Memories of Bug Fixes

Sunghun Kim
Department of Computer Science
University of California, Santa Cruz
Santa Cruz, CA, USA
hunkim@cs.ucsc.edu

Kai Pan
Department of Computer Science
University of California, Santa Cruz
Santa Cruz, CA, USA
pankai@cs.ucsc.edu

E. James Whitehead, Jr.
Department of Computer Science
University of California, Santa Cruz
Santa Cruz, CA, USA
ejw@cs.ucsc.edu

ABSTRACT
The change history of a software project contains a rich collection of code changes that record previous development experience. Changes that fix bugs are especially interesting, since they record both the old buggy code and the new fixed code. This paper presents a bug finding algorithm using bug fix *memories*: a project-specific bug and fix knowledge base developed by analyzing the history of bug fixes. A bug finding tool, BugMem, implements the algorithm. The approach is different from bug finding tools based on theorem proving or static model checking such as Bandera, ESC/Java, FindBugs, JLint, and PMD. Since these tools use pre-defined common bug patterns to find bugs, they do not aim to identify project-specific bugs. Bug fix memories use a learning process, so the bug patterns are project-specific, and project-specific bugs can be detected. The algorithm and tool are assessed by evaluating if real bugs and fixes in project histories can be found in the bug fix memories. Analysis of five open source projects shows that, for these projects, 19.3%-40.3% of bugs appear repeatedly in the memories, and 7.9%-15.5% of bug and fix pairs are found in memories. The results demonstrate that project-specific bug fix patterns occur frequently enough to be useful as a bug detection technique. Furthermore, for the bug and fix pairs, it is possible to both detect the bug and provide a strong suggestion for the fix. However, there is also a high false positive rate, with 20.8%-32.5% of non-bug containing changes also having patterns found in the memories. A comparison of BugMem with a bug finding tool, PMD, shows that the bug sets identified by both tools are mostly exclusive, indicating that BugMem complements other bug finding tools.

Categories and Subject Descriptors
D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement – *Restructuring, reverse engineering, and reengineering,* D.2.8 [**Software Engineering**]: Metrics – *Product metrics,* K.6.3 [**Management of Computing and Information Systems**]: Software Management – *Software maintenance*

General Terms
Algorithms, Measurement, Experimentation

Keywords
Fault, Bug, Fix, Bug finding tool, Prediction, Patterns

SIGSOFT'06/FSE-14, November 5–11, 2006, Portland, Oregon, USA.

1. INTRODUCTION
Bugs are prevalent in software. As a result, any technique that can automatically detect software bugs and suggest fixes will lead to fewer delivered bugs and improved software quality. Many automatic bug finding tools have been proposed, including Bandera [6], ESC/Java [11], FindBugs [14], JLint [1], and PMD [5]. They use a range of techniques to detect bugs and suggest fixes, including pre-defined bug patterns [1, 14], theorem proving [11], and model-checking [6]. These bug finding tools adopt a horizontal approach, using techniques that are applicable across all projects. To date, there are very few tools using the vertical approach of leveraging patterns in a specific project and performing project-specific bug finding. Recent work using this vertical approach includes [18], which focuses on detecting bugs in method usage pairs, and [28] which focuses on return value checking. In this paper, we present a vertical bug finding approach that extracts and memorizes a broad range of patterns in buggy code and uses the previous bug patterns of a specific project to find project-specific bugs in new changes or other parts of the source code.

One of the common bugs detectable by horizontal bug finding tools is the *null dereferencing bug,* shown in Figure 1. The code tries to reference '*bar*' when it is null. The correct behavior is to check whether '*bar*' is null, printing the '*foo'* field only if this is not the case.

```
if (bar==null) {
  System.out.println(bar.foo);
}
```

Figure 1. Example null dereferencing bug.

The bug in Figure 1 is easily detected using horizontal bug finding techniques, and the null dereferencing bug is one of many kinds of bugs that exist across software projects. However, we believe that there are many project-specific bugs, since different projects have different requirements, business logic, and semantics. Consider two bug fix examples from the Eclipse JDT project, shown in Figure 2. Lines starting with "-" show buggy code, and lines starting with "+" show the corresponding bug fix.

The example shows two separate instances in the history of two different files where an incorrect condition check, *isOpen(),* is removed and replaced with the correct condition check, *hasJavaNature()*.This example is representative of a large class of bugs that are project-specific and involve the use of project-specific abstractions and conventions. These bugs *cannot* be detected by existing horizontal bug finding tools [1, 5, 6, 11, 25], since these kinds of design or implementation details are usually not formally described, and change over time. The knowledge used to perform the bug fix shown in Figure 2 is common among the Eclipse core developers, part of their collective memory. It is not easy for new developers to learn such knowledge, and even

core developers sometimes forget, committing the same mistakes over again.

```
JavaProject.java at transaction 2024 (Fix for bug 28434)
- if (requiredProjectRsc.exists() &&
-     requiredProjectRsc.isOpen()) {;
+if(JavaProject.hasJavaNature(requiredProjectRsc))
```

```
DeltaProcessor.java at transaction 1945 (Fix for bug 27499)
- boolean isOpened=proj.isOpen();
- if (isOpened && this.hasJavaNature(proj))
+ if (JavaProject.hasJavaNature(proj))
```

Figure 2. Repeated bug fix examples in Eclipse. The '–' and highlighting indicate buggy code and the '+ 'indicates a corresponding fix. The change log messages for each transaction indicate they are bug fixes.

We can learn from previous mistakes to keep project-specific bugs from occurring again. A long-developed project usually has a software configuration management (SCM) repository that records a great number of bug fix changes. These bug fix changes record the location of bugs as well as their fixes, the solutions to the bugs. By extracting and saving the code patterns found in buggy code, it is possible to detect potential bugs in new changes. This paper presents an approach for building project-specific bug and fix memories from project change histories.

The term "memories" is used to describe a database that stores bug and fix pattern instances extracted from bug fix changes in a project's development history. An algorithm extracts pattern instances from bug fix changes by parsing, normalizing and filtering the code in the bug or fix area. The parsing step extracts syntax components from the code, the normalization process generalizes the syntax structure for matching similar code, and the filtering step eliminates noise in component matching. These extracted pattern instances are stored in the memories database for matching future bugs. A bug finding tool, BugMem, uses the memories for detecting project-specific bugs and suggesting fixes.

After applying our approach on five open source projects, we find that 19.3%-40.3% of the bugs and 7.9%-15.5% of bug and fix pairs repeat in the history. The results demonstrate that project-specific bug fix patterns occur frequently enough to be useful as a bug detection technique. Furthermore, for the bug and fix pairs, it is possible to both detect the bug and provide a strong suggestion for the fix.

We compared BugMem with a bug finding tool based on a static syntax checker, PMD, and found that the identified bug sets by PMD and by BugMem are largely exclusive. This indicates that BugMem is not meant to replace conventional bug finding tools and can be used with other bug finding tool to maximize the bug detecting ability.

The remainder of the paper begins by presenting algorithms to build bug fix memories from project change histories (Section 2) and then evaluates how well the memories match real bug fixes in a project's change history (Section 3). We next describe the BugMem tool (Section 4). The paper ends with related work (Section 5) and conclusions (Section 6).

2. BUILDING BUG FIX MEMORIES

To build memories of bug fixes, we must identify those changes in a software project history where a bug was fixed. The first step is to extract source code, change logs, and source code changes (deltas) from a project's SCM repository. Kenyon [3], a system that extracts source code change histories from CVS and Subversion, is used for this step. Kenyon automatically checks out the source code of each transaction—the set of file changes in one commit to the SCM system—and extracts change information such as the change log message, author, change date, source code, and change deltas.

A file change contains a list of region pairs that show the differences between two file versions; each region is called a *hunk* ***(H)***, as shown in Figure 3. A hunk, ***H***, consists of a set of source code lines. Within SCM systems, a file change that involves modification of a single line is recorded as the deletion of the old line, and addition of the new line. We capture this notion of changes recorded as deletions and additions with the concept of a hunk pair (***HP***). A hunk pair consists of a deleted hunk (***DH***) representing lines deleted from the prior version and the corresponding added hunk (***AH***) with lines added in the new version, i.e. ***HP*** = ***(DH, AH)***. We exclude hunks that include only import statements, comments, or code format changes, since most of these changes do not affect program behavior.

Traditionally, bugs are identified in software by examining test executions for incorrect output, performing software inspections, or running static analysis tools. Our method for bug identification is somewhat different, in that we assume that developers have been using these traditional methods for bug identification throughout a project's evolution, and have been fixing the buggy code. We use prior bug fix experience to identify future bugs.

Bug fix changes are identified by mining SCM change log messages. Two approaches are used for this step: searching for keywords such as "Fixed" or "Bug" [20] and searching for references to bug reports stored in a bug tracking system, such as "#42233" [7, 10, 26]. This allows us to identify whether an entire transaction contains a bug fix.

In bug fix changes, we start by assuming that deleted hunks (***DH***) are bug hunks (***BH***), and added hunks (***AH***) are fix hunks (***FH***), since by deleting the lines in ***DH*** a bug was removed, and by adding the lines in ***AH*** a bug was fixed. Formally, ***BH*** = ***DH*** and ***FH*** = ***AH*** if it is a bug fix change.

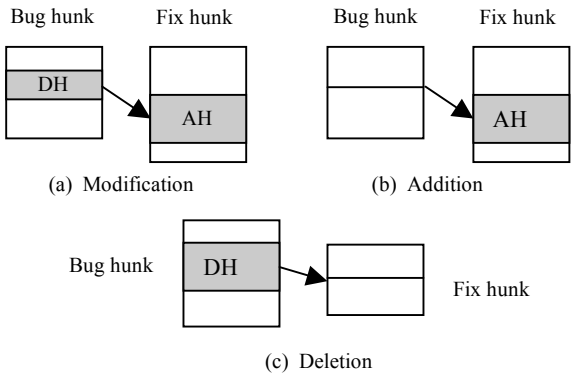


Figure 3. The three types of hunk pairs in bug fix changes

As is shown in Figure 3, there are three types of hunk pairs:

$$type(HP) = \begin{cases} addition, \text{ if } |DH| = 0 \\ deletion, \text{ if } |AH| = 0 \\ modification, \text{ otherwise} \end{cases}$$

All addition types of hunk pairs are ignored since bugs are identified by examining the deleted hunk, which is empty in addition hunks (i.e., |DH|=0).

The bug fix memories (***M***) are the union of the bug memories (***BM***) and fix memories (***FM)***: **M** = ***BM*** ∪ ***FM***. Bug memories (***BM***) and fix memories (***FM***) are the union of *components* extracted from the bug hunks or fix hunks in a project (a detailed description of the component extraction process is given in Section 2.1):

$$\boldsymbol{BM} = \bigcup extract(BH), \boldsymbol{FM} = \bigcup extract(FH)$$

Put another way, the memories database for a specific project is built by iterating over each bug fix transaction and applying the component extraction algorithm. Extracted components are saved in the memories database. The overall process for building the bug fix memories database is sketched in Figure 4.

```
M = null
for (n = 1 to N) {
  gather bug and fix hunks for transaction n
  M = M ∪ extract(BH) ∪ extract(FH) for each bug and fix
hunk pair in transaction n
}
```

Figure 4. The process for building bug fix memories. The total number of transactions is *N*.

Once constructed, the bug fix memories can be used to detect potential bugs in new or existing source code and provide corresponding fix examples. The *find* function, which takes components and memories as inputs and returns a matched hunk pair set (***MatchedHP***), is defined as:

$$find(component, M) \rightarrow MatchedHP$$

$$find(extract(BH), M) \rightarrow MatchedHP$$

$$find(extract(FH), M) \rightarrow MatchedHP$$

The next sections detail the process of building a project's bug fix memories database.

2.1 Extracting Components from Hunks

The code in bug hunks represents the mistakes developers have made throughout a project's history, while the code in fix hunks contains solutions to these mistakes. We want to learn from a project's history so we can prevent mistakes similar to those already observed and provide useful suggestions for how to repair these errors. Intuitively, to achieve this goal we must record the code found in bug hunks as well as the corresponding fixes. But how?

Naively, we can directly save all the code found in a bug hunk for use in future bug detection. Unfortunately, code in a new change typically does not have an exact match with the buggy code in the history. As a result, this approach would miss cases in which new code has a similar, but not equivalent, structure as the stored buggy code. In order to take a full advantage of the buggy code in the history, it is necessary to perform a series of steps that break the code down into its constituent parts, then abstract these parts into more general patterns.

Central to our approach is an algorithm for extracting syntax patterns, called *components*, from hunks, which are saved to the *memories* database. The extraction process can be expressed as: extract (***H***) → {c | c is a component in ***H***}. More specifically, extract (***BH***) → {c | c is a component in ***BH***}, and extract (***FH***) → {c | c is a component in ***FH***} for bug and fix hunks. To extract components from hunks, the entire source code file is parsed to extract components. Then we collect only those components that fall into hunks. Currently, bug fix memories can only be constructed from programs written in Java, due to the ease of parsing this language.

The rest of this section explains the component extraction algorithm, which is an implementation of the function, extract (***H***). The algorithm consists of four steps, raw component extraction, normalization, information filtering, and diff filtering.

2.1.1 Raw Component Extraction

In this step, the source code inside a hunk is parsed to burst out the individual syntactic elements found there. We begin by preprocessing the code to remove all whitespace and blank lines, so that formatting differences do not affect patterns. We differentiate between composite statements such as if, for, while, etc. and those that are simple such as method calls, assignment, etc. Code is further processed to concatenate all multi-line simple statements into a single line. Additional processing ensures that the conditional predicates of if, for, while, etc. all lie on a single line. This yields us basic *syntax lines*.

```
if (foo.flag>=5 && foo.ready()) {
  i=1;
  foo.create("example");
  initiate(5,bar);
}
```

Figure 5. Example code in a bug hunk (the type of the variable *foo* is *Foo*, the type of *i* is *int*, and the type of *bar* is unknown).

Figure 5 shows an example of code in a bug hunk, used as a running example. Four basic syntax lines are extracted from this code: (1) *if (foo.flag>=5 && foo.ready())*; (2) *i=1*; (3) *foo.create ("example")*; and (4) *initiate(5,bar)*.

In the implementation of the component extractor, the Java parser only performs a single-pass scan of individual files, so types of some variables are unknown. This is the reason why the type of a variable or an expression is sometimes unknown, as with *bar* in Figure 5 above.

Raw components are extracted based on the abstract syntax tree of a basic syntax line. More specifically, the set of non-leaf nodes in a syntax line's abstract syntax tree are its raw components. For example, from the syntax line *if (foo.flag>=5 && foo.ready())* in Figure 5, six raw components are extracted: (1) *foo.flag*, (2) *foo.flag>=5,* (3) *ready()*, (4) *foo.ready()*, (5) *foo.flag>=5 && foo.ready()*, and (6) *if (foo.flag>=5 && foo.ready())*. Note that *foo.flag* is the parent node of leaf nodes *foo* and *flag*, and string literals are treated as non-leaf nodes.

A raw component can be one of the four high-level kinds: *static Java call*, *Java call*, *user-defined call*, or *non-call*. If a component does not represent a method call, it is of the non-call kind. If a component represents a method call, and the method call is an invocation of a Java core class or a static field of a Java core class, it is a *static Java call*. For example, the components *System.out.println("Hello")* and *Integer. parseInt("12")* are static Java calls. If a component represents a method call to a user-defined object whose type is in the Java core classes, it is called a *Java call*. A method call to a user-defined method is a *user-defined call*. This classification of components is used in setting component search options, described in Section 2.2.

The data for a component consists of a component string and an actual parameter list. A non-call component has only component string data. For example, the component string for *foo.flag* is "*foo.flag"*. For a method call component, the method name and actual parameter list are represented separately. The component string for a method call component carries method name and parameter number information. For example, the component string for method calls *initiate()*, *initiate(5),* and *initiate(5, 9)* are *initiate()*, *initiate(.)*, and *initiate(,)* respectively. The actual parameter list for *initiate(5, 9)* contains "5" and "9". Method call components are represented in this way to support component matching, discussed further in Section 2.2. All the raw components extracted from the example code in Figure 5 are shown in the left part of Figure 6. Note that for each method call component there is an actual parameter list following the component string.

foo.flag	*Foo.flag*,
foo.flag>=5	*Foo.flag>=5*
	Foo.flag>=int
ready()	*ready()*
foo.ready()	*Foo.ready()*
foo.flag>=5 && foo.ready()	*Foo.flag>=5 && Foo.ready()*
	Foo.flag>=int && Foo.ready()
if (foo.flag>=5 && foo.ready())	*if (Foo.flag>=5 && Foo.ready())*
	if (Foo.flag>=int && Foo.ready())
i=1	*int=1*
	int=int
"example"	*"example"*
	String
create(.) "example"	*create(.) String*
foo.create(.) "example"	*Foo.create(.) String*
initiate(,) 5, bar	*initiate(,) int, **

Figure 6. Raw components (left) and components after normalization (right).

2.1.2 Normalization

The purpose of extracting components is to discover characteristics from the code in a bug or fix hunk so they can be used to match similar characteristics in new changes.

To extend the possibility of matching similar code, a normalization step is performed on the extracted raw components. The normalization process follows the rules below.

1. If the type of a variable in a raw component is known, the variable is normalized to its type in the resulting component. For example, given an object *foo* of type *Foo*, the raw component *foo.flag* will be normalized to *Foo.flag*. If the type is numeric (i.e. *int*, *float*, *double*, etc.), it is further normalized to *int*. Variables with unknown types are not normalized.

2. For a raw component that contains a numeric, boolean, or char literal, two components are generated: one without normalization for the literal and one with the literal replaced by *int*, *boolean*, or *char* accordingly.

3. For a raw component that contains a string literal, two components are generated: one without normalization that includes the original string literal and one with normalization where the string literal is replaced with *String*.

4. For a method call, each actual parameter is normalized to the type of the parameter. If the type of a parameter value is unknown, a * indicates that this parameter can be any type. For example, the component string for component *initiate(5,bar)* is *initiate(,)* and the parameter list of this component is *int* and *, supposing the type of *bar* is unknown.

A *normalization level* value is computed to indicate a component's degree of normalization among its peers extracted from the same raw component. Due to the normalization of literals, several components may be generated for a raw component. For example, *Foo.flag>=5* and *Foo.flag>=int* are two components generated by normalizing the component *foo.flag>=5*. Since *Foo.flag>=5* has less normalization than *Foo.flag>=int*, a normalization level of 0 is assigned to *Foo.flag>=5*, and a normalization level of 1 is assigned to *Foo.flag>=int*. The normalization level field is used in setting the component searching option.

Figure 6 shows the resulting components after normalizing the raw component list in the left hand side of the figure.

2.1.3 Information Filtering

After normalization, the resulting components are candidate components for storage into the database. One problem that arises after the normalization step is that commonly occurring statement types are normalized to commonly occurring components. For example, integer assignment statements will generate *int=int,* which is very common. It is undesirable to add these common components to the database, since they will cause many false alarms. Therefore, a filtering step weeds out components such as these that carry little unique information.

Table 1. The information value for detailed elements in components.

Detailed Element	Condition	Information Value	Example
if predicate	Construct	1	if ()
do predicate	Construct	1	while ()
while predicate	Construct	1	while ()
for expression	Construct	1	for ()
Conditional expression	Construct	1	i>0? i: 1
return statement	Construct	1	return i
case expression	Construct	1	case 5:
switch expression	Construct	1	switch ()
synchronized expression	Construct	1	synchronized ()
throw statement	Construct	1	throw new Exception()
string literal	Length>8	2	"compiler.problem.Messages"
string literal	Length between 3 and 8	1	"example"
numeric literal		1	10
method call		2	initiate()
class name or variable type	User-defined class	1	Foo
variable name or field name		1	flag
Other	Does not match any other category	0	int=int

A component's *information value* indicates how much unique information it carries. The information value for a component is determined by summing the information values of its constituent elements. Table 1 lists the information value for different kinds of syntax constructs, identifiers and literals.

Components possessing little unique information are filtered by defining an information value threshold, that is, we only keep components whose information value is greater than or equal to 2.

Following this rule, four components are filtered from the resulting components listed in the right hand side of Figure 6, since their individual information values are less than 2.: *int=1* (information value of 1, from the numeric literal 1), *int=int* (information value 0, an "other"), *"example"* (information value 1, string literal), and *String* (information value 0, an "other").

2.1.4 Diff Filtering

After the information filtering step, we obtain a list of components that pass our threshold for carrying sufficient unique information from the code in the bug hunk. A further filtering step is to determine the components that exist in the bug hunk but not in the fix hunk. That is, code characteristics that are common between the bug hunk and the fix hunk are not saved, since they are unchanged.

Bug Hunk

```
if (foo.flag>=5 && foo.ready()) {
    i=1;
    foo.create("example");
    initiate(5,bar);
```

Fix Hunk

```
if (foo.flag>=7 || foo.ready()) {
    create("example");
    initiate(5);
```

Figure 7. Example code in a bug hunk and the corresponding code in the fix hunk.

```
Foo.flag>=5
Foo.flag>=5 && Foo.ready()
Foo.flag>=int && Foo.ready()
if (Foo.flag>=5 && Foo.ready())
if (Foo.flag>=int && Foo.ready())
Foo.create(.)    String
initiate(,)      int, *
```

Figure 8. Components in the bug hunk but not in the fix funk for the example in Figure 7.

Figure 7 shows example code in a bug hunk and its corresponding code in a fix hunk. Figure 8 shows the resulting components after diff filtering the components from the code in Figure 7. Several components have been filtered out in this step. One example is the component *foo.ready()*, which exists in both the bug hunk and the fix hunk. The components remaining at the end of this step are saved to the database. For the example in Figure 7, the components listed in Figure 8 will be saved to the memories database.

2.2 Storing and Searching Memories

Section 2.1 describes the algorithm for extracting components from the source code in hunks. This algorithm is applied to all of the bug and fix hunks in bug fix transactions, with the resulting components being saved to the memories database.

Using a populated memories database, it is possible to perform bug detection and change suggestion. Bugs in new or existing code are found by searching for matching patterns in bug hunks, while change suggestions are made by returning the code in the corresponding fix hunk. For example, suppose the component *if (Foo.flag>=5 && Foo.ready())* is found in a new change. The database is searched for records whose *Component_string* value is also *if (Foo.flag>=5 && Foo.ready())* and whose *In_bug_hunk* value is *true*. If any matching records are found, it is possible to alert developers that their new change may contain a bug. We can additionally provide developers with suggestions on how to fix the bug by presenting the fix hunk code in the *Fix_hunk* field.

Note that to match method call components, it is necessary to compare the parameter list as well as the component string. Special handling is also needed to match parameter types recorded as * in the parameter list, indicating that they match any parameter type.

There are several options for component searching, which adjust the degree of exact/close matching and omission of very common components. The options are listed in Table 2.

Table 2. Options for component searching.

Option	Description
0	Only match components whose normalization_level is 0 and exclude all static Java call or Java call component kinds.
1	Only match components whose normalization_level is 0 and exclude all static Java component kinds.
2	Only match components whose normalization_level is 0.
3	Match components with any normalization_level.

Option 0 has the strictest matching rule for component searching. This option searches components that have the least normalization, and ignores all *static Java call* and *Java call* component kinds. The component *System.out.println()* is a commonly occurring example of a static Java call component. The component *String.length(),* normalized from *str.length(),* is an example of a Java call kind, also a common pattern. Option 0 ignores components such as *System.out.println()* and *String.length()* to avoid the false positives they cause, since we believe developers typically do not make mistakes in these kinds of components. Option 1 is less strict than option 0, since it does not ignore Java call components, such as *String.length()*. Option 3 provides the most permissive component search, searching all components in the database. Option 3 yields the highest hit rates for component searching, but at the cost of more false positives.

Table 3. Analyzed open source projects. The period shows the analyzed project timespan. The number of transactions indicates the number of transactions that contain at least one file change. The number of hunks indicates the total number of hunks. The number of bug fix hunks indicates the number of hunks in the bug fix changes. The number of components shows the component count after building memories. For the Eclipse project we use only the core.jdt module due to the large size of the entire project.

Project	Software type	Period	# of transactions	# of hunks	# of bug fix hunks (%)	# of components in bug memories	# of components in fix memories
ArgoUML	UML editor	01/1998 ~ 09/2005	4,685	56,476	9,682 (17.1%)	64,552	86,347
Columba	Mail client	11/2002 ~ 12/2005	2,362	23,090	2,646 (11.5%)	10,919	17,177
Eclipse	IDE	06/2001 ~ 01/2006	6,394	72,215	23,223 (32.2%)	126,930	189,049
jEdit	Editor	09/2001 ~ 01/2006	1,190	18,966	5,060 (26.7%)	30,729	39,076
Scarab	Issue tracker	12/2000 ~ 02/2006	2,962	14,939	2,549 (17.1%)	13,632	19,859

3. EVALUATION

Before diving into the details of how we evaluate the effectiveness of bug fix memories, it is important to understand how they are intended to be used within a software project. A developer working on a project in their favorite development environment will receive feedback whenever the code they are developing matches one of the stored bug patterns. The tool that performs this matching is called BugMem, described in Section 4. The memories database developers are querying is *always up to date*. This is due to the inclusion, at checkin time, of new bug fix information. Since the component extraction process is computationally inexpensive, and requires no manual intervention, it is integrated into post-checkin processing for a project. As a result, the bug fix memories can be viewed as a kind of *on-line* learning algorithm.

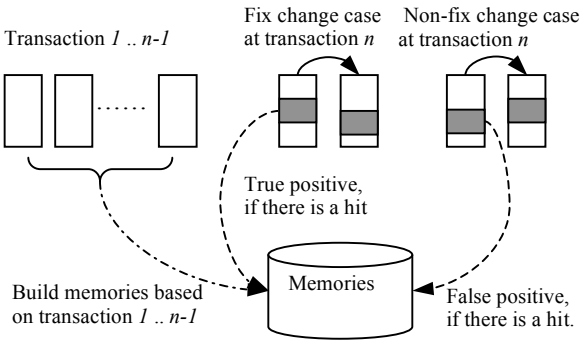


Figure 9. Evaluation of true positives and false positives

Since the intended use of the memories is to find bugs and suggest changes in the current transaction by leveraging the information in *all* prior transactions, traditional approaches for evaluation are ill suited for this situation. In a project with *n* transactions, a typical approach would be to train on 90% of the transactions and then evaluate on the remaining 10%. It is also possible to pick different parts of the project to be the 90% by cycling the 10% evaluation set through different portions of the transaction history—k-fold cross-validation [22]. The drawback with these traditional evaluation approaches is they don't reflect the actual use conditions of the memories database, since in normal use the transactions in the evaluation set would contribute to training the memories.

Another common approach for evaluating bug finding techniques is to train a model on one project, then evaluate it on another. This is also ill-suited for evaluating the bug fix memories. Since a project's memories are comprised of source code patterns, it is inherently specific to that project. In general, evaluation of vertical bug finding techniques involves training on one project and then assessing performance on that same project.

The approach chosen for evaluating bug fix memories is sketched in Figure 9. We walk through the transaction history of a project, evaluating at each transaction how well the approach works when *using only the information available as of that transaction*. To evaluate the bug memories at transaction *n*, the memories are built using the bug fix hunks from transactions *1* to *n-1*. We then determine whether transaction *n* is a bug fix. If so, a check is performed to see if a component in the bug hunk at transaction *n* is found in the memories database. If found, it is called it a *true positive hit*, which means that the bug is found in the previous memories. If transaction *n* is a non-fix change, and a component in the non-bug hunk is found in the memories, it is called a *false positive hit*, since code that does not contain a bug matches the bug memories. Hit rates are used to evaluate the bug finding and suggestion generation capabilities of BugMem.

The bug fix memories approach is also compared to a horizontal bug finding tool, PMD [5]. This permits an evaluation of how well BugMem and PMD perform at finding the actual bugs in the analyzed projects, and whether they find the same kinds of bugs. This is described in Section 3.3.

3.1 Setup

Project change histories from five open source projects, ArgoUML, Columba, Eclipse, jEdit, and Scarab, were extracted using the Kenyon infrastructure [3]. Subsequently, bug fix memories were built for each project using the approach described in Section 2. Table 3 summarizes the analyzed projects.

3.2 Bug Fix Memory Hit Rates

As described above, hit rates at a transaction *n* are computed by searching for matches in the memories built from transactions *1* to *n-1*. To precisely describe this process, notations are added to the bug memories, *M*, indicating which transactions have contributed information to the memories. A per-transaction memory, M_i, represents only those components extracted from transaction *i*. Per-transaction memories are used to define the memories as of a given transaction, *n*:

$$M^n = \bigcup_{i=1}^{n} M_i$$

This permits a refinement of the *find* function to describe searches at a given transaction, *n*:

$$find_n(extract(H), M^{n-1}) \rightarrow MatchedHP$$

This states that the find function for transaction *n* only uses memories built using components from transactions *1* to *n-1*.

All bug and fix hunks are extracted at transaction *n*, and checked to see if they match against M^{n-1}. The process iterates from transaction 1 to the end transaction, *N*. The overall process is sketched in Figure 10.

```
For (n = 1 to N) {
  If (transaction n is identified as a bug fix) {
    Get bug or fix hunks in transaction n
    Try find_n(extract(H), M^(n-1)) for each bug and fix hunk in
transaction n
  }
}
```

Figure 10. Process for evaluating hit rates. The total number of transactions is *N*.

To determine half and full hit rates, we start by extracting matched hunk pairs (*Half_hit_MatchedHP*) using deleted hunks (DH) from the memories:

$$Half\_hit\_MatchedHP_n(HP) = find_n(extract(DH), M^{n-1})$$

If a component in a bug hunk is found in the memories, it is defined as a half hit. A half hit indicates that the same kind of bug has been seen in previous transactions (memories):

$$Half\_hit_n(HP) \;\; iff \;\; Half\_hit\_MatchedHP_n(HP) \neq \Phi$$

If a component in a fix hunk is also found in *Half_hit_MatchedHP*, it is a full hit. A full hit indicates that the exact bug and fix pair has been seen in previous transactions (memories).

$$Full\_hit_n(HP) \; iff \; Half\_hit_n(HP) \wedge$$
$$(find(extract(AH), \; Half\_hit\_MatchedHP_n(HP)) \neq \Phi \vee$$
$$type(HP) = deletion)$$

More concisely, in order to have a full hit, a half hit must have occurred first. Additionally, the added (fix) hunk must be found in the bug and fix hunk pairs returned by the half hit database query (*find(…)*). In the case where there is no added hunk (code was deleted, but not added when fixing a bug), the hunk pair is of type deletion (*type(HP)=deletion*). Since it is not possible to find an empty added hunk in the half hit hunk pairs, a half hit is assumed to be a full hit in this case.

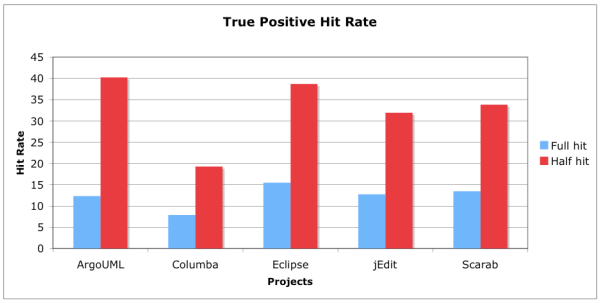


Figure 11. Full and half hit rates, search option 3.

Figure 11 shows the full and half hit rates of analyzed projects. Using different component search options yields different hit rates. Use of option 3 yields the results in Figure 11, which represent the highest possible hit rates. Full hits vary from 7.9% - 15.5%, indicating that this many bug and fix pairs repeat over the project's history. Half hit rates vary from 19.3%-40.3%, indicating that this many bug hunks are found in previous transactions (memories).

From the developer's perspective, the half hit rate indicates that in 19.3%-40.3% of bug fix changes, the BugMem tool can find code in the change that matches an existing pattern. These matching lines can be highlighted for the developer as code that is likely to be buggy. Additionally, the full hit rates indicate that for 7.9%-15.5% of the changes, it is possible for the BugMem tool to provide suggested changes, observed from the development history, to fix the identified bug.

Table 4. True positive hit rates.

Option / Projects	0	1	2	3
Argouml	10.5% / 34.8%	10.8% / 35.3%	12.3% / 40.2%	12.4% / 40.3%
Columba	5.9% / 15.1%	6.3% / 15.9%	7.9% / 19.2%	7.9% / 19.3%
Eclipse	14.2% / 33.9%	14.6% / 34.6%	15.5% / 38.6%	15.5% / 38.7%
jEdit	9.5% / 24.5%	10.1% / 26%	12.8% / 31.8%	12.8% / 31.9%
Scarab	7.6% / 27%	8.3% / 28.8%	13.5% / 33.8%	13.5% / 33.8%

To get a sense of the impact of different search options, hit rates were evaluated for all search options. Table 4 shows detailed full and half hit rates for search options 0-3. In general, there is little difference between options 2 and 3, but significant difference between options 0 and 3.

To get a sense of the false positive rates of BugMem, the analysis shown in Figure 10 was repeated, this time using non bug fix hunks. If components in the non bug fix hunks are found in the memories, such components are false positives, since those hunks are not bug fixes, and hence not supposed to match any components in the memories. Figure 12 shows an overview of the hit rates.

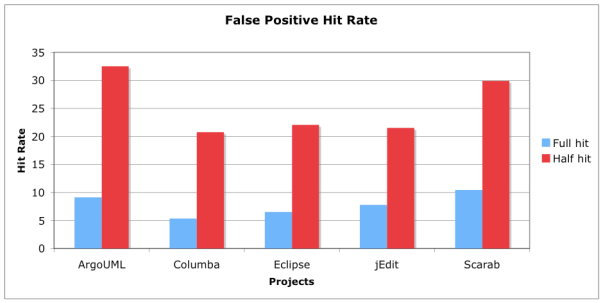


Figure 12. False positive hit rates, search option 3.

Table 5 shows detailed false positive full and half hit rates for search options 0-3.

Table 5. False positive hit rates.

Option / Projects	0	1	2	3
Argouml	7% / 26.5%	7.2% / 26.9%	9% / 32.4%	9.1% / 32.5%
Columba	3.5% / 14.2%	3.8% / 15%	5.3% / 20.7%	5.3% / 20.8%
Eclipse	5.9% / 18.8%	6% / 19.1%	6.5% / 21.9%	6.5% / 22.1%
jEdit	5.8% / 16.7%	6.1% / 17.4%	7.7% / 21.5%	7.8% / 21.5%
Scarab	7.8% / 21.6%	8.1% / 22.6%	10.4% / 29.9%	10.5% / 29.9%

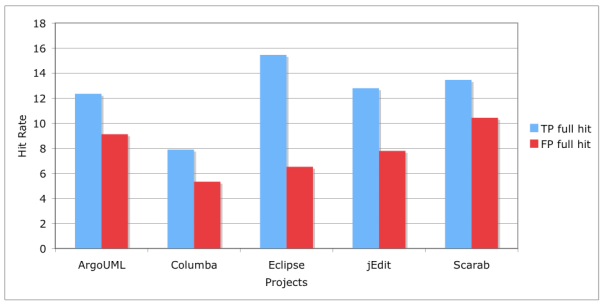


Figure 13. True positive (TP) and false positive (FP) full hit rates of analyzed projects.

We compared the true positive (TP) full hit rates and false positive (FP) full hit rates, with results in Figure 13. Overall, the false positive full hit rates range from 7.8%-10.5%, which is 2.6%-8.9% lower than the true positive hit rates. Even though the false positive hit rates are relatively high, BugMem is still useful, since it provides not only warning flags, but also bug fix examples (on average 1.7 fix examples per warning in jEdit). Developers can quickly decide if they want to accept or reject the warnings by

examining the provided examples. Holmes et al. [13] and Mandelin et al. [19] show that providing code examples is beneficial for understanding software. BugMem always provides bug fix examples along with warnings.

3.3 Comparison with PMD

To compare a horizontal bug finding tool with the vertical approach used by BugMem, we identify bugs using a bug finding tool, PMD [5], comparing them with those identified by BugMem. PMD is chosen because it does not require annotation and only requires Java source code as its input. Other bug finding tools, such as FindBugs and JLint, take Java class files as their input, which would require source code compilation for every transaction. This is computationally very expensive. Since PMD performs syntactic checks on source code, it mostly catches stylistic bugs. Comparing BugMem with other bug finding tools such as FindBugs and JLint remains future work.

PMD can identify potential bugs using pre-defined syntactic error patterns such as 'empty if statement', 'misplaced null check', or 'no null check in the equal method' [5]. To compute its hit rate, potential bugs are located using PMD and then checked to see if the bugs are fixed in the project's change histories.

To compute full and half hit rates, PMD was run to obtain detected violations that fall into a hunk. The violation count of a hunk is defined as *VC(**H**)*. For example, if a hunk includes 5 violations, *VC(**H**)* is 5. All bug and fix hunks were extracted and violation counts were measured for each hunk, following the process sketched in Figure 14.

```
For (r = 1 to N) {
   Get bug fix hunks in transaction r
   Compute VC(H) for each bug and fix hunk in transaction r
}
```

Figure 14. Process for evaluating PMD hit rates. The total number of transactions is *N*.

A half hit occurs if the violation count of a bug hunk is greater than 0, since PMD correctly identifies some bugs in the bug hunks. To be a full hit, the violation number of a bug hunk must be reduced in the fix hunk. A full hit indicates that there are violations in the bug hunk, but the violations are removed in the fix hunk. If the hunk pair type is deletion and there is a hit on the bug hunk, it is assumed to be a full hit, since the violated code is removed. Formally, half and full hits are defined as:

$$Half\_hit(HP) \;\; iff \;\; VC(AH) > 0$$

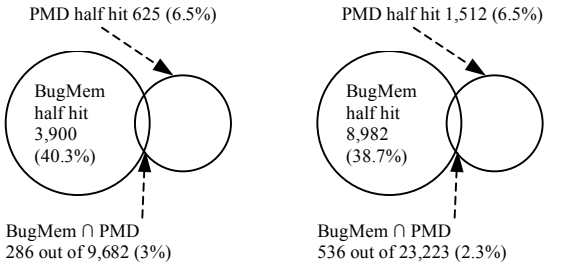
$$Full\_hit(HP) \;\; iff \;\; Half\_hit(HP) \wedge (VC(DH) < VC(AH) \vee type(HP) = deletion)$$

Table 6. PMD hit rates of analyzed project. The first number indicates full hit rates, and the second means half hit rates.

Priority / Projects	1	2 or less	all
Argouml	0.2% / 0.4%	1.9% / 6.4%	2% / 6.5%
Columba	0.3% / 0.6%	2% / 6.2%	2% / 6.3%
Eclipse	0.3% / 0.5%	1.4% / 6.5%	1.4% / 6.5%
jEdit	0.1% / 0.4%	1.8% / 6.8%	1.8% / 6.8%
Scarab	0.1% / 0.4%	2.7% / 7.2%	2.7% / 7.2%

PMD assigns each violation a priority ranging from 1 to 3. A priority of 1 indicates a serious warning, while a priority of 3 reflects less important warnings. We observe hit rate variances by using warnings at a specific priority level. Detailed full and half hit rates with priority combinations are shown in Table 6.

Bug sets correctly identified by PMD and BugMem (half hits) were compared to see if the two sets are exclusive. Figure 15(a) shows the entire bug hunk space of ArgoUML, comprised of 9,682 bug hunks. Among them, 3,900 (40.3%) of the bug hunks are correctly identified by BugMem (see Table 4, half hit at option 3), and 625 (6.5%) are correctly identified by PMD (see Table 6, half hit at priority 3 or less). We then observed the intersection of the two correctly identified sets to see how these two tools can complement each other. Surprisingly, only 3% of the total identified hunks were common between PMD and BugMem.



(a) ArgoUML (b) Eclipse

Figure 15. Identified bug hunk sets of two projects using PMD and BugMem.

Figure 15(b) shows bug hunk sets identified by BugMem and PMD in Eclipse. Similar to ArgoUML, only 2.3% of the identified hunks were common in Eclipse. Intersections for other projects are shown in Table 7. The intersections of the identified bugs are about 1.1~3% of the total bug hunks. The results indicate that the hunk sets identified by PMD and BugMem are largely exclusive. We conclude that BugMem is not meant to replace prior bug finding tools. BugMem can find bugs that cannot be identified by PMD and vice-versa. There is considerable synergy in using a combination of vertical and horizontal bug finding tools together.

Table 7. Identified bug hunks (half hit) by PMD, BugMem, and their intersection.

Project	BugMem hit (%)	PMD hit (%)	BugMem ∩ PMD (%)
Argouml	3,900 (40.3%)	625 (6.5%)	286 (3%)
Columba	510 (19.3%)	166 (6.3%)	30 (1.1%)
Eclipse	8,982 (38.7%)	1,512 (6.5%)	536 (2.3%)
jEdit	1,615 (31.9%)	342 (6.8%)	118 (2.3%)
Scarab	862 (33.8%)	184 (7.2%)	55 (2.2%)

3.4 Limitations of Memories

3.4.1 Missing Memories.

Since there are limitations inherent in line-based text diffs, our approach misses some kinds of bug fixes. For addition type hunk pairs (|DH| = 0), it is hard to find the buggy part of the source code. Suppose there is a bug fix change as shown in Figure 16. The *if condition* is added to check whether *foo* is *null*, so the fix hunk contains the *if condition* line and the bug hunk is empty. In fact, *print(foo.a)* at revision *n-1* is buggy code, since there should

be an *if condition* before it to perform the null dereference check. Due to the empty bug hunk, this bug is missing in the memories. Similar cases include addition of try/catch statements, addition of an *else* branch, addition of a method call in a sequence of method calls, etc. In ongoing work, we are developing a set of these commonly occurring bug fix patterns.

print(foo.a);	if (foo!=null) {
	print(foo.a);
	}

Revision n-1	Revision n

Figure 16. Addition hunk type example.

3.4.2 Limitation in the initial stage of a project

The bug fix memories approach is applicable only when a project has been under development for awhile, and hence some bug fixes have been collected in the memories. In contrast, horizontal bug finding tools can work immediately from transaction 1 of a newly started project. How many transactions must pass before BugMem achieves a reasonable hit rate? True positive half hit rates (option 3) were observed over several thousand transactions, as shown Figure 17. Hit rates dramatically increase around transaction 200-900 and then continue growing slowly as transactions accumulate. Waiting 200-900 transactions before using BugMem appears to be a reasonable rule-of-thumb.

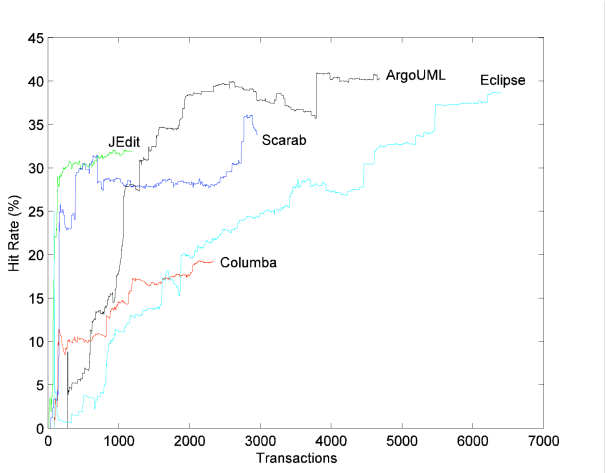


Figure 17. True positive half hit rates (option 3) over transactions.

3.4.3 Only file-by-file basis

Our approach is based on comparing a source code file of two versions, so the bugs captured and fixes suggested are only file-by-file based. The cross-file relationships of bugs and fixes are not revealed.

3.5 Threats to Validity

There are four major threats to the validity of this work.

Systems examined might not be representative. We examined 5 systems, and it is still possible that we accidentally chose systems that have better (or worse) than average bug fix memories hit rates. Since we intentionally only chose systems that had some degree of linkage between change tracking systems and the text in the change log (so we could determine bug fix changes and hunks), we have a project selection bias. As our dataset increases the severity of this threat will diminish.

Systems are all open source. The systems examined in this paper all use an open source development methodology, and hence might not be representative of all development contexts. It is possible that the stronger deadline pressure, different personnel turnover patterns, and different development processes used in commercial development could lead to different memories hit rates.

All systems are written in Java. Extracting components from hunks and building memories requires a complete programming language parser. As a result, BugMem currently only supports the Java language. Other programming languages may have different bug patterns and memories hit rates.

Bug fix data is incomplete. Even though we selected projects that have change logs with good quality, we still are only able to extract a subset of the total number of bugs (typically only 40%-60% of those reported in the bug tracking system). The identified bug-fix data is not the oracle set. It may include false positives and false negatives. Incomplete bug fix data may increase (or decrease) false positive rates, and prevent the development of complete bug fix memories.

4. USING BUG FIX MEMORIES

Bug fix memories can be used to construct a bug finding tool and perform IDE integration. A project's bug fix memories can also be used as a code example repository to provide awareness to developers. We describe each application in detail.

4.1 Bug Finding Tool

We implemented a bug finding tool, BugMem, using memories of bug fixes. Like other bug finding tools such as ESC/Java, FindBug, and PMD, the BugMem tool is provided source code and generates warning messages. Figure 18 shows example BugMem output which indicates that using *setSelectText()* might be a potential bug, and recommends changing it to *insertTab()* based on previous bug fix instances. BugMem provides real fix examples for identified bugs using fix data from the project's bug fix memories.

```
$ bugmem Test.java

Warning in addText at Test.java  (line 10)   Found 4 memories
Type: call "setSelectedText(.)"
================================================================
org/gjt/sp/jedit/textarea/JEditTextArea.java at Rev: 114 in jedit
================================================================
                -          else setSelectedText("\t");
                +          else insertTab();
……
```

Figure 18. Simple output of the BugMem command line tool.

An IDE (Eclipse) integration of BugMem has also been implemented. During a source editing session using the IDE, BugMem can point out potentially buggy lines and provide real bug fix examples for those lines.

4.2 Bug and Fix Understanding

The memories of bug fixes are very useful for developers who are new to software projects. Core developers who know and remember all previous bugs and fixes may be able to avoid making the same mistakes again. For new developers, however, the memories of bug fixes are essential to guide their future development. When they does not know the right method or

constant to use, automatically recovered memories of bug fixes can help correct mistakes and suggest correct examples.

5. RELATED WORK

In this section, we discuses related work on finding bugs, locating buggy areas, using project history to detect bugs, and using code examples to assist development.

5.1 Bug Finding Tools

Many bug finding tools such as Bandera, ESC/Java [11], PMD [5], JLint [1], and FindBugs [14] have been proposed and are in wide use [25]. Most of these tools use syntactic pattern matching, model checking, or theorem proving. They are similar to BugMem in that they perform static analysis, find bugs, and then suggest correct code. They are good at detecting commonly known bugs, such as null dereferencing errors. However, they do not detect high-level project-specific bugs.

While prior bug finding tools use built-in and pre-defined bug patterns, BugMem learns project-specific bug patterns by analyzing an ongoing development history. Additionally, BugMem can suggest correct code to repair detected buggy code.

5.2 Using Project History to Detect Bugs

We used project histories to build memories to detect bugs and suggest fixes. Project histories are widely used to build project knowledge [7, 8], detect common bug patterns [18, 28], and find association rules among bugs [27].

Hipikat is a tool that recommends relevant software artifacts to developers based on project histories comprised of artifacts such as source code changes, mailing list messages, bug tracking entries, and written documentation [7, 8]. The Hipikat approach is similar to BugMem in that it builds up a repository of information from the project's history. However, we explicitly identify bad (bug) and good (fix) memories to detect potential bugs and suggest fixes. Hipikat tries to provide related references to developers rather than identify good or bad memories. Hipikat uses lexical information (which is often automatically extracted) to search memories while BugMem analyzes source code and extracts components automatically.

Williams and Hollingsworth use project histories to improve existing bug finding tools [28]. When a function returns a value, using the value without checking it may be a bug. The problem in this approach is that there are too many false positives, due to the generation of warnings about all source code that uses an unchecked return value. To remove these false positives, Williams and Hollingsworth use project histories to determine what kinds of function return values must be checked. For example, if the return value of the function '*foo*' was always checked in the project history, but not checked in current source code, it is very suspicious.

Livshits and Zimmermann combined software repository mining and dynamic analysis to discover common method usage patterns that are likely to encounter violations in Java applications [18]. Their approach employs dynamic analysis and is more specific in finding violation patterns on method usage pairs. For example, *blockSignal()* and *unblockSingal()* should always be paired in the source code.

The approaches in [28] and [18] are vertical bug finding techniques similar to ours, since they both analyze project-specific patterns. However, they only focus on a small set of bug patterns, such as the return value checking in [28] and the method usage pairs in [18]. In contrast, BugMem uses all kinds of components to build memories and detect bugs, and the kinds of components keep growing along with the development process.

Song et al. find association rules among six bug types from project histories [27]. Using these association rules, they can predict future bugs. For example, suppose bug types *A* and *B* are often found together in the history. Then if we find only bug type *A* in the source code, we assume the code contains bug type *B* as well. BugMem uses components from bug hunks to detect bugs, and does not use any bug association rules. Using buggy component association rules may increase hit rates; testing this idea remains future work.

Brun and Ernst extract properties from buggy code and feed it to machine learning algorithms to train a bug prediction model [4]. They use the Daikon invariant extractor [9] to extract invariant information. Their approach is similar in that they try to capture properties of buggy code and use it for future prediction. However, they use invariant information for their code properties, while we use syntactic information.

5.3 Identifying Buggy Areas

Identifying buggy code areas is quite useful for improving software quality, and many approaches have been proposed. Some approaches use software complexity metrics to identify buggy areas, assuming that complex software has more potential bugs [15, 23, 24]. Other approaches leverage a project's bug history, change history, or code co-changes to identify buggy areas [2, 12, 21]. Prediction accuracy for these approaches range from 60-80%, but the areas predicted to be buggy are quite coarse, ranging from modules to binaries, files, or functions. Even though BugMem has lower accuracy (hit rate), it precisely locates bugs at the line level and provides suggestions for fixes.

CP-Miner [17] is an approach that finds copy-paste code clone regions in source code and detects "forgot-to-change" bugs in them. In contrast, BugMem is able to identify bugs in any changed source code region.

5.4 Using Code Examples to Assist Development

Holmes and Murphy proposed an approach to extract structural components from example code and use them to assist coding when developers are working on similar code [13]. Mandelin et al. introduces a jungloid mining approach that automatically generates jungloid code fragments by mining library and example code to provide common API use examples [19]. The input to the jungloid mining is the input and output types of APIs. The output of jungloid mining is examples of method call sequences extracted from sample client programs or synthesized from API method signatures. The jungloid mining approach focuses solely on method calls. Source code search engines [16] are also widely used to find source code examples. BugMem is similar to source code example approaches, since we extract components from source code examples in the history (hunks). However, we identify both bad (from bug hunks) and good (from fix hunks) examples, using them to detect bugs. Prior code example approaches assume that all examples are good source code, but the existence of as-yet undiscovered bugs in all projects means that this is not true.

6. CONCLUSIONS

We presented BugMem, a project-specific bug finding tool using memories of bug fixes. BugMem detects potential bugs and suggests corresponding fixes. We evaluate BugMem by computing bug fix memories hit rates. We found that 19.3%-40.3% of bugs (half hit) appear repeatedly, and 7.9%-15.5% of bug and fix pairs (full-hit) appear repeatedly in the history. We also compared identified bug sets by PMD and by BugMem, and found the two identified sets are mostly exclusive. We conclude that prior bug finding tools and BugMem should be used together to maximize bug detection capability.

Source code repositories such as CVS and Subversion are typically used to store histories and make backups. In our view, a source core repository contains knowledge that can be used to discriminate between good and bad source code. So far, the knowledge available in source code repositories has not yet been fully leveraged. Our approach of computing memories of bug fixes provides a useful way to extract and deploy the knowledge latent in source code repositories. We harness this information to improve the quality of source code and provide detailed guidance to developers.

7. ACKNOWLEDGMENTS

We thank Gail Murphy and anonymous reviewers for their valuable feedback on this paper. We thank Jennifer Bevan and Aaron Tomb for their comments on this paper. We especially thank Kevin Greenan and the Storage Systems Research Center at UCSC for allowing the use of their cluster for our research.

8. REFERENCES

[1] C. Artho, "Jlint - Find Bugs in Java Programs," 2006, http://jlint.sourceforge.net/.

[2] J. Bevan and E. J. Whitehead, Jr., "Identification of Software Instabilities," Proc. of 10th Working Conference on Reverse Engineering (WCRE 2003), Victoria, Canada, pp. 134-145, 2003.

[3] J. Bevan, E. J. Whitehead, Jr., S. Kim, and M. Godfrey, "Facilitating Software Evolution with Kenyon," Proc. of the 2005 European Software Engineering Conference and 2005 Foundations of Software Engineering (ESEC/FSE 2005), Lisbon, Portugal, pp. 177-186, 2005.

[4] Y. Brun and M. D. Ernst, "Finding Latent Code Errors via Machine Learning over Program Executions," Proc. of 26th International Conference on Software Engineering (ICSE 2004), Scotland, UK, pp. 480-490, 2004.

[5] T. Copeland, *PMD Applied*: Centennial Books, 2005.

[6] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng, "Bandera: Extracting Finite-state Models from Java Source Code," Proc. of 22nd International Conference on Software Engineering (ICSE 2000), Limerick, Ireland, pp. 439-448, 2000.

[7] D. Cubranic and G. C. Murphy, "Hipikat: Recommending pertinent software development artifacts," Proc. of 25th International Conference on Software Engineering (ICSE 2003), Portland, Oregon, pp. 408-418, 2003.

[8] D. Cubranic, G. C. Murphy, J. Singer, and K. S. Booth, "Hipikat: A Project Memory for Software Development," *IEEE Trans. Software Engineering*, vol. 31, no. 6, pp. 446-465, 2005.

[9] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon System for Dynamic Detection of Likely Invariants," *Science of Computer Programming*, 2006.

[10] M. Fischer, M. Pinzger, and H. Gall, "Populating a Release History Database from Version Control and Bug Tracking Systems," Proc. of 19th International Conference on Software Maintenance (ICSM 2003), Amsterdam, The Netherlands, pp. 23-32, 2003.

[11] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended Static Checking for Java," Proc. of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, Berlin, Germany, pp. 234 - 245, 2002.

[12] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting Fault Incidence Using Software Change History," *IEEE Transactions on Software Engineering*, vol. 26, no. 7, pp. 653-661, 2000.

[13] R. Holmes and G. C. Murphy, "Using Structural Context to Recommend Source Code Examples," Proc. of 27th International Conference on Software Engineering (ICSE 2005), St. Louis, MO, USA, pp. 117-125, 2005.

[14] D. Hovemeyer and W. Pugh, "Finding Bugs is Easy," Proc. of the 19th Object Oriented Programming Systems Languages and Applications (OOPSLA '04), Vancouver, British Columbia, Canada, pp. 92-106, 2004.

[15] T. M. Khoshgoftaar and E. B. Allen, "Ordering Fault-Prone Software Modules," *Software Quality Control Journal*, vol. 11, no. 1, pp. 19 - 37, 2003.

[16] Koders, "Koders - Source Code Search Engine," 2006, http://www.koders.com/.

[17] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: finding Copy-paste and Related Bugs in Large-scale Software Code," *IEEE Trans. Software Engineering*, vol. 32, no. 3, pp. 176-192, 2005.

[18] B. Livshits and T. Zimmermann, "DynaMine: Finding Common Error Patterns by Mining Software Revision Histories," Proc. of the 2005 European Software Engineering Conference and 2005 Foundations of Software Engineering (ESEC/FSE 2005), Lisbon, Portugal, pp. 296-305, 2005.

[19] D. Mandelin, L. Xu, R. Bodik, and D. Kimelman, "Jungloid Mining: Helping to Navigate the API Jungle," Proc. of Conference on Programming Language Design and Implementation (PLDI 2005), Chicago, Illinois, USA, pp. 48-61, 2005.

[20] A. Mockus and L. G. Votta, "Identifying Reasons for Software Changes Using Historic Databases," Proc. of 16th International Conference on Software Maintenance (ICSM 2000), San Jose, California, USA, pp. 120-130, 2000.

[21] A. Mockus and D. M. Weiss, "Predicting Risk of Software Changes," *Bell Labs Technical Journal*, vol. 5, no. 2, pp. 169-180, 2002.

[22] A. W. Moore, "Cross-Validation," 2005, http://www.autonlab.org/tutorials/overfit.html.

[23] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Predicting the Location and Number of Faults in Large Software Systems," *IEEE Transactions on Software Engineering*, vol. 31, no. 4, pp. 340-355, 2005.

[24] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Where the Bugs Are," Proc. of 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis, Boston, Massachusetts, USA, pp. 86 - 96, 2004.

[25] N. Rutar, C. B. Almazan, and J. S. Foster, "A Comparison of Bug Finding Tools for Java," Proc. of 15th IEEE International Symposium on Software Reliability Engineering (ISSRE'04), Saint-Malo, Bretagne, France, pp. 245-256, 2004.

[26] J. Śliwerski, T. Zimmermann, and A. Zeller, "When Do Changes Induce Fixes?" Proc. of Int'l Workshop on Mining Software Repositories (MSR 2005), Saint Louis, Missouri, USA, pp. 24-28, 2005.

[27] Q. Song, M. Shepperd, M. Cartwright, and C. Mair, "Software Defect Association Mining and Defect Correction Effort Prediction," *IEEE Trans. Software Engineering*, vol. 32, no. 2, pp. 69-82, 2006.

[28] C. C. Williams and J. K. Hollingsworth, "Automatic Mining of Source Code Repositories to Improve Bug Finding Techniques," *IEEE Trans. Software Engineering*, vol. 31, no. 6, pp. 466-480, 2005.