Inferring Project-Specific Bug Patterns for Detecting Sibling Bugs

Guangtai Liang[1], Qianxiang Wang[1], Tao Xie[2], Hong Mei[1]
[1]Institute of Software, School of Electronics Engineering and Computer Science
Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education
Peking University, Beijing, 100871, China
{lianggt08, wqx, meih}@sei.pku.edu.cn
[2]Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA
taoxie@illinois.edu

ABSTRACT
Lightweight static bug-detection tools such as FindBugs, PMD, Jlint, and Lint4j detect bugs with the knowledge of generic bug patterns (e.g., objects of java.io.InputStream are not closed in time after used). Besides generic bug patterns, different projects under analysis may have some project-specific bug patterns. For example, in a revision of the Xerces project, the class field "fDTDHandler" is dereferenced without proper null-checks, while it could actually be null at runtime. We name such bug patterns directly related to objects instantiated in specific projects as **Project-Specific Bug Patterns (PSBPs)**. Due to lack of such PSBP knowledge, existing tools usually fail in effectively detecting most of this kind of bugs. We name bugs belonging to the same project and sharing the same PSBP as **sibling bugs.** If some sibling bugs are fixed in a fix revision but some others remain, we treat such fix as an incomplete fix. To address such incomplete fixes, we propose a PSBP-based approach for detecting sibling bugs and implement a tool called Sibling-Bug Detector (SBD). Given a fix revision, SBD first infers the PSBPs implied by the fix revision. Then, based on the inferred PSBPs, SBD detects their related sibling bugs in the same project. To evaluate SBD, we apply it to seven popular open-source projects. Among the 108 warnings reported by SBD, 63 of them have been confirmed as real bugs by the project developers, while two existing popular static detectors (FindBugs and PMD) cannot report most of them.

Categories and Subject Descriptors
D.2.5 **[Software Engineering]**: Software/Program Verification; F.3.1 **[Theory of Computation]**: Specifying and Verifying and Reasoning about Programs

General Terms
Languages, Reliability, Verification.

Keywords
Project-specific bug patterns, sibling-bug detection, incomplete fixes.

ESEC/FSE'13, August 18–26, 2013, Saint Petersburg, Russia

1. INTRODUCTION
Lightweight static bug-detection tools such as FindBugs [4], PMD [17], Jlint [7], and Lint4j [12] detect bugs with the knowledge of generic bug patterns. For example, for Java projects under analysis, there is a generic bug pattern: "objects of *java.io.InputStream* are not closed in time after used". As shown in this example, most generic bug patterns are related to classes provided by common third-party libraries (e.g., JDK for Java projects), and are usually applicable for all projects based on the same third-party libraries.

Besides generic bug patterns, different projects under analysis may have various project-specific bug patterns. When writing object-oriented code for a project, developers would define many project-specific objects in their newly-written interfaces, classes, or methods. Those user-defined objects may also need to be used under some certain constraints. For example, in a revision of the Xerces project, the object "fDTDHandler", an object of the class XMLDTDHandler (defined within Xerces), is dereferenced without proper null-checks, while it could actually be null at runtime. As another example, in a revision of the Tomcat project, the local object "cometEvent", an object of the class CometEventImpl (defined within Tomcat), is not closed after used in a certain method. We name such patterns directly related to objects instantiated in specific projects as **Project-Specific Bug Patterns (PSBPs)**. Note that focused objects in PSBPs could also be objects of classes defined in some third-party library. Existing tools usually fail in detecting most of those bugs sharing PSBPs due to the lack of related PSBP knowledge.

Bugs sharing the same PSBP are typically related, involved with the same object of the same class. We name such bugs sharing the same PSBP as **sibling bugs** in this paper**.** If some sibling bugs are fixed in a fix revision but some others remain, such fix can be considered as an incomplete fix. Recent studies have revealed that incomplete bug fixes are common in bug-fixing processes. For example, Kim *et al.* [9] identified that, among 26 (17) attempted fixes, four (three) Null-Pointer-Exception (NPE) fixes are incomplete in the ANT (Lucene) project.

Figure 1 shows an incomplete-fix example from the Xerces project. The lines added by commits are in bold, underlined, and labeled with the "+" symbols. In this example, the first fix revision (*Revision 318586*) fixes an NPE bug on the class-field object "fDTDHandler" in the method "setInputSource" of the class XMLDTDScannerImpl. However, a sibling NPE bug on the same object remains in the method "startEntity" of the same class. Four months later, the remaining bug is finally fixed in a later commit (*Revision 318859*).

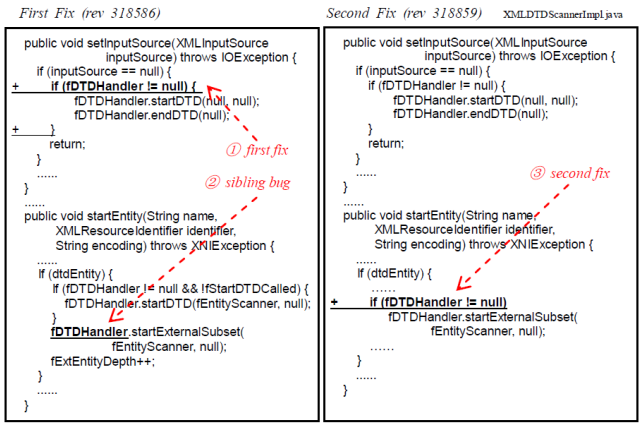

Figure 1. An incomplete NPE fix from Xerces.

Based on such incomplete-fix phenomenon, we find that static-analysis techniques actually could leverage existing fixes to infer PSBPs and then detect their related remaining sibling bugs for the projects under analysis. For example, based on the first fix shown in Figure 1, we can infer a PSBP: "the class field 'fDTDHandler' could be null at runtime, but it is dereferenced without proper null-checks". Then, with the help of such inferred PSBP, we can easily detect a sibling bug: "fDTDHandler" is still dereferenced directly without a proper null-check in the method "startEntity".

To automatically detect such remaining sibling bugs, in this paper, we propose a PSBP-based approach and implement a static-analysis tool called Sibling-Bug Detector (SBD). Given a fix revision, SBD first leverages bug-pattern templates to effectively identify its actually-fixed bugs, and then infers PSBPs based on the identified bugs. After that, based on the inferred PSBPs, SBD detects whether some other sibling bugs still remain. To evaluate SBD, we apply SBD to seven popular well-maintained open-source projects. SBD identifies 108 sibling bugs, and 63 of them have been confirmed as real bugs by the project developers, while two existing popular static bug-detection tools (FindBugs [4] and PMD [17]) cannot report most of them.

This paper makes the following main contributions:

- The first to propose the concept of Project-Specific Bug Patterns (PSBPs).
- An effective technique for inferring PSBPs based on bug-pattern templates.
- A PSBP-based Sibling-Bug Detector (SBD), which applies the inferred PSBPs to detect sibling bugs for the project under analysis.
- Empirical evaluations on seven popular real-world open-source projects to demonstrate the effectiveness of SBD.

2. APPROACH

This section presents the proposed approach. Figure 2 illustrates the approach overview. Given a fix revision, our approach first infers PSBPs based on its actual fix activities, and then applies the inferred PSBPs to detect whether some other sibling bugs remain in the project under analysis.

Section 2.1 describes the bug-pattern templates predefined for different well-known bug types in our approach. Sections 2.2 and 2.3 describe the process of inferring project-specific bug patterns and the process of detecting sibling bugs.

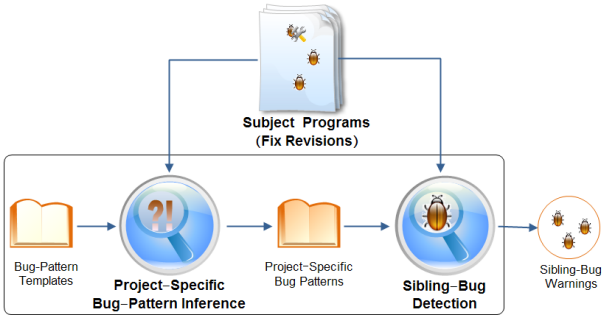

Figure 2. Approach Overview.

2.1 Defining Bug-Pattern Templates

Since directly inferring arbitrary PSBPs from a given fix revision could be challenging, we leverage bug-pattern templates to help infer PSBPs. In our approach, a bug-pattern template contains two parts: *featured statements* and *usage scenarios*.

Featured statements are important statement changes (additions and/or deletions) introduced by the given fix revision. Such featured statements can provide strong hints to understand the actually-fixed bug. Note that not all those statement changes introduced by the fix revision are equally important to characterize the actually-fixed bug. *Usage scenarios* are characterized with Finite State Automata (FSA) to summarize typical usage scenarios for the focused objects of a bug-pattern template. Such usage scenarios can further help understand fix revisions.

Figure 3 and Figure 4 visually illustrate the bug-pattern templates currently implemented for our approach: the NPE template and the resource-leak template. Their corresponding formal specification files can be found on the project website[1].

For the NPE template, its featured statements are the newly-added null-check statements since developers typically introduce them on the NPE-triggering objects to avoid NPE bugs. Meanwhile, its usage scenarios summarize typical usage scenarios of NPE-related objects. In usage scenarios, an expression between "[" and "]" describes the precondition that the corresponding state-transition event should satisfy, and a black-colored (gray-colored) state represents a buggy (correct) state. The buggy usage scenario for an NPE-related object "a" is that "a" is dereferenced directly when "a" is null (from State 1 to State 4), while the correct usage scenarios include (1) "a" is dereferenced when it is not null (from State 1 to State 3); (2) "a" is first null-checked (from State 1 to State 2) and then safely dereferenced (from State 2 to State 3).

For the resource-leak template, its featured statements include the newly-added object-releasing method calls since developers typically call such methods on resource objects to avoid leaks. Meanwhile, its usage scenarios summarize the typical usage scenarios of the leak-related objects. Its correct usage scenarios include (1) the focused object "a" is first created by constructors or other method calls (from State 1 to State 2), and then it is safely released with a release method (from State 2 to State 3); (2) "a" is first assigned and then returned as a return value (from State 2 to State 3); (3) "a" is first assigned and then it will not be closed since it is null (from State 2 to State 3). While its buggy usage scenario is that after "a" is assigned, "a" is not closed and also not returned when "a" is not null (from State 2 to State 4).

[1] http://sa.seforge.org/SiblingBugDetetor

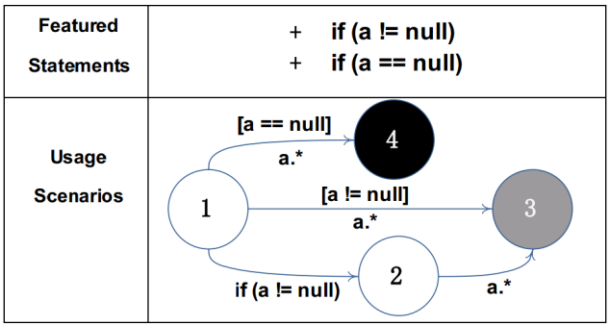

Figure 3. The bug-pattern template for NPE bugs.

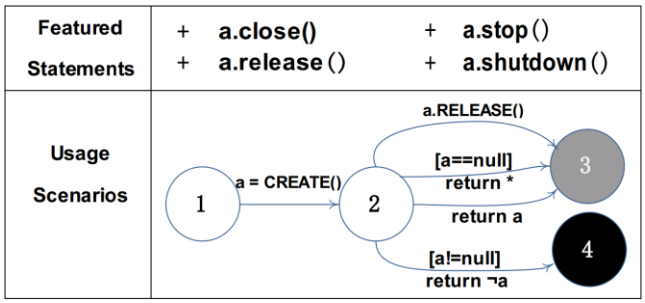

Figure 4. The bug-pattern template for resource-leak bugs.

2.2 Inferring Project-Specific Bug Patterns

With the help of the predefined bug-pattern templates, we infer PSBPs based on each given fix revision with three steps: featured-statement identification (Section 2.2.1), focused-path extraction (Section 2.2.2), and pattern generation (Section 2.2.3).

2.2.1 Featured-Statement Identification

Given a fix revision, the first step of our PSBP inference is to identify its featured statements among its newly-added statements (currently we do not handle fix revisions containing only statement deletions, which we plan to support in future work). In this step, we first retrieve all of its newly-added statements. Then, we try to syntactically match them with the featured statements defined in each template. Based on such matching, we treat the matched statements as the featured statements of the given fix revision.

Figure 5 shows another NPE fix example, which is from the fix revision (Revision 657135) on the class file "ListLevel.java" in the POI project. In Figure 5, the added/deleted lines are labeled with the "+"/"-" symbols. For this example, we first identify its newly-added statements: Lines 224-228 and 232. Then, we try to match each of them with the predefined featured statements of each bug-pattern template. Based on such matching, we identify ***"if (_numberText == null)"*** (Line 224) as a featured statement since it matches with the featured statement ***"if (a == null)"*** of the NPE template. Meanwhile, we treat ***_numberText*** as its focused object.

2.2.2 Focused-Path Extraction

In this step, guided by each featured statement (identified by the previous step), we first extract a limited number of its covered paths (e.g., up to the first five traversed paths that include the featured statement), and then slice each of them based on the variable corresponding to the focused object of the featured statement. In this step, we treat such sliced paths covered by featured statements as focused paths.

```
198       public byte[] toByteArray()
199       {
200         byte[] buf = new byte[getSizeInBytes()];
...       .....
222         offset += _cbGrpprlPapx;
223
224   +     if (_numberText == null) {
225   +       // TODO - write junit to test this flow
226   +       LittleEndian.putUShort(buf, offset, 0);
227   +     } else {
  -   -       LittleEndian.putShort(buf, offset, (short)_numberText.length);
228   +       LittleEndian.putUShort(buf, offset, _numberText.length);
229           offset += LittleEndian.SHORT_SIZE;
230           for (int x = 0; x < _numberText.length; x++)
231           {
  -   -         LittleEndian.putShort(buf, offset, (short)_numberText[x]);
232   +         LittleEndian.putUShort(buf, offset, _numberText[x]);
233           offset += LittleEndian.SHORT_SIZE;
234         }
235       }
236       return buf;
237     }
```

Figure 5. An NPE fix example from the fix revision (Revision 657135) on the file "ListLevel.java" in POI.

Algorithm 1: Focused Path Extraction

Input:
Rev : the fix commit revision number
L : the featured-statement locations
Output:
P : the extracted featured paths

```
1  locate the featured methods M of the given fix commit
   (Revision Rev) according to L;
2  foreach featured method m ∈ M do
3      generate its control-flow graph cfg_m;
4      foreach featured statement s_f ∈ m do
5          extract its covered paths P_covered;
6          get the focused object O_s of s_f;
7          foreach covered path p_covered ∈ P_covered do
8              slice p_covered based on O_s as p_featured;
9              P ⟵ p_featured;
10         end
11     end
12 end
```

Algorithm 1 shows the algorithm for extracting focused paths. It takes the revision number *Rev* of a fix revision under analysis and the locations *L* of the featured statements of the fix revision as inputs, and produces the extracted focused paths *P* as output.

In the algorithm, we first locate the methods where the featured statements reside as the featured methods *M* (Line 1). For each featured method *m* in *M*, we first generate its control-flow graph (Line 3). After that, for each featured statement s_f in *m*, we extract up to the first five traversed paths of its covered paths as $P_{covered}$ (Line 5), and then we get the focused object O_s of s_f (Line 6). Based on O_s, we slice each of its covered paths as a focused path $p_{featured}$ without considering control dependencies (Line 8): starting with s_f, we first slice backward the path portion preceding s_f and then slice forward the path portion succeeding s_f; the backward/forward slicing is based on O_s. During the intra-procedural slicing process, the statements irrelevant to O_s (with respect to the concept of slicing) are discarded, resulting in that each focused path includes only the statements relevant to O_s. Finally, we save the extracted focused path $p_{featured}$ into *P* (Line 9).

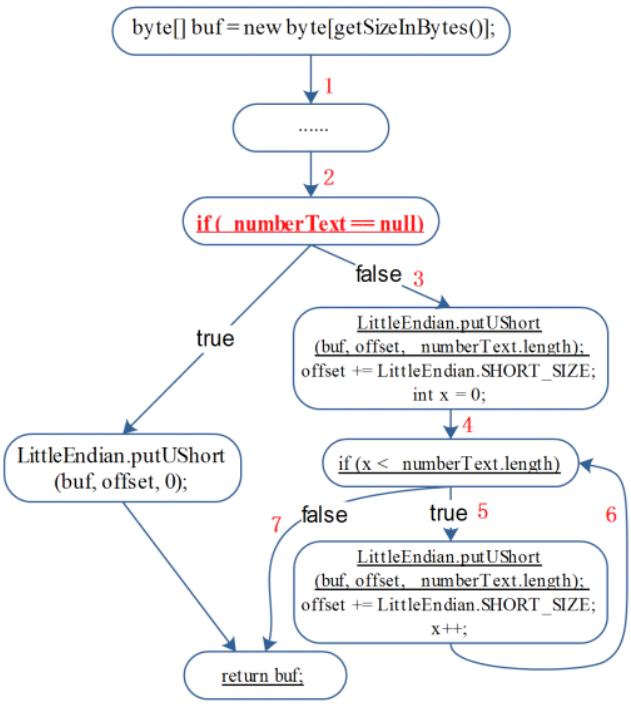

Figure 6. The extracted focused path for the fix example shown in Figure 5.

Figure 6 shows an example about the focused path extraction for the fix revision shown in Figure 5. In it, one of the paths covered by the featured statement "if (_numberText == null)" (Line 224) is illustrated. The internal flow edges of the path are labeled with numbers among the control-flow graph of the example method "toByteArray". Based on the focused object $O_{\_numberText}$ of the featured statement, we slice the covered path as a focused path. The focused path includes only five statements (underlined in Figure 6) and their line numbers are 224, 228, 230, 232, and 236 (the return statements would be forcedly kept during the slicing).

2.2.3 Pattern Generation

In this step, we first try to match each focused path with the predefined usage scenarios of each bug-pattern template. Based on such matching, we further confirm whether the given fix revision fixes some real bug instance(s) conforming to some certain bug-pattern template(s). Based on the confirmed bug instances, we then generate project-specific bug patterns correspondingly.

Algorithm 2 shows the pattern-generation algorithm. In the algorithm, we take a focused path P and its included featured statement S_f as inputs, and produce the generated project-specific bug patterns *PSBPs* as output. In particular, we first load the proper bug-pattern template based on the featured statement S_f (Line 1). For example, if S_f is a null-check statement, we would load the predefined NPE template as *BPT* since S_f is a featured statement for NPE bugs. After that, guided by the usage scenarios *Sce* of the loaded template *BPT*, we track the usage states of the objects used in P (Lines 2-25). In the state-tracking process, we set the initial state of each focused object as State 1, and then update its states by visiting forward the statements in P.

For each statement s in P, we repeat the same analysis process as follows. If statement s_p preceding s tries but fails to create an object o (the flow edge between these two statements in the CFG is an exception-throwing edge), the effects of s_p impacting previously on the points-to information and the usage state of o are actually invalid. In such case, we would roll back such invalid effects (Lines 3-5). If s dereferences an object o, we use the dereference event e and its current precondition to update the usage state of o (Lines 6-8). In the algorithm, such state-updating processes are all guided by *Sce*: we determine whether and how to update the state of o by checking whether and how the corresponding state of *Sce* is transited under the same certain event and precondition. If s calls a method m on an object o, we use the method-call event e and its current precondition to update the usage state of o (Lines 9-11). If s makes an equality check (i.e., using operator "= =") on an object o, we first check whether the subsequent statement of s is located in the false branch of s. If yes, we negate the equality operator of the *if* statement (Lines 12-15). After that, we use the equality-check event e and its current precondition to update the usage state of o (Lines 16). If s assigns some value to an object o, we first update points-to information according to s, and then use the assignment event e to update the usage state of o (Lines 18-21). If s returns an object o, we use the return event e and its current precondition to update the usage state of o (Lines 22-24).

After analyzing the statements in P, for each focused object o, we check whether it is in a "correct" usage state at the exit of P and also whether its state-transition process involves at least one featured statement (Lines 26-27). If yes, we affirm that a bug instance on o conforming to *BPT* is really fixed in the given path. Based on such identified real bug instance, we further generate a project-specific bug pattern *BSBP* by concretizing *BPT* with o, and save it into *BSBPs*.

Note that, when S_f calls a method m, we would also generate project-specific bug patterns by concretizing *BPT* with the inner[2] and outer[2] objects of o that also define a method named as m. For example, the typical featured statement for the resource-leak bugs calls the method "*close()*". If the given fix revision fixes a resource-leak bug on a resource object o_b, we generate project-specific bug patterns not only for o_b, but also for its inner and outer objects that also define the same method (named as *"close()"*). The reason is that once a resource object is closed, its inner and outer resource objects usually should also be closed safely (sharing the same bug pattern).

For the fix example shown in Figure 5, we would match its focused path shown in Figure 6 with the usage scenarios of the NPE template. The focused path includes five statements (Lines 224, 228, 230, 232, and 236). After visiting Line 224, we update the usage state of the object $O_{\_numberText}$ that the variable *_numberText* points to as State 2 since $O_{\_numberText}$ is null-checked in the statement. After visiting Line 228, we update the usage state of $O_{\_numberText}$ as State 3 since $O_{\_numberText}$ is dereferenced in the statement. However, visiting the remaining statements does not change the usage state of $O_{\_numberText}$ further. Therefore, $O_{\_numberText}$ is finally in State 3, a "correct" usage state, at the exit of the focused path. Therefore, we affirm that the fix example fixes an NPE bug on $O_{\_numberText}$. Based on such bug instance, we concretize the NPE template with $O_{\_numberText}$ to generate a project-specific bug pattern (shown in Figure 7): "the class field

[2] When an object o is initiated, if another object o' is assigned to a field of o, we say that o' is nested into o, o nests o', o' is an inner object of o, and o is an outer object of o'.

'*_numberText*' could be null at runtime and dereferencing it without proper null-checks would lead to NPE bugs".

2.3 Detecting Sibling Bugs

In this phase, based on the inferred PSBPs, we detect sibling bugs remaining in the same project under analysis.

To collect basic program information for sibling-bug detection, we first conduct some basic analyses: object-nesting analysis, points-to analysis, and precondition analysis.

The object-nesting analysis performs a forward intra-procedural dataflow analysis to identify the information on nesting[2] relationships between objects. The points-to analysis performs a forward inter-procedural context-sensitive dataflow analysis to identify the points-to information for all reference variables. The points-to information of a variable provides the potential object(s) that it may point to, its potential actual type(s), and its potential aliases. The precondition analysis performs a forward intra-procedural dataflow analysis to identify the execution preconditions (i.e., groups of predicates) for each statement.

Sibling-bug detection performs a flow-sensitive inter-procedural CFG-based dataflow analysis, whose algorithm is similar to the defect-analysis algorithm described in our previous work [11]. By matching CFG paths of the method under analysis with the usage scenarios *Sce* of each inferred PSBP, we track the usage state of each focused object at any path point, guided by *Sce*. If any focused object is in a buggy state at the exit of a path, we treat the path as a buggy one and then report a sibling-bug warning for the focused object.

For each method under analysis, we use a forward worklist algorithm [14] over its CFG blocks to iteratively compute a fixed point over the usage states of all focused objects. In order to flexibly control the precision-cost tradeoff, we define the maximum depth of inter-procedural analysis as *maxDepth* (=3) and the maximum number of intra-procedural iterations as *maxIter* (=10). Before reaching the fixed point, we iteratively visit CFG blocks in the reverse-post order until *maxIter* is reached.

For each block *b* of *CFG*, we update its inflow value D_b^{in}, the possible usage states of all focused objects at the inflow side of *b*, by merging all the outflow values of its inflow blocks. However, if some inflow block tries to create an object *o* but fails, we would roll back its previous impact on the points-to information and the usage state of *o*. After computing D_b^{in}, we visit each statement *s* in *b* forward to propagate their impacts on D_b^{in} guided by *Sce*.

In this process, we propagate impacts for five types of statements: the method-call statement, the equality-check statement, the object-dereference statement, the assignment statement, and the return statement. Moreover, we propagate the impact of a statement only if its precondition does satisfy the required effect-taking precondition. In our predefined templates, a state-transition statement can be assigned with an effect-taking precondition, only under which the statement takes effect. For example, in the resource-leak bug-pattern template, the return statement takes effect only when the focused object is null.

After the analysis for each method under analysis, we check whether any focused object *o* is still in a buggy state at the exit of the method. If yes, we report a sibling-bug warning for the corresponding buggy path.

To improve the efficiency, we detect sibling bugs within proper scopes: if the focused object o_b of *PSBP* is a local object of a specific method, we detect bugs only within the method; if it is a private class field, we detect bugs within its belonging class file; if it is a protected class field, we detect bugs within its belonging package; if it is a public class field, we detect within the whole project under analysis.

Algorithm 2: Pattern Generation

Input:
P : the extracted focused path
S_f : the featured statement
Output:
$PSBPs$: the inferred program-specific bug patterns

```
1  load the proper bug pattern template BPT according to S_f;
2  foreach statement s ∈ P do
3      if the preceding statement of s tries but fails to create an object o then
4          roll back its effect on the points-to information and the usage state of o State_o based on the usage scenarios Sce of BPT;
5      end
6      if s dereferences an object o then
7          update State_o with such dereference event and its current precondition based on Sce;
8      end
9      if s calls a method m on an object o then
10         update State_o with such method-call event and its current precondition;
11     end
12     if s makes an equality-check on an object o then
13         if the subsequent statement of s falls in the false branch of s then
14             negate the equality-check operator of s;
15         end
16         update State_o with such equality-check event and its precondition;
17     end
18     if s assigns some value to an object o then
19         update points-to information according to s;
20         update State_o with such assignment event;
21     end
22     if s returns an object o then
23         update State_o with such return event and its current precondition;
24     end
25 end
26 foreach object o in the "correct" usage state do
27     if the state-transition process of o contains a featured statement then
28         generate a PSBP by concretizing BPT with o;
29         PSBPs ⟵ PSBP;
30     end
31 end
```

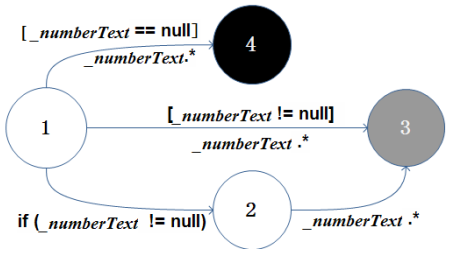

Figure 7. The inferred project-specific NPE bug pattern from the fix revision shown in Figure 5.

3. EVALUATIONS

Based on the proposed approach, we implemented a PSBP-based Sibling-Bug Detector (SBD) and conduct evaluations on it. This section presents our evaluation setup and evaluation results on SBD.

3.1 Evaluation Setup

We design our evaluations to address the following research questions:

- RQ1 (detection effectiveness): Can SBD effectively detect sibling bugs for real-world projects?
- RQ2 (tool complementarity): How high percentage of true SBD warnings cannot be detected by traditional bug detectors (e.g., FindBugs and PMD)?
- RQ3 (bug fixability): How difficult is it to fix SBD warnings?

In our evaluations, we select seven projects including Lucene, Tomcat, ANT, James, Maven, Xerces, and POI as the projects under analysis. We choose these projects because they are commonly used in the literature [8, 9, 10, 19, 27] and they are relatively mature and well-maintained open-source projects under the Apache Software Foundation.

To detect sibling bugs for these projects, we predefine two bug-pattern templates in SBD: the NPE bug-pattern template and the resource-leak bug-pattern template (shown in Section 2.1).

We predefine these two bug-pattern templates for two reasons. First, these two types of bugs tend to introduce multiple fixes [30]. For a resource-leak or NPE bug, its focused object can potentially leak or be null-dereferenced in many code locations, resulting in many sibling bugs. Therefore, fixing a resource-leak or NPE bug completely usually needs to handle all of its sibling bugs. However, developers can easily miss fixing some sibling bugs, resulting in submitting incomplete fix revisions. Second, traditional bug detectors are usually ineffective in detecting resource-leak or NPE bugs. Theoretically, all kinds of objects (resource objects) instantiated in any method under analysis may throw null-pointer exceptions (lead to resource leaks). However, in real-world projects, only a certain subset of objects (resource objects) should be null-checked (released). Without such concrete knowledge, traditional bug detectors usually produce too many false negatives (shown in Section 3.3) as well as false positives.

3.2 Detection Effectiveness

In this section, we evaluate the detection effectiveness of SBD by measuring the number of sibling bugs that SBD can effectively detect.

Figure 8 shows the evaluation methodology. Guided by the fix revisions whose log messages contain the keyword "NPE" or "resource leak", we randomly select 50 NPE fix revisions and 50 resource-leak fix revisions from the source-code repositories of the subjects (during this process, if a fix revision involves multiple changed files, we treat each file change as a standalone fix revision). Based on the selected fix revisions, we run SBD to detect sibling bugs within proper scopes. Then we manually verify the reported sibling-bug warnings based on the following process. We first check whether they have been fixed in subsequent historical revisions. If some warning has been fixed, we treat it as a true warning. If some warning has not been fixed and remains in the head revision, we report it as a new bug issue to the project community. If a reported warning is confirmed or fixed by developers, we also treat it as a true warning.

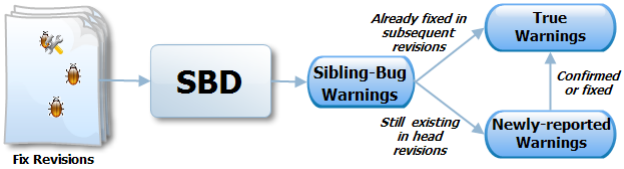

Figure 8. Evaluation methodology.

Table 1. The result summary for the SBD warnings on the 100 selected fix revisions on NPE or resource-leak bugs.

Bug Type	Sibling-Bug Warnings			
	All	Already-Fixed	Reported	Confirmed
NPE	41	8	26	7
Resource Leak	67	11	48	37
All	108	19	74	44

Table 1 summarizes the SBD warnings on the subjects. For each bug type, Columns "All", "Already-Fixed", "Reported", and "Confirmed" represent the total number of the related warnings, the number of the related warnings that have been already fixed by subsequent historical revisions in source-code repositories, the number of the related warnings that are reported as new bug issues, and the number of the related newly-reported warnings that have been confirmed or fixed by developers, respectively. SBD totally reports 41 NPE warnings. Through manual verification, we confirm that eight of them have already been fixed in subsequent historical revisions. We report 26 warnings remaining in head revisions as new bug issues to the related project issue-tracking systems. So far, 7 of the 26 newly-reported issues have been confirmed or fixed by developers. Similarly, SBD reports 67 resource-leak warnings. We confirm that 11 of these 67 warnings have already been fixed in subsequent revisions. We report 48 warnings remaining in head revisions as new issues. Among the 48 newly-reported warnings, 37 of them have been confirmed or fixed by their developers.

Table 2 shows the details of the already-fixed warnings. Columns "Bug Type", "Subject", and "Fixed Location" present the fixed bug type, the project name, and the fixed location of each fix revision under analysis, respectively. Column "# of SBD Warnings" presents the number of the warnings that are reported by SBD on each fix revision. Column "Already Fixed in" shows information about the subsequent revision(s) that the corresponding warnings were fixed in. For example, the revision 886113 of the POI project fixed an NPE bug in the method *"toString"* of the class *"LbsDataSubRecord"*. Based on such revision, SBD reports one NPE warning, and the warning has been fixed by a subsequent revision (Revision 892461).

Table 3 shows the details of the 42 bug issues that we newly report for the 74 SBD warnings (note that we may report several warnings in an issue) and their resolution statuses. For each subject, Column "#W" presents the number of the SBD warnings that still exist in the head revision, Column "Reported-Issue ID" presents the ID information of the issue that we report for the corresponding warnings, and Column "Status" presents the resolution status of the related issue.

Table 2. The 19 already-fixed SBD sibling-bug warnings.

Fix Revision			# of SBD Warning	Already Fixed in
Bug Type	**Subject**	**Fixed Location**		
Null Pointer Exception (NPE)	POI-886113	LbsDataSubRecord::toString	1	rev892461
	POI-657135	ListLevel::getSizeInBytes	1	rev1022456
	Xerces-318356	AbstractDOMParser::startDocument	2	rev318567
	Xerces-318586	XMLDTDScannerImpl::setInputSource	1	rev318859
	Lucene-476359	SegmentInfos::run	2	rev602055
	Maven-562710	AbstractJavadocMojo::getSourcePaths	1	rev562714
Resource Leak	Xerces-319282	XIncludeHandler::handleIncludeElement	1	rev319304
	Tomcat-423920	WebappClassLoader::findResourceInternal	1	rev915581
	Tomcat-423920	StandardServer::await	1	rev1066310
	Maven-740164	LatexBookRenderer::writeSection	1	rev1003021
	James-107920	MimeMessageJDBCSource::getInputStream	2	rev107975
	ANT-269449	FixCRLF::execute	1	rev269909
	ANT-269827	Replace::processFile	2	rev269961
	ANT-270637	ReplaceRegExp::doReplace	2	rev272826,905179

Table 3. The 74 newly-reported SBD sibling-bug warnings (reported in 42 bug issues).

Fix Revision			#W	Reported-Issue ID	Status
Bug Type	**Subject**	**Fixed Location**			
Null Pointer Exception (NPE)	POI-1142762	CharacterRun::getFontName	1	52662	Fixed[rev1243907]
	POI-1171628	MAPIMessage::set7BitEncoding	1	52664	Fixed[rev1244449]
	POI-1179452	ZipFileZipEntrySource::close	2	52665	Fixed[rev1244450]
	Xerces-319317	XSWildcardDecl::getNsConstraintList	1	XERCESJ-1551	Need Test Case
	Xerces-928735	RangeToken::dumpRanges	3	XERCESJ-1552	Need Test Case
	Xerces-320527	IdentityConstraint::getSelectorStr	1	XERCESJ-1554	Confirmed (Already Avoided)
	Lucene-219387	MultipleTermPositions::skipTo	3	LUCENE-3779	In Progress
	Lucene-407851	ParallelReader::getTermFreqVector	1	LUCENE-3780	In Progress
	Lucene-407851	ParallelReader::next,read,skipTo,close	2	LUCENE-3781	In Progress
	Lucene-499089	Directory::clearLock	1	LUCENE-3782	In Progress
	Lucene-698487	NearSpansUnordered::isPayloadAvailable	7	LUCENE-3783	In Progress
	Maven-554202	AbstractJavadocMojo::getSourcePaths	2	MJAVADOC-342	Fixed[rev1385200]
	Maven-712569	WebappStructure::getDependencies	1	MWAR-275	In Progress
Resource Leak	ANT-272185	XMLResultAggregator::writeDOMTree	1	52738	Fixed[rev1294340]
	ANT-272583	Javadoc::execute	1	52740	Fixed[rev1294345]
	ANT-269827	Replace::execute	4	52742	Fixed[rev1294360]
	ANT-270637	ReplaceRegExp::doReplace	1	52743	Fixed[rev1294780,rev1297127]
	James-108172	NNTPHandler::handleConnection	3	JAMES-1381	Confirmed
	James-108172	POP3Handler::handleConnection	2	JAMES-1382	Confirmed
	James-108172	RemoteManagerHandler::handleConnection	3	JAMES-1383	Confirmed
	James-108172	SMTPHandler::handleConnection	5	JAMES-1384	Confirmed
	Tomcat-730178	Catalina::stopServer,load	4	52724	Fixed[rev1297209]
	Tomcat-423920	MemoryUserDatabase::open	1	52726	Fixed[rev1297717]
	Tomcat-423920	HostCong::deployWAR	1	52727	Fixed[rev1297722]
	Tomcat-640273	CometConnectionManagerValve::lifecycleEven	1	52729	Fixed[rev1297729]
	Tomcat-1043157	JDTCompiler::getContents	1	52731	Fixed[rev1297769]
	Tomcat-1043157	ExpressionFactory::getClassNameServices	1	52732	Fixed[rev1297768]
	Tomcat-424429	NioEndpoint::run	1	52718	In Progress
	Tomcat-423920	WebappClassLoader::validateJarFile	1	52719	Fixed[r1298140, r1304483]
	Tomcat-777567	ManagerBase::run,setRandomFile	2	52720	Confirmed
	Tomcat-423920	StandardContext::cacheContext	1	52721	No Need to Fix (Unused code now)
	Tomcat-412780	HTMLManagerServlet::cacheContext	1	52722	Invalid
	Tomcat-907502	StandardManager::doUnload	2	52723	Fixed[rev1299036]
	Xerces-319937	ObjectFactory::findJarServiceProvider	1	XERCESJ-1556	In Progress
	Maven-935344	PmdReport::execute	1	MPMD-144	Fixed[rev1341161]
	Maven-729532	PmdReportTest::readFile	1	MPMD-145	Fixed[rev1340576]
	Maven-730089	CpdReport::writeNonHtml	1	MPMD-146	Fixed[rev1340575]
	Maven-1134539	Verifier::loadFile,displayLogFile	2	MVERIFIER-12	In Progress
	Maven-740164	LatexBookRenderer::writeSection	2	DOXIA-461	In Progress
	Maven-1003021	XdocBookRenderer::renderSection	1	DOXIA-464	In Progress
	Maven-740164	XHtmlBookRenderer::renderBook	1	DOXIA-462	In Progress
	Maven-1085807	TestUtils::readFile	1	MPLUGINTESTING-20	In Progress

For example, for the fix revision (Revision 1171628) of the POI project, there is one SBD warning that still exists in the head revision. We report it as a new issue with ID as 52664. Based on the issue, a POI developer fixes it with a new fix revision (Revision 1244449) and also expresses his appreciation to us on reporting the issue. For the warnings in Xerces-320527 and Tomcat-777567, the developers reply that the warnings have already been avoided by historical code changes, so there is no need to fix them in the head revisions.

However, some warnings are not confirmed. For Xerces-319317, we report one warning but the Xerces developers require us to

Table 4. Detection-effectiveness results of FindBugs and PMD on the SBD warnings already fixed or confirmed by developers.

Fix Revision			SBD	FindBugs	PMD
Bug Type	**Subject**	**Fixed Location**			
Null Pointer Exception (NPE)	POI-886113	LbsDataSubRecord::toString	1		
	POI-657135	ListLevel::getSizeInBytes	1		
	POI-1142762	CharacterRun::getFontName	1		
	POI-1171628	MAPIMessage::set7BitEncoding	1		
	POI-1179452	ZipFileZipEntrySource::close	2		
	Xerces-318356	AbstractDOMParser::startDocument	2		
	Xerces-318586	XMLDTDScannerImpl::setInputSource	1	1	
	Xerces-320527	IdentityConstraint::getSelectorStr	1		
	Lucene-476359	SegmentInfos::run	2		
	Maven-562710	AbstractJavadocMojo::getSourcePaths	1		
	Maven-554202	AbstractJavadocMojo::getSourcePaths	2		
Resource Leak	ANT-269449	FixCRLF::execute	1		
	ANT-269827	Replace::processFile	6		
	ANT-270637	ReplaceRegExp::doReplace	3	2	
	ANT-272185	XMLResultAggregator::writeDOMTree	1		
	ANT-272583	Javadoc::execute	1		
	Xerces-319282	XIncludeHandler::handleIncludeElement	1		
	Tomcat-423920	WebappClassLoader::findResourceInternal	1		
	Tomcat-423920	StandardServer::await	1		
	Tomcat-423920	WebappClassLoader::validateJarFile	1		
	Tomcat-423920	StandardContext::cacheContext	1		1
	Tomcat-423920	MemoryUserDatabase::open	1		1
	Tomcat-777567	ManagerBase::run,setRandomFile	2		
	Tomcat-907502	StandardManager::doUnload	2		2
	Tomcat-730178	Catalina::stopServer,load	4	2	1
	Tomcat-640273	CometConnectionManagerValve ::lifecycleEvent	1		
	Tomcat-1043157	JDTCompiler::getContents	1		
	Tomcat-1043157	ExpressionFactory::getClassNameServices	1		
	Maven-740164	LatexBookRenderer::writeSection	1		
	Maven-935344	PmdReport::execute	1		1
	Maven-729532	PmdReportTest::readFile	1		
	Maven-730089	CpdReport::writeNonHtml	1		1
	James-107920	MimeMessageJDBCSource::getInputStream	2		1
	James-108172	NNTPHandler::handleConnection	3		
	James-108172	POP3Handler::handleConnection	2		
	James-108172	RemoteManagerHandler::handleConnection	3		
	James-108172	SMTPHandler::handleConnection	5		
Total # of detected SBD warnings already fixed or confirmed			63	5	8

provide test cases to demonstrate the related bug's existence at runtime. For the warning in Tomcat-412780, a Tomcat developer resolves it as invalid because he considers that the corresponding resource leak would never happen at runtime. Other warnings are still in process and have not been resolved by their developers yet.

In summary, as shown in Table 1, SBD reports a total of 108 sibling-bug warnings for all subjects. Among these 108 warnings, 19 warnings have already been fixed in their subsequent historical revisions. Such result directly confirms that these warnings are true. In addition, 44 of the 74 newly-reported SBD warnings have also been confirmed or fixed by their developers. Note that although open-source projects are usually well maintained, SBD reports 44 new real bugs in their head revisions. Overall, at least 58.3% ((19+44)/108) of the SBD warnings are true. Table 2-3 indicate that the remaining sibling bugs are common for the two well-known bug types, and SBD can effectively locate remaining sibling bugs for these well-known bug types. Table 3 shows that 28 SBD warnings have been fixed by community developers with new fix revisions (i.e., the warnings shown in the rows with Column "Status" as "Fixed[revision_no]"; note that a row, representing a newly-reported issue, may correspond to multiple warnings), 16 warnings have been confirmed as real bugs (i.e., the warnings shown in the rows with Column "Status" as "Confirmed"), and 13 issues are still in progress of being investigated (i.e., the issues shown in the rows with Column "Status" as "In Progress"). Such result indicates that at least 59.5% ((28+16)/74) of the newly-reported warnings are true. Among the true warnings, about 63.6% (28/(28+16)) of them have been fixed by community developers.

The evaluation results show that SBD can effectively locate sibling bugs for real-world projects based on their existing fix revisions. With the help of SBD, developers can fix bugs more comprehensively and systematically.

3.3 Tool Complementarity

Section 3.2 shows that SBD can effectively detect NPE and resource-leak sibling bugs. In this section, we investigate how well SBD complements two existing widely-used tools FindBugs and PMD by measuring how high percentage of true SBD warnings cannot be detected by these two existing bug detectors.

In this evaluation, we first run FindBugs and PMD on the subjects to collect their reported NPE and resource-leak warnings. Then,

we manually check whether each of the true SBD warnings is also reported by FindBugs or PMD. During this process, we consider only the already-fixed or confirmed SBD warnings (shown in Tables 2 and 3) as true SBD warnings.

Table 4 summarizes the number of the true SBD warnings for each related fix revision, and also the number of the true SBD warnings that FindBugs or PMD also reports for the related fix revision (when the number is 0, we leave the table cell as empty). For example, for James-107920, SBD reports 2 true NPE warnings for the class file "MimeMessageJDBCSource.java". However, FindBugs reports none of the 2 true warnings, and PMD reports only 1 of the 2 true warnings. For Tomcat-730178, SBD reports 4 true resource-leak warnings, while FindBugs reports only 2 of the 4 true warnings and PMD reports only 1 of the 4 true warnings.

In total, FindBugs and PMD report only 13 (5+8) of the 63 true SBD warnings. Among the true SBD warnings, FindBugs cannot report 92.1% ((63-5)/63) of them and PMD cannot report 87.3% ((63-8)/63) of them. Such result shows that SBD is complementary with these existing tools since most of the true warnings that SBD reports cannot be detected by these existing tools.

3.4 Bug Fixability

In this section, we investigate the difficulty to fix SBD warnings. The difficulty of fixing bugs is indeed case by case. However, we believe that fixing SBD warnings would not be difficult, since these warnings are detected based on existing fixes, which can provide good references to fix these warnings.

This section uses two examples of actual fixes to show the simplicity of the fixing process for SBD warnings. For the example from the POI project, Figure 9 shows the first fix revision, the sibling bug identified by SBD, and the second fix revision. The added lines of each revision are in bold and labeled with the "+" symbols while the deleted lines are labeled with the "-" symbols. The first fix revision fixes an NPE bug on the class field "nameIdChunks" in the method "set7BitEncoding". Such fix indicates that "nameIdChunks" could be null at runtime and should be null-checked before dereferenced. However, it is still dereferenced directly without a null-check in the method "has7BitEncodingStrings". Based on such fix revision, SBD reports an NPE warning on the "nameIdChunks" field. After we report the warning as a new issue for the POI project, a POI developer fixes it with a new fix revision by simply committing the same fix activities as the first fix. Figure 10 shows another example from Tomcat. The first fix revision fixed a resource-leak bug on the local object "cometEvent" in the method "lifeCycleEvent". However, when the statements in the TRY block (e.g., commetEvent.setEvent Type(…)) throw exceptions, the statement "cometEvent.close()" would have no chance to be executed. In such cases, the "cometEvent" object would leak. Based on such fix revision, SBD reports a resource-leak warning on "cometEvent". After we report the warning as a new issue for the Tomcat project, a Tomcat developer fixes it by just simply closing the resource object "cometEvent" in the FINALLY block instead of the TRY.

These two examples of actual fixes show that the fixing process on a SBD warning would not be difficult: to fix a SBD warning, developers tend to easily imitate one of its related existing fixes, by simply replicating it or making a similar fix around the reported location(s).

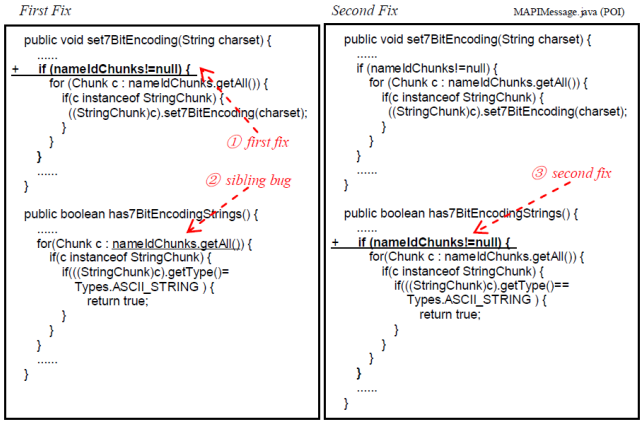

Figure 9. An example of actual fixes from the POI project on an NPE sibling-bug warning reported by SBD.

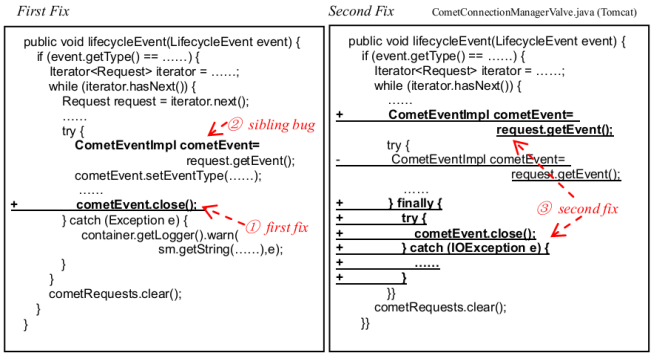

Figure 10. An example of actual fixes from the Tomcat project on a resource-leak sibling-bug warning reported by SBD.

4. THREATS TO VALIDITY

In this section, we summarize main threats to external, internal, and construct validity of our evaluations. There are two main threats to external validity. The first one is about the representativeness of the subject projects that we select in the evaluations. The results of our evaluations may be specific only to these projects. To reduce this threat, we choose different types of projects as subjects and evaluate SBD on multiple subjects. The second one is about the extendability of SBD on other bug types. In the evaluations, we show that SBD can effectively detect sibling bugs for NPE bugs and resource-leak bugs. However, SBD may not be applied easily for other bug types. To reduce this threat, we separate the specification process of bug-pattern templates for different bug types from the detection-logic implementation in SBD. We also propose a specification notation for bug-pattern templates, and such notation is applicable or can be easily extended to support other bug types (e.g., array index out-of-bounds, uninitialized variable reads, and unmatched lock/unlock pairs). Based on the notation, we define a bug-pattern template for each bug type with a standalone XML file in SBD. The main threat to internal validity is about the predefined bug-pattern templates for NPE and resource-leak bugs. In the evaluations, we manually summarize the featured statements and the usage scenarios for the two well-known bug types. However, our summarization may not be complete or accurate enough. To reduce this threat, we carry out the summarization by studying

plenty of actual fix revisions for NPE and resource-leak bugs. The main threat to construct validity is that, in the evaluations, we treat a warning as a true one if a real developer commits an actual fix on it. However, since typical fixes for some warnings are quite simple (e.g., adding a guarding *if* statement or invoking a resource-releasing method on a resource object in a finally block) and harmless, developers may choose to "fix" these warnings conservatively even if the "fixes" could be redundant or unnecessary. In such cases, the related warnings would be mistreated as true ones.

5. RELATED WORK

Identification of Incomplete Fixes. Kim *et al.* [9] propose an approach to identify incomplete fixes for exception bugs (e.g., null-pointer-exception bugs), with the concept of "bug neighborhood". A bug neighborhood refers to a set of related flows of invalid values [9]. Their approach requires external users to pinpoint concrete statement pairs that can introduce bugs in a program under analysis. Based on each statement pair, their approach detects whether the statement pair has been fixed in the modified version of the program. In contrast, our approach does not require users to manually pinpoint anything, and our approach can systematically detect remaining sibling bugs for subject programs. In addition, our approach is general-purpose and extensible for various well-known bug types.

Mining of Generic Bug Patterns. Various specification-mining approaches [28, 29, 31] have been proposed in the literature. Specifications mined by these approaches can be used to guide the bug-pattern extraction: each violation to a specification can be treated as a bug pattern. Existing approaches mine specifications for API libraries mainly from three kinds of sources: API client programs, API library source code and historical revisions, and API library documents. However, the specifications that these approaches can produce are mainly about the usage rules of classes defined in third-party libraries. Specifications that are mined by these approaches on a certain library would be applicable for all programs relying on the same library. Therefore, these approaches are suitable for mining generic bug patterns. In contrast, our PSBP-inference process aims to infer project-specific bug patterns, which are directly related to objects instantiated in specific projects under analysis.

Static Detection of NPEs and Resource Leaks. Most existing static bug-detection tools (e.g., ESC/Java [3], FindBugs [4], PMD [17], JLint [7], and Lint4J [12]) also provide NPE and resource-leak detectors. Among them, ESC/Java is a specification-based violation checker, which requires specifications to be manually provided by developers. ESC/Java tries to find all violations to a specified null/non-null annotation, and usually produces too many false positives. Other tools use typical static-analysis techniques to detect NPE or resource-leak warnings based on generic defect patterns. These tools usually report too many false positives or negatives. Besides these popular tools, various research approaches on detecting NPE and resource-leak bugs have been proposed. Spoto *et al.* [23] propose a technique for inferring non-null annotations to improve the precision of their null-pointer analysis. Their inference of non-null annotations is based on some heuristics (e.g., the initialized instances or static fields are treated as always non-null). Hovemeyer *et al.* [6] propose an approach based on non-standard NPE analysis. In their approach, they also use annotations as a convenient lightweight mechanism to improve the precision of their analysis. Weimer and Necula [27] propose an approach for detecting system resource-leak problems (in Java programs) resulted from incorrect exception handlings. Their approach includes an unsound but simple path-sensitive intra-procedural static analysis to find resource leaks. Shaham *et al.* [20] propose a conservative static analysis based on canonical abstraction to verify the safety of the synthesized free operations for dynamically-allocated objects. Their analysis could be used to automatically insert resource-releasing operations to prevent resource leaks. Charem and Rugina [1] propose a similar approach with a less-expensive analysis. Dillig *et al.* [2] propose the CLOSER approach to perform a modular and flow-sensitive analysis to determine "live" system resources at each program point. Torlak *et al.* [25] propose a scalable flow-sensitive context-sensitive inter-procedural resource-leak analysis, which relies on less-expensive alias abstractions. Compared with these approaches, our approach infers PSBPs from actual fix revisions, and then applies the inferred PSBPs to detect the remaining sibling bugs.

6. CONCLUSION

In this paper, we have proposed a general-purpose approach for detecting sibling bugs for various bug types, and implemented a tool called Sibling-Bug Detector (SBD). Based on existing fix revisions, our approach first automatically infers Project-Specific Bug Patterns (PSBPs) hidden and reflected by the fix revisions, and then applies the inferred PSBPs to detect their related remaining sibling bugs for the projects under analysis. Through evaluations, we have shown that SBD effectively reports 63 true sibling-bug warnings for seven real-world open-source projects, while two existing popular static bug detectors (FindBugs and PMD) cannot report most of them.

Although our approach has been shown to be effective, the approach still has some limitations. First, our proposed specification notation for bug-pattern templates is general-purpose for bug types involving only one single object. Currently, for each bug type, we use a Finite State Automaton (FSA) to summarize the typical usage scenarios of its focused object. However, once a bug type involves *multiple* interacting objects (e.g., *an array list of java.util.ArrayList cannot be updated when it is being traversed by an iterator of java.util.Iterator*) [13], our approach is not applicable. Second, in SBD, the predefined bug-pattern templates are still limited. With more pattern templates predefined and applied, the generality of SBD could be further improved.

To make SBD more practical, we plan to extend SBD to locate sibling bugs for more well-known bug types (e.g., data race, dead lock, and buffer overflow) in our future work. During this process, we plan to keep exploring better approaches to specifying bug-pattern templates, inferring PSBPs, and detecting sibling bugs.

7. ACKNOWLEDGMENTS

The authors from Peking University are sponsored by the National Basic Research Program of China (973) (Grant No. 2009CB320703, 2011CB302604), the National Natural Science Foundation of China (Grant No. 61121063, 61033006), the High-Tech Research and Development Program of China (Grant No. 2012AA011202, 2013AA01A213) and the Key Program of Ministry of Education, China under Grant No. 313004. Tao Xie's work is supported in part by NSF grants CCF-0845272, CCF-0915400, CNS-0958235, CNS-1160603, an NSA Science of Security Lablet grant, a NIST grant, and NSF of China No. 61228203. We are thankful for the tremendous advice from Sung Kim in the initial phase of this work.

8. REFERENCES

[1] S. Cherem and R. Rugina. Region analysis and transformation for Java programs. In Proc. of the 4th Int. Symposium on Memory Management, ISMM '04, pages 85-96, New York, NY, USA, 2004. ACM.

[2] I. Dillig, T. Dillig, E. Yahav, and S. Chandra. The CLOSER: automating resource management in Java. In Proc. of the 7th Int. Symposium on Memory Management, ISMM '08, pages 1-10, New York, NY, USA, 2008. ACM.

[3] ESC/Java. http://en.wikipedia.org/wiki/ESC/Java.

[4] FindBugs. http://findbugs.sourceforge.net/.

[5] Z. Gu, E. T. Barr, D. J. Hamilton, and Z. Su. Has the bug really been fixed? In Proc. of the 32nd ACM/IEEE Int. Conf. on Software Engineering - Volume 1, ICSE'10, pages 55-64, New York, NY, USA, 2010. ACM.

[6] D. Hovemeyer, J. Spacco, and W. Pugh. Evaluating and tuning a static analysis to find null pointer bugs. In Proc. of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE'05, pages 13-19, New York, NY, USA, 2005. ACM.

[7] Jlint. http://jlint.sourceforge.net/.

[8] S. Kim and M. D. Ernst. Which warnings should I fix first? In Proc. of the 6th joint meeting of the European Software Engineering Conf. and the ACM SIGSOFT Symposium on Foundations of Software Engineering, ESEC/FSE'07, pages 45-54, New York, NY, USA, 2007. ACM.

[9] M. Kim, S. Sinha, C. Görg, H. Shah, M. J. Harrold, and M. G. Nanda. Automated bug neighborhood analysis for identifying incomplete bug fixes. In Proc. of the 3rd Int. Conf. on Software Testing, Verification and Validation, ICST '10, pages 383-392, Washington, DC, USA, 2010. IEEE Computer Society.

[10] G. Liang, L. Wu, Q. Wu, Q. Wang, T. Xie, and H. Mei. Automatic construction of an effective training set for prioritizing static analysis warnings. In Proc. of the 25th IEEE/ACM Int. Conf. on Automated Software Engineering, ASE'10, pages 93-102, New York, NY, USA, 2010. ACM.

[11] G. Liang, Q. Wu, Q. Wang, and H. Mei. An effective defect detection and warning prioritization approach for resource leaks. In Proc. of the 36th Annual IEEE Computer Software and Applications Conference (COMPSAC 2012), pages 119-128, Izmir, Turkey, July 16-20, 2012.

[12] Lint4j. http://www.jutils.com/.

[13] N. A. Naeem and O. Lhotak. Typestate-like analysis of multiple interacting objects. In Proc. of the 23rd ACM SIGPLAN Conf. on Object-Oriented Programming Systems Languages and Applications, OOPSLA'08, pages 347-366, New York, NY, USA, 2008.

[14] F. Nielson, H. R. Nielson, and C. Hankin. Principles of Program Analysis. Springer Publishing Company, Incorporated, 2010.

[15] A. Orso, N. Shi, and M. J. Harrold. Scaling regression testing to large software systems. SIGSOFT Software Eng. Notes, 29(6), pages 241-251, Oct. 2004.

[16] K. Pan, S. Kim, and E. J. Whitehead, Jr. Toward an understanding of bug fix patterns. Empirical Software Engineering. 14(3), pages 286-315, June 2009.

[17] PMD. http://pmd.sourceforge.net/.

[18] R. Purushothaman and D. E. Perry. Toward understanding the rhetoric of small source code changes. IEEE Trans. Softw. Eng., 31(6):511-526, June 2005.

[19] N. Rutar, C. B. Almazan, and J. S. Foster. A comparison of bug finding tools for Java. In Proc. of the 15th Int. Symposium on Software Reliability Engineering, ISSRE'04, pages 245-256, Washington, DC, USA, 2004. IEEE Computer Society.

[20] R. Shaham, E. Yahav, E. K. Kolodner, and M. Sagiv. Establishing local temporal heap safety properties with applications to compile-time memory management. In Proc. of the 10th Int. Conf. on Static Analysis, SAS'03, pages 483-503, Berlin, Heidelberg, 2003. Springer-Verlag.

[21] J. Sliwerski, T. Zimmermann, and A. Zeller. HATARI: raising risk awareness. SIGSOFT Softw. Eng. Notes, 30(5):107-110, Sept. 2005.

[22] J. Sliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? SIGSOFT Softw. Eng. Notes, 30(4):1-5, May 2005.

[23] F. Spoto. Precise null-pointer analysis. Softw. Syst. Model., 10(2):219-252, May 2011.

[24] L. Tan, D. Yuan, G. Krishna, Y. Zhou. /*iComment: Bugs or bad comments?*/. In Proc. of the 21st Symposium on Operating Systems Principles, pp.145-158, Stevenson, USA, Oct. 14-17, 2007.

[25] E. Torlak and S. Chandra. Effective interprocedural resource leak detection. In Proc. of the 32nd ACM/IEEE Int. Conf. on Software Engineering - Volume 1, ICSE'10, pages 535-544, New York, NY, USA, 2010. ACM.

[26] J. Tucek, W. Xiong, and Y. Zhou. Efficient online validation with delta execution. SIGPLAN Not., 44(3):193-204, Mar. 2009.

[27] W. Weimer and G. C. Necula. Mining temporal specifications for error detection. In Proc. of the 11th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, pp.461-476, Edinburgh, UK, Apr. 4-8, 2005.

[28] W. Weimer and G. C. Necula. Finding and preventing run-time error handling mistakes. In Proc. of the 19th Annual ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA'04, pages 419-431, New York, NY, USA, 2004. ACM.

[29] Q. Wu, G. Liang, Q. Wang, T. Xie, and H. Mei. Iterative mining of resource-releasing specifications. In Proc. of the 26th IEEE/ACM Int. Conf. on Automated Software Engineering, ASE'2011, pages 233-242, 2011.

[30] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram. How do fixes become bugs? In Proc. of the 19th ACM SIGSOFT Symposium and the 13th European Conf. on Foundations of Software Engineering, ESEC/FSE'11, pages 26-36, New York, NY, USA, 2011.

[31] H. Zhong, L. Zhang, T. Xie, and H. Mei. Inferring resource specifications from natural language API documentation. In Proc. of the 24th Int. Conf. on Automated Software Engineering, pp.307-318, Auckland, New Zealand, Nov. 16-20, 2009.