutation operators related to alteration in `try` blocks. We refer to *CEH* as the call exception hierarchy of a Java program *P* in our discussions.

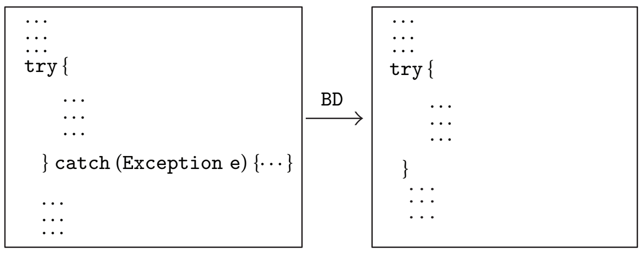
3.1. BD: Deletion of `catch` *Block.* The mutation operator *BD* is proposed to delete a `catch` block, say *CB,* following a `try` block, say *TB*. Deleting a `catch` block $< S2 >$ using the mutation operator *BD* can be formally stated as

$$BD \Rightarrow [TB:]\texttt{try}\{\ldots\}[CB:]\texttt{catch}(E)\{\ldots\} \rightarrow [TB:]\texttt{try}\{\ldots\}.$$

Figure 4 explains the operation of the mutation operator *BD*. If there are more than one `catch` block, then it can delete in an arbitrary way one or more `catch` block(s). The objective of this mutation operator is to alter the exception propagation path. When an exception, say *E*, is thrown in *TB*, the mutant program with *BD* should not handle the exception *E* appropriately because of the absence of *CB*. As a consequence, the exception raised at *TB* should be back propagated to the caller exception, if any in the *CEH*. The exception *E* then may be handled at an upper level if the exception type of *E* matches with some other exception at upper level. It may be noted that the original program and a mutant version may cause same exception meeting some reachability at the `try` block *TB*. In that case, deleting the whole `catch` block *CB* would not produce any distinctions between the two programs for a test case. Hence, this operator may produce equivalent mutants.

Table 1: Exception handling related mutation operators.

Operator	Description
BD	Catch block deletion
BR	Catch block replacement
BI	Catch block insertion
RE	Catch and rethrow exception
TD	Deletion in try block
TR	Replacement in try block
TI	Insertion in try block
TA	Alter try block


```
...
...
...
try {
    ...
    ...
    ...
  } catch (Exception e) {...}
  ...
  ...
  ...
```

```
...
...
...
try {
    ...
    ...
    ...
  }
  ...
  ...
```

Figure 4: Mutation operator *BD*.

3.2. BR: Replacement of `catch` *Block.* The mutation operator *BR* is proposed to cover faults such that a program does not handle exception correctly. Let *CB* be the `catch` block to be altered. The mutation operator *BR* attempts to deviate from the exception propagation path. The mutation operator *BR* creates mutant program(s) depending on the level of the occurrence of the `catch` block *CB*. Also, see Figure 5, which states the three different levels of *CBs* and their possible mutants. Three rules are to be followed.

Rule 1. If the *CB* is at the *root node* in the *CEH*, then *BR* replaces the exception parameter type of *CB* with that of *DCE*, where *DCE* denotes the *direct callee exception* of the exception of *CB*.

Rule 2. If the *CB* is at any *leaf* node in the *CEH*, then *BR* replaces the exception parameter type of *CB* with that of *DPE*, where *DPE* denotes the *direct caller exception* of the exception of *CB*.

Rule 3. If the *CB* is at an intermediate level, that is, neither at the root nor at leaf of the *CEH*, then *CBR* replicates the `catch` block *CB* into two blocks *CB1* and *CB2*, such that *BR* replaces

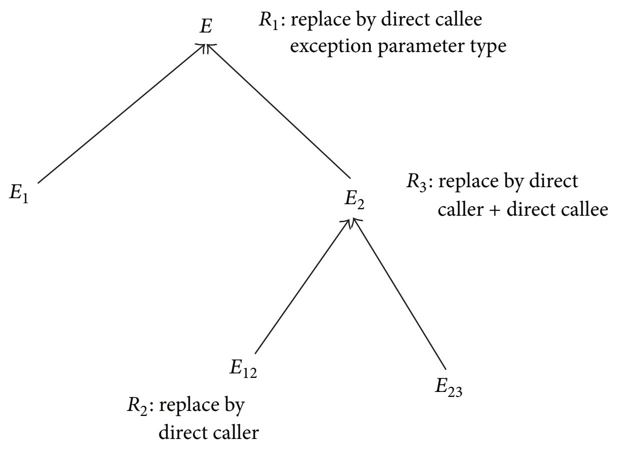

E
R_1: replace by direct callee exception parameter type
E_1
E_2
R_3: replace by direct caller + direct callee
E_{12}
E_{23}
R_2: replace by direct caller

FIGURE 5: Mutation operator *BR*.

the exception parameter type of *CB* with that of *DCE* and *DPE* as *CB1* and *CB2*, respectively. Here, *DPE* and *DCE* denote the *direct caller* and *direct callee* exceptions of the exception type of *CB* in *CEH*, respectively.

The mutation operator *BR* can be formally stated as follows.

Rule 1. If *CB* is at the root of *CEH*

$$BR \Rightarrow [TB:]\texttt{try}\{\ldots\}[CB:]\texttt{catch}(E)\{\ldots\} \rightarrow [TB:]\texttt{try}\{\ldots\}[CB:]\texttt{catch}(E')\{\ldots\}$$

where E' is the *DCE* of *E*.

Rule 2. If *CB* is any leaf node in *CEH*

$$BR \Rightarrow [TB:]\texttt{try}\{\ldots\}[CB:]\texttt{catch}(E)\{\ldots\} \rightarrow [TB:]\texttt{try}\{\ldots\}[CB:]\texttt{catch}(E')\{\ldots\}$$

where E' is the *DPE* of *E*.

Rule 3. If *CB* is any node other than root and leaf node in *CEH*

$$BR \Rightarrow [TB:]\texttt{try}\{\ldots\}[CB:]\texttt{catch}(E)\{\ldots\} \rightarrow [TB:]\texttt{try}\{\ldots\}[CB1:]\texttt{catch}(E')\{\ldots\}[CB2:]\texttt{catch}(E'')\{\ldots\}$$

where E' and E'' are the *DCE* and *DPE* of *E*, respectively.

3.3. BI: Insertion in `catch` *Block.* The purpose of the mutation operator `catch` block insertion *BI* is to catch all possible exceptions that might arise at a `try` block *TB*. In other words, if *CB* is the `catch` block associated with the `try` block *TB*, then *BI* tries to create mutant programs, which hide all exceptions in the `try` block *TB*. The mutation operator *BI* inserts catch blocks $CB_1, CB_2, \ldots, CB_n$ for all the exception parameter types $E_1, E_2, \ldots, E_n$ where $E_1, E_2, \ldots, E_n$ are the callee exceptions in the *CEH* for the exception at the `try` block *TB*.

TABLE 2: Mutation operator *TD*.

Operator	Description	Level
AOD	Arithmetic operator deletion	
COD	Conditional operator deletion	Method-level
LOD	Logical operator deletion	
IHD	Information hiding deletion	
IOD	Overriding method deletion	
ISD	Super keyword deletion	
OMD	Overloading method deletion	Class-level
JTD	This keyword deletion	
JSD	Static modifier deletion	
JID	Member variable initialization deletion	

It can be formally stated as

$$BI \Rightarrow [TB:]\texttt{try}\{\ldots\}[CB:]\texttt{catch}(E)\{\ldots\} \rightarrow [TB:]\texttt{try}\{\ldots\}[CB_i]\texttt{catch}(E_i)\{\ldots\}$$

where CB_i is a catch block with exception E_i for each $E_i \in CEH$.

The operation of *BI* mutation operator is depicted in Figure 6. Since the mutation operator *BI* hides some exceptions to propagate upward, it may produce equivalent mutants.

3.4. RE: Catch and Rethrow Exception. A situation that may arise in a program is that exception raised in a `try` block may not be propagated down the exception handling stack. To cover such a fault, the mutation operator called *catch and rethrow exception* and denoted by *RE* is proposed. The mutation operator *RE* for a `catch` block, say *CB,* associated with a `try` block, say *TB*, catches the exception type *E* of *CB* and rethrows the exception *E* inside the *CB* (see Figure 7, the operation of *RE*).

Rethrowing an exception using the mutation operator *RE* can be formally stated as

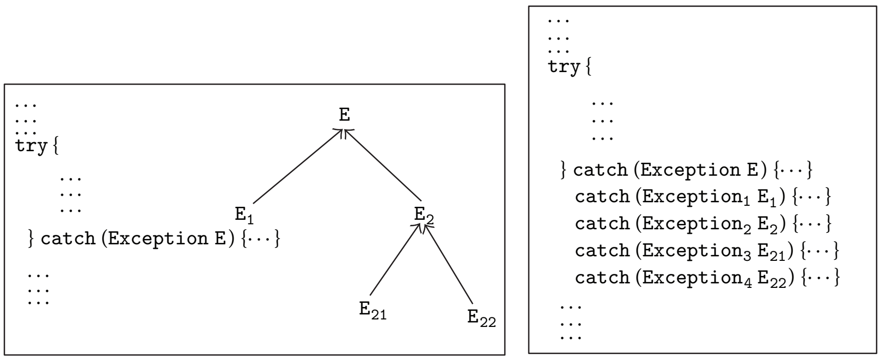
$$RE \Rightarrow [TB:]\texttt{try}\{\ldots\}[CB:]\texttt{catch}(E)\{\ldots\} \rightarrow [TB:]\texttt{try}\{\ldots\}[CB:]\texttt{catch}(E)\{\ldots; throw E_i; \ldots\}$$

for all $E_i \in CEH$.

Let *E* be the exception type at the `catch` block *CB*. Also, let $E_1, E_2, \ldots, E_n$ be the callee exceptions of the exception *E* according to call exception hierarchy *CEH*. The mutation operator *RE* creates the mutant programs by inserting a throw statement to throw the exception E_i for each $E_i \in E, E_1, E_2, \ldots, E_n$. Note the difference between *BI* and *RE*. In *BI*, a `catch` block is inserted with an exception, whereas, in *RE*, it inserts a throw statement in the `catch` block to throw exception of callee exception type of the exception at `catch` block including itself.

3.5. TD: Deletion in `try` *Blocks.* The mutation operator *TD* consists of two types of operators: method-level and class-level mutation operators. Concepts of these operators are taken from [27]. These operators are listed in Table 2.

The first three mutation operators in Table 2 are the method-level mutation operators and they modify Java

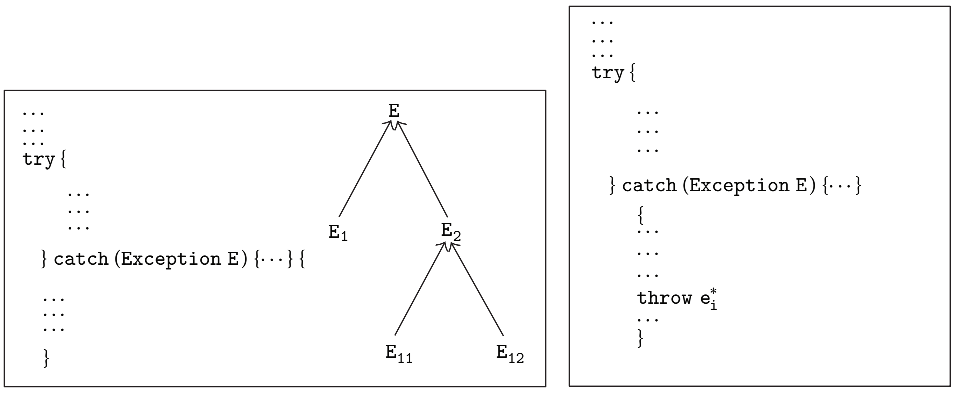


```
...
...
...
try {
     ...
     ...
     ...
 } catch (Exception E) {···}
 ...
 ...
 ...
```

```
...
...
...
try {
     ...
     ...
     ...
 } catch (Exception E) {···}
   catch (Exception_1 E_1) {···}
   catch (Exception_2 E_2) {···}
   catch (Exception_3 E_21) {···}
   catch (Exception_4 E_22) {···}
 ...
 ...
 ...
```

(i) Original program with *CEH* (ii) Mutant program with *BI*

FIGURE 6: Mutation operator *BI*.



```
...
...
...
try {
     ...
     ...
     ...
 } catch (Exception E) {···} {
 ...
 ...
 ...
 }
```

```
...
...
...
try {
     ...
     ...
     ...
 } catch (Exception E) {···}
     {
     ...
     ...
     ...
     throw e_i*
     ...
     }
```

*For each $e_i\ \varepsilon\ \{E, E1, E2, E11, E12\}$

(i) Original program with *CEH* (ii) Mutant program with *RE*

FIGURE 7: Mutation operator *RE*.

expressions by deleting primitive operators in Java programming language [19]. The rest of the mutation operators in Table 2 are the class-level mutation operators dealing with object-oriented features such as inheritance, polymorphisms, and dynamic binding. These operators are to modify declarations, modifiers, initializations, and so forth. A detailed description of these mutant operators can be found in [28].

The mutant operator *TD* can be mathematically stated as follows:

$$TD \Rightarrow [TB:]\texttt{try}\{\ldots;S;\ldots\}[CB:]\texttt{catch}(E)\{\ldots\} \rightarrow [TB:]\texttt{try}\{\ldots;S';\ldots\}[CB:]\texttt{catch}(E)\{\ldots\}$$

where S' is an altered statement of S after deleting one or more operator(s) or declaration(s).

3.6. TR: Replacement in `try` *Blocks.* Like the mutation operator *TD*, the mutation operator *TR* consists of method-level and class-level mutation operators [27, 28], which are listed in Table 3. As shown in Table 3, the first six mutation operators are to modify expressions by replacing some compatible Java primitive operators. The remaining mutation operators in Table 3 are to replace modifiers, assignments, initialization, and so forth [28].

The mutant operator *TR* can be formally stated as follows:

$$TR \Rightarrow [TB:]\texttt{try}\{\ldots;S;\ldots\}[CB:]\texttt{catch}(E)\{\ldots\} \rightarrow [TB:]\texttt{try}\{\ldots;S';\ldots\}[CB:]\texttt{catch}(E)\{\ldots\}$$

where S' is an altered statement of S after replacing one or more operator(s) or declaration(s).

3.7. TI: Insertion in `try` *Blocks.* Different mutation operators belonging to the proposed mutation operator *TI* are shown in Table 4. In Table 4, the mutation operators in first three rows are to modify expressions by inserting some compatible Java primitive operators. The rest of the operators in Table 4 are related to insertions of modifiers, assignments, initializations, and so forth [28].

The mutant operator *TD* can be mathematically stated as follows:

$$TI \Rightarrow [TB:]\texttt{try}\{\ldots;S;\ldots\}[CB:]\texttt{catch}(E)\{\ldots\} \rightarrow [TB:]\texttt{try}\{\ldots;S';\ldots\}[CB:]\texttt{catch}(E)\{\ldots\}$$

where S' is an altered statement of S after inserting one or more operator(s) or declaration(s).

3.8. TA: Alter `try` *Block.* The mutation operators discussed in the previous subsections are proposed to modify some

Table 3: Mutation operator *TR*.

Operator	Description	Level
AOR	Arithmetic operator replacement	Method-level
ROR	Relation operator replacement	
COR	Conditional operator replacement	
LOR	Logical operator replacement	
ASR	Assignment operator replacement	
SOR	Shift operator replacement	
IOP	Overridden method calling position change	Class-level
IOR	Overridden method rename	
PCC	Cast type change	
OMR	Overloading method content change	
EOA	Reference and assignment replacement	
EOC	Reference and content replacement	
EAM	Accessor method change	
EMM	Modifier method change	

Table 4: Mutation operator *TI*.

Operator	Description	Level
AOI	Arithmetic operator insertion	Method-level
COI	Conditional operator insertion	
LOI	Logical operator insertion	
IHI	Hiding variable information	Class-level
ISI	Super keyword insertion	
PCI	Type cast operation insertion	
JTI	This keyword insertion	
JSI	Static modifier insertion	

statements within `try` blocks. In the mutation operator *TA*, we propose to include some statements, which are outside `try` blocks, or move some statements outside `try` blocks. Figure 8(ii) shows the working of the mutation operator *TA*. In Figure 8(i), P_1 is a Java program and there is a `try` block such that a variable, say *in*, is initialized in the `try` block and it is referenced later outside the `try` block. Suppose that *in* is initialized with a null pointer. In that case, it may throw null pointer exception [29]. We mutate the program P_1 to P_1' by including the statement of reference within the `try` block with the objective that whether the `catch` block corresponding to `try` block is able to catch the exception type which might raise due to the reference of *in*. Here, the variable *in* is initialized and referenced within the `try` block and exception raised say due to null pointer assignment in statement marked as (A) may be caught in the `catch` block. Alternatively, let us consider the case of Java program P, where a variable x is initialized and referenced in a `try` block. We apply the mutation operator *TA* to P to have a mutant program P' such that the reference statement is moved from the `try` block and place it outside the `try-catch` block. The objective of this type of mutation is to check whether the exception raised may be propagated and catch in any caller exception or not. In the mutant program, it is now placed outside the `try` block and which is now beyond the scope of the `catch` block of the `try` block.

The mutant operator *TA* can be mathematically stated as follows:

$$TI \Rightarrow [TB:]\texttt{try}\{\ldots;S;\ldots\}[CB:]\texttt{catch}(E)\{\ldots\} \rightarrow [TB:]\texttt{try}\{\ldots;S';\ldots\}[CB:]\texttt{catch}(E)\{\ldots\}$$

where S' is an altered statement of S obtained after either keeping both declaration and reference of an object within the same `try` block if they were not together or declaration in the `try` block but its reference outside the `try` block, when they were originally together within a `try` block.

4. Mutation Analysis

In the last section, we have discussed the proposed mutation operators to test the effectiveness of test cases for testing exception handling constructs in Java. In this section, we discuss our analysis to evaluate the effectiveness of the proposed mutation operators. To do this, we have considered a number of Java programs. We then apply mutation operators to create a number of mutant programs for each Java program. A number of test cases which are used in our analysis is then discussed. All these steps are discussed in the following subsections.

4.1. Subject Programs. In order to verify the effectiveness of our approach, we have carried out a number of experiments. In our experiments, we have considered five different types of subject programs from different application domains. The Java programs considered are:

(i) library information systems (LIS): automating the regular routines in library management,

(ii) cell phone system (CPS): designing hand-set services for mobile communications,

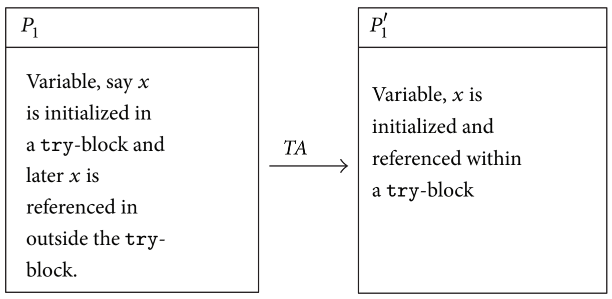
(iii) trading house automation system (TAS): automating business processes of trading houses,

(iv) conference management systems (CMS): automating various activities in conference organization,

(v) vending automation system (VAS): automating vending processes.

The designs of *LIS* and *CPS* are followed from [30] and [31], respectively. The applications, namely, *TAS* and *CMS,* are designed in UML 2.0 by the students of postgraduate course (software engineering) in Computer Science and Engineering Department, Indian Institute of Technology Kharagpur [32, 33]. The UML models of the application VAS are taken from [34]. We follow the designs from the above-mentioned sources and implemented them in Java. The implementation characteristics are summarized in Table 5.

The Java programs we have considered are neither very small nor very large, but of moderate sizes. In column 3 of


P_1
Variable, say x is initialized in a try-block and later x is referenced in outside the try-block.
TA
P_1'
Variable, x is initialized and referenced within a try-block

(i)

```
...
...
public void openFile (String fname)
 throw FileNotFoundException (“file not found”) {
 ...
 ...
byte readData [ ] = NULL;
...
 try {
 in = new FileInputStream (frame); ... (A)
 ...
 } catch (IOException ioe)
 ...
 in.read (readData, 0, 255);
  ...
```

(ii) Original program P_1

```
...
...
public void openFile (String fname)
 throw FileNotFoundException (“file not found”) {
 ...
 ...
byte readData [ ] = NULL;
...
 try {
 in = new FileInputStream (frame);
 ...
  in.read (readData, 0, 255); ... (B)
 } catch (IOException ioe) {...}
 ...
```

(iii) Muatnt program P_1' of P_1

Figure 8: Mutation operator *TA*.

Table 5, the numerator indicates the number of entity and boundary classes and the denominator indicates the number of modal classes also called controller classes (classes which exhibit distinct behavior in different states). The numbers of total `try` blocks, `catch` blocks are shown in columns 4 and 5 of Table 5, respectively. The number of exceptions thrown in Java methods, which are not within the scope of `try` block, is shown in column 6 of Table 5.

4.2. Mutant Programs. In our mutation analysis, we have considered fault-based mutation analysis technique. For this, we have generated the mutant programs of the considered

Table 5: Characteristics of subject programs.

Subject program	Size (LOC)	Number of classes	Exception handling constructs		
			Try blocks	Catch blocks	Outside exceptions
LIS	3356	16/04	26	33	18
TAS	2448	16/06	30	38	29
CPS	2257	21/07	24	46	21
CMS	2146	25/09	19	32	27
VAS	1864	10/04	11	25	31

programs (in Table 5). The mutant programs are generated by seeding faults randomly to the programs based on a fault model. The fault model we have followed in our work is stated below.

(1) Catch block misses exception: the exception thrown in a `try` block does not match with the exception parameter type of the `catch` block associated with the `try` block.

(2) Catch block handles exceptions incorrectly: the body of a `catch` block, that is, the exception handling routine, is not proper to handle the exceptions.

(3) Catch block rethrows exceptions: catch block handles an exception and rethrows the same or another exception(s).

(4) Catch block catches exceptions: exception thrown matches with the exception parameter type in a `catch` block and does not perform any routine but avoids abnormal exit.

(5) Try block throws exceptions implicitly: due to faults in expression, keyword, declaration, initialization, and assignment, some exceptions may be generated implicitly.

(6) Try block misses or throws explicit exceptions: some exceptions are missed and thrown outside `try` block; some exceptions are thrown deliberately inside `try` block.

Based on the above fault model, we have applied the mutation operators as discussed in Section 3, and the number of mutant programs created for each programs is shown in Table 6.

We have also generated a number of mutant programs with Jumble [17, 26]. The mutant programs with Jumble for each sample program is shown in the last row of Table 6. According to the proposed mutation operators we create 149 mutant programs for *LIS* (see column 2, Table 6), 152 mutant programs for *TAS* (column 3, Table 6), 163 mutant programs for *CPS* (column 4, Table 6), 162 mutant programs for *CMS* (column 5, Table 6), and 131 mutant programs for *VAS* (column 6, Table 6).

4.3. Generating Test Cases. To test the subject programs and their mutant versions, we have considered several test cases for each subject program. Generation of test cases is not the focus of this work. We have studied the existing test case generation approaches and followed an appropriate approach as discussed in the following.

Table 6: Mutant programs.

Mutation operator	Subject programs				
	LIS	*TAS*	*CPS*	*CMS*	*VAS*
BR	16	14	19	22	14
BD	17	19	20	26	18
BI	12	18	21	22	15
RE	8	11	16	10	13
TR	34	27	29	26	21
TD	23	26	21	22	20
TI	30	26	25	24	22
TA	9	11	12	10	8
Jumble	98	105	102	118	91

Structural testing techniques [35] develop test cases to cover various structural elements of a program such as control-flow, data-flow, branch, and path. Control-flow based structural testing criteria [36] use the flow of control in a program to generate test cases. Data-flow-based testing criteria [37–40] use the data-flow relationships to select the test cases. In branch testing [36, 41], test cases are developed by considering inputs that cause certain branches in the program under test to be executed. Similarly, in path testing [42, 43], test cases are generated to execute certain paths in programs. All the existing structural testing techniques, however, not necessarily cover new structural element such as exception handling constructs, all statements that raise exceptions, and those that catch exceptions. Please note that these criteria may be similar to the traditional coverage criteria [36] that require the coverage of statements or branches. Nevertheless, the criteria such as branch testing [41] have weak fault detection capabilities [44]. As a consequence, all-throw, all-catch coverage criteria are not necessarily strong coverage criteria. In fact, this coverage criterion does not cover the various exceptional control flow paths [4]. Sinha and Harrold [9] proposed a family of exception testing criteria to adequately test the behavior of exception handling constructs. Based on the proposed criteria [9], Sinha and Harrold [4] proposed a testing mechanism dealing with exception handling constructs. In [4], a technique has been proposed to construct control-flow graph representation of programs with explicit exception occurrences, that is, exceptions that are raised explicitly through `throw` statements and exception handling constructs. This

Table 7: Results of mutation analysis.

Program under test	Number of test cases	Mutant killed		Equivalent mutants		Mutation score	
		Jumble	Our approach	Jumble	Our approach	Jumble	Our approach
LIS	35	18	116	32	16	27	87
TAS	48	14	129	28	12	18	92
CPS	56	23	112	26	15	30	76
CMS	44	29	117	29	23	33	84
VAS	32	20	108	23	16	29	94
Average		**21**	**116**	**28**	**16**	**27**	**87**

control-flow representation is useful to trace various exceptional control flow paths and cover different types of exception that can be raised at a statement or the complex control and data interactions, both intra- and intermethods, that may have in the presence of exception handling constructs. In our work, we propose the test case generation scheme following the control flow representation of programs proposed in [4, 9].

From the control-flow representation, we can specify the paths to be tested; however, test data to exercise a path is a challenging task, particularly, for complex and large number of paths. The existing approaches of test data generation can be divided into three classes: random, static, and dynamic. Random test data generation is easy to automate but problematic [45]. First, it produces a statistically insignificant sample of the possible paths through the program under test. Second, it may be expensive to generate the expected output data for the large amount of input data produced. Finally, given that exceptions occur only rarely, the input domain which causes an exception is likely to be small. Random test data generations may never hit on this small area of the input domain.

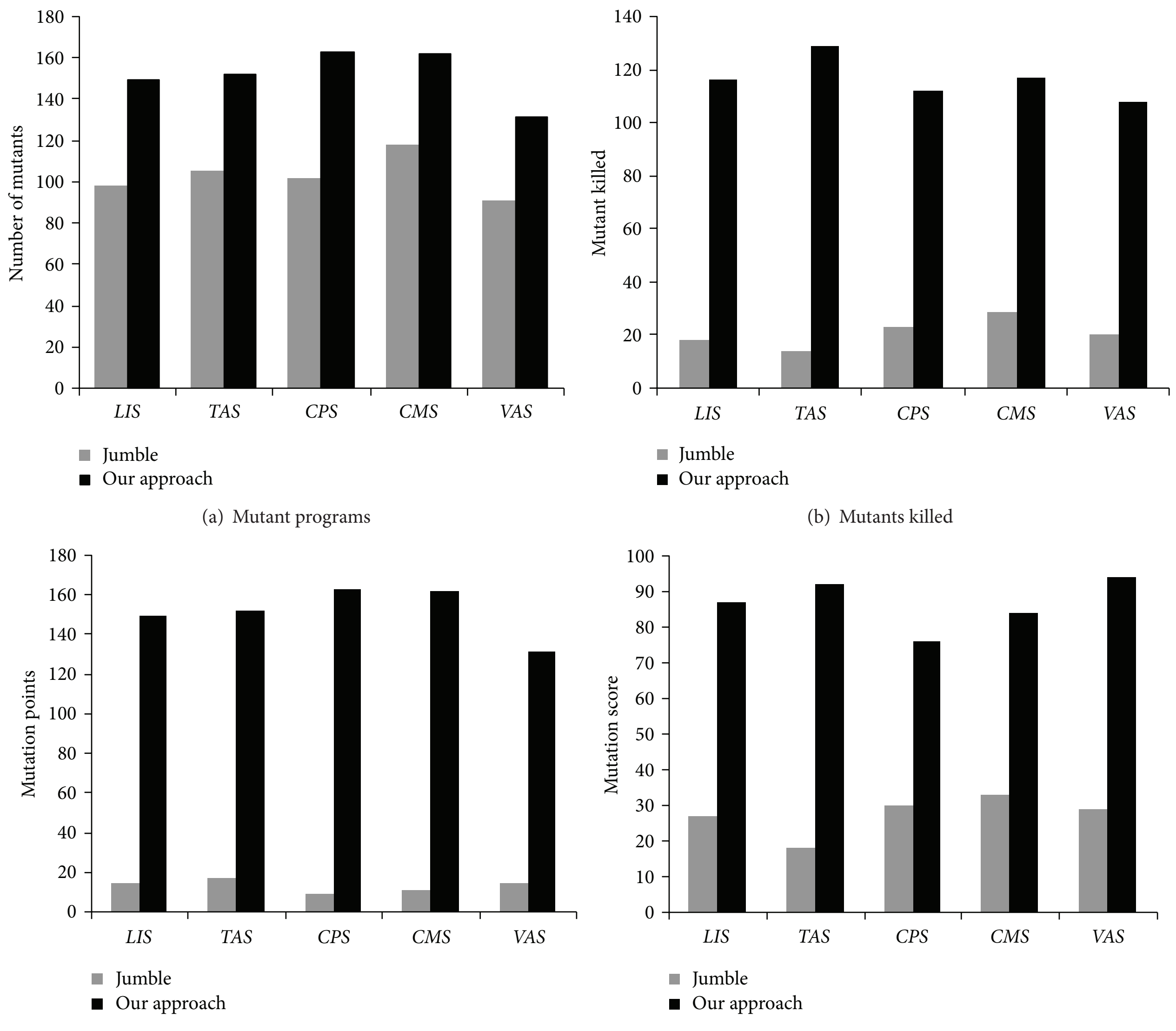
Static approaches to test data generation generally use symbolic execution. Many test data generation approaches presented in the literature use symbolic execution to obtain structural test data [46–50]. Symbolic execution works by traversing a control flow graph of the program under test and building up symbolic representations of the internal variables in terms of the input variables for the desired path. Branches within the code introduce constraints on the variables. Solutions to these constraints represent the desired test data. A number of problems exist with this approach. Using symbolic execution it is difficult to analyze recursion, array, and indices which depend on input data and some loop structures. Also, the problem of solving arbitrary constructs is known to be undecidable.

Dynamic test-data generation involves execution of the program under test and a directed search for test data that meets the desired criterion [51]. Local search techniques only work effectively for linear continuous functions. Consequently, these techniques are likely to become stuck at a local optimum [52]. DeMillo and Offutt [53] and Offutt et al. [54] proposed approaches for test data generation for mutation testing. The technique proposed in [53] is based on mutation analysis to create test data approximating the test adequacy. The technique is to automatically generate constraints and solve them to create test cases for unit testing. Another technique proposed in [54] is an extension of previous research [53] in constraint-based testing. This later method takes an initial set of values for each input and then refine the values through the control-flow graph of the program and resolve the path constraint dynamically. In this work, to generate the test data for test cases we follow the approach proposed in [54].

5. Results and Discussion

Test suites for each program considered in our work were generated as discussed in Section 4. Mutant programs for each program is also developed using the mutation operators discussed in Section 3. In this section, we discuss our experiments and result is obtained. Let T be the test suite for a program P and $\phi(P)$ its set of mutant programs. For each test case $t \in T$, we execute the program P and all mutant programs $Q \in \phi(P)$. If the observed output of the mutant Q is different than that of P, then we increase the count of the number of mutants killed. This way, we calculate the total numbers of mutants killed by a given test suite T. Further, we also calculate the equivalent mutants, such that two mutant programs $Q_1 \in \phi(P)$ and $Q_2 \in \phi(P)$ are not distinguishable as far as the output of the two is concerned. We determine whether a mutant $Q \in \phi(P)$ is an equivalent mutant or not. To do this, we look through the semantic exception hierarchy (as discussed in Section 2.2). If the exception parameter type in any original `catch` block is a super class of all the derived types (according to Java exception hierarchy) in the mutant `catch` block, then we judge the mutant Q to be an equivalent mutant. This is because the base type of exception can also accommodate all its subclass types. We also check if some mutants due to the mutation operators *BD*, *BI*, *TD*, *TR*, and *TI* are equivalent or not.

Our experimental results of mutation analysis with five considered subject programs are presented in Table 7. The number of mutants killed, equivalent mutants detected, and mutation score for each program and its number of test cases in test suite are shown in Table 7. It is observed that the number of mutants killed by Jumble with the mutant programs created by Jumble is on average 21. On the other hand, the number of mutants killed with mutant programs created by our proposed mutant operators is on average 87. It is also observed that mutant killed by Jumble is mainly related to logical, arithmetic, and conditional expression, whereas



(a) Mutant programs (b) Mutants killed (c) Mutation points (d) Mutation scores

Figure 9: Mutation analysis: Jumble versus our approach.

mutant killed with our approach is largely related to exception handling constructs. The performance of our mutation analysis is attributed in Figure 9. Figure 9 compares the mutant programs created, mutant killed, mutation points with exception handling constructs, and mutation score according to Jumble and our approach.

It may be noted that JUnit considers one module at a time. In contrast, we have followed the system testing approach. This is because the unit testing approach is not so effective to test an object-oriented program as multiple objects are in collaboration to solve a problem [55]. Moreover, our hierarchy of call exception is not necessarily limited to a single class but rather distributed across several classes. As a consequence, we need to have access to all the relevant modules so that from the callee class we can determine the actual location from where an exception has been thrown and caught. With modular approach it cannot be determined. Hence, in our approach, we have considered the whole program, that is, a nonmodular analysis technique. We admit that modular approach is faster but the nonmodular analysis at the use case level is quite practical and usually followed in object-oriented software engineering development environment [56].

Time Complexity. The running time complexities of both Jumble and our approach are $O(n)$, where n is the number of exceptional handling constructs in a program.

6. Related Work and Comparison

Gallagher and Narasimhan [8] proposed an approach for testing exception handling constructs. They proposed structural analysis to select the effective test cases from a given set of test cases to test exception handling constructs in Java programs. In their structural analysis, the whole structure of a program under test is analyzed to retrieve all possible execution paths. They define a fitness function to measure the performances of test cases. All test cases are executed and then evaluated one by one and the high quality test cases are selected based on their fitness measures.

Ji et al. [15] proposed four mutation operators for Java exception handling constructs. They aimed to develop effective mutant programs based on those mutation operators. The major difference with the work of Ji et al. [15] and our work is that they proposed the mutation operators to test whether the Java exceptional handling constructs will be able to catch the exceptions or not whereas objective of our work is to test the adequacy of test suites through mutation analysis on Java exceptions. Further, in Ji et al. [15] work, they considered exceptions, which are raised implicitly, that is, calling Java library routines. On the other hand, in addition to implicit raise of exceptions, we consider the generation of exceptions explicitly using arithmetic and logical control statements. That is why we consider many more conventional mutation operators other than the mutation operators as proposed by Ji et al. [15]. Moreover, to ensure that there is a sufficient number of exceptions raised, we have given extra effort to develop test cases accordingly, which is grossly ignored in the work of Ji et al. [15]. More significantly, Ji et al. [15] adopted a trivial experiment with JUnit to test the effectiveness of their proposed mutation operators, that is, whether their proposed mutation operators really capable of causing mutant programs or not. It may be noted that JUnit tests a program at a unit level (i.e., one class at a time), which in fact can catch exceptions raised by the methods in that class only. But, in general, an exception can be raised from method(s) located not necessarily in the same class it is actually caught. Indeed, it is a difficult task to catch the exceptions unless we have a full tracking of all paths of exceptions from the point of cause of exception to the caller routine (and hence main class). This requires entire program to be taken into account rather than only a particular unit. Furthermore, in our approach, emphasize has been given to locate the cause of exception at any point on a path of execution.

Sinha and Harrold [9] described a class of test adequacy criteria that can be used to test the behavior of exception handling constructs. They present a subsumption hierarchy of the criteria and point out the relationship of the criteria to those found in traditional subsumption hierarchies. They also proposed techniques for generating the testing requirements for the criteria using some control flow representations.

Buhr [21] proposed some exception handling mechanisms in the context of developing rich language features for developing robust programs. They also discussed the exception propagation mechanism and proposed propagation models such as static, dynamic, and propagation.

Sinha and Harrold [4] extended their work reported in [9] to analyze and test programs with exception handling constructs. They proposed techniques to construct control-flow representations for Java programs with explicit exceptions occurrences. They used these representation to perform some analysis such as static and dynamic slicing, structural and regression testing, and dynamic execution profiling.

Tracey et al. [6] proposed an optimization technique for automatically generating test data to test exceptions. They considered Ada as the model language in their approach. The proposed approach is based on the application of a dynamic global optimization based search for the required test data.

Adamopoulos et al. [57] reported on the cause of equivalent mutants. Offutt and Pan [58] proposed a technique to detect equivalent mutants and infeasible paths.

Andrews et al. [14] made a thorough investigation of a fundamental assumption that has been underlying much of experimental software testing research that whether faults generated by hand or from mutation operators is representative of real faults. They observed that the use of mutation operators is able to yield trustworthy results; that is, generated mutants are similar to real faults. Mutants appear, however, to be different from hand seeded faults that seem to be harder to detect than real faults.

Ma et al. [27] introduced a number of mutation operators to create mutant programs in the context of Java programming language. They propose a tool for entire mutation process and testing [28]. The mutant operators proposed in [27] only modify expressions by replacing, deleting, and inserting primitive features of Java programming language at method-level and object-oriented specific features such as inheritance, polymorphisms, and dynamic binding at class-level. However, Ma et al. did not consider any mutation operators to handle exceptions related faults.

Work of Gallagher and Narasimhan [8] and Sinha and Harrold [4] followed static analysis and their approaches not necessarily locate all exception points, particularly those exceptions raised implicitly. For example, an exception can be raised implicitly from calling Java library routine. As a consequence, it is very difficult to draw control flow representations or path identification to cover all problems. Hence, test case selection according to these approach may not be adequate. The proposed method, on the other hand, does not determine exception paths explicitly. We attempt to change exception propagation path according to the characteristic of Java exception handling constructs.

Jumble provides mutation operators for arithmetic expression, conditional predicate, initialization, assignment, return values, and control constructs such as if-then, if-then-else, switch, and loop. It is to be noted that the mutation operators according to Jumble are not sufficient enough for exception handling constructs.

7. Conclusions

Exception handling is an important mechanism to develop robust and reliable programs. It is a difficult task to test whether all exception points are covered and all exception handler routines are arranged appropriately or not. It is also not possible to identify all exception paths precisely because there are some exceptions which can be raised implicitly. This paper proposes a number of mutation operators to create mutant programs. With these mutant programs, we verify the effectiveness of a given test suite, particularly to verify the exception handling constructs. In other words, mutation operators used in this paper can generate effective mutant programs which can raise and handle different type of exceptions. The test suite which can kill more mutant programs is with a higher mutation score. Effectiveness of the proposed mutation operators is comparable to that of Jumble. Jumble also reports the quantity of mutants killed as the score of a test suite. Jumble provides mutations operators, which are mainly limited to arithmetic expressions, conditions, return values, and control statements such as decision and loop. The main limitation in Jumble is that there is no explicit mutation operators in Jumble that can take care exception handling constructs. Our experiments with five different programs substantiate that the proposed mutation operators are 69% (on average, mutation score in Jumble and our approach is 27 and 87, resp.) more effective to build mutants and then kill the mutant programs than the mutation analysis technique in Jumble. Nevertheless, this result is based on experiments with moderate sized Java programs from different application domains and can be taken as indicative. To have conclusive decision we should go for more experiments with more subjects.

Using the proposed mutation operators and mutation analysis method, our work can be further extended to guide automatic synthesis of test cases for testing exception handling constructs. Further, the mutation operators are proposed considering Java as the model language. We can extend the approach for other advanced programming languages.

Conflict of Interests

The author declares that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

This work was supported by Department of Science and Technology, Government of India, under Women Scientist Award (WOS-A) granted to the author. The author is thankful to the postgraduate students who helped in designing and implementing the applications used in this work.

References

[1] J. B. Goodenough, "Exception handling: issues and proposed notations," *Communications of the ACM*, vol. 18, no. 12, pp. 683–696, 1975.

[2] J. D. Musa, *Software Reliability Engineering*, McGraw-Hill, London, UK, 1998.

[3] W. N. Toy, "Fault tolerant design of local ESS processor," in *The Theory and Practice of Reliable System Design*, 1981.

[4] S. Sinha and M. J. Harrold, "Analysis and testing of programs with exception handling constructs," *IEEE Transactions on Software Engineering*, vol. 26, no. 9, pp. 849–871, 2000.

[5] B. G. Ryder, D. Smith, U. Kremer, M. Gordon, and N. Shah, "A static study of java exceptions using JSEP," Tech. Rep. DCS-TR-403, Rutgers University, November 1999.

[6] N. Tracey, J. Clark, K. Mander, and J. McDermid, "Automated test-data generation for exception conditions," *Software: Practice and Experience*, vol. 30, pp. 61–79, 2000.

[7] A. Bertolino, "Software testing research: achievements, challenges, dreams," in *Proceedings of the Future of Software Engineering (FoSE '07)*, pp. 85–103, May 2007.

[8] M. J. Gallagher and V. L. Narasimhan, "Adtest: a test data generation suite for ada software systems," *IEEE Transactions on Software Engineering*, vol. 23, no. 8, pp. 473–484, 1997.

[9] S. Sinha and M. J. Harrold, "Criteria for testing exception-handling constructs in Java programs," in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM '99)*, pp. 265–275, September 1999.

[10] M. P. Robillard and G. C. Murphy, "Analyzing exception flow in java TM programs," in *Proceedings of the Joint 7 th European Software Engineering Conference and the 7 th ACM SIGSOFT Internation al Symposium on the Foundations of Software Engineering (LNCS '99)*, vol. 1687, pp. 322–337, Toulouse, France, Septembe 1999.

[11] J.-D. Choi, D. Grove, M. Hind, and V. Sarkar, "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 (PASTE '99)*, pp. 21–31, 1999.

[12] T. A. Budd and D. Angluin, "Two notions of correctness and their relation to testing," *Acta Informatica*, vol. 18, no. 1, pp. 31–45, 1982.

[13] S.-W. Kim, J. A. Clark, and J. A. McDermid, "Investigating the effectiveness of object-oriented testing strategies using the mutation method," *Software Testing Verification and Reliability*, vol. 11, no. 4, pp. 207–225, 2001.

[14] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *Proceedings of the 27th International Conference on Software Engineering (ICSE '05)*, pp. 402–411, IEEE, May 2005.

[15] C. Ji, Z. Chen, B. Xu, and Z. Wang, "A new mutation analysis method for testing java exception handling," in *Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC '09)*, vol. 2, pp. 556–561, IEEE Computer Society Press, July 2009.

[16] http://junit.org/.

[17] http://www.jumble.sourceforge.net/.

[18] http://www.java.com/.

[19] D. Samanta, *Object-Oriented Programming with C++ and Java*, Prentice Hall of India, New Delhi, India, 2003.

[20] J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*, Addison-Wesley, Reading, Mass, USA, 1996.

[21] P. A. Buhr, "Advanced exception handling mechanisms," *IEEE Transactions on Software Engineering*, vol. 26, no. 9, pp. 820–836, 2000.

[22] G. J. Myers, *The Art of Software Testing*, John Wiley & Sons, Hoboken, NJ, USA, 2nd edition, 2004.

[23] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: help for the practicing programmer," *IEEE Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[24] M. E. Delamaro, J. C. Maldonado, and A. P. Mathur, "Interface mutation: an approach for integration testing," *IEEE Transactions on Software Engineering*, vol. 27, no. 3, pp. 228–247, 2001.

[25] J. B. Rainsberger and S. Stirling, *JUnit Recipes: Practical Methods for Programmer Testing*, Manning, Greenwich, UK, 2005.

[26] S. A. Irvine, T. Pavlinic, L. Trigg, J. G. Cleary, S. Inglis, and M. Utting, "Jumble java byte code to measure the effectiveness of unit tests," in *Proceedings of the IEEE Proceedings of Testing: Academic and Industrial Conference—Practice and Research Techniques*, pp. 169–175, September 2007.

[27] Y.-S. Ma, J. Offutt, and Y.-R. Kwon, "MuJava: a mutation system for java," in *Proceedings of the 28th International Conference on Software Engineering (ICSE '06)*, pp. 827–830, ACM, May 2006.

[28] http://cs.gmu.edu/~offutt/mujava/.

[29] K. Dobolyi and W. Weimer, "Changing java's semantics for handling null pointer exceptions," in *Proceedings of the 19th International Symposium on Software Reliability Engineering (ISSRE '08)*, pp. 47–56, November 2008.

[30] L. Briand and Y. Labiche, "A UML-based approach to system testing," Tech. Rep. TR SCE-01-01, Version 4, Carleton University, 2002.

[31] A. Abdurazik, J. Offutt, and A. Baldini, "A comparative evaluation of tests generated from different UML diagrams: diagrams and data," Tech. Rep. ISE-TR-05-04, George Mason University, Fairfax, Va, USA, 2005.

[32] M. Sarma, *Automatic test specification generation for state-based system testing [Ph.D. thesis]*, Indian Institute of Technology, Kharagpur, India, 2008.

[33] M. Sarma, "System state model generation from UML 2.0," Tech. Rep. CSE-TR-04-07, Indian Institute of Technology, Kharagpur, India, April 2007.

[34] L. C. Briand, J. Cui, and Y. Labiche, "Towards automated support for deriving test data from UML statecharts," in *Proceedings of the Unified Modeling Language Conference (UML '03)*, vol. 2863 of *Lecture Notes in Computer Science*, pp. 249–264, Springer, San Francisco, Calif, USA, October 2003.

[35] S. C. Ntafos, "A comparison of some structural testing strategies," *IEEE Transactions on Software Engineering*, vol. 14, no. 6, pp. 868–874, 1988.

[36] J. C. Huang, "An approach to program testing," *ACM Computing Surveys*, vol. 7, no. 3, pp. 114–128, 1975.

[37] P. G. Frankl and E. J. Weyuker, "An applicable family of data flow testing criteria," *IEEE Transactions on Software Engineering*, vol. 14, no. 10, pp. 1483–1498, 1988.

[38] M. J. Harrold and M. L. Soffa, "Interprocedural data flow testing," in *Proceedings of the ACM SIGSOFT of 3rd Symposium of Software Testing, Analysis and Verification (SIGSOFT '89)*, pp. 158–167, December 1989.

[39] J. W. Laski and B. Korel, "A data flow oriented program testing strategy," *IEEE Transactions on Software Engineering*, vol. 9, no. 3, pp. 347–354, 1983.

[40] S. Rapps and E. J. Weyuker, "Selecting software test data using data flow information," *IEEE Transactions on Software Engineering*, vol. 11, no. 4, pp. 367–375, 1985.

[41] P. G. Frankl and S. N. Weiss, "Experimental comparison of the effectiveness of branch testing and data flow testing," *IEEE Transactions on Software Engineering*, vol. 19, no. 8, pp. 774–787, 1993.

[42] W. E. Howden, "Methodology for the generation of program test data," *IEEE Transactions on Computers*, vol. 24, no. 5, pp. 554–560, 1975.

[43] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. 2, no. 4, pp. 308–320, 1976.

[44] P. G. Frankl and E. J. Weyuker, "Provable improvements on branch testing," *IEEE Transactions on Software Engineering*, vol. 19, no. 10, pp. 962–975, 1993.

[45] B. Beizer, *Software Testing Techniques*, Thomson Computer Press, New York, NY, USA, 2nd edition, 1990.

[46] R. Boyer, B. Elspas, and K. Levitt, "SELECT—a formal system for testing and debugging programs by symbolic execution," in *Proceedings of International Conference on Reliable Software*, pp. 234–245, 1975.

[47] L. A. Clarke, "A system to generate test data and symbolically execute programs," *IEEE Transactions on Software Engineering*, vol. 2, no. 3, pp. 215–222, 1976.

[48] J. C. King, "Symbolic execution and program testing," *Communications of the Association for Computing Machinery*, vol. 19, no. 7, pp. 385–394, 1976.

[49] C. V. Ramamoorthy, S.-B. F. Ho, and W. T. Chen, "On the automated generation of program test data," *IEEE Transactions on Software Engineering*, vol. 2, no. 4, pp. 293–300, 1976.

[50] R. A. DeMillo and A. J. Offutt, "Experimental results from an automatic test case generator," *ACM Transactions on Software Engineering and Methodology*, vol. 2, no. 2, pp. 109–127, 1993.

[51] B. Korel, "Automated software test data generation," *IEEE Transactions on Software Engineering*, vol. 16, no. 8, pp. 870–879, 1990.

[52] B. F. Jones, H.-H. Sthamer, and D. E. Eyres, "Automatic structural testing using genetic algorithms," *Software Engineering Journal*, vol. 11, no. 5, pp. 299–306, 1996.

[53] R. A. DeMillo and A. J. Offutt, "Constraint-based automatic test data generation," *IEEE Transactions on Software Engineering*, vol. 17, no. 9, pp. 900–910, 1991.

[54] A. J. Offutt, Z. Jin, and J. Pan, "The dynamic domain reduction procedure for test data generation," *Software—Practice and Experience*, vol. 29, no. 2, pp. 167–193, 1999.

[55] R. V. Binder, *Testing Object Oriented Systems: Models, Patterns and Tools*, The Object Technology, Addison-Wesley, 1999.

[56] J. Z. Gao, H.-S. J. Tsao, and Y. Wu, *Testing and Quality Assurance for Component-Based Software*, Artech House, Norwood, Mass, USA, 2003.

[57] K. Adamopoulos, M. Harman, and R. M. Hierons, "How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution," in *Proceedings of the Genetic and Evolutionary Computation Conference (LNCS '04)*, vol. 3103, pp. 1338–1349, Seattle, Wash, USA, June 2004.

[58] A. J. Offutt and J. Pan, "Automatically detecting equivalent mutants and infeasible paths," *Software Testing Verification and Reliability*, vol. 7, no. 3, pp. 165–192, 1997.