Fault Localization and Repair for Java Runtime Exceptions

Saurabh Sinha
IBM India Research Lab
New Delhi, India
saurabhsinha@in.ibm.com

Hina Shah
Georgia Tech
Atlanta, Georgia, U.S.A.
hinashah@cc.gatech.edu

Carsten Görg
Georgia Tech
Atlanta, Georgia, U.S.A.
goerg@cc.gatech.edu

Shujuan Jiang
CUMT
Xuzhou, Jiangsu, China
shjjiang@cumt.edu.cn

Mijung Kim
Georgia Tech
Atlanta, Georgia, U.S.A.
mijung.kim@cc.gatech.edu

Mary Jean Harrold
Georgia Tech
Atlanta, Georgia, U.S.A.
harrold@cc.gatech.edu

ABSTRACT

This paper presents a new approach for locating and repairing faults that cause runtime exceptions in Java programs. The approach handles runtime exceptions that involve a flow of an incorrect value that finally leads to the exception. This important class of exceptions includes exceptions related to dereferences of null pointers, arithmetic faults (e.g., ArithmeticException), and type faults (e.g., ArrayStoreException). Given a statement at which such an exception occurred, the technique combines dynamic analysis (using stack-trace information) with static backward data-flow analysis (beginning at the point where the runtime exception occurred) to identify the source statement at which an incorrect assignment was made; this information is required to locate the fault. The approach also identifies the source statements that may cause this same exception on other executions, along with the reference statements that may raise an exception in other executions because of this incorrect assignment; this information is required to repair the fault. The paper also presents an application of our technique to null pointer exceptions. Finally, the paper describes an implementation of the null-pointer-exception analysis and a set of studies that demonstrate the advantages of our approach for locating and repairing faults in the program.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis*

General Terms

Algorithms, Experimentation, Measurement

Keywords

Fault localization, runtime exceptions, static analysis, null dereference

1. INTRODUCTION

Programming languages, such as Java, that are designed for robustness provide an exception-handling mechanism. Under this mechanism, when a semantic constraint of the programming language is violated, (e.g., trying to access an element outside the bounds of an array), an exception is raised to indicate the error, the regular flow of control is interrupted, and there is an attempt to transfer control to a designated part of the program that tries to recover from the error. If recovery is possible, the program continues within the regular flow of control.

Exceptions in Java can be classified as: *application exceptions* that are explicitly thrown in response to exceptional conditions in an application, and *runtime exceptions* that are generated by the Java runtime environment. There has been considerable research on application exceptions that has resulted in techniques for analyzing them to provide useful information to developers (e.g., [5, 11, 14]). However, there has been little research that studies runtime exceptions, although they occur often in executing Java programs. Such exceptions represent programming errors, and include arithmetic exceptions, such as dividing by zero, and dereferences of null pointers. Because Java does not require that methods specify or catch such exceptions, when they occur during execution, there is typically no exception handler available to handle the condition, and the program terminates.

To illustrate, consider the example program in Figure 1. Executing the program with input value `x` results in the following null pointer exception and stack trace:

```
Exception in thread "main" java.lang.NullPointerException
  at RTEExample.method3(RTEExample.java:30)
  at RTEExample.method4(RTEExample.java:33)
  at RTEExample.main(RTEExample.java:5)
```

The stack trace shows that the null pointer exception was thrown in `method3` at line 30 (i.e., the dereferenced field `string1` was `null`). The stack trace also shows that `method4` and `main` were still on the stack when the exception occurred, that `main` called `method4` at line 5, and that `method4` called `method3` at line 33.

When these runtime exceptions occur, the Java runtime environment stores information in the stack trace that can help the developer locate and fix the fault that causes the exception. The *stack trace* includes information about which line of code threw the exception and the method that contains that line of code. The stack trace also contains all methods currently on the runtime stack, along with the statements in those methods at which the method calls were made.

Although the stack trace can be useful in locating the code or the conditions that cause the exception to be raised, the granularity of the information provided in the stack trace may be too coarse to

```
    public class RTEExample {
      String string1,string2;
      int val;
1.    public static void main( String args[] ) {
2.      RTEExample rte =
          new RTEExample(args[0].length());
3.      if ( args.length == 1 ) {
4.        rte.method2();
5.        rte.method4();
        } else {
6.        int num = Integer.parseInt( args[1] );
7.        rte.method1( num );
        }
8.      rte.method3();
9.    }
10.   public RTEExample( int i ) {
11.     val = i;
12.     string1 = null; // NP assignment
13.     string2 = null; // NP assignment
14.     if ( val > 1 ) {
15.       string2 = new String( "abc" );
        }
16.   }
```

```
17.   public void method1( int j ) {
18.     if ( j == 0 ) {
19.       String tempStr = null; // NP assignment
20.       string1 = tempStr;
        } else {
21.       string1 = null;  // NP assignment
        }
22.     string2.charAt(0); // NP dereference
23.   }
24.   public void method2() {
25.     string1 = string2;
26.   }
27.   public void method3() {
28.     if ( val > 1 ) {
29.       string2.charAt(0);
        }
30.     string1.charAt(0);   // NP dereference
31.   }
32.   public void method4() {
33.     method3();
34.   }
    }
```

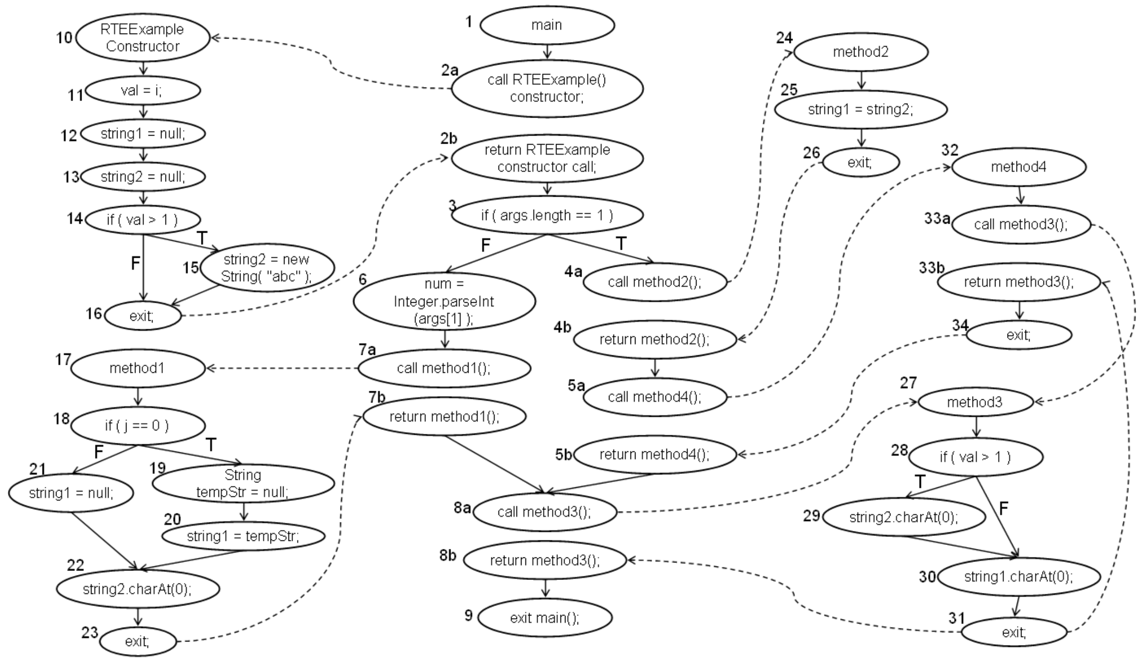

Figure 1: Java program `RTEExample` that illustrates runtime exceptions and the interprocedural control-flow graph for the program.

assist in locating the cause of the exception. The stack trace contains only the methods that are involved in the execution and the statement at which a method call was made, but does not contain information about the flow of control through the method. Thus, to locate the cause of the exception, the developer must inspect the execution manually, using the stack trace, and attempt to understand the flow of control through the calling methods. Furthermore, methods that have been called during the execution and have already returned will not appear in the stack trace. Thus, during the manual inspection of the methods to locate the cause of the exception, the developer may miss methods that were involved in the execution, or may not understand the complex flow of control in the program. Finally, after the location of the cause of the exception is found, the stack trace provides little information to assist the developer in repairing the fault. The developer must locate manually other statements in the program that may cause the exception to be raised in other executions and, therefore, must be considered to fix the fault that caused the exception in this execution.

To illustrate, consider again the stack trace resulting from executing the program in Figure 1 with input value `x`. Further inspection shows that none of the methods in the stack trace contains an assignment to `string1`, the field whose dereference in statement 30 resulted in the exception. Thus, the stack-trace information alone is insufficient for finding the cause of this exception. Figure 1 displays the interprocedural control flow graph[1] for our example program. The execution path for input value `x` consists of:

> 1, 2a, 10, 11, 12, 13, 14, 16, 2b, 3, 4a, 24, 25, 26, 4b,
> 5a, 32, 33a, 27, 28, 30.

A careful examination of this execution shows that the null assignment that caused the null pointer exception occurred at statement 13, where `string2` was assigned a null value. This value was copied to `string1` at statement 25, and then dereferenced at statement 30. Neither the constructor that contains statement 13

[1]A *control-flow graph* (CFG) for a method contains nodes that represent statements and edges that represent the flow of control between statements. An *interprocedural control-flow graph* (ICFG) contains a CFG for each method in the program. A call site is represented using a call node and a return node. At each call site, a call edge connects the call node to the entry node of the called method; a return edge connects the exit node of the called method to the return node.

nor the method that contains statement 25 is on the stack when this exception is thrown. Further examination of the rest of the program shows that the null values assigned at statements 19 and 21 could also reach the dereference at statement 30 in some other execution; such statements should be considered while repairing the fault.

To address the limitations of existing techniques, we have developed, and present in this paper, a new approach that automatically identifies the cause of a runtime exception, and provides context information to assist the developer in repairing the fault. Our approach identifies the source statement(s) responsible for the exception in the failing execution; this information is required to *locate the fault*. Our approach also identifies (1) other source statements that may cause this same exception to be raised on other executions, and (2) other reference statements whose execution may result in exceptions in other executions because of the same source statement; this information is required to *repair the fault*.

The approach combines dynamic analysis (using stack-trace information) with static backward data-flow analysis (starting at the point where the runtime exception occurred). The approach handles runtime exceptions that involve a flow of an incorrect value that finally leads to the exception. This important class of exceptions includes (1) exceptions caused by dereferences of null pointers, (2) exceptions related to arithmetic faults, such as ArithmeticException, and (3) exceptions related to type faults, such as ArrayStoreException. To illustrate our approach, we applied it to null pointer exceptions. Our technique extends an interprocedural path-sensitive and context-sensitive analysis that finds potential dereferences of null values [10]. The extensions include incorporating the use of the stack trace for a failing execution and finding context information required to repair the fault.

In this paper, we also present the results of studies that we performed on a set of open-source programs. Our studies evaluate the effectiveness of our technique in finding the null pointer assignment(s) that cause a runtime exception in Java programs, and show that our technique is more effective than using a static-analysis approach or the stack-trace information alone. Our studies also show the additional information that our technique finds that can assist in repairing the fault.

The main benefit of our approach is that it automates the search for the cause of runtime exceptions using readily available dynamic information about the execution in the form of the stack trace. Thus, it reduces the manual effort required to locate the fault, which reduces debugging time. Another benefit of our approach is that it automates the identification of code that is not responsible for the exception on this execution but that may cause the same exception on other executions. Thus, it provides the developer with the necessary context for fixing the fault, and ensuring that the exception does not occur on subsequent executions. A third benefit of our approach is that it applies to an important class of runtime exceptions that occur often in practice. Thus, it can be an important tool for use by developers that will reduce the time to find and fix faults caused by this class of runtime exceptions.

The contributions of the paper include

- A presentation of a novel approach for locating faults that cause runtime exceptions and for providing context information required for fixing the faults
- A description of a technique that applies the approach to null pointer exceptions
- The results of analytical and empirical evaluations that show that our technique is more effective than other techniques based on static analysis or stack-trace information alone

2. OVERVIEW OF OUR APPROACH

In this section, we present a high-level overview of our approach for supporting fault localization and fault repair using a combination of static and dynamic analysis. As mentioned earlier, currently, our approach is limited to failures caused by the flow of an incorrect value from a source statement to the program point where the value causes a failure.

Our approach consists of two phases. In the first phase, our technique assists in fault localization by providing the location of the source statement that is responsible for the runtime exception. In the second phase, our technique assists in fault repair by providing context information about other statements that are related to this exception, and may be involved in runtime exceptions in other executions. We present an overview of these phases in turn.

In execution $\mathcal{E}$ in which an exception was raised at reference statement $s_{r(\mathcal{E})}$, there is exactly one statement $s_{a(\mathcal{E})}$ at which the incorrect value that caused the exception is generated. The goal of Phase 1 of our technique is to locate this source statement. For example, for a NullPointerException at statement `x.f`, $s_{a(\mathcal{E})}$ is the statement that assigned a null value to `x`. For an ArithmeticException at statement `z = (x-y)`, $s_{a(\mathcal{E})}$ is the statement at which the expression `(x-y)` became zero—this statement could be an assignment to either `x` or `y`. For an ArrayStoreException at statement `a[i] = x`, $s_{a(\mathcal{E})}$ is the statement that assigned an incorrect type to `x` that caused the exception.

To locate $s_{a(\mathcal{E})}$, our approach performs a backward interprocedural data-flow analysis of the program, starting at $s_{r(\mathcal{E})}$. The analysis is guided by the available dynamic information about the failing execution $\mathcal{E}$. In most cases of field failures, the only dynamic information available is the stack trace associated with $\mathcal{E}$. Therefore, we illustrate the analysis using stack-trace information. However, when additional dynamic information, such as branch traces, are available, our approach can use such information.[2]

The result of this stack-trace-driven analysis lets us classify the source statements $s_{a(\mathcal{E})}$ into two categories: definite and possible. In some cases, the analysis computes a unique source statement at which the incorrect value that reaches $s_{r(\mathcal{E})}$ is generated. For this case, this statement definitely caused the exception in execution $\mathcal{E}$. Thus, we classify $s_{a(\mathcal{E})}$ as a *definite* incorrect assignment, and we know that it is the cause of the exception. In other cases, the analysis computes more than one source statement at which the incorrect value that reaches $s_{r(\mathcal{E})}$ in $\mathcal{E}$ could have been generated. Because the stack trace provides no control-flow information within the methods, there may be multiple possible source statements that could have caused the exception at $s_{r(\mathcal{E})}$ in $\mathcal{E}$. For example, if a method M that is executed in $\mathcal{E}$ has a conditional statement c where an incorrect value is generated along both branches of c, the analysis cannot determine which of the branches was executed in $\mathcal{E}$. Although in such cases the analysis cannot determine which of a set of source statements assigned the incorrect value that caused the exception at $s_{r(\mathcal{E})}$, it can determine that at least one of the set of statements did cause the exception. Thus, we classify statements $s_{a(\mathcal{E})}$ as *possible* incorrect assignments, and we know that at least one of them caused the exception on execution $\mathcal{E}$.

Figure 2 illustrates the information that our technique provides. In the figure, the shaded area represents the execution $\mathcal{E}$ on which the runtime exception was thrown at $s_{r(\mathcal{E})}$. The figure shows the case in which the analysis was able to identify the exact statement

[2]In some cases, the inputs that cause failures may be available. For example, some of the BUGZILLA reports for the `Ant` project that we examined contained the build files on which null pointer exceptions occurred. In such cases, more detailed dynamic information can be computed in-house by instrumenting the code and executing it on the failing test inputs.

Table 1: Failing executions, corresponding stack traces, and NPA and NPR classifications, for the example program.

Execution Number	Input	Execution Path	Stack Trace	Definite NPA	Possible NPA	Maybe NPA	Maybe NPR	Definite/Possible NPAs in a Method in the Stack Trace
1	`<x>`	1, 2a, 10, 11, 12, **13**, 14, 16, 2b, 3, 4a, 24, 25, 26, 4b, 5a, 32, 33a, 27, 28, 30	`method3(RTEExample.java:30)` `method4(RTEExample.java:33)` `main(RTEExample.java:5)`	13	–	19, 21	22	–
2	`<xx,0>`	1, 2a, 10, 11, 12, 13, 14, 15, 16, 2b, 3, 6, 7a, 17, 18, **19**, 20, 22, 23, 7b, 8a, 27, 28, 29, 30	`method3(RTEExample.java:30)` `main(RTEExample.java:8)`	–	19, 21	13	–	–
3	`<xx,1>`	1, 2a, 10, 11, 12, 13, 14, 15, 16, 2b, 3, 6, 7a, 17, 18, **21**, 22, 23, 7b, 8a, 27, 28, 29, 30	`method3(RTEExample.java:30)` `main(RTEExample.java:8)`	–	19, 21	13	–	–
4	`<x,1>`	1, 2a, 10, 11, 12, **13**, 14, 16, 2b, 3, 6, 7a, 17, 18, 21, 22	`method1(RTEExample.java:22)` `main(RTEExample.java:7)`	13	–	–	30	–

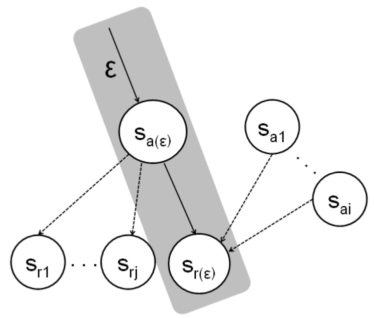

Figure 2: Categories of assignments and references identified by our technique.

$s_{a(\mathcal{E})}$ that caused the exception to be raised at $s_{r(\mathcal{E})}$. Thus, in the figure, $s_{a(\mathcal{E})}$ is a *definite* incorrect assignment.

After Phase 1 of our analysis, the developer can choose Phase 2 of the analysis, which provides context for fixing the fault. In particular, Phase 2 provides information about other statements that are related to both $s_{a(\mathcal{E})}$ and $s_{r(\mathcal{E})}$ and should be considered when fixing the fault. For $s_{r(\mathcal{E})}$, our technique finds all statements s_a that could assign incorrect values that could reach $s_{r(\mathcal{E})}$ and cause an exception on an execution other than $\mathcal{E}$. For $s_{a(\mathcal{E})}$, our technique locates all statements s_r that the incorrect value at $s_{a(\mathcal{E})}$ could reach on an execution other than $\mathcal{E}$ and cause an exception to be raised. We classify statements, such as s_a, as *maybe* assignments and statements, such as s_r, as *maybe* references.

To illustrate, consider again Figure 2. The figure shows that a set of statements, $s_{a1}, \ldots, s_{ai}$, have been identified that could assign incorrect values that may flow to $s_{r(\mathcal{E})}$, and cause an exception to be raised on executions other than $\mathcal{E}$. The figure also shows that a set of statements, $s_{r1}, \ldots, s_{rj}$, could reference the incorrect value assigned at $s_{a(\mathcal{E})}$ in other executions and cause a runtime exception to be raised. Both kinds of statements provide context for fixing the fault related to the exception that was raised at $s_{r(\mathcal{E})}$.

3. APPLICATION OF OUR APPROACH TO NULL POINTER EXCEPTIONS

To demonstrate our approach that uses static analysis (in the form of backward data-flow analysis) guided by dynamic information (in the form of the stack trace) to assist in fault localization and fault repair, we applied it to null pointer exceptions. Our approach uses the null-dereference analysis implemented in a tool called XYLEM [10], which we refer to as XYLEM analysis.

In this section, we first discuss null pointer assignments and dereferences. Then, we provide an overview of the XYLEM analysis (details can be found in Reference [10]) and discuss the modifications we made to it for use in our new algorithm. After that, we present the details of our two-phase algorithm.

3.1 Null Assignments and Dereferences

In Java, a NullPointerException is thrown at a statement that dereferences a variable or a field that has a null value. A *null pointer assignment* (NPA) is a statement at which a null value originates; examples of null pointer assignments include statements "`x = null`," "`return null`," and "`foo(null)`" (a `null` value for an actual parameter at a method call). A *null pointer dereference* (NPR) is a dereference statement at which the dereferenced variable could potentially be null. In the example in Figure 1, statement 13 is a null pointer assignment, and statement 22—at which `string2` can potentially be null—is a null pointer dereference.

In keeping with the classification of source statements related to the runtime exception at $s_{r(\mathcal{E})}$ for execution $\mathcal{E}$, we classify an NPA as definite, possible, or maybe: $s_{a(\mathcal{E})}$ is a *definite NPA* if the null value generated at $s_{a(\mathcal{E})}$ was the one dereferenced at $s_{r(\mathcal{E})}$ in $\mathcal{E}$; $s_{a(\mathcal{E})}$ is a *possible NPA* if the null value generated at $s_{a(\mathcal{E})}$ could have been dereferenced at $s_{r(\mathcal{E})}$ in $\mathcal{E}$; s_a is a *maybe NPA* if the null value generated at s_a was definitely not dereferenced at $s_{r(\mathcal{E})}$ in $\mathcal{E}$, but could be dereferenced at $s_{r(\mathcal{E})}$ in other executions. There exists a unique definite NPA for each null pointer exception, and the goal of Phase 1 of our approach is to identify that NPA.

Table 1 lists four failing executions that result in null pointer exceptions for `RTEExample` (Figure 1). For each execution, the table shows an execution number (column 1), the program input (column 2), the path traversed by the execution (column 3), the stack trace (column 4), the definite, possible, and maybe NPAs (columns 5–7) identified by our analysis (using the stack trace), and whether the definite or possible NPAs were contained in a method that is in the stack trace for that execution (column 9). (We discuss column 8 later.) In each execution path, the last statement is the NPR and the statement shown in boldface is the NPA at which the dereferenced null value originates. For example, in execution 1, the null assignment at statement 13 causes the null dereference at statement 30. For this execution, our analysis identifies statement 13 as the definite NPA, no statements as possible NPAs, and statements 19 and 21 as maybe NPAs. Note that, statically, the null values generated at each of statements 13, 19, and 21, reach statement 30. However, based on the stack configuration, the analysis determines that statement 13 is the NPA that caused the null pointer exception in execution 1 and, therefore, that statements 19 and 21 definitely did not cause the execution to fail.

For execution 2, the analysis cannot identify the definite NPA. The stack trace contains insufficient information for the analysis to determine which branch from condition statement 18 was taken during the execution. Therefore, the analysis assumes that either branch could have been taken, and classifies statements 19 and 21 as possible NPAs. The analysis can, however, determine that statement 13 definitely could not have caused the null pointer exception in execution 2, and thus, classifies it as a maybe NPA.

To support the repair of null-dereference faults, in addition to the maybe NPAs, our approach computes maybe NPRs for a definite NPA. For a definite NPA $s_{a(\mathcal{E})}$ that caused a runtime exception in execution $\mathcal{E}$, a *maybe* NPR is a statement that did not dereference the null value generated at $s_{a(\mathcal{E})}$ in $\mathcal{E}$, but that could dereference the null value in a different execution. Column 8 of Table 1 lists the maybe NPRs for the four executions of `RTEExample`. For example, for definite NPA 13 in execution 1, statement 22 is a maybe NPR. Although statement 22 does not dereference the null value in execution 1, it can dereference that value in a different execution, as illustrated by execution 4.

3.2 Overview of the XYLEM Analysis

The XYLEM analysis is purely static and, for an NPR, it continues its analysis until one NPA is located. This NPA is one that can occur on some execution of the program [10]. For our approach, we modified the original XYLEM analysis in two ways: (1) instead of stopping when it finds one NPA, the analysis continues until it finds all NPAs on the paths that it considers; (2) the analysis is parameterized so that the method for finding which calling methods and which CFG predecessors to consider in the analysis is specified as a parameter to the analysis.

Starting at a statement s_r that dereferences variable v, XYLEM performs a backward, path-sensitive analysis to determine whether v could be null at s_r. During the analysis, it propagates a set of abstract state predicates backward in the CFG. The analysis starts with a predicate asserting that v is null, and updates states during the path traversal. If the updated state becomes inconsistent, the path is infeasible and the analysis stops traversing the path.

Figure 3 presents algorithm, `ComputeNPA`, for identifying NPAs. The algorithm takes as input the dereference statement s_r of variable v from which to start the traversal, along with a dispatcher $\mathcal{D}$ that locates the methods into which the backward analysis continues when the entry of a method is reached and the CFG predecessors from which the analysis continues. The algorithm returns as output N, a set of NPAs. For each s_a in N, the algorithm has found a path along which the null value assigned at s_a may be dereferenced at s_r.

The algorithm initializes the state Γ with predicates $\langle v = \texttt{null}\rangle$ (v is the variable dereferenced at s_r) and $\langle \texttt{this} \neq \texttt{null}\rangle$ (line 1), and then calls procedure `analyzeMethod` (line 2), which uses a standard worklist-based approach to compute a fix-point solution over the abstract state predicates (lines 4–22). To perform efficient analysis, XYLEM abstracts away arithmetic expressions; thus, the number of generated predicates is bounded and the analysis is guaranteed to terminate. The analysis traverses a loop until the state no longer changes from one iteration to the next. Because, integer arithmetic over the loop induction variable is abstracted away, the analysis of a loop is bounded. The analysis uses back substitutions to update state predicates. For example, at a statement `x = y`, each predicate on `x` in the incoming state is transformed to a corresponding predicate on `y`. We refer the reader to Reference [10] for details of the XYLEM analysis.

The initial predicate $\langle v = \texttt{null}\rangle$ is called the *root predicate* and the dereferenced variable v the *root reference*. The goal of procedure `analyzeMethod` is to identify null assignments to the root reference. At an assignment statement $v = w$, the root reference is updated to w and the root predicate becomes $\langle w = \texttt{null}\rangle$. `analyzeMethod` removes an element (s, Γ) from the worklist, and uses the dispatcher, passed in as a parameter, to select the predecessors of s (lines 6–7). In the original XYLEM analysis, the procedure selects all predecessors of s. If a predecessor is neither the entry node of the method nor a call node (line 9), the procedure computes the state transformation induced by s_p (line 10). When the procedure encounters a null assignment to the root reference, it has found an NPA—the root predicate becomes $\langle \texttt{null} = \texttt{null}\rangle$, which is represented as $\langle \texttt{true}\rangle$—and the algorithm adds s_p to the set of NPAs (line 11).

```
algorithm ComputeNPA
input  $s_r$        Dereference of variable $v$ for which to compute NPAs
       $\mathcal{D}$        Dispatcher that selects the predecessors of a node to traverse
global $N$        NPAs for $s_r$
       $CS$       call stack of methods
       $\sigma(s,\Gamma)$  summary information at a call site $s$ that maps an
                 incoming state $\Gamma$ to an outgoing state
begin
 1. $\Gamma$ = {$\langle v = \texttt{null}\rangle$,$\langle \texttt{this} \neq \texttt{null}\rangle$}
 2. analyzeMethod($s_r, \Gamma$)
 3. return $N$
procedure analyzeMethod ($s$: starting statement, $\Gamma$: state)
 4. initialize $worklist$ with $(s, \Gamma)$
 5. while $worklist \neq \emptyset$ do
 6.     remove $(s, \Gamma)$ from $worklist$
 7.     $preds$ = get predecessors of $s$ from $\mathcal{D}$
 8.     foreach $s_p$ in $preds$ do
 9.         if $s_p$ is not the entry and not a call then
10.             compute $\Gamma'$ for the transformation induced by $s_p$
11.             if root-predicate($\Gamma'$) = $\langle \texttt{true}\rangle$, add $s_p$ to $N$ // found NPA
12.             add $(s_p, \Gamma')$ to $worklist$ if not visited
13.         else if $s_p$ is a call that invokes $M$ then
14.             $\Gamma' = \sigma(s_p, \Gamma)$
15.             if $\Gamma' = \emptyset$ then // no summary exists
16.                 push $M$ onto $CS$
17.                 $\Gamma'$ = map $\Gamma$ to the exit of $M$
18.                 analyzeMethod(exit node of $M, \Gamma'$)
19.                 pop $CS$
20.                 $\Gamma'$ = map state at the entry of $M$ to $s_p$
21.                 add $\Gamma'$ to $\sigma(s_p, \Gamma)$
22.             add $(s_p, \Gamma')$ to $worklist$ if not visited
23. if $CS = \emptyset$ then // method not being analyzed in a specific context
24.     $cs$ = get call sites from $\mathcal{D}$
25.     foreach $s_c$ in $cs$ do
26.         $\Gamma'$ = map $\Gamma$ to $s_c$
27.         analyzeMethod($s_c, \Gamma'$)
end
```

Figure 3: The XYLEM analysis extended to use stack traces to support our fault-localization approach.

At a call site, the procedure uses summary information, which maps an incoming state Γ to an outgoing state Γ' to which the called method transforms Γ. Using the summary information, the procedure avoids analyzing a method multiple times for the same state. On reaching a call site, the procedure first checks whether summary information exists for the current state (lines 13–15). If no summary exists, the algorithm descends into the called methods to analyze them (lines 16–20). It uses a call stack to ensure context-sensitive processing of called methods.[3] After returning from the called method, the analysis saves the summary information for reuse in subsequent traversals (line 21).

Consider the backward path (30, 28, 27, 33a, 32, 5a, 4b, 26, 25) traversed by the algorithm for the dereference of `string1` at statement 30 in `RTEExample` (see Figure 1). Statement 25 transforms the root predicate $\langle \texttt{string1} = \texttt{null}\rangle$ to $\langle \texttt{string2} = \texttt{null}\rangle$. Continuing backward from 25 along path (24, 4a, 3, 2b, 16, 14, 13), the algorithm reaches statement 13, which transforms predicate $\langle \texttt{string2} = \texttt{null}\rangle$ to $\langle \texttt{true}\rangle$; the algorithm, thus, identifies statement 13 as an NPA.

[3]A *context-sensitive* analysis propagates states along interprocedural paths that consist of valid call–return sequences only—the path contains no pair of call and return that denotes control returning from a method to a call site other than the one that invoked it.

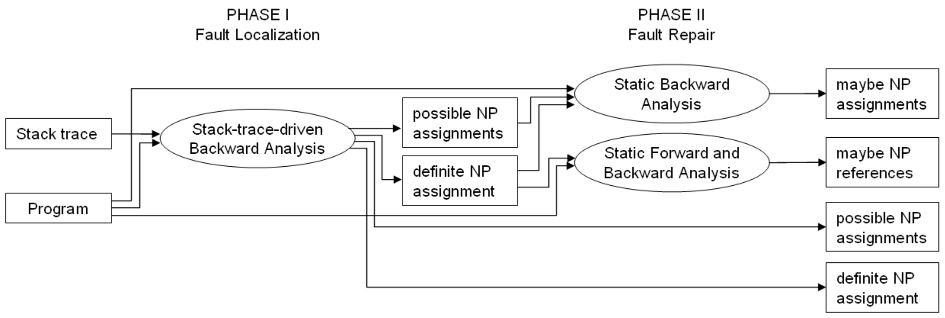

Figure 4: The two phases of the application of our approach to null pointer exceptions.

```
algorithm FaultLoc
 input  $\mathcal{P}$            program
        $ST$                    stack trace for execution $\mathcal{E}$
 output $A_{definite}$          definite NPA for $\mathcal{E}$
        $A_{possible}$          possible NPAs for $\mathcal{E}$
        $A_{maybe}$             maybe NPAs for $\mathcal{E}$
        $R_{maybe}$             maybe NPRs for $\mathcal{E}$
begin
      // Phase 1 analysis: Identify definite and possible NPAs
  1.  let $s_{r(\mathcal{E})}$ be the NPR for $\mathcal{E}$
  2.  let $\mathcal{D}_{cs}$ be a dispatcher that selects the caller in $ST$ for a method
      and all CFG predecessors of a node
  3.  $N_d$ = computeNPA($s_{r(\mathcal{E})}, \mathcal{D}_{cs}$)
  4.  if $|N_d| = 1$ then $A_{definite} = N_d$; else $A_{possible} = N_d$

      // Phase 2 analysis: Identify maybe NPAs and NPRs
  5.  let $\mathcal{D}_a$ be a dispatcher that selects all callers of a method and all
      CFG predecessors of a node
  6.  $N_s$ = computeNPA($s_{r(\mathcal{E})}, \mathcal{D}_a$)
  7.  $A_{maybe} = N_s - N_d$
  8.  if $|A_{definite}| = 1$ then
  9.      let $s_a$ be the NPA in $A_{definite}$
 10.      perform a forward analysis from $s_a$ to identify reachable
          dereferences $R$; mark traversed nodes $\mathcal{N}$
 11.      let $\mathcal{D}_t$ be a dispatcher that selects nodes in $\mathcal{N}$
 12.      foreach $s_r$ in $R$ do
 13.          $N$ = computeNPA($s_r, \mathcal{D}_t$)
 14.          if $N$ contains $s_a$ then
 15.              $R_{maybe} = R_{maybe} \cup s_r$
end
```

Figure 5: Algorithm for performing Phase 1 and Phase 2 analyses of our approach.

On reaching the entry to the method, the algorithm considers calling methods. The algorithm uses the dispatcher to determine which methods to consider (line 24). For the original XYLEM analysis, `analyzeMethod` is recursively called on each caller of the current method (lines 25–27).

3.3 Our Two-phase Algorithm

Figure 4 presents an overview of the application of our approach to support fault localization and fault repair for Java null pointer exceptions. Phase 1 inputs a program and the stack trace from a failed execution, $\mathcal{E}$, of that program in which a null pointer exception was raised. This phase performs a backward analysis, driven by the stack trace, to identify and output definite and possible NPAs. Phase 2 of the approach also inputs the program and stack trace for $\mathcal{E}$, along with the NPAs identified in Phase 1. This phase performs two analyses: a backward analysis to identify and output maybe NPAs; and a combined forward and backward analysis to identify and output maybe NPRs. Both phases of the approach leverage the XYLEM analysis.

Figure 5 presents the algorithm that performs the Phase 1 and Phase 2 analyses of our approach. Given a program $\mathcal{P}$ and the stack trace ST for execution $\mathcal{E}$, the algorithm computes and returns the definite NPA, the possible NPAs, the maybe NPAs, and the maybe NPRs for $\mathcal{E}$. This section discusses each of these phases in turn.

3.3.1 Phase 1: Identifying definite and possible NPAs

To identify definite and possible NPAs for the exception raised at $s_{r(\mathcal{E})}$ for execution $\mathcal{E}$, Phase 1 leverages the modified XYLEM analysis. We refer to the NPAs computed by the XYLEM analysis as *static NPAs* and the NPAs computed by the stack-trace-guided version of the analysis as *dynamic NPAs*.

In Phase 1 (lines 1–4), the algorithm computes dynamic NPAs by invoking `ComputeNPA` with a dispatcher $\mathcal{D}_{cs}$ that guides the analysis along the call chain contained in the stack trace.

Lines 23–27 of `ComputeNPA` (Figure 3) illustrate the stack-trace-guided steps of the analysis. If the current method is not being analyzed in a specific context, the algorithm examines the stack trace and identifies the call site into which it ascends. For example, when the algorithm is invoked for execution 1 shown in Table 1, `ComputeNPA` starts traversal from statement 30. On reaching the entry of `method3()`, it ascends to call site 33a in `method4()` only—it does not ascend to call site 8a in `main()`. On reaching the entry of `method4()`, it ascends to call site 5a; finally, it identifies the null assignment at statement 13. For another example, when invoked with the stack trace for execution 2, the analysis ascends to call site 8a from the entry of `method3()`. At node 7b, it descends into `method1()` and identifies NPAs 19 and 21.

At the end of Phase 1 of our algorithm, there is either one definite NPA or a set of possible NPAs identified (line 4 in Figure 5). Consider again Table 1. For execution 1, only one dynamic NPA (statement 13) is computed; therefore, that NPA is a definite NPA. For execution 2, two dynamic NPAs (statements 19 and 21) are computed; thus, those NPAs are possible NPAs.

The type of dynamic information that is available about the failing execution affects the accuracy of our approach. With more detailed dynamic information, the approach will compute fewer possible NPAs. For example, for executions 2 and 3 in Table 1, the stack traces contain insufficient information for `FaultLoc` to determine which branch from condition statement 18 was taken. However, if branch traces were available for these executions, the algorithm would classify statement 19 as the definite NPA for execution 2 and statement 21 as the definite NPA for execution 3.

3.3.2 Phase 2: Identifying maybe NPAs and NPRs

To identify maybe NPAs for the exception raised at $s_{r(\mathcal{E})}$, and maybe NPRs for the definite NPA at $s_{a(\mathcal{E})}$, Phase 2 of `FaultLoc` (lines 5–15) again leverages the XYLEM analysis.

First, the algorithm uses the XYLEM analysis—passing in a dispatcher $\mathcal{D}_a$ that returns all callers of a method and all CFG predecessors of a node—to identify static NPAs (lines 5–6). The algorithm computes the statements in the set difference between the static and dynamic NPA sets; these statements constitute the maybe

NPAs (line 7). For example, for execution 1 of `RTEExample`, the dynamic NPA set contains statement 13 only; the static NPA set of statement 30 includes statements 13, 19, and 21. The difference of the two sets (statements 19 and 21) forms the maybe NPAs for execution 1.

Next, the algorithm identifies maybe NPRs (for a definite NPA s_a) in two steps. In the first step (line 10), the algorithm performs a forward analysis, starting at s_a, to identify reachable dereferences of the null value generated at s_a. The algorithm marks the nodes that are traversed during the analysis. For example, consider the definite NPA for execution 1 of `RTEExample`. Starting at statement 13, the algorithm walks forward in the ICFG[1] to identify statements that may dereference `string2`, either directly or indirectly through transitive assignments and parameter passing. In this step, the algorithm identifies statements 22 and 29 as the reachable dereferences.

In the second step (lines 12–15), for each NPR identified in the first step, the algorithm leverages the backward path-sensitive XYLEM analysis with the modification that it constrains the analysis (using dispatcher $\mathcal{D}_t$) to follow along only those paths that were traversed in step 1. If for an NPR s_r, s_a is identified as an NPA during the backward analysis, the algorithm adds s_r to the set of maybe NPRs for $\mathcal{E}$.

Continuing with the example of execution 1, the algorithm invokes the XYLEM analysis once each for statements 22 and 29. When executed for statement 22, the analysis identifies statement 13 as an NPA. When executed for statement 29, the analysis—because it is path-sensitive—does not identify statement 13 as an NPA. On traversing edge (28, 29), the analysis collects the predicate $\langle \texttt{val} > 1 \rangle$. Eventually, the analysis reaches statement 16, which has two predecessors (statements 14 and 15). Following back along the in-edge from statement 15, the algorithm encounters the assignment of a new object to `string2`, which invalidates the root predicate $\langle \texttt{string2} = \texttt{null} \rangle$. Along edge (14, 16), the predicate $\langle \texttt{val} \leq 1 \rangle$ is generated, which also causes the state to become inconsistent. Therefore, the analysis traverses no further.

The benefit of the two-step approach for computing maybe NPRs is that the second step, using the XYLEM analysis, can potentially filter out infeasible reachable dereferences that may be identified in the first step.

Note that Phase 2 computes maybe NPRs for only the definite NPA (if any) that was identified in Phase 1. If Phase 1 computes possible NPAs, the developer would first want to identify which of those possible NPAs definitely caused the exception, and then compute the maybe NPRs for that NPA.

4. EVALUATION

To evaluate our approach, we performed both an analytical and an empirical study of it. This section first presents the analytical evaluation, then presents the empirical study, and finally, discusses the limitations and threats to the validity of the evaluation.

4.1 Analytical Study of Our Technique

The goal of this study is to compare our approach with other approaches that locate faults or provide context information for Java null pointer exceptions to determine (1) the differences in the NPA information that they compute and (2) the relative expense involved in computing these NPAs.

To accomplish this goal, we considered the different patterns of occurrences of definite, possible, and maybe NPAs. For each pattern, we determined (1) which of the NPAs could be identified by a path-sensitive, context-sensitive static analysis [10] alone, (2) which of the NPAs could be identified using the stack trace alone, and (3) which of the NPAs could be identified by our technique that combines the two approaches. In cases where different approaches computed the same information, we also considered the relative costs of computing the NPAs. Note that none of the existing techniques provide maybe NPR information. Thus, we did not compare our maybe-NPR analysis with others.

We considered patterns based on four aspects of the NPAs. The first three are the types of NPAs (definite, possible, or maybe), where for each type, there can be no NPAs or some number of NPAs. The fourth is whether the NPA occurs in a method in the stack trace. With these four aspects of NPAs, there are 16 configurations. Eight of them cannot occur. The four that have a definite NPA and possible NPAs as part of the configuration cannot occur because, if a definite NPA is found, there can be no possible NPAs. The four that have no definite NPA and no possible NPAs as part of the configuration also cannot occur because, if a null pointer exception is raised, at least one of them is always identified by the analysis. The other eight configurations are shown in Table 2. In the table, the first column assigns a number to each pattern. The next three columns indicate whether NPAs of that type are identified by the analysis. The fifth column indicates whether the definite/possible NPAs occur in methods in the stack trace. The sixth column shows what would be found using the stack trace alone, the seventh column shows what would be found with the static analysis alone, and the last column shows what our technique would find.

Consider Pattern 1 in which there is a definite NPA, no possible NPAs, no maybe NPAs, and no NPAs in any of the methods in the stack trace. Examining the stack trace alone does not directly reveal the NPA because the NPA does not occur in a method that is in the stack trace. Moreover, the stack trace provides no information about maybe NPAs. Static analysis alone can compute this NPA and, because it identifies only one NPA, it also knows that no maybe NPAs exist. If the definite NPA occurs in a stack method (Pattern 2), the stack trace can be used to identify it. However, finding the NPA by inspecting the stack trace can require significant manual effort and the stack trace provides no information about maybe NPAs. For Patterns 1 and 2, our stack-trace-guided technique provides the same information about NPAs that the static-analysis approach provides. However, our technique can find this NPA more efficiently than the static analysis because it performs the analysis considering only the executed methods, whereas the static analysis must consider all calling methods, and thus, potentially traverses many more paths than our analysis.

In Patterns 3 and 4, there are maybe NPAs along with a definite NPA. In such cases, although static analysis can identify all NPAs, it cannot distinguish the definite NPA from the maybe NPAs. Therefore, it cannot identify which of the NPAs definitely caused the null pointer exception to occur. Our technique can distinguish the two NPA types and, therefore, can help localize the fault and provide context information about related NPAs. In these patterns, the stack trace provides the same information as it does for Patterns 1 and 2. Thus, it may not be efficient for finding the definite NPA and it does not help in finding the maybe NPAs.

In Patterns 5–8, there is no definite NPA. In such cases, static analysis cannot distinguish between possible and maybe NPAs. Consequently, it cannot guide developers by letting them focus on the NPAs that could potentially have caused the null pointer exception, while avoiding the NPAs that definitely could not have caused the exception. The stack trace provides similar information as it does for Patterns 1 and 2—it can help in directly finding the possible NPAs only if the NPAs occur in methods in the stack trace.

Overall, the stack-trace approach is limited in that it cannot provide any information about the occurrences of maybe NPAs. Moreover, if the definite or possible NPAs do not occur in stack methods,

Table 2: **Patterns of occurrences of NPAs, and whether the different NPAs can be identified using (1) only the stack trace, (2) only the static analysis, and (3) our combined technique.**

	Definite NPA	Possible NPA	Maybe NPA	Definite/Possible NPAs in a Method in the Stack Trace	Using Only the Stack Trace		Using Only the Static Analysis	Using Our Combined Technique
1	Yes	No	No	No	Does not find Definite directly	Finds nothing about Maybe(s)	Finds the Definite, Finds that there are no Maybe(s)	Finds the Definite, Finds that there are no Maybe(s)
2	Yes	No	No	Yes	Finds Definite			
3	Yes	No	Yes	No	Does not find Definite directly,		Finds but cannot distinguish the Definite and Maybe(s)	Finds the Definite and distinguishes Maybe(s)
4	Yes	No	Yes	Yes	Finds Definite			
5	No	Yes	No	No	Does not find Possible(s) directly		Finds but cannot distinguish Possible(s) and Maybe(s)	Finds and distinguishes Possible(s) and Maybe(s)
6	No	Yes	No	Yes	May find Possible(s) directly			
7	No	Yes	Yes	No	Does not find Possible(s) directly			
8	No	Yes	Yes	Yes	May find Possible(s) directly			

the approach requires the data flow to be traced backward through called methods to find the NPAs, which may require significant manual effort. The static-analysis approach identifies all NPAs. However, if it computes more than one NPA, it cannot classify the NPA as definite, possible, or maybe. For example, if static analysis computes two NPAs, it cannot determine whether (1) both are possible NPAs or (2) one is a definite NPA and the other is a maybe NPA. In contrast to using these approaches alone, in most cases, our technique (which combines them), is more effective in locating the NPAs that could have caused the null pointer exception. Furthermore, our technique also provides context information in the form of NPAs that may cause the exception in other executions, and can be useful for repairing the fault.

4.2 Empirical Study of Our Technique

To evaluate our technique empirically, we implemented it for null pointer exceptions using the XYLEM tool, and we conducted an empirical study using open-source projects and the BUGZILLA defect reports for those projects. This section first describes our experimental set up and then presents the results of the study.

4.2.1 *Experimental setup*

Each subject for our experiment is a pair that consists of an application and a stack trace for a failing execution of the application. To gather subjects for our study, we used the following process. We browsed the BUGZILLA repository for defect reports that show occurrences of null pointer exceptions. Of these reports, we filtered out those that neither listed the stack trace nor listed the code version on which the failure was observed. For some of the reports, the listed code revisions were unavailable for download, or, if available, the source line numbers did not match the line numbers that appeared in the stack trace; we filtered out such reports as well. We also filtered out a report if the stack trace contained native methods or the JDK API methods.[4] Table 3 lists the 13 executions—(project release, BUGZILLA ID) pairs—that we used for the study. The sizes of the applications in terms of the number of nodes in the ICFG[1] vary from approximately 3,200 for `NanoXML` to over 185,000 for `Ant-1.6.5`.[5]

We implemented our approach using the XYLEM tool [10], which implements the null-dereference analysis described in Section 3.2. The XYLEM tool uses the WALA analysis infrastructure[6] to construct the call graph and the ICFG. XYLEM performs the analysis in two steps. In the first step, it performs points-to analysis, escape analysis, and control-dependence analysis. In the second step, it uses the results of the first step and performs null-dereference analysis. In this step, given a dereference from which to start the analysis, XYLEM performs the backward analysis to identify a path along which a null value can flow to the dereference.

To implement the Phase 1 analysis, we extended the second step of the XYLEM tool to use stack-trace information while performing the program traversal. To implement the Phase 2 analysis, for the computation of maybe NPAs, we parameterized the analysis so that it does not return after identifying the first NPA, but performs a comprehensive traversal for a given NPR. The implementation of this phase currently computes all the maybe NPAs. We are implementing the component of Phase 2 that computes the maybe NPRs.

4.2.2 *Distribution of NPA types and patterns*

The goal of this study is to determine the distribution of NPA types and patterns for program executions in which null pointer exceptions are thrown.

To accomplish this goal, we analyzed each execution using the extended XYLEM tool and computed definite, possible, and maybe NPAs. We also identified the NPAs that occurred in methods in the stack traces. Table 3 presents the data that we collected. The table shows, for each subject, a subject number, the subject version and bug number (columns 1–2), and the number of definite, possible, and maybe NPAs (columns 3–5) that were computed. Column 6 lists the number of definite and possible NPAs that occurred in stack methods, and column 7 shows the pattern of occurrence of the NPAs (Table 2 lists the patterns).

We highlight what the data indicate about the applicability of the three techniques for locating faults and context information: static analysis, stack trace, and our approach.

For five of the 13 executions, for which static analysis computes more than one NPA, our approach provides the benefit of distinguishing definite/possible NPAs from maybe NPAs. For example, for execution 4, our approach classifies the three NPAs that static analysis computes as one definite and two maybes. Similarly, for execution 8, our approach classifies the two NPAs as one definite and one maybe. Static analysis cannot distinguish the NPAs that it computes. Thus, in such cases, using our approach, the developer would know which NPA definitely caused the exception and,

[4]Although we browsed the BUGZILLA repository extensively for several projects, we filtered out many of the bug reports for the reasons stated.

[5]Most of the project distributions come with several jar files. We analyzed only a subset of those jar files based on the contents of the stack traces.

[6]`http://wala.sourceforge.net`

Table 3: Distribution of NPA types for our subjects.

	Subject		Definite NPAs	Possible NPAs	Maybe NPAs	Definite/ Possible NPAs in a Method in the Stack Trace	Pattern type
	Project release	Bug					
1	Ant 1.5	10360	1	0	0	0	1
2	Ant 1.5.1	15994	1	0	0	1	2
3	Ant 1.6.0	25826	1	0	0	1	2
4	Ant 1.6.2	31840	1	0	2	1	4
5	Ant 1.6.3	34878	0	2	0	0	5
6	Ant 1.6.5	38622	1	0	0	0	1
7	Fop 0.92beta	39553	1	0	0	1	2
8	NanoXML	–	1	0	1	1	4
9	Tomcat 5.0.18	27077	1	0	0	1	2
10	Tomcat 5.0.25	29688	1	0	2	1	4
11	Tomcat 5.0.28	32130	1	0	0	1	2
12	Tomcat 5.5.12	37425	0	2	0	0	5
13	Xerces 1.3.0	2252	1	0	0	0	1

therefore, could focus on that NPA to determine why the null value reached the NPR. Using static analysis alone, the developer would have to identify first which of the static NPAs is the definite NPA, before investigating further why the exception occurred.

For the remaining eight executions, static analysis computes one NPA—these correspond to Patterns 1 and 2. For these cases, our approach provides the same information about NPA occurrences as static analysis. For five of the executions, the definite/possible NPAs do not appear in a stack method. For such cases, the approach of inspecting stack methods would require manual tracing of data values back through called methods to identify the NPAs.

4.2.3 *Effort required to locate NPAs*

The goal of this study is to illustrate the effort that might be required, using alternative approaches, to locate the NPA that assigned the null value.

To do this, we discuss two example executions from Table 3. First, consider execution 5 from the table. For this execution, our approach identifies two possible NPAs and no maybe NPAs (Pattern 5). The stack trace of that execution shows that the null dereference occurs at line 875 in method `checkIncludePatterns()` of class `DirectoryScanner`:

```
...ant.DirectoryScanner.checkIncludePatterns 875
...ant.DirectoryScanner.scan 808
...ant.types.AbstractFileSet.getDirectoryScanner 358
...ant.taskdefs.Copy.execute 404
...ant.UnknownElement.execute 275
...ant.Task.perform 364
...ant.Target.execute 341
...ant.Target.performTasks 369
...ant.Project.executeSortedTargets 1216
...ant.Project.executeTarget 1185
...ant.helper.DefaultExecutor.executeTargets 40
...ant.Project.executeTargets 1068
...ant.Main.runBuild 668
...ant.Main.startAnt 187
...ant.launch.Launcher.run 246
org.apache.tools.ant.launch.Launcher.main 67
```

Line 875 of `checkIncludePatterns()` contains a dereference of a variable that is assigned the return value from a method:

```
checkIncludePatterns()
  [874] File f = findFile(...);
  [875] if (f.exists()) {
```

To locate the NPA, the developer would have to trace back into the called method `findFile()` to see whether the method may return a null value. Method `findFile()` in turn returns the value from another method. On examining method `findFile2()`, the developer would find that it can return null values at two statements.

```
findFile()
  [1504] return findFile2(...);
findFile2()
  [1522] if (!base.isDirectory()) {
  [1533]    return null;
   ...
  [1541] return null;
```

Next, consider execution 10 from Table 3 for which our approach computes a definite NPA (that is on the stack trace), and two maybe NPAs (Pattern 4). The partial stack trace for the execution, along with the source line containing the NPR, are shown below:

```
...catalina.startup.HostConfig.deployDescriptors 445
...catalina.startup.HostConfig.deployApps 427
...catalina.startup.HostConfig.check 1064
...catalina.startup.HostConfig.lifecycleEvent 327
...catalina.util.LifecycleSupport.fireLifecycleEvent 119
...catalina.core.StandardHost.backgroundProcess 800
...
deployDescriptors(..., String[] files)
  [445]  for ( int i=0; i<files.length; i++ )
```

The code for `deployDescriptors()` shows that the statement at line 445 dereferences a formal parameter. The stack trace illustrates that `deployDescriptors()` is called from `deployApps()`, which passes in a null value for the formal parameter `files`. Line 426 shown below returns the value from a JDK API method, which in this execution was null.

```
deployApps()
  [426] String configFiles[] = configBase.list();
  [427] deployDescriptors(configBase,configFiles);
```

In this case, the stack trace contains sufficient information to identify the definite NPA, and the developer does not have to navigate to other methods (that are not on the stack) to trace the flow of the null value. However, the stack provides no information that there are two other callers of `deployDescriptors()` that can also potentially pass in null values, as shown in the following code:

```
deployWARs()
  [556] deployDescriptors(configBase(),configBase.list());
start()
  [970] String configFiles[] = configBase.list();
  [971] deployDescriptors(configBase,configFiles);
```

One of the key strengths of our approach is that it identifies context information, in the form of maybe NPAs and maybe NPRs, that can help the developer in fixing a fault.

4.3 Discussion

Our evaluation shows that, for the subjects that we studied, our two-phase algorithm, applied to null pointer exceptions, can be more effective in locating the null pointer assignment than using either static analysis or the stack trace alone.

However, there are several threats to the validity of the evaluation. Threats to internal validity arise when factors affect the dependent variables without the researchers' knowledge. In our case, our implementation could have flaws that would affect the accuracy of the results we obtained. However, we are confident in the results we obtained for two reasons. First, our implementation is based on the XYLEM tool [10] that has been used by industrial practitioners, and has been used for significant experimentation. Second, we manually checked most of the results to verify their correctness.

Threats to external validity arise when the results of the experiment cannot be generalized to other situations. One external threat concerns the ability to generalize the results for null pointer exceptions, based on the subjects we used. In our study, we used 13 programs and runtime exceptions, and thus, we are unable to conclude that our results will hold for programs in general. In the current implementation, when a reference that is retrieved from a collection class (e.g., Hashtable) causes a null pointer exception, we report the statement that retrieves the reference from the collection as an NPA. Future extensions to our tool could perform additional analysis to identify the statements where the null value was generated and then added to the collection. Furthermore, our implementation does not handle reflection. Future enhancements could incorporate dynamic information from the stack trace to guide static analysis in resolving reflection. With some of these improvements to our implementation, studies can be performed on more and varied programs to validate our results.

Another external threat concerns the ability to generalize the results for other types of runtime exceptions that are based on assignment and flow of an incorrect value. Our implementation, and thus our empirical evaluation, applies to null pointer exceptions only. Therefore, we cannot definitively state that our approach will achieve the same results when applied to other types of exceptions. Only experiments with implementations of other such analyses can answer this question.

5. RELATED WORK

There is much research in the area of locating program faults that is related to our work.

Many techniques have been developed for finding common bugs, including null dereferences. Fähndrich and Leino [3] propose an approach to retrofit languages, such as Java, by declaring two types of references—possibly-null and non-null. The non-nullity of a reference is enforced by the type system at compile time, through the use of object invariants. Thus, their technique addresses the null-dereference problem by modifying existing languages to incorporate new reference types. In contrast, our technique is applicable to Java without modification. Furthermore, our technique provides context information to assist in repairing the fault.

Xie and Engler [17] propose an approach that uses redundancy checkers to find hard errors, such as null dereferences, potential deadlocks, and security violations, that may crash a system. Their approach first finds redundancy errors using redundancy checkers, and then finds correlations between these errors and other hard errors using a statistical technique. Unlike their approach, our technique uses a combination of static and dynamic analysis to assist not only in locating the faults, but also in repairing the faults. Thus, it avoids the limitations of a purely static approach.

Other techniques use static error-detection methods that analyze a program to detect potential faults without executing it. Hovemeyer, Spacco, and Pugh [7] use forward interprocedural dataflow analysis and annotations to find bugs related to null-pointer dereferences. Evans [2] created LCLINT to incorporate annotations that help in detecting errors, such as uses of dead storage and dangerous aliasing, in C programs. ESC/JAVA [4] is a compile-time program checker that finds inconsistencies and potential runtime errors between the programmer's design decisions expressed in an annotation language and the actual code. Unlike our technique, the above techniques require user-provided program annotations. Thus, the quality of the fault localization they provide depends on the annotations. Furthermore, their approaches provide no context information for repairing the fault.

Loginov and colleagues [9] present a sound null-dereference analysis, based on abstract interpretation, that gradually expands the interprocedural scope of analysis to establish the safety of a dereference. Unlike their approach, our approach performs a backward analysis (starting at a point of failure) that is guided by the stack trace, and provides context information to assist with fault repair.

PREFIX [1] performs accurate interprocedural analysis for detecting a broad class of memory errors, including null dereferences, in C programs. The technique features a bottom-up analysis of procedures to compute summaries, and a forward path-sensitive analysis within each procedure that prunes out infeasible paths. However, that approach is purely static. Thus, unlike our technique, it suffers from the common problems of purely static approaches. Also, it does not compute specific context information that can help in repairing a fault.

Rountev, Kagan, and Gibas [12] discuss imprecision of static analysis and propose approaches for evaluating such imprecision. A common approach for addressing the imprecision problem of static analysis is to combine static analysis with dynamic analysis. Our technique adopts this approach—it combines the dynamically generated information (from the stack trace) with the static analysis performed by XYLEM.

Tomb, Brat, and Visser [15] propose a technique for finding runtime errors in Java programs by combining static and dynamic analyses. The approach performs forward interprocedural symbolic execution to find constraints that may reveal a bug, and then attempts to generate test cases to trigger that bug. Our approach is similar in that it also leverages an interprocedural path-sensitive analysis. However, our analysis is backward; it starts from a statement where it knows that a bug has caused a failure, and it uses the available runtime information, such as stack traces, to guide the backward analysis. Additionally, our technique provides context information for assistance in repairing the fault.

Hangal and Lam [6] developed a tool, DIDUCE, that uses dynamic program invariant detection to find the root causes of software bugs. Their approach attempts to infer bugs by dynamic program invariant detection but it provides no information for repairing other similar bugs in the program. Furthermore, gathering of dynamic invariants is an expensive analysis that may not scale to large programs. Our approach, in contrast, uses readily available, lightweight, dynamic information—the stack trace—to locate faults.

Program slicing [16] is a widely studied technique for debugging. A static program slice [16], computed with respect to the dereferenced variable v at an NPR, identifies all statements that could affect the value of v, which includes all NPAs. However, the developer would be faced with the task of examining the slice to identify the definite NPA. A dynamic slice [8], computed with respect to the execution trace of a failing execution $\mathcal{E}$, excludes statements, and NPAs, that do not execute in $\mathcal{E}$. Thus, it could exclude some of the possible NPAs identified by our stack-trace-guided approach. However, dynamic slicing requires the program to be rerun, with instrumentation, on the failing inputs. Moreover, for applications that are long-running or that use concurrency, dynamic slicing would not be practical. Our approach uses readily available stack-trace information, and requires no reexecution of the program.

6. CONCLUSION AND FUTURE WORK

In this paper, we presented a new approach for locating and repairing faults that cause runtime exceptions in Java programs due to incorrect assignment of a value that finally leads to the exception. Our approach assists in fault localization: it performs a backward stack-trace-guided dataflow analysis, starting at the point where the exception was raised, to locate the source statement that is responsible for the exception. Our approach also assists in fault repair: it performs backward and forward analysis to identify other statements that are related to the exception and its cause, and that should be considered while fixing the fault that caused the exception.

We applied our approach to Java null pointer exceptions to identify three types of null pointer assignments (definite, possible, and maybe) and one type of null pointer dereference (maybe). Our analysis is based on a static backward path-sensitive, context-sensitive null-dereference analysis [10] that we modified to consider the dynamic, stack-trace information in performing the analysis. By using the dynamic information to guide the static analysis, our technique can achieve better results than either the static analysis or the inspection of the stack trace alone. Our analytical study and our empirical study support this claim.

However, as discussed in Section 4.3 there are several areas of future work that we plan to conduct that will let us perform additional studies with a greater number, and a more diverse set, of programs and runtime exceptions. This work includes providing extensions to the analysis to include features of Java programs, such as use of containers, libraries, and reflection. The work also includes applying our approach to other types of runtime exceptions that are based on assignment and flow of an incorrect value.

The current implementation of our two-phase algorithm is an implementation of Phase 1 to find definite and possible NPAs and the static backward analysis component of Phase 2 to find maybe NPAs. We have partially implemented the static forward and backward analysis component of Phase 2 that will compute the maybe NPRs. We plan to complete this implementation, and extend our studies to include identification of these NPRs. Currently in our empirical evaluation, we note whether the NPAs are contained in a method that is in the stack trace. However, we cannot assess the difficulty developers may have in locating the NPAs. Additionally, although we report the maybe NPAs that our algorithm identifies, we cannot assess how useful this information will be to developers in repairing the fault. With the completion of the implementation of our two-phase algorithm, we plan to conduct user studies to assess both the difficulty in locating the NPAs and the usefulness of the additional context information.

Identifying the NPA is a key first step in investigating the cause of a runtime exception. After identifying the NPA, the developer needs to understand whether (1) the NPA should not have been reached at all in the execution; (2) if reached, the null value should have been overwritten along the path to the NPR; or (3) the NPR should not have been reached from the NPA. In the first case, the fault lies before the NPA, whereas, in the second and third cases, the fault lies along the paths between the NPA and the NPR. Thus, developing analyses to support the developer in performing this task is an important extension to our approach, which we plan to address in future work.

Another aspect of our current technique is its use of the stack-trace to guide the analysis to find the NPAs. However, in some cases, this dynamic information is insufficient to identify the definite NPA that caused the null pointer exception. If Phase 1 of our analysis identifies only possible NPAs and the inputs that cause the exception are available,[2] we can use those NPAs to instrument the program to determine which of those possible NPAs was actually executed. By executing the program again, with this lightweight instrumentation, our technique can determine the definite NPA that caused the exception. We plan to implement this instrumentation technique and evaluate its efficiency.

A final area of future work involves creating effective ways to present the results to developers. Currently, we present the results of our analysis as plain-text output to the developer. Because of the distributed nature of exception flow and the complexity of the problem, textual outputs are not always easy to understand, and an interactive visualization may help developers explore and understand the analysis results. In previous work [13], some of the authors developed a technique for visualizing exception-handing constructs and the related flow at three different abstract levels. A similar interactive visualization could support developers in the fault-localization and fault-repair process.

Acknowledgements

This work was supported in part by awards from National Science Foundation under CCF-0429117, CCF-0541049, and CCF-0725202, IBM by a Software Quality Innovation Faculty Award, Key Project of Chinese Ministry of Education under 108063, and Natural Science Foundation of Jiangsu Province China under BK-2008124.

References

[1] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software—Practice and Experience*, 30(7):775–802, 2000.

[2] D. Evans. Static detection of dynamic memory errors. In *Proceedings of the ACM SIGPLAN Conference on Programming Languages, Design, and Implementation*, pages 44–53, May 1996.

[3] M. Fähndrich and K. R. M. Leino. Declaring and checking non-null types in an object-oriented language. In *Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 302–312, Oct 2003.

[4] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 234–245, June 2002.

[5] C. Fu and B. G. Ryder. Exception-chain analysis: Revealing exception handling architecture in Java server applications. In *Proceedings of the 29th International Conference on Software Engineering*, pages 230–239, May 2007.

[6] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the International Conference on Software Engineering*, pages 291–301, May 2002.

[7] D. Hovemeyer, J. Spacco, and W. Pugh. Evaluating and tuning a static analysis to find null pointer bugs. In *Proceedings of the 2005 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 13–19, September 2005.

[8] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, Oct. 1988.

[9] A. Loginov, E. Yahav, S. Chandra, N. Fink, S. Rinetzky, and M. G. Nanda. Verifying dereference safety via expanding-scope analysis. In *Proedings of the International Symposium on Software Testing and Analysis*, pages 213–223, July 2008.

[10] M. G. Nanda and S. Sinha. Accurate interprocedural null-dereference analysis for Java. In *Proceedings of the 31st International Conference on Software Engineering*, pages 133–143, May 2009.

[11] M. P. Robillard and G. C. Murphy. Static analysis to support the evolution of exception structure in object-oriented systems. *ACM Transactions on Software Engineering and Methodology*, 12(2):191–221, 2003.

[12] A. Rountev, S. Kagan, and M. Gibas. Evaluating the imprecision of static analysis. In *Proceedings of the 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 14–16, June 2004.

[13] H. Shah, C. Görg, and M. J. Harrold. Visualization of exception handling constructs to support program understanding. In *Proceedings of the ACM Symposium on Software Visualization*, pages 19–28, September 2008.

[14] S. Sinha, A. Orso, and M. J. Harrold. Automated support for development, maintenance, and testing in the presence of implicit control flow. In *Proceedings of the 26th International Conference on Software Engineering*, pages 336–345, May 2004.

[15] A. Tomb, G. P. Brat, and W. Visser. Variably interprocedural program analysis for runtime error detection. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 97–107, July 2007.

[16] M. Weiser. Program slicing. *IEEE Trans. Softw. Eng.*, 10(4):352–357, July 1984.

[17] Y. Xie and D. R. Engler. Using redundancies to find errors. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 51–60, November 2002.