Automated Bug Neighborhood Analysis for Identifying Incomplete Bug Fixes

Conference Paper · January 2010

DOI: 10.1109/ICST.2010.63 · Source: DBLP

6 authors, including:

Hina Shah
Georgia Institute of Technology
13 PUBLICATIONS **219** CITATIONS
SEE PROFILE

Mary Jean Harrold
Georgia Institute of Technology
188 PUBLICATIONS **11,718** CITATIONS
SEE PROFILE

Mangala Gowri Nanda
IBM
38 PUBLICATIONS **870** CITATIONS
SEE PROFILE

2010 Third International Conference on Software Testing, Verification and Validation

Automated Bug Neighborhood Analysis for Identifying Incomplete Bug Fixes

Mijung Kim*, Saurabh Sinha†, Carsten Görg*, Hina Shah*, Mary Jean Harrold*, and Mangala Gowri Nanda†
*Georgia Institute of Technology
Email: {mijung.kim|goerg|hinashah|harrold}@cc.gatech.edu
†IBM Research – India
Email: {saurabhsinha|mgowri}@in.ibm.com

***Abstract*—Although many static-analysis techniques have been developed for automatically detecting bugs, such as null dereferences, fewer automated approaches have been presented for analyzing whether and how such bugs are fixed. Attempted bug fixes may be incomplete in that a related manifestation of the bug remains unfixed. In this paper, we characterize the "completeness" of attempted bug fixes that involve the flow of invalid values from one program point to another, such as null dereferences, in Java programs. Our characterization is based on the definition of a bug neighborhood, which is a scope of flows of invalid values. We present an automated analysis that, given two versions P and P' of a program, identifies the bugs in P that have been fixed in P', and classifies each fix as complete or incomplete. We implemented our technique for null-dereference bugs and conducted empirical studies using open-source projects. Our results indicate that, for the projects we studied, many bug fixes are not complete, and thus, may cause failures in subsequent executions of the program.**

I. Introduction

Java programs often contain bugs (or faults), such as dereferences of null values and array accesses with incorrect index values, that cause the Java Virtual Machine to throw runtime exceptions. These bugs involve the flow of invalid values from one program point to another program point where the invalid values cause runtime exceptions. Although many automated techniques have been developed to detect such bugs statically (e.g., [1], [2], [3], [4], [5], [6]), fewer techniques have been presented for identifying whether and how such bugs get fixed [7][8]. An attempted bug fix may be "incomplete" in that the fix leaves related manifestations of a bug that could occur in other executions unfixed.

To illustrate the problem of incomplete bug fixes, consider a null-pointer exception, which occurs because a null-pointer assignment reaches a null-pointer dereference. 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-pointer dereference* (NPR) is a statement at which the dereferenced variable could potentially be null. In Figure 1, the gray box in the graph on the left depicts a null-pointer exception that has occurred in program P where null-pointer assignment NPA_1 reaches null-pointer dereference NPR_1. NPR_2 represents a null-pointer dereference that may be reached by NPA_1 in P', depending on how NPR_1 is fixed, and NPR_3 represents a null-pointer dereference that can be reached from NPA_1 in another execution of P or P'.

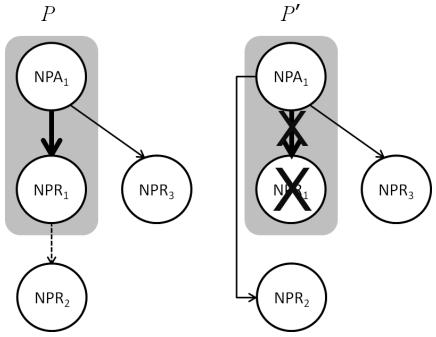

Figure 1. Example to illustrate an incomplete bug fix.

Suppose a developer tries to fix this null-pointer exception by adding a null check around NPR_1 to produce P'. As the graph on the right in Figure 1 shows, this change removes the flow of the null value from NPA_1 to NPR_1, and it guarantees that an exception will not occur at NPR_1 on any execution of P'. However, this fix is incomplete in that other exceptions may still occur because of NPA_1. Because NPR_1 no longer throws an exception, the null value at NPA_1 may now reach NPR_2, and cause an exception. Because the null value at NPA_1 can still flow to NPR_3, NPA_1 may reach NPR_3 on another execution, and cause an exception. We consider an attempted bug fix, with respect to a pair (NPA, NPR) to be *incomplete* if either the NPA or the NPR could cause an exception to occur in another execution of P'.

Existing research that has investigated bug fixes (e.g., [7], [8]) does not consider this completeness aspect of a bug fix. Ayewah and colleagues [7] studied, for three software systems, the bugs that were present in one build, but not reported in the next build. They also examined the types of program changes that caused the bugs to be deleted. However, they did not investigate whether the bug fixes were complete. Similarly, Spacco, Hovemeyer, and Pugh [8] examined the evolution of bugs over successive releases of Sun's JDK core runtime library, but they performed no evaluation of how the bugs that appeared in one version and not in a subsequent version were fixed. However, in our empirical studies, we found many instances in which a bug

978-0-7695-3990-4/10 $26.00 © 2010 IEEE
DOI 10.1109/ICST.2010.63

was fixed in an incomplete way. In this paper, we address these limitations of previous approaches.

First, we present a characterization of completeness of a bug fix. Our characterization is based on the definition of a bug neighborhood, which intuitively is a set of related flows of invalid values. For an attempted bug fix to be complete, all related value flows in the bug neighborhood of the flow must be fixed as well. We describe different ways in which the bug neighborhood may be affected by code changes that are targeted toward fixing a bug. Our technique is applicable to the class of bugs that involves the flow of an invalid value from one program point to another, where the value causes a runtime exception. Examples of such exceptions include null-pointer exceptions, array-index exceptions, and class-cast exceptions. In this paper, we illustrate the application of the approach to null-pointer exceptions.

Second, we present an automated analysis that identifies whether an (NPA, NPR) pair in program P has been fixed in a modified version P' of the program. For an attempted bug fix, the analysis also identifies whether the fix is complete according to our definition of completeness. Given an (NPA, NPR) pair in P, the analysis technique identifies the matching pair (NPA$'$, NPR$'$) in P'.[1] Next, the technique determines whether there are other NPRs in P' related to NPA$'$ and other NPAs in P' related to NPR$'$ that could cause null-pointer exceptions in other executions of P'. To do this, the technique performs a backward analysis from NPR$'$ and a forward analysis from NPA$'$, and determines whether a bug fix has been attempted with respect to (NPA, NPR). If a fix has been attempted, it classifies the fix as complete or incomplete. This step of the analysis leverages the interprocedural null-dereference analysis implemented in a tool called XYLEM [5] and extensions to that analysis [9], which we presented in previous work.

In this paper, we also present the results of empirical studies that we performed using open-source and commercial software. Our studies show that bug neighborhoods can be quite large and that there are many instances in which bug fixes are incomplete. These studies illustrate the need for a technique like ours that can alert the developer to attempted bug fixes that are incomplete and that can cause exceptions to occur in other executions.

The main benefit of our technique is that it can automatically detect incomplete bug fixes, and highlight parts of the program that should be examined, and potentially changed, to ensure that a fix is complete. Our technique could be implemented in an interactive debugging tool that, for a given bug, suggests to the developer the types of fixes that would be complete, and identifies the parts of the program that should be examined to implement the fix. Thus, using the technique, bug fixes can be made more effective.

[1]It is possible that the NPA, the NPR, or both are deleted and, therefore, do not occur in P'.

The main contributions of the paper are:

- An approach for classifying attempted bug fixes as complete or incomplete that provides information to assist the developer in getting a complete fix.
- An implementation of the technique for null-dereference bugs.
- Empirical results that show that, for the subjects we studied, bug neighborhoods for (NPA, NPR) pairs are large in size and varied in type, and that many attempted bug fixes are incomplete.

II. Completeness of an attempted bug fix

In this section, we define completeness of an attempted bug fix for a null-pointer dereference in a Java program. First, we present the definition and analysis of a bug neighborhood, and use the bug neighborhood to define a complete fix (Section II-A). Then, we illustrate the potential resulting bug neighborhoods after various kinds of fixes are attempted for a null-pointer bug (Section II-B).

A. Bug neighborhoods and complete fixes

A null-dereference bug occurs with respect to an (NPA, NPR) pair in a program P. As we illustrated in the Introduction, an attempted fix for such a bug can leave unresolved manifestations related to the pair in the modified version P', although it removes the current bug in this execution. Those manifestations, which could cause null-pointer exceptions related to the (NPA, NPR) in other executions, constitute a bug neighborhood for the (NPA, NPR) pair.

Figure 2 illustrates the concept of a bug neighborhood. The figure shows that the bug neighborhood for (NPA_1, NPR_1) can contain (NPA, NPR) pairs consisting of four types of NPAs and NPRs related to (NPA_1, NPR_1): Maybe NPAs, Maybe NPRs, Forward NPRs, and Backward NPAs. In the figure, Maybe NPAs and Maybe NPRs are connected to NPR_1 and NPA_1, respectively, with solid edges to indicate that they are reachable in other executions. Forward NPRs and Backward NPAs are connected to NPR_1 and NPA_1, respectively, with dashed edges to indicate that they may be reachable in other executions, depending on how the (NPA_1, NPR_1) is fixed. Forward NPRs and Backward NPAs are also connected to Maybe NPRs and Maybe NPAs.

Given (NPA_1, NPR_1), our bug-neighborhood analysis automatically identifies the other NPAs and NPRs that induce its bug neighborhood.

1) Maybe NPRs: Maybe NPRs are those null-pointer dereferences that might be reached by NPA_1 in another execution if the (NPA_1, NPR_1) pair is not fixed completely. Our technique performs a forward analysis from NPA_1 to find these Maybe NPRs. In Figure 2, there are two Maybe NPRs associated with NPA_1—Maybe NPR_2 and Maybe NPR_3—resulting in (NPA_1, Maybe NPR_2) and (NPA_1, Maybe NPR_3) being added to the bug neighborhood for (NPA_1, NPR_1).

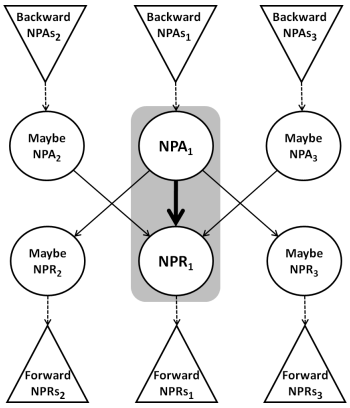

Figure 2. Bug neighborhood for (NPA_1,NPR_1) pair.

2) Maybe NPAs: Maybe NPAs are analogous to Maybe NPRs, and are those null-pointer assignments that might reach NPR_1 in another execution if the (NPA_1, NPR_1) pair is not fixed completely. Our technique performs a backward analysis from NPR_1 to find these Maybe NPAs. In Figure 2, there are two Maybe NPAs associated with NPR_1—Maybe NPA_2 and Maybe NPA_3—resulting in (Maybe NPA_2, NPR_1) and (Maybe NPA_3, NPR_1) being added to the bug neighborhood for (NPA_1, NPR_1).

3) Forward NPRs: Forward NPRs are NPRs that may become reachable from NPA_1 if the program is changed so that NPR_1 is no longer an NPR. These Forward NPRs are *masked* by NPR_1: if NPR_1 is changed so that it cannot be reached by NPA_1 in P', these Forward NPRs may now be reachable from NPA_1. Additionally, Forward NPRs can mask other Forward NPRs. Thus, Forward NPRs are computed transitively until no possible NPRs can be reached from NPA_1 because of fixes to NPR_1 and to other Forward NPRs. This transitive step results in a tree of Forward NPAs rooted at NPR_1, which is represented in Figure 2 by the triangle under NPR_1. For n Forward NPRs, (NPA_1, Forward NPR_1), ... , (NPA_1, Forward NPR_n) are added to the bug neighborhood for (NPA_1, NPR_1).

To obtain a complete fix, the bug-neighborhood analysis also requires a similar computation of Forward NPRs associated with Maybe NPRs. Otherwise, even if (NPA_1, NPR_1) has been fixed, it is possible that the Forward NPRs associated with Maybe NPRs can be reached, and they may cause other null-pointer exceptions after the attempted fix. This computation generates additional (NPA_1, Forward NPR_i) pairs, and adds them to the bug neighborhood for (NPA_1, NPR_1).

4) Backward NPAs: Backward NPAs are analogous to Forward NPRs, and are NPAs that may reach NPR_1 if the program is changed so that NPA_1 no longer generates a null value. These Backward NPAs are *masked* by NPA_1: if NPA_1 is changed so that it cannot reach NPR_1 in P', these Backward NPAs may now reach NPR_1. Additionally, Backward NPAs could mask other Backward NPAs. Thus, Backward NPAs are computed transitively until no possible NPAs can reach NPR_1 because of fixes to NPA_1 and to other Backward NPAs. This transitive step results in a tree of Backward NPAs rooted at NPA_1, which is represented in Figure 2 as the triangle above NPA_1. For n Backward NPAs, (Backward NPA_1, NPR_1), ... , (Backward NPA_n, NPR_1) are added to the bug neighborhood (NPA_1, NPR_1).

Like Forward-NPR computation, computing the bug neighborhood also requires computing Backward NPAs associated with Maybe NPAs. This computation generates additional (Backward NPA_i, NPR_1) pairs, and adds them to the bug neighborhood for (NPA_1, NPR_1).

5) Definitions: We now define bug neighborhood and complete fix for a null-reference bug represented by the pair (NPA_1, NPR_1).

Definition 1: A *bug neighborhood* for (NPA_1, NPR_1) is the set of (NPA, NPR) pairs that are induced by manifestations of Maybe NPAs, Maybe NPRs, Forward NPRs, and Backward NPAs related to (NPA_1, NPR_1). The size of the bug neighborhood for (NPA_1, NPR_1) is the number of pairs in the neighborhood.

Definition 2: An attempted bug fix (i.e., change from P to P') for (NPA_1, NPR_1) in P is a *complete fix* if the bug neighborhood for the associated pair in P' is empty. Note that our analysis does not determine correctness of fixes. Thus, it may be possible that a complete fix is incorrect.

B. Effects of attempted fixes on bug neighborhoods

To illustrate the way in which an attempted bug fix can change the neighborhood, we present four general types of attempted fixes for an (NPA, NPR) pair: removing both the NPA and the NPR, removing the flow between the NPA and the NPR, removing only the NPA, and removing only the NPR. For each type, we provide examples of changes that we have seen in practice to fix the (NPA, NPR) pair, show how the change affects the bug neighborhood, and discuss how the computation of the bug neighborhood is affected for specific types of attempted fixes.

1) Removing both the NPA and the NPR: A fix for an (NPA, NPR) pair might result in the deletion of both the NPA and the NPR. This fix could occur, for example, if the method containing the pair is deleted from P in creating P'. In such cases, no null-pointer exceptions associated with either the NPA or the NPR could occur in P'. Therefore, the bug neighborhood for (NPA, NPR) in P' will be empty, and thus, the fix is a complete fix.

2) Removing the flow between the NPA and the NPR: A fix for an (NPA, NPR) pair might involve removing the flow from the NPA to the NPR, so that the NPR is not reachable from the NPA in P'. This fix could occur, for example, if the NPA is in method M_i, the NPR is in method M_j, and the method call in M_i to M_j, on which the NPA flows, is deleted from P in creating P'. In this case, there may still be Maybe

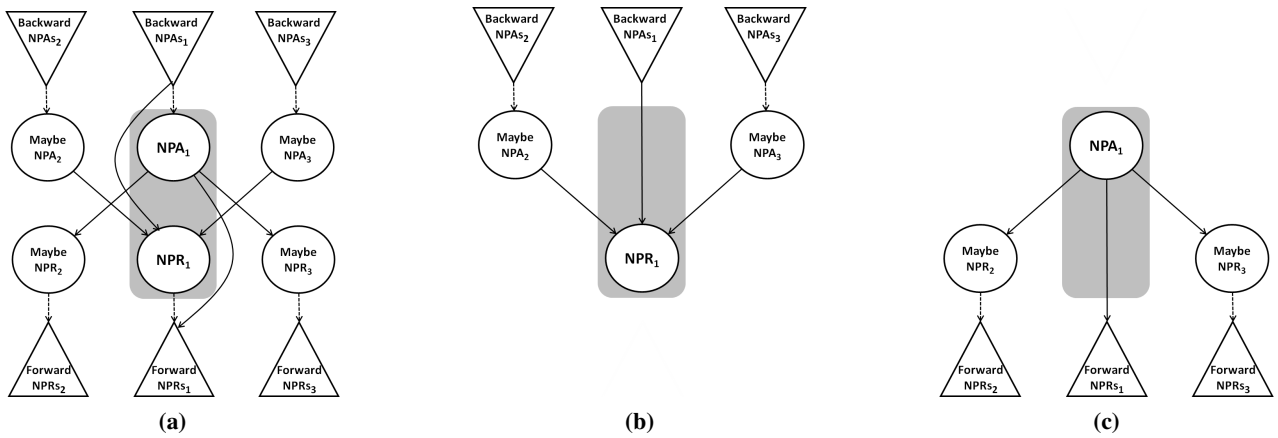

Figure 3. Potential bug neighborhoods after attempted fixes: (a) removing the flow between the NPA and the NPR, (b) removing only the NPA, and (c) removing only the NPR.

NPAs and Maybe NPRs to consider, and thus, the analysis will search for them in P'. Additionally, because neither the NPA nor the NPR is removed, the Backward NPAs and Forward NPRs must be considered in the computation of the bug neighborhood.

Figure 3(a) illustrates the potential components of the bug neighborhood that result for this type of attempted fix. In the figure, the flow between NPA_1 and NPR_1 has been removed by the change. Because of this, Backward NPAs_1 may now reach NPR_1 and Forward NPRs_1 may now be reached from NPA_1; all these must be considered in the computation of the bug neighborhood. Moreover, the Maybe NPRs reachable from NPA_1, (i.e., Maybe NPR_2 and Maybe NPR_3), along with their Forward NPRs, must be considered. Similarly, the Maybe NPAs that reach NPR_1 (i.e., Maybe NPA_2 and Maybe NPA_3), along with their Backward NPAs, must be considered.

3) Removing only the NPA: A fix for an (NPA, NPR) pair might involve removing only the NPA so that this NPA cannot reach the NPR in P'. For example, this fix could occur if the NPA is deleted from P in creating P'. For another example, this fix could occur if a new object is created at the NPA so that a null value cannot flow from this location to NPR in P'. In this case, there may still be Maybe NPAs to consider because they could reach the NPR. However, because the NPA has been removed, there are no Maybe NPRs or Forward NPRs to consider. Thus, the analysis for computing the bug neighborhood is reduced to searching only for components of the bug neighborhood related to the NPR.

Figure 3(b) illustrates the potential components of the bug neighborhood that result for this type of attempted fix. In the figure, NPA_1 has been removed by the change. Because NPA_1 is removed, Backward NPAs_1 may now reach NPR_1, and must be considered. The figure also shows that the Maybe NPAs that reach NPR_1, along with the Backward NPAs associated with them, must also be considered.

4) Removing only the NPR: A fix for an (NPA, NPR) pair might involve removing only the NPR so that this NPR cannot be reached by the NPA in P'. For example, this fix could occur if the NPR is deleted from P in creating P'. For another example, this fix could occur if a condition is added around the NPR in P' so that the NPA cannot flow to the NPR in P'. In this case, there may still be Maybe NPRs to consider because they could be reached from the NPA. However, there are no Maybe NPAs or Backward NPAs to consider. Thus, the analysis for computing the bug neighborhood is reduced to searching only for components of the bug neighborhood related to the NPA.

Figure 3(c) illustrates the potential components of the bug neighborhood that result for this type of attempted fix. In the figure, NPR_1 has been removed by the change. Therefore, Forward NPRs_1 may now be reached by NPA_1, and must be considered. The figure also shows that the Maybe NPRs that are reachable from NPA_1, and the Forward NPRs associated with them, must also be considered.

III. Automated analysis

In this section, we present the bug-neighborhood analysis that, given a program P and a modified version P' of P, identifies the null-dereference bugs in P for which fixes have been attempted in P', and classifies each attempted fix as complete or incomplete. The analysis leverages the null-dereference analysis, implemented in XYLEM [5]. First, we provide an overview of the XYLEM analysis (Section III-A); Reference [5] contains details. Then, we present the bug-neighborhood analysis (Section III-B).

Before presenting the analysis, we present an example we use to illustrate the XYLEM and the bug-neighborhood analyses. In Figure 4, the code fragment on the left shows the original version of function `foo()`; the fragment on the right shows a modified version `foo'()` in which a null check has been added at line 6.

```
   foo( int i, j ) {
[1]   x = null; // NPA
[2]   if ( i == 0 )
[3]      x = new C();
[4]   y = x;
[5]   if ( j > 10 ) {

[7]      x.m1(); // NPR
[8]      x.m2(); // F-NPR
      } else {
[9]      if ( i == 0 )
[10]        y.m();
[11]     x.m3(); // M-NPR
[12]     x.m4(); // M-F-NPR
      }
   }
```

```
   foo'( int i, int j ) {
[1]   x = null; // NPA
[2]   if ( i == 0 )
[3]      x = new C();
[4]   y = x;
[5]   if ( j > 0 ) {
[6]      if ( x != null )
[7]         x.m1();
[8]      x.m2(); // F-NPR
      } else {
[9]      if ( i == 0 )
[10]        y.m();
[11]     x.m3(); // M-NPR
[12]     x.m4(); // M-F-NPR
      }
   }
```

Figure 4. Example to illustrate XYLEM and bug-neighborhhood analyses.

A. Overview of Xylem Analysis

Starting at a statement s that dereferences a variable v, the XYLEM analysis, traverses backward to identify a path over which a null value for v can flow to s. During the analysis, it propagates a set of abstract state predicates backward in the interprocedural control-flow graph.[2] The analysis starts with a predicate asserting that v is null at s, and updates states during the path traversal. If the updated state becomes inconsistent, no null value for v can flow to s along that path. Thus, the analysis does not traverse further along that path.

Consider the application of the XYLEM analysis for the dereference of `x` at line 7 in `foo()`. The analysis initializes the state to contain predicate $\langle \texttt{x} = \texttt{null} \rangle$. Traversing backward, the analysis reaches the conditional statement at line 5 and adds $\langle \texttt{j} > 10 \rangle$ to the state. The statement at line 4 does not update the state. There are two backward paths from line 4. Along the path to line 3, the analysis finds an assignment of a new object to `x`, which contradicts the existing predicate $\langle \texttt{x} = \texttt{null} \rangle$; therefore, the traversal along this path stops. Along the other backward path from line 4, the analysis reaches conditional statement 2 along its false branch, and adds $\langle \texttt{i} \neq 0 \rangle$ to the state. Next, the analysis reaches line 1 where it finds the null assignment to `x`, and identifies statement 7 as an NPR.

B. Bug-neighborhood Analysis

Figure 5 presents, `BugNeighborhoodAnalysis`, the algorithm that performs the bug-neighborhood analysis. The algorithm inputs an (NPA, NPR) pair, (s_a, s_r), from the original program P. The algorithm determines whether the bug fix with respect to (s_a, s_r) has been attempted. If not, the bug is reported as "unfixed." If there has been an attempted fix, the algorithm determines whether the fix is complete or incomplete in P', and reports it as "complete" or "incomplete," respectively.

The algorithm first (lines 1–2) maps s_a and s_r to the respective null-assignment and dereference statements (if any) in P'. The computation of this mapping is orthogonal to our analysis; it can be computed using different analyses (e.g., [10], [11]), which vary in accuracy and cost. After computing the mapped statements s'_a and s'_r, the remainder of the algorithm determines whether an attempted fix has been applied, and if so, whether the fix is complete.

[2]The *control-flow graph* (CFG) for a method contains nodes that represent statements and edges that represent the flow of control between statements. The *interprocedural control-flow graph* (ICFG) contains a CFG for each method in the program.

algorithm `BugNeighborhoodAnalysis`
input (s_a, s_r): (NPA, NPR) pair in the original program P
output $FixStatus$: {"unfixed", "complete fix", "incomplete fix"}
declare $BugNeighborhood'$: pairs of reaching NPAs for s'_r (mapped from s_r), reachable NPRs for s'_a (mapped from s_a) in modified program P'
begin
1. s'_a = matching null-assignment statement in P'
2. s'_r = matching dereference statement in P'
3. $BugNeighborhood' = \emptyset$ // initialization
   // check whether s'_r has reaching NPAs in P'
4. **if** $s'_r \neq$ `null` **then** // s'_r exists in P'
5. S'_a = NPAs identified by the XYLEM analysis starting at $s_r{}'$
6. **foreach** $a' \in S'_a$ **do** add (a', s'_r) to $BugNeighborhood'$
   // check whether s'_a has reachable NPRs in P'
7. **if** $s'_a \neq$ `null` **then** // s'_a exists in P'
8. S'_{deref} = dereferences (excluding s'_r) reachable from s'_a in P'
9. **foreach** $d' \in S'_{deref}$ **do**
10. S'_a = NPAs identified by the XYLEM analysis starting at d'
11. **if** $s'_a \in S'_a$ **then** add (s'_a, d') to $BugNeighborhood'$
12. **foreach** (NPA,NPR) $\in BugNeighborhood'$ **do**
    find Forward NPRs, Backward NPAs
    // set $FixStatus$ for (s_a, s_r)
13. **if** $(s'_a, s'_r) \in BugNeighborhood'$ **then** $FixStatus$ = "unfixed"
14. **else if** $BugNeighborhood' = \emptyset$ **then** $FixStatus$ = "complete fix"
15. **else** $FixStatus$ = "incomplete fix"
16. **return** $BugNeighborhood'$, $FixStatus$
end

Figure 5. Algorithm for the bug-neighborhood analysis.

After initializing $BugNeighborhood'$ (line 3), lines 4–6 of the algorithm check whether there exist reaching NPAs for s'_r in P'. If there exists a matching dereference statement s'_r (i.e., s'_r is not null), the algorithm performs the XYLEM analysis, starting at s'_r, to identify S'_a—the set of reaching NPAs, (lines 4–5). For all statements a' in S'_a, it adds (a', S'_a) to $BugNeighborhood'$ (line 6).

To illustrate, consider applying the algorithm to (NPA, NPR) pair (1,7) in function `foo()`. In `foo'()`, this NPR has been fixed by adding the null check in line 6. After matching the NPR to line 7 in `foo'()`, the algorithm invokes the XYLEM analysis, which, because of the added null check, finds no reaching NPAs. Thus, S'_a is empty in this case.

Lines 7–11 of the algorithm determine whether there exist reachable NPRs for s'_a. The algorithm computes, in two steps, the set of NPRs in P' that can dereference the null value generated at s'_a. In the first step, the algorithm performs a forward reachability analysis, starting at s'_a, to identify reachable dereferences of the null value generated at s'_a (line 8). In the second step (lines 9–10), for each dereference identified in the first step, the algorithm uses the XYLEM analysis to determine whether for a dereference d', s'_a is identified as an NPA. If this is the case, the algorithm adds (s'_a, d') pair to $BugNeighborhood'$ (line 11). The benefit of the two-step approach is that the second

step, using the XYLEM analysis, can potentially filter out infeasible reachable dereferences that may be identified in the first step [9]. Line 12 of the algorithm uses similar backward and forward analysis to find the Forward NPRs and the Backward NPAs for each pair in $BugNeighborhood'$, and adds these pairs to $BugNeighborhood'$.

Lines 13–15 of the algorithm determine and set the value of $FixStatus$. If (s'_a, s'_r) is in $BugNeignborhood'$, then there is still a flow of a null value from the pair associated with (s_a, s_r), and thus, the pair has not been fixed. Thus, the algorithm sets $FixStatus$ as "unfixed." Otherwise, if $BugNeighborhood'$ is empty, the algorithm sets $FixStatus$ as "complete fix," else ($BugNeighborhood'$ is not empty), the algorithm sets $FixStatus$ as "incomplete fix." Finally, in line 16, the algorithm returns $BugNeighborhood'$ and $FixStatus$.

Consider again the invocation of the algorithm for pair (1, 7) in `foo()`. Starting at line 1 (s'_a) in `foo'()`, the algorithm performs forward reachability analysis to identify dereferences 7, 8, 10, 11, and 12. For each of these dereferences, the algorithm invokes XYLEM. For dereferences 8, 11, and 12, the XYLEM analysis identifies statement 1 as a reaching NPA, whereas for dereference 7 and 10, it finds no reaching NPA. Therefore, at line 13 of the algorithm, $Bugneighborhood' = \{(1, 8), (1, 11), (1, 12)\}$: statement 8 is a Forward NPR that has been uncovered by the attempted fix, statement 11 is an unfixed Maybe NPR, and statement 12 is a Forward NPR associated with statement 11. Thus, a fix is attempted for pair (1, 7) because the pair does not occur in $BugNeighborhood'$. But, the fix is incomplete because $BugNeighborhood'$ is not empty.

IV. Empirical Evaluation

To evaluate our approach, we implemented it, and conducted two empirical studies to investigate the occurrences of bug neighborhoods and the completeness of attempted bug fixes. After describing the experimental setup (Section IV-A), we present the results of the two studies (Sections IV-B and IV-C). Finally, we discuss threats to the validity of our empirical evaluation (Section IV-D).

A. Experimental Setup

Our experimental subjects consist of three open-source projects (Ant-1.6.0, Lucene-2.2.0, and Tomcat-4.1.27)[3] and three proprietary commercial products (referred to as App-A, App-B, and App-C). Table I lists the subjects, along with the numbers of classes, methods, bytecode instructions, and (NPA, NPR) pairs in each subject.

We integrated our analysis into the XYLEM tool. The implementation consists of two main components: the mapping component and the bug-neighborhood analysis component. Suppose, for our discussion, that the null-pointer bug in P is represented by the pair (NPA_0, NPR_0).

[3]Available at apache.org

Table I
SUBJECTS USED IN THE EMPIRICAL STUDIES.

Subject	Classes	Methods	Bytecode Instructions	(NPA, NPR) Pairs
Ant-1.6.0	1858	17204	443254	167
Lucene-2.2.0	381	2815	72691	86
Tomcat-4.1.27	260	4077	101075	97
App-A	278	3933	98225	63
App-B	169	1876	46286	119
App-C	2488	13746	340896	107

Table II
EIGHT BUG-NEIGHBORHOOD CATEGORIES FOR AN (NPA, NPR) PAIR.

Category	Maybe NPA Present	Maybe NPR Present	Forward NPR Present
1	no	no	no
2	no	no	yes
3	no	yes	no
4	no	yes	yes
5	yes	no	no
6	yes	no	yes
7	yes	yes	no
8	yes	yes	yes

The mapping component maps statements (bytecode instructions) based on their signatures and order of occurrences in the source code. The result is (1) a mapping between (NPA_0, NPR_0) and a pair in P' or (2) an indication that there is no matched statement in P' for either NPA_0 or NPR_0.

The component for performing the bug-neighborhood analysis uses the XYLEM analysis to identify the bug neighborhood for the pair in P' associated with (NPA_0, NPR_0). The component implements algorithm `BugNeighborhoodAnalysis` (Figure 5), and performs the appropriate backward and forward analyses to find the Maybe NPAs, Maybe NPRs, and Forward NPRs associated with the mapped pair in P'. Currently, the implementation does not compute Backward NPAs.

Using these NPAs and NPRs in P' and the pair in P' mapped from (NPA_0, NPR_0), the implementation creates new null-pointer bug pairs in P' and adds them to the bug neighborhhod for the mapped pair in P'. Using the bug neighborhood for the mapped pair in P', the implementation classifies (NPA_0, NPR_0) as not fixed (i.e., no attempt was made to fix the bug), attempted but incomplete, or complete.

B. Study 1: Neighborhood Categories and Sizes

1) Goal and Method: The goal of this study is to examine the consistency and sizes of bug neighborhoods in practice. To do this, we ran our implementation on each subject (Table I). For each (NPA, NPR) pair in a subject, we computed the bug neighborhood and the neighborhood size.

To understand the consistency of bug neighborhoods, we created a classification that consists of eight categories, shown in Table II. Each category is defined based on whether a Maybe NPA, a Maybe NPR, or a Forward NPR is present in a neighborhood (recall that our current implementation

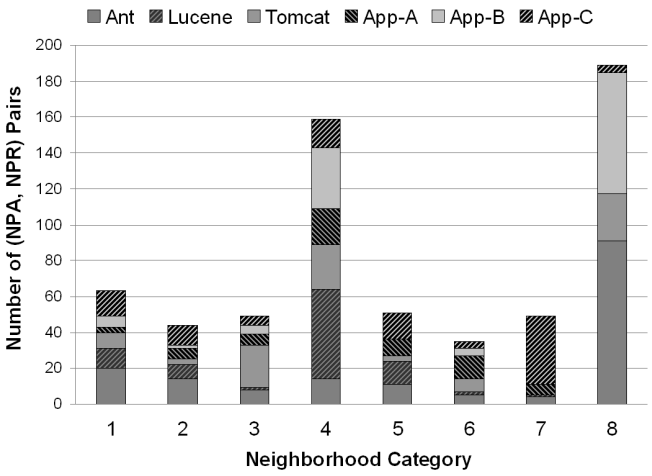

Figure 6. Number of occurrences of bug-neighborhood categories, aggregated over the subjects. Each bar segment shows the number of (NPA, NPR) pairs in a bug-neighborhood category that occur in a subject.

does not compute Backward NPAs). For example, Category 1 represents the simplest neighborhood, in which no Maybe NPAs, Maybe NPRs, or Forward NPRs occur. Thus, for this category, each type of attempted fix shown in Figure 3 is a complete fix. Category 8 represents the most complex neighborhoods in which all three of maybe NPAs, maybe NPRs, and forward NPRs are present.

To present the data on the sizes of the bug neighborhoods, we created three groups of neighborhood sizes: *small* (five or fewer (NPA, NPR) pairs), *medium* (greater than five but not greater than 15 (NPA, NPR) pairs), and *large* (greater than 15 (NPA, NPR) pairs).

2) Results and Analysis: First, we present the data on occurrences and distribution of bug-neighborhood categories. Then, we present the data on bug-neighborhood sizes.

Figure 6 presents data that show the number of occurrences of each neighborhood category, aggregated over the subjects. The horizontal axis lists the eight categories; the vertical axis represents the number of (NPA, NPR) pairs whose neighborhoods are of a particular category. The data for a category are represented as a segmented bar, in which the segments represent the number of pairs in each subject; the height of the bar shows the total number of pairs for the category. The order of the subjects is the same in all the bars: `Ant` is the segment at the bottom, `App-C` is the segment at the top. Over all subjects, neighborhood Category 1 occurs 63 times. Of these, 20 occur in `Ant`, 11 occur in `Lucene`, and 9 occur in `Tomcat`; `App-A`, `App-B`, and `Ant-C` contain 3, 6, and 14 Category 1 neighborhoods, respectively. Category 8 occurs 189 times in total, of which 91 occur in `Ant`, 26 in `Tomcat`, 68 in `App-B`, and 4 in `App-C`.

The data illustrate that Category 8 neighborhoods, which are the most complex neighborhoods, occur most frequently in our subjects. Category 4, which is also fairly complex (neighborhoods of this category contain Maybe NPRs and Forward NPRs), has the second highest occurrence. Another interesting observation is that neighborhood categories that have Maybe NPRs but no Forward NPRs (represented as the sum of Categories 3 and 7) occur often. For such neighborhoods, a developer can target the NPR and maybe NPRs in an attempted fix, without being concerned about whether the fix would unmask forward NPRs.

In Figure 7, we present a different view of the data. The horizontal axis lists the subjects. For each subject, the chart contains one bar for each category. The vertical axis, as in Figure 6, represents the number of (NPA, NPR) pairs whose bug neighborhoods are of a particular category. The segments in each bar show the number of pairs that have small, medium, and large bug neighborhoods.

As the data illustrate, `Ant` and `App-B` contain a large percentage of Category 8 bug neighborhoods: more than half their (NPA, NPR) pairs have neighborhoods of Category 8. Two subjects (`Ant` and `App-C`) contain pairs for all neighborhood categories; all subjects except `App-B` have at least seven of the eight categories. `App-C` has a different distribution of neighborhood categories from all the other subjects: most of its (NPA, NPR) pairs belong to Category 7 whereas, in all other subjects, Category 7 is one of the categories with the fewest (NPA, NPR) pairs.

The data also show a relationship between bug-neighborhood size and bug-neighborhood categories: all large neighborhoods occur in the most complex neighborhood categories (Categories 4 and 8). Medium neighborhoods occur in categories that contain Maybe NPAs or Maybe NPRs (Categories 3 to 8); small neighborhoods are distributed across all categories. With two exceptions, the large neighborhoods in our subjects have less than 35 (NPA, NPR) pairs: `Ant` has Categroy 8 neighborhoods of size 72 and `Lucene` has Category 4 neighborhoods of size 46.

Overall, the data indicate that complex bug neighborhoods can occur frequently in practice and be large in size. Therefore, an approach, such as ours, that points the developer to bug-neighborhood statements can be useful in practice as it provides context information about a bug that can help the developer ensure that an attempted fix is complete.

C. Study 2: Completeness of Attempted Bug Fixes

1) Goal and Method: The goal of this study is to investigate the existence and frequency of incomplete fixes. We considered successive releases of the subjects, and identified fixes made between each pair of a release and its successive release. For each P and P',[4] we ran the analysis to compute the bugs in P for which fixes were attempted in P' and the classification of each fix as complete or incomplete.

Our mapping component for identifying (NPA, NPR) pair association in P and P' produces some false positives

[4]For consistency in terminology, we refer to the releases in a pair as the original program, denoted by P, and the modified program, denoted by P'.

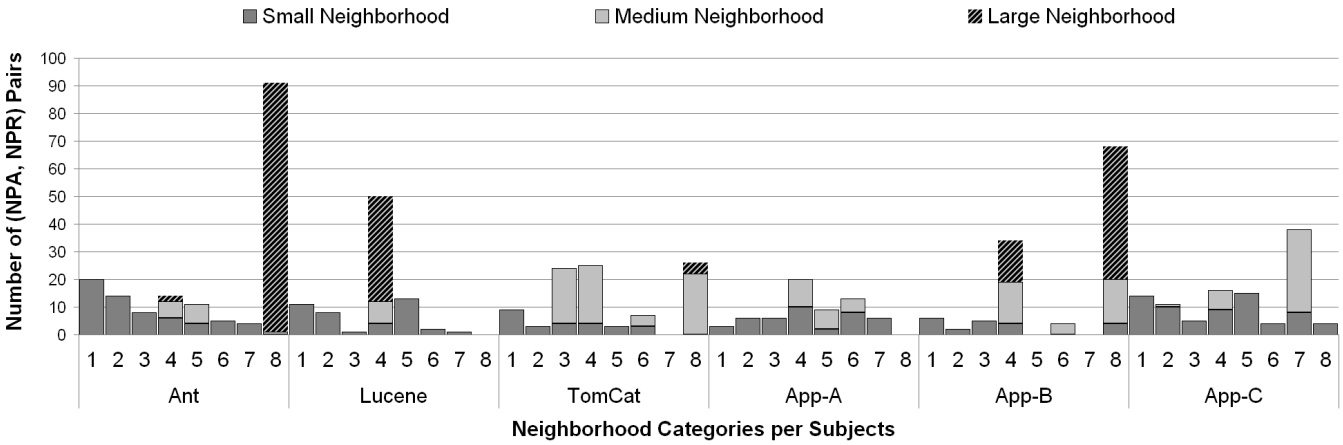

Figure 7. Number of occurrences of bug-neighborhood categories per subject. The bar segments show the number of (NPA, NPR) pairs in a bug-neighborhood category that have small, medium, and large neighborhoods.

Table III
NUMBER OF INCOMPLETE BUG FIX ATTEMPTS AND THEIR NEIGHBORHOOD CATEGORIES AND SIZES.

Subject	Number of Incomplete/ Attempted Fixes	Neighborhood Categories of Incomplete Fixes	Neighborhood Sizes of Incomplete Fixes
Ant	4 / 26	2, 4, 4, 5	Small, 3 x Medium
Lucene	3 / 17	4, 4, 4	Small, 2 x Large
Tomcat	0 / 9	—	—
App-A	0 / 7	—	—

(reported as attempted fixes but not actually attempted). Because we want to report only precise results, we manually verified our analysis results for four subjects—`Ant`, `Lucene`, `Tomcat`, `App-A`—and removed these false positives. We report the detailed results for these four subjects.

2) Results and Analysis: Table III illustrates the number of incomplete and attempted bug fixes and shows the neighborhood categories and neighborhood sizes of the incomplete fixes. In `Ant`, four of 26 attempted bug fixes are incomplete; in `Lucene`, three of 17 attempted bug fixes are incomplete. Five of the seven incomplete fixes belong to Category 4, the other two incomplete fixes belong to Category 2 and Category 5. The (NPA, NPR) pairs of the incomplete fixes have small, medium, and large bug neighborhoods, the largest neighborhood has size 36. In `Tomcat` and `App-A`, all attempted bug fixes—9 and 7 respectively—are complete.

The data from this study show that, in practice, attempted bug fixes can be incomplete and, thus, null-pointer bugs related to the attempted fix of one null-pointer bug can occur in other executions of the program. Moreover, the bug neighborhoods of these incomplete fixes can be large and, therefore, difficult to identify manually. For example, for two of the incomplete fixes, the bug neighborhoods were larger than 30. Without automated analysis, determining whether these additional pairs related to the (NPA, NPR) exist, and then locating these pairs manually can be a time-consuming task. Even in the cases where the fix is complete, the developer may want to know this to avoid searching for related NPAs and NPRs manually.

D. Threats to Validity

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. We have used the XYLEM analysis for many experiments and checked its results manually. Thus, we are confident in the results it produced. We discovered that, with the mapping we used in this implementation, our results contained some false positives. Thus, we verified our results manually, and reported only those results in Study 2.

Threats to external validity arise when the results of the experiment cannot be generalized to other situations. One external threat involves the ability to generalize our results to other programs. In our study, we used six programs and thus, we are unable to conclude that our results will hold in general. However, these programs are used in practice and are somewhat diverse. Thus, we can conclude that incomplete fixes for null-pointer bugs do exist in practice, and that their bug neighborhoods can be significant in size.

Another external threat concerns the ability to generalize the results for null-pointer exceptions to other types of bugs that result from incorrect flows of values. We believe that a similar analysis could provide information for other related types of bugs. However, only an implementation and experimentation with such other bugs would confirm our belief in the applicability of our technique.

V. RELATED WORK

In previous work [9], we presented an approach for identifying and fixing faults that cause runtime exceptions. Given a statement at which a null-pointer exception occurs,

the approach uses the XYLEM analysis, guided by the stack trace of the failing execution, to identify definite and possible NPAs that could have caused the exception. The approach also identifies maybe NPAs and maybe NPRs that could cause exceptions in other executions. Unlike our previous work, in this paper, we investigate how bugs are fixed. We extend our previous approach to define a bug neighborhood and characterize the completeness of a bug fix in terms of the bug neighborhood. We also present analysis that compares two code versions to identify attempted fixes and classify each fix as complete or incomplete.

There has been much research on mining bug and source-code repositories to study the history of bug fixes. Information about past bug fixes is used for many applications, such as assessing and improving the effectiveness of static bug detection (e.g., [7], [12]), prioritizing warnings (e.g., [13], [14]), recommending bug fixes and pertinent software artifacts (e.g., [15], [16], [17]), assigning defect reports to developers (e.g., [18]), identifying code changes that subsequently lead to bug fixes (e.g., [19]), and estimating the fault-proneness of code components (e.g., [20], [21]). We discuss a sample of this research that is related to different aspects of our work.

The main distinguishing aspect of our work from the previous research is that none of the existing work has investigated whether bugs fixes are complete and how incomplete bug fixes could be made complete. Much of this previous research has focused on the bugs reported in bug repositories, which are more general than the class of bugs (involving flows of invalid values) to which our approach is applicable. Determining completeness for general bugs, such as those related to application functionality, is not addressed by our approach.

Ayewah and colleagues [7] investigated, for three software systems, how the bugs reported by FINDBUGS [22] were fixed across different builds of the system. They manually classified the bugs as those that could have little, some, or substantial functional impact. They also examined the types of program changes that caused the bugs to be fixed and determined whether such a change was localized and intended to remove the bug. However, they did not evaluate whether a bug fix was complete.

Spacco, Hovemeyer, and Pugh [8] examined the evolution of bugs over successive builds of Sun's JDK core runtime library. They presented two approaches for mapping bugs from one version to another, and reported trends about defect lifetimes, decay in defects reported, and defect density over successive builds. However, they performed no evaluation of whether the bugs that were removed were completely fixed.

Williams and Hollingsworth [12] developed a checker to detect bugs where the return value of a function was used before being tested (e.g., dereferencing a returned pointer without performing a null check). They also developed a ranking technique that orders the warnings based on whether the relevant function was involved in a past bug fix that added a check on its return value, and how often the return value of the function is tested before use. In their work, a notion of completeness similar to ours could be used: if a bug fix added a check on the return value of a function at one call site, but not at another call site to the same function, the fix could be marked incomplete. However, the goal of that work was not to classify fixes as complete or incomplete.

Existing work on recommending systems performs several activities: suggests fixes for bugs based on patterns of past fixes [15]; suggests relevant software artifacts, based on a memory of the artifacts created during the development of a software system, that should be examined to perform a task, such as fixing a bug [16]; or provides context information for a bug, based on mining information (from different data repositories) about how similar bugs have been fixed in the past [17]. Our work is related to such approaches in that it can be used to recommend parts of the code that should be examined to ensure that a fix is complete. However, unlike these techniques, our current approach is not based on mining information from software repositories but on an analysis of the code.

Researchers have explored using the history of bug fixes to prioritize new bugs based on machine learning [13] or statistical modeling [14]. Our approach could be used to infer correlations between characteristics of bug neighborhoods and the likelihood of attempted fixes, which could be used to prioritize a new bug based on its neighborhood characteristics.

VI. Conclusion and Future Work

In this paper, we presented a new approach for automatically identifying incomplete bug fixes in Java programs. We examined bugs that involve a flow of invalid values causing a runtime exception. We introduced the concept of a bug neighborhood that lets us categorize bugs. In addition, a bug neighborhood also lets us determine the completeness of attempted bug fixes: if a bug neighborhood is not empty after an attempted fix, the fix is incomplete.

Our approach can assist debugging in two different scenarios. During maintenance, our approach can identify incomplete bug fixes by analyzing two (or more) revisions of a program. If incomplete fixes are found, developers can utilize the remaining information in the bug neighborhood to complete the incomplete fixes. During development, our approach can check whether incomplete bug neighborhoods exist before or after developers attempt to fix a bug, and guide them through the process of completing the fixes. Thus, our approach can prevent the introduction of incomplete fixes into new revisions.

To evaluate our approach we implemented it and conducted two empirical studies on three open source and three industrial subjects. Our analysis results indicate that, for the

subjects considered, large and complex bug neighborhoods occur frequently, and attempted fixes can be incomplete.

There are several areas of future work; we discuss some of them here. First, we will address the imprecision of the mapping component we use to identify the association of the (NPA, NPR) pairs in P and P'. We will investigate whether other mapping techniques, such as JDiff [10], can produce better and more reliable results.

Second, we will gather more analysis results from other subjects and we will investigate several interesting open questions: is there a correlation between incomplete bug fixes and the size of a bug neighborhood? Is there a correlation between incomplete bug fixes and a bug-neighborhood category? Are incomplete bug fixes in open-source projects different from incomplete fixes in industrial projects? Do incomplete fixes occur more frequently in open-source software than in industrial projects?

Third, we will investigate other types of bugs that involve a flow of invalid values that cause a runtime exception. We will investigate whether there exist well-defined fault categories and fix categories for these types of bugs, and how these categories correlate. Our goal is to support developers further in addressing incomplete bug fixes. For some fault categories, we may be able to apply the fixes automatically, for other fault categories, we may be able to recommend possible fixes to developers or, at least, provide additional information that helps the developers to fix the bug.

Finally, we will enhance our analysis to guide developers in addressing an incomplete fix and making it complete. We will investigate prioritizing (NPA, NPR) pairs using several approaches, and presenting the neighborhoods using intuitive visualization techniques. Such techniques may reduce the manual process in debugging and save time and cost.

ACKNOWLEDGMENT

This work was supported in part by awards from NSF under CCF-0429117, CCF-0541049, and CCF-0725202, and IBM by a Software Quality Innovation Faculty Award.

REFERENCES

[1] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended static checking for Java," in *Proc. of PLDI*, Jun. 2002, pp. 234–245.

[2] D. Hovemeyer and W. Pugh, "Finding more null pointer bugs, but not too many," in *Proc. of PASTE*, Jun. 2007, pp. 9–14.

[3] D. Hovemeyer, J. Spacco, and W. Pugh, "Evaluating and tuning a static analysis to find null pointer bugs," in *Proc. of PASTE*, Sep. 2005, pp. 13–19.

[4] A. Loginov, E. Yahav, S. Chandra, N. Fink, S. Rinetzky, and M. G. Nanda, "Verifying dereference safety via expanding-scope analysis," in *Proc. of ISSTA*, Jul. 2008, pp. 213–223.

[5] M. G. Nanda and S. Sinha, "Accurate interprocedural null-dereference analysis for Java," in *Proc. of ICSE*, May 2009, pp. 133–144.

[6] A. Tomb, G. Brat, and W. Visser, "Variably interprocedural program analysis for runtime error detection," in *Proc. of ISSTA*, Jul. 2007, pp. 97–107.

[7] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou, "Evaluating static analysis defect warnings on production software," in *Proc. of PASTE*, Jun. 2007, pp. 1–8.

[8] J. Spacco, D. Hovemeyer, and W. Pugh, "Tracking defect warnings across versions," in *Proc. of Intl. Workshop on MSR*, May 2006, pp. 133–136.

[9] S. Sinha, H. Shah, C. Görg, S. Jiang, M. Kim, and M. J. Harrold, "Fault localization and repair for Java runtime exceptions," in *Proc. of ISSTA*, Jul. 2009, pp. 153–164.

[10] T. Apiwattanapong, A. Orso, and M. J. Harrold, "JDiff: A differencing technique and tool for object-oriented programs," *ASE*, vol. 14, no. 1, pp. 3–36, Mar. 2007.

[11] S. Raghavan, R. Rohana, D. Leon, A. Podgurski, and V. Augustine, "Dex: A semantic-graph differencing tool for studying changes in large code bases," in *Proc. of ICSM*, Sep. 2004, pp. 188–197.

[12] C. C. Williams and J. K. Hollingsworth, "Automatic mining of source code repositories to improve bug finding techniques," *IEEE TSE*, vol. 31, no. 6, pp. 466–480, Jun. 2005.

[13] S. Kim and M. Ernst, "Which warnings should I fix first?" in *Proc. of ESEC/FSE*, Sep. 2007, pp. 45–54.

[14] J. R. Ruthruff, J. Penix, J. D. Morgenthaler, S. Elbaum, and G. Rothermel, "Predicting accurate and actionable static analysis warnings: An experimental approach," in *Proc. of ICSE*, May 2008, pp. 341–350.

[15] S. Kim, K. Pan, and E. J. Whitehead, Jr., "Memories of bug fixes," in *Proc. of Intl. Symp. on FSE*, Nov. 2006, pp. 35–45.

[16] D. Čubranić, G. C. Murphy, J. Singer, and K. S. Booth, "Hipikat: A project memory for software development," *IEEE TSE*, vol. 31, no. 6, pp. 446–465, Jun. 2005.

[17] B. Ashok, J. Joy, H. Liang, S. K. Rajamani, G. Srinivasa, and V. Vangala, "Debugadvisior: A recommender system for debugging," in *Proc. of ESEC/FSE*, Aug. 2009, pp. 373–382.

[18] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *Proc. of ICSE*, May 2006, pp. 361–370.

[19] S. Kim, T. Zimmermann, K. Pan, and E. J. Whitehead Jr., "Automatic identification of bug-introducing changes," in *Proc. of ASE*, Sep. 2006, pp. 81–90.

[20] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE TSE*, vol. 26, no. 7, pp. 653–661, Jul. 2000.

[21] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Predicting the location and number of faults in large software systems," *IEEE TSE*, vol. 31, no. 4, pp. 340–355, Apr. 2005.

[22] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *ACM SIGPLAN Notices (Proceedings of Onward! at OOPSLA 2004)*, vol. 39, no. 10, pp. 92–106, Dec. 2004.