BugFix: A Learning-Based Tool to Assist Developers in Fixing Bugs

Dennis Jeffrey* jeffreyd@cs.ucr.edu
Min Feng* mfeng@cs.ucr.edu
Neelam Gupta guptajneelam@gmail.com
Rajiv Gupta* gupta@cs.ucr.edu

*University of California, CSE Department, Riverside, CA

Abstract

We present a tool called `BugFix` *that can assist developers in fixing program bugs. Our tool automatically analyzes the debugging situation at a statement and reports a prioritized list of relevant bug-fix suggestions that are likely to guide the developer to an appropriate fix at that statement.* `BugFix` *incorporates ideas from* machine learning *to automatically learn from new debugging situations and bug fixes over time. This enables more effective prediction of the most relevant bug-fix suggestions for newly-encountered debugging situations. The tool takes into account the static structure of a statement, the dynamic values used at that statement by both passing and failing runs, and the* interesting value mapping pairs *[17] associated with that statement. We present a case study illustrating the efficacy of* `BugFix` *in helping developers to fix bugs.*

1. Introduction

Software Debugging is the process of fixing program bugs. This is an important and necessary step of software development, because programming is a complicated, human-intensive activity that is prone to error. Unfortunately, debugging can be a difficult task. Locating an error from among hundreds, thousands, or even millions of lines of source code can be daunting without any automated assistance. When an error is located, it may still take considerable time for a developer to fully understand the problem so that an appropriate fix can be made. Techniques to help automate the debugging process can assist developers in more efficiently producing robust and reliable software.

Most prior research in automated software debugging has focused on *fault localization*, narrowing or guiding the search for bugs to help developers identify faulty statements more quickly. *Dynamic Program Slicing* [2, 23, 30, 35, 36] can be used to compute a subset of program statements that directly or indirectly affect the erroneous output produced by a program during a failing execution. *Delta Debugging* [6, 32, 33] can analyze differences in program state between successful and failing executions to isolate the error that caused a failure. Other approaches [17, 21, 24, 25] use runtime information to rank program entities in the order of their likelihood of being faulty.

Techniques to help developers understand program behavior can also help them to debug more efficiently. Ko and Myers [22] developed a debugging tool called `The Whyline` to help developers better understand program behavior. This tool allows developers to select a question concerning the output of a program, and the tool then uses a combination of static and dynamic analysis techniques to search for possible explanations.

To our knowledge, there is very little prior work that focuses specifically on assisting developers in *changing programs to fix bugs*. He and Gupta [13] developed an approach to automatically generate program modifications that can correct an erroneous statement in a function. Their approach requires that a formal precondition and postcondition be specified for the function in terms of first-order theory formulas. Abraham and Erwig [1] developed a debugging tool for spreadsheets in which a user can specify the expected value for a cell that contains an incorrect value; the tool then identifies change suggestions that can correct the error. In general, suggestions for modifying program code (which we call *bug-fix suggestions*) have the potential to guide developers to appropriate fixes more quickly. Our goal in the current work is to automatically generate such suggestions using a machine-learning approach that considers knowledge gained from previous bugs that have already been identified and fixed. We implemented our approach in a tool called `BugFix`.

`BugFix` requires as input a faulty program and a corresponding test suite containing at least one failing test case. The goal of our tool is to compute and report a prioritized list of *bug-fix suggestions* for a given *debugging situation* at a program statement that is suspected of being faulty. A *debugging situation* can be thought of as a characterization of the particular static and dynamic details of a suspicious statement being debugged (described in detail in Section 3.1). A *bug-fix suggestion* is a textual description of how to modify a given statement such that a bug in the statement

is likely to be fixed. An actual fix performed by a developer is textually represented by a *bug-fix description*.

Our tool is built upon concepts from the *machine learning* community, which allow the tool to *learn* about new debugging situations and their corresponding bug fixes that are encountered over time. Through continued use, the ability of the tool to report highly relevant bug-fix suggestions for new debugging situations is expected to improve. This is accomplished by maintaining a *database of bug-fix scenarios* describing the different debugging situations and corresponding bug-fix descriptions previously encountered by the tool. From this database, a machine-learning algorithm for learning association rules can be applied to automatically generate a *knowledgebase of rules*, mapping different (general and specific) debugging situations to corresponding bug-fix descriptions. Each rule is also associated with a confidence value indicating how likely the rule is to be correct (i.e., how likely a particular debugging situation should indeed map to the given bug-fix description).

Given a new debugging situation, our tool automatically analyzes it in conjunction with the knowledgebase of rules to compute and report a prioritized list of relevant bug-fix suggestions. Once the appropriate fix is made by the developer, then this new information about the current debugging situation and corresponding bug fix is added to the database of bug-fix scenarios. This enables a revised set of rules to be computed that can lead to more effective results when the tool is used again in the future. We believe that our tool has the potential to help a developer more quickly discover, apply, and verify the appropriate fix for a bug.

There are two main contributions of this paper.

1. A new machine-learning tool to assist developers in fixing program bugs, which complements prior work on locating and understanding bugs.
2. A case study demonstrating the efficacy and potential of our tool in helping developers properly fix bugs.

In the next section, we describe some background information that is necessary to understand `BugFix`. The tool itself is then described in detail (Section 3). The efficacy and potential of the tool is illustrated through a case study (Section 4). Related work is presented (Section 5), and finally we conclude (Section 6).

2. Background Information

`BugFix` makes use of two concepts that we first describe: (1) *association rule learning*; and (2) *interesting value mapping pairs* [17].

2.1. Association Rule Learning

In the machine-learning community, *association rule learning* [4] is a popular method for discovering the relationship between variables in databases. It has been widely used in many diverse application areas, such as marketing, intrusion detection, genetic engineering, and (in the current work) software debugging.

Let $I = \{i_1, i_2, \ldots, i_n\}$ be a set of n attributes called *items*, and $T = \{t_1, t_2, \ldots, t_m\}$ be a set of m *transactions* comprising the *database*. Each transaction in T is a subset of the items in I (a set of items is commonly referred to as an *itemset*). *Association rules* are derived from these transactions in the database. An association rule is defined in the form $X \rightarrow Y$ where $X, Y \subseteq I$ and $X \cap Y = \emptyset$. X and Y are called the *antecedent* and the *consequent*, respectively. A rule intuitively means that if the items in set X are present/true, then it is probable that the items in set Y are also present/true. For example, an association rule in the supermarket domain could be $\{eggs, bread\} \rightarrow \{milk\}$, which implies that if a customer buys $eggs$ and $bread$, then the customer probably buys $milk$ as well.

The notion of *confidence* has been introduced to measure the significance of a rule. The confidence $conf$ of a rule $X \rightarrow Y$ is defined as follows:

$$conf(X \rightarrow Y) = supp(X \cup Y)/supp(X)$$

where $supp(X)$ is the *support* of itemset X, which is equal to the fraction of transactions in the database containing X. This confidence can be interpreted as an estimate of the probability $P(Y|X)$. It allows one to select a subset of the most interesting rules from a set of all possible rules.

A variety of techniques have been developed for learning association rules. `apriori` [5] is the most popular algorithm. Given a database of transactions, `apriori` identifies association rules through two key steps.

Step 1. Find the *frequent itemsets*. These are sets of items for which the associated support values are at least a specified minimum value. The algorithm iteratively generates candidate itemsets and prunes out those containing subsets of items that are known to be infrequent.

Step 2. Generate association rules from the frequent itemsets. For each frequent itemset X, `apriori` enumerates all non-empty subsets of X. For each such subset $Y \subseteq X$, the algorithm calculates the confidence of rule $Y \rightarrow (X - Y)$ and outputs the rule if the associated confidence value is larger than a specified minimum confidence.

`BugFix` uses the `apriori` algorithm to identify rules mapping debugging situations to bug-fix descriptions. These rules are then analyzed in conjunction with a newly-encountered debugging situation to identify the most relevant bug-fix suggestions from among the bug-fix descriptions currently known.

2.2. Interesting Value Mapping Pairs

In our prior work [17], we showed how effective fault localization could be performed using a technique called *value replacement*, which involves replacing the set of values used at a statement instance in a failing run with an al-

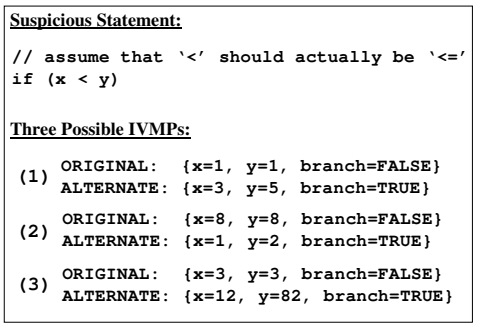

Figure 1. Example IVMPs at a statement.

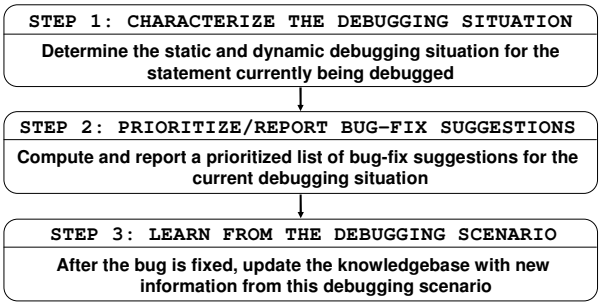

Figure 2. The three main steps of `BugFix`.

ternate set of values, then checking to see whether the failing run changes to become passing. If so, then this value replacement is represented by an *interesting value mapping pair* (IVMP), showing the original set of values used at the statement instance, and the corresponding set of alternate values that can be substituted to cause the failing run to pass.

Fig. 1 shows three possible IVMPs for a given suspicious statement. The suspicious statement in this case is an `if` condition in which the $<$ operator is mistakenly used instead of $<=$. The effect of this error is that whenever the operand values x and y are identical, then the condition will erroneously evaluate to $false$ when it should have evaluated to $true$. As a result, all original sets of values in the IVMPs have identical values for x and y, and the condition evaluating to $false$. However, all alternate sets of values have different values for x and y that instead cause the condition to evaluate to the expected outcome of $true$. These alternate values can cause a failing run to pass (assume that neither x nor y are subsequently referenced and that there are no other bugs in the program).

IVMPs, besides being useful for locating faulty statements [17], can exhibit patterns that provide hints about how to fix bugs. Such patterns among the IVMPs occurring at a suspicious statement are useful as one way to help describe a particular debugging situation.

3. BugFix: A Tool for Debugging Assistance

Our `BugFix` tool assumes that an existing fault localization technique – such as the IVMP-based approach we previously proposed [17] – is first used to locate a suspicious statement that is likely to be faulty. Once such a statement is found, then our tool can be used to assist a developer in fixing a bug at that statement. This is accomplished by performing the main steps shown in Fig. 2.

In the first step of our tool, the current *debugging situation* is analyzed and characterized in terms of both static (structure of the statement) and dynamic (patterns in the values associated with the statement) information. In the second step, a *knowledgebase of rules* mapping various debugging situations to relevant bug-fix descriptions is queried. Based on the current analyzed debugging situation and the confidence values associated with each rule in the knowledgebase, a prioritized list of bug-fix suggestions relevant to the current debugging situation is computed and reported to the user. In the third step, once the user fixes the bug in the statement, the knowledgebase of rules is updated with new information concerning the most recently-encountered debugging situation and the corresponding bug fix. We now describe each of these three main steps in detail.

3.1. Analyzing the Debugging Situation

A *debugging situation* is a characterization of a particular suspicious statement that is being examined during debugging. It is represented by a set of atomic entities that describe certain static and dynamic details of the statement. We call these entities *situation descriptors*. Intuitively, when two debugging situations are similar to each other, then they will have similar sets of situation descriptors. These situation descriptors represent a way to automatically characterize and compare different debugging situations to see how similar they are.

Situation descriptors can be either static or dynamic, and we consider three types in the current work: (1) those associated with the static structure of the given statement; (2) those associated with patterns in the IVMPs [17] associated with the statement; and (3) those associated with patterns in the values used at the statement by failing and passing runs. We now define each of these types of situation descriptors.

Statement structure situation descriptors. The descriptors pertaining to statement structure are derived from the (unordered) *tokens* comprising the statement, as obtained by tokenizing the statement according to the programming language. To limit the total number of possible descriptors, some tokens are represented abstractly as general situation descriptors. For example, there are an infinite number of different variable names and constant values, so we represent these with general descriptors such as "int-VAR" or "char-CONST". Other tokens such as keywords and operators come from a limited set of possibilities for the given language, so we represent these as situation descriptors named after the keywords/operators themselves. Finally, comments and formatting tokens such as semicolons, parentheses, and curly braces are ignored.

C Structure Entity	Examples	How to Treat as Situation Descriptor	Situation Descriptor
keyword	if switch for case while default return	use as is	(the keyword itself)
operator	+ - * / && || & ~ <= ! -> .	use as is	(the operator itself)
ref / deref	& *	rename	REF DEREF
variable name	foo bar i j sum total average max	generalize	VAR *x*-VAR (*x* is the variable type)
constant value	17 "rabbit" 'y' 4 3.21 0 'z' -3	generalize	CONST *x*-CONST (*x* is the constant type)
function call	foo() bar(x, 53) fprintf("res: %d", a)	generalize	FUNC_CALL
array access	x[10] foo[a+7] a[3]	generalize	ARRAY_ACCESS
cast	(int)a (char)(2+c)	generalize	CAST
format tokens	, { (;] :) [	ignore	(none)

Other general situation descriptors:
ASSIGN_STMT (for assignment statements)
COND_STMT (for conditional statements)
DECL_STMT (for declaration statements)

Figure 3. Deriving situation descriptors from various C program structure entities.

Fig. 3 describes different C program structure entities and how they are treated as situation descriptors by our tool. From this figure, notice that for variable names and constant values, we actually associate two general situation descriptors each: one without a type specifier, and one with a type specifier. This is because, for example, even if two different situations use constants of different type, then they should still be regarded as "slightly similar" because they both at least involve constant values. On the other hand, if both situations use constants of the same type, then they should be regarded as "very similar". The reference and dereference operators "&" and "*" need to be renamed as situation descriptors to avoid conflicts with the bitwise-and and multiplication operators, respectively. There are also general descriptors for assignment statements, conditional statements, and declaration statements. Since C allows for user-defined types, we specify such types as "user-defined" for situation descriptors that require a type to be specified.

Fig. 4 shows an example of some C statements and how they are represented by situation descriptors. The middle column shows how the C statement is tokenized into descriptors from left-to-right. The right-most column shows the final, unordered set of descriptors obtained by removing duplicate descriptors. In structures such as casts, function calls, and array references, the tokens contained *within* these structures are tokenized into descriptors as well.

IVMP pattern situation descriptors. These situation descriptors are derived from patterns that are observed in the IVMPs associated with the given statement. Our tool uses available test cases to search for IVMPs at the current statement [17]. Then, the IVMPs are analyzed for patterns that can be represented by situation descriptors. We consider a *pattern* to occur when corresponding values in the IVMPs compare to each other in the same way across all IVMPs at a statement. The previous example in Fig. 1 involved a pattern in which the two used values are always the same in the original sets of values in the IVMPs. Another pattern occurs when a particular original value always corresponds to a *larger* alternate value in the IVMPs.

C Statement	Tokenized/Converted into Situation Descriptors	Final Set of Descriptors
int x = a + b;	ASSIGN_STMT, int, VAR, int-VAR, =, VAR, int-VAR, +, VAR, int-VAR	ASSIGN_STMT, int, VAR, int-VAR, =, +
c = (char)(2 + *y);	ASSIGN_STMT, VAR, char-VAR, =, CAST, char, CONST, int-CONST, +, DEREF, VAR, int*-VAR	ASSIGN_STMT, VAR, char-VAR, =, CAST, char, CONST, int-CONST, +, DEREF, int*-VAR
if (foo(x) + a[3] < 0)	COND_STMT, if, FUNC_CALL, VAR, int-VAR, +, ARRAY_ACCESS, VAR, int*-VAR, CONST, int-CONST, <, CONST, int-CONST	COND_STMT, if, FUNC_CALL, VAR, int-VAR, +, ARRAY_ACCESS, int*-VAR, CONST, int-CONST, <

Figure 4. Example of C situation descriptors (assume type "int" when unspecified).

To identify patterns in the IVMP values, we consider how pairs of values compare to each other in terms of whether they are *less than*, *greater than*, or *equal to* each other. We do this by looking at pairs of values in three different ways: (1) within just the original sets of values in the IVMPs; (2) within just the alternate sets of values; and (3) between corresponding values in the original and alternate sets of values. Fig. 5 shows an example of how these pairs of values are compared in the three columns labeled "Value Comparisons". These comparisons are computed for each IVMP associated with a statement (there are three IVMPs shown in Fig. 5). If corresponding comparisons match across all IVMPs at the statement, then it is considered to be a pattern and is therefore designated as a situation descriptor (these are highlighted in Fig. 5). We use general names to represent the IVMP values, such as $origDef$ or $altUse2$, so that the names in the descriptors do not vary among different statements or programs. We also look for three additional patterns in IVMPs that we represent with "special" descriptors: descriptor *OTHER-BRANCH* when a branch outcome always changes to the alternate outcome in the IVMPs; descriptor *ONE-TO-ANY* when a single unique value in the original value sets always changes to some other value in the alternate value sets; and descriptor *ANY-TO-ONE* when some original value always changes to a single unique alternate value. In Fig. 5, none of the "special" situation descriptors applied.

Value pattern situation descriptors. These situation descriptors represent patterns in the values involved at a

IVMPs				Value Comparisons		
	Def	Use1	Use2	Original	Alternate	Corresponding
orig:	5	4	1	origDef > origUse1	altDef > altUse1	origDef < altDef
alt:	8	5	3	origDef > origUse2	altDef > altUse2	origUse1 < altUse1
				origUse1 > origUse2	altUse1 > altUse2	origUse2 < altUse2
orig:	10	3	7	origDef > origUse1	altDef > altUse1	origDef < altDef
alt:	11	10	1	origDef > origUse2	altDef > altUse2	origUse1 < altUse1
				origUse1 < origUse2	altUse1 > altUse2	origUse2 > altUse2
orig:	1	0	1	origDef > origUse1	altDef > altUse1	origDef < altDef
alt:	3	0	3	origDef = origUse2	altDef = altUse2	origUse1 = altUse1
				origUse1 < origUse2	altUse1 < altUse2	origUse2 < altUse2
Patterns (situation descriptors) found:				origDef > origUse1 altDef > altUse1 origDef < altDef		

Figure 5. Example of identifying patterns (situation descriptors) in IVMPs.

Exercised Value Sets						Value Comparisons	
Failing Runs			Passing Runs			Failing Runs	Passing Runs
Def	Use1	Use2	Def	Use1	Use2		
0	0	2	1	4	4	failDef = failUse1 failDef < failUse2 failUse1 < failUse2	passDef < passUse1 passDef < passUse2 passUse1 = passUse2
1	8	2	1	13	2	failDef < failUse1 failDef < failUse2 failUse1 > failUse2	passDef < passUse1 passDef < passUse2 passUse1 > passUse2
1	7	2	0	0	0	failDef < failUse1 failDef < failUse2 failUse1 > failUse2	passDef = passUse1 passDef = passUse2 passUse1 = passUse2
			0	7	5		passDef < passUse1 passDef < passUse2 passUse1 > passUse2
Patterns (situation descriptors) found:						failDef < failUse2	
Special patterns (situation descriptors) found:						ONE-FAIL-VALUE	

Figure 6. Example of identifying patterns (situation descriptors) in exercised value sets.

given statement when exercised by both passing and failing runs. Our tool uses executions of the available test cases to identify the various sets of values used at the given statement, and these value sets are classified into two groups: those coming from failing runs, and those coming from passing runs. We search for patterns among these values in a similar way as was done for the IVMP pattern situation descriptors. First, for each set of values exercised by failing runs, we see how pairs of values compare to each other and then determine which comparisons are consistent across all failing-run value sets; the consistent relationships are designated as situation descriptors. Next, we do the same for the passing run value sets. Finally, we check for four additional patterns that we represent using the following "special" situation descriptors: descriptors *ALL-FAIL-SMALLER* and *ALL-FAIL-LARGER* if a particular value from the failing run value sets is respectively always smaller or always larger than the corresponding value from the passing run value sets; descriptors *ONE-FAIL-VALUE* and *ONE-PASS-VALUE* if a particular value is the same in all failing run value sets or in all passing run value sets, respectively.

Fig. 6 shows an example with three exercised value sets from failing runs, and four exercised value sets from passing runs. The comparisons between all pairs of values in the failing and passing value sets are shown. In this case, one value comparison pattern is consistent across all failing runs and is represented by a situation descriptor, and the special *ONE-FAIL-VALUE* situation descriptor applies as well because value $Use2$ in all the failing runs is 2.

3.2. Prioritizing Bug-Fix Suggestions

Once the set of situation descriptors to characterize the current debugging situation has been determined, `BugFix` queries a *knowledgebase of rules* that map various debugging situations to bug-fix descriptions. The result of this query is a prioritized list of bug-fix suggestions that is relevant to the current debugging situation. We first describe this knowledgebase of rules, and then we show how it can be used to compute a prioritized list of bug-fix suggestions relevant to the current debugging situation.

The knowledgebase of rules. The knowledgebase of rules is derived from a *database of bug-fix scenarios* that is maintained by our tool. This database is initially created through *training data* composed of some set of known debugging situations and their corresponding bug-fix descriptions. Each time our tool encounters a new debugging situation and its corresponding bug fix, this new scenario is added to the database. Whenever the database is altered, it is passed as input to the `apriori` association rule learning algorithm [5] (described previously in Section 2.1) to compute a revised knowledgebase of rules.

Each rule in the knowledgebase maps a particular *debugging situation* to a corresponding *bug-fix description*. We showed in the previous section that a debugging situation is represented by a set of atomic situation descriptors. A bug-fix description is simply an atomic textual description of how to modify a statement to fix a bug. Note that a particular bug fix can be described in more general or more specific terms. For example, changing an operator from $<$ into $<=$ can be described generally as an "operator mutation", and more specifically as "change $<$ into $<=$". To account for this, we allow a developer to describe a bug fix using multiple bug-fix descriptions. This allows the tool to be more versatile in reporting the most relevant bug-fix suggestions for a debugging situation: sometimes, a more general bug-fix suggestion may be appropriate, while a more specific suggestion may be misleading.

The `apriori` algorithm also associates a *confidence* value with each rule, indicating how likely it is that a rule properly maps a debugging situation to an appropriate bug-fix description.

The knowledgebase of rules is such that if a rule R exists that has a particular debugging situation S and bug-fix description F, then other rules will exist in which various subsets of S map to the same F. However, the confidence values associated with these other rules must be less than or equal to the confidence of rule R. For example, assume

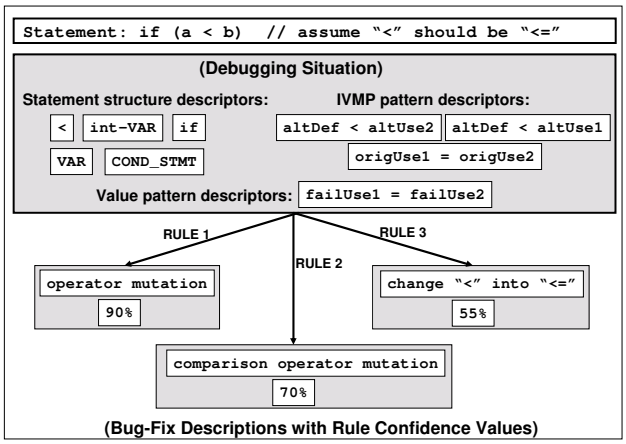

Figure 7. Example of 3 rules (one debugging situation mapped to 3 bug-fix descriptions).

Current debugging situation: A B D F

RULE	PRIO-1	PRIO-2	RULE	PRIO-1	PRIO-2
* A -> a (33%)	4	8	* A D -> a (100%)	1	2
* A -> c (33%)	4	8	A E -> c (100%)		
* A -> e (34%)	3	6	B C -> a (100%)		
* B -> a (14%)	6	11	* B D -> a (71%)	2	3
* B -> b (12%)	7	12	* B D -> b (29%)	5	9
* B -> c (33%)	4	8	B E -> b (12%)		
* B -> d (7%)	8	13	B E -> c (33%)		
* B -> e (34%)	3	6	B E -> d (55%)		
C -> a (100%)			C D -> a (100%)		
* D -> a (71%)	2	4	D E -> b (100%)		
* D -> b (29%)	5	10	A B C -> a (100%)		
E -> b (12%)			* A B D -> a (100%)	1	1
E -> d (55%)			A B E -> c (100%)		
E -> c (33%)			A C D -> a (100%)		
* A B -> a (33%)	4	7	B C D -> a (100%)		
* A B -> c (33%)	4	7	B D E -> b (100%)		
* A B -> e (34%)	3	5	A B C D -> a (100%)		
A C -> a (100%)					

Prioritized list of bug-fix suggestions (w/o duplicates): a e c b d

Figure 8. Example of prioritizing bug-fix suggestions for a given debugging situation.

that a person who buys eggs and bread almost certainly also buys milk. Then if a person does indeed buy eggs and bread, we would have high confidence that the person will also buy milk. However, if another person buys only eggs, then that person may also buy milk but we would have less confidence that this will be the case.

Fig. 7 shows an example of what three rules might look like in the knowledgebase of rules for a conditional statement. In the figure, there is a single debugging situation composed of 9 situation descriptors that is mapped to three different bug-fix descriptions (from more general to more specific), each with a different confidence value.

Prioritizing the bug-fix suggestions. Given a current debugging situation, our tool prioritizes the bug-fix descriptions in the knowledgebase of rules by performing four steps: (1) identifying rules to consider; (2) sorting rules by confidence values; (3) breaking ties by number of situation descriptors; and (4) reporting the prioritized bug-fix descriptions as suggestions.

Identifying rules to consider. First, the subset of rules to be considered for prioritization is identified. These rules are those in which the debugging situation associated with the rule is a *subset* of the current debugging situation. Only these rules are considered because any rule that is *not* a subset will involve at least one situation descriptor that *does not apply* to the current debugging situation.

Fig. 8 shows an abstract example in which there are 35 rules in the knowledgebase, and the bug-fix descriptions are ranked with respect to a current debugging situation. In the figure, each rule is shown with capital letters to represent situation descriptors, and lower-case letters for bug-fix descriptions. Confidence values are in parentheses after each rule. An asterisk (*) before a rule indicates that the rule is considered for prioritization since its set of situation descriptors is a subset of the current debugging situation.

Sort rules by confidence values. The rules being considered are ranked in decreasing order of confidence value. In Fig. 8, the computed ranking is shown in the columns labeled "PRIO-1" (rank value 1 is the highest rank).

Break ties by number of situation descriptors. Any ties are broken by ordering rules in decreasing order of situation descriptor set size. Our rationale for breaking ties in this way is the following: if two rules have the same confidence value, then the rule that has more situation descriptors in common with the specified debugging situation is likely to be associated with a more-relevant bug-fix description. In Fig. 8, the ranking computed in this step is shown in the columns labeled "PRIO-2".

Report prioritized bug-fix descriptions as suggestions. Finally, the bug-fix descriptions are reported as suggestions in order of their associated prioritized rules. If there are duplicate bug-fix descriptions, then only the first occurrence of each one in the sorted list is reported.

3.3. Learning from the Debugging Scenario

Once a developer fixes a bug in the current statement, the final step of our tool is to *learn* from this newly-encountered debugging situation and the corresponding bug fix. This is done by allowing the developer to describe the bug fix in terms of one or more bug-fix descriptions, and then adding a new entry representing the current debugging scenario to the *database of bug-fix scenarios*. The database is then passed as input to the `apriori` algorithm, which computes a revised *knowledgebase of rules*.

BugFix is designed so that it can become more effective over time at accurately predicting the most relevant bug fixes for debugging situations. It is fully automated except for the step of actually making the proper bug fix at a faulty statement and specifying that bug fix in terms of bug-fix descriptions. The tool currently assumes that a bug can be fixed by modifying a single source code statement.

4. Case Study

We now present a case study illustrating the use of our BugFix tool. This case study is designed to show the potential benefit of BugFix; we leave a thorough experimental evaluation for future work. In this study, we show how our tool can be used to derive helpful bug-fix suggestions for several debugging situations (assuming that the faulty statement has been located). We used an implementation of the apriori association rule learning algorithm obtained from [15]. For the faulty programs to be debugged, we used a subset of the Siemens benchmark programs [16] described in Table 1. We selected these faults because they can be easily and clearly described in detail, and because they highlight interesting aspects of our approach that show the potential benefit of BugFix. To enable identification of IVMP and value pattern situation descriptors for the debugging situations, we associated a branch-coverage adequate test suite with each faulty program consisting of at least 5 failing runs and 5 passing runs, selected from test case pools associated with each Siemens program.

Prog. Name	LOC	Program Description	Faulty Versions Used
tcas	138	altitude separation	v6, v9, v20
totinfo	346	statistic computation	v16
sched	299	priority scheduler	v3
sched2	297	priority scheduler	v7
replace	516	pattern substituter	v1, v23

Table 1. Siemens benchmark programs used in our case study.

Initial training. Our tool is designed to become more effective over time at reporting the most relevant bug-fix suggestions for a given debugging situation. However, initially the tool must be *trained* using a set of known debugging situations and their corresponding bug fixes. This ensures that an initial knowledgebase of rules will exist. With more training, the tool is expected to perform more effectively on new debugging situations. To illustrate training in our case study, we used the following faulty programs and their known bug fixes: tcas v6, replace v1, schedule v3, and totinfo v16. For tcas v6, we show the full information (faulty statement, debugging situation, and bug-fix descriptions) in Fig. 9. For the remaining training programs, we show only the faulty statements and corresponding bug-fix descriptions in the figure. Notice in the figure that for tcas v6, the IVMP pattern situation descriptors "$origDef < altDef$" and "$origDef > altDef$" seem contradictory. This is because IVMPs are computed with respect to binary instructions in our implementation, and we include IVMP patterns from different binary instructions if they are associated with the same program statement.

To learn from the four debugging scenarios in Fig. 9, we create four different *itemsets* – one for each of the four debugging scenarios – by taking the union of the debugging situation descriptors and the bug-fix descriptions. Then we simply pass these four itemsets (comprising the current *database of bug-fix scenarios*) to the apriori algorithm [5] so that the rules can be automatically derived. When invoking apriori, we instruct the algorithm to only report rules in which antecedents are comprised of only situation descriptors, and consequents are each comprised of a single bug-fix description. This ensures that all rules map debugging situations to bug-fix descriptions. We do not limit the considered rules based on *support* value, but we do limit the considered rules to those with *confidence* value at least 80% (this value was found to yield good results in our case study, based on the training data).

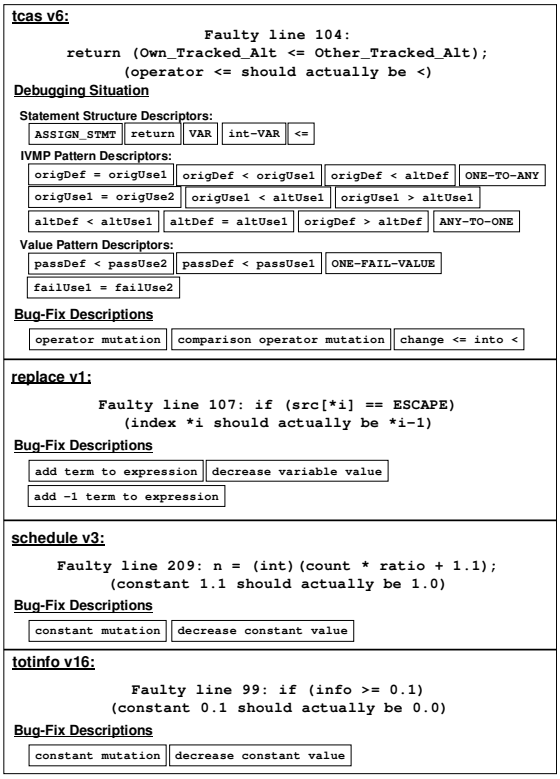

Figure 9. Four faulty program debugging scenarios used to train our tool.

Table 2 shows the bug-fix descriptions known to our tool for this case study. Each description has an abbreviation as specified in the "Abbrev" column. The "When Learned" column describes when the corresponding description is learned by the tool (either during initial training, or else through the remainder of this case study as new debugging situations are encountered).

Encountering new debugging situations. Based on the knowledgebase of rules obtained from the initial training, we are now ready to see how our tool behaves when encountering a new debugging situation. For this, we will look at four new debugging scenarios, described in Fig. 10 (the as-

Bug-Fix Description	Abbrev.	When Learned
operator mutation	*opm*	training
comparison operator mutation	*copm*	training
change <= into <	$<= \mid <$	training
change >= into >	$>= \mid >$	new situation
add term to expression	$e+t$	training
decrease variable value	$v-$	training
increase variable value	$v+$	new situation
add -1 term to expression	$e-1$	training
add +1 term to expression	$e+1$	new situation
constant mutation	$c+-$	training
decrease constant value	$c-$	training

Table 2. Bug-fix descriptions known during this case study.

sociated situations descriptors in the figure are omitted due to space limitations). Notice that `tcas` v9 and `tcas` v20 have faulty statements that look identical, but they are actually distinct statements occurring at two different source code lines in the program.

`tcas` *v9.* This faulty program involves a bug in which a comparison operator $>=$ at line 90 should actually be $>$. For this debugging situation, we identify the situation descriptors and then query the (trained) knowledgebase to obtain a prioritized list of relevant bug-fix suggestions. `BugFix` reports the prioritized list [$<= \mid <$, *copm*, *opm*, $e-1$, $v-$, $e+t$, $c-$, $c+-$], with associated rank values [1, 1, 1, 2, 2, 2, 3, 3] (in other words, the first 3 suggestions are tied for rank 1, the next 3 suggestions are tied for rank 2, and the last 2 suggestions are tied for rank 3). These results imply that potential bug-fixes $<= \mid <$, *copm*, and *opm* should be considered first by a developer. Suggestions *copm* and *opm* are indeed effective, since the current situation does require a comparison operator mutation. Suggestion $<= \mid <$ is less effective, but it is similar to the expected fix (the fix in this case, changing $>=$ into $>$, is not yet known to the tool). We believe these results can quickly guide a developer to the appropriate fix in this faulty statement. After the fix is made, suppose the developer describes the bug fix with three descriptors: *operator mutation*, *comparison operator mutation*, and *change* $>=$ *into* $>$. The tool then learns from this current debugging scenario.

`tcas` *v20.* This faulty statement looks identical to the one just seen in `tcas` v9, but since it is a distinct statement at a different source code line, it turns out that the debugging situation is slightly different due to some differences in the IVMP patterns. However, we expect that the knowledge just learned from scenario `tcas` v9 should be beneficial in helping the tool to report highly relevant bug-fix suggestions for this new situation. Indeed, the prioritized bug-fix suggestions reported by our tool for this new situation are [$>= \mid >$, *copm*, *opm*, $<= \mid <$], with associated rank values [1, 1, 1, 2]. In this case, the other bug-fix descriptions contained in the knowledgebase are not reported

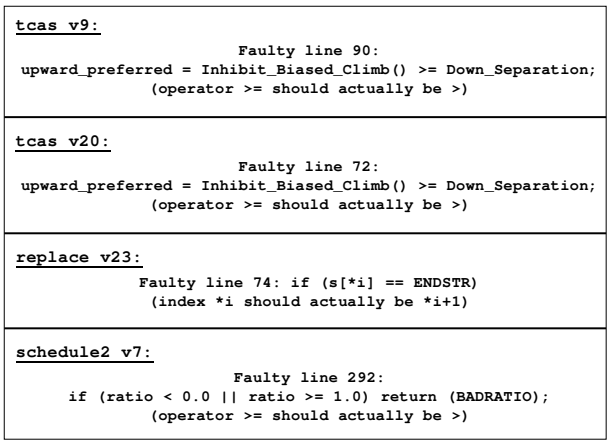

Figure 10. Four new debugging scenarios after the initial training.

in the prioritized list because their associated rules all have confidence values less than the specified minimum threshold. All 3 suggestions with highest rank in the prioritized list precisely match the expected fix to make at this faulty statement. Thus, this demonstrates how the results reported by `BugFix` can improve over time as more knowledge is automatically learned through continued use of the tool.

`replace` *v23.* In the faulty statement associated with this faulty program, an array index $*i$ should actually be $*i+1$. This bug has some similarities to `replace` v1 that was involved during training of our tool. The prioritized list of bug-fix suggestions in this case turns out to be [$e-1$, $v-$, $e+t$, $>= \mid >$, *copm*, *opm*, $c-$, $c+-$], with associated ranks [1, 1, 1, 2, 2, 2, 3, 3]. The highest-ranked suggestions ($e-1$, $v-$, and $e+t$) are, as might be expected, the same three bug-fix descriptions associated with the similar scenario `replace` v1 encountered during training. In this case, suggestion $e+t$ is appropriate because a term should be added to the array index expression. However, suggestions $e-1$ and $v-$ are not quite consistent with the expected fix, since here, the value of a variable should actually be *increased* by adding a $+1$ term to the index expression. On the other hand, these expected bug-fix suggestions ($v+$ and $e+1$) are not yet known to the tool, and therefore could not have been reported by the tool. However, after the bug is fixed, bug-fix suggestions $v+$ and $e+1$ will henceforth be known to the tool, so more effective suggestions can be reported in a similar situation in the future.

`schedule2` *v7.* Here we encounter a debugging situation in a completely new subject program that has not yet been encountered by our tool. Although the bug in this case (comparison operator $>=$ erroneously used instead of $>$) is familiar, the statement itself is rather unique as compared to what has been previously encountered. Based upon all knowledge learned from the previously-encountered debugging scenarios, our tool reports for the current scenario

the following prioritized list of bug-fix suggestions: [$c-$, $c+-$, $>= \mid >$, $copm$, opm, $<= \mid <$, $e-1$, $v-$, $e+t$, $e+1$, $v+$], with associated ranks [1, 1, 2, 2, 2, 3, 3, 3, 3, 4, 4]. In this case, it seems the expected fix is represented by suggestions $>= \mid >$, $copm$, and opm, which are all given rank 2. However, unexpectedly for us, we discovered that rank-1 suggestion $c+-$ also implies another possible fix: mutating the constant value 1.0. Indeed, instead of changing the predicate $ratio >= 1.0$ into $ratio > 1.0$ (the expected change), it may also be appropriate to instead mutate the constant so the predicate becomes $ratio >= 1.001$ (an unexpected change). It turns out that with the latter change, all available test cases pass. However, the latter change is not semantically equivalent to the former expected change. A developer must determine whether such a change is indeed appropriate, given the specification of the program.

5. Related Work

Fixing bugs. The focus of our current work is to automatically generate suggestions for modifying source code to fix bugs. He and Gupta [13] developed an approach that uses the notion of path-based weakest preconditions to automatically generate program modifications to correct an erroneous statement in a function. Unlike their approach which requires a formal specification for a function, our tool requires only a faulty program and at least one failing test case. Abraham and Erwig's debugging tool [1] identifies change suggestions for users to correct errors in spreadsheets. Our tool, on the other hand, is implemented to work on general `C` programs and can be adapted to handle other programming languages as well.

Locating Bugs. Bugs must first be located in program code before they can be fixed.

`Slicing-based approaches`. *Static Slicing* [31] identifies a subset of program statements that may influence the value of a variable at a program location. The related concepts of *Dynamic Slicing* [2, 23, 30, 35, 36] and *Relevant Slicing* [3, 10] have also been studied. Slices can be used to identify a subset of statements that are likely to contain a faulty statement. `The Whyline` debugging tool [22] uses a combination of slicing and other analysis techniques to help explain program output.

`State-altering approaches`. In the *Delta Debugging* framework, failure-inducing input is identified [33] that allows for the computation of cause-effect chains for failures [32] which can in turn be linked to faulty code [6]. This is accomplished by swapping program state (the values of variables) between a successful and failing run. Misherghi and Su [26] recently proposed an improved Delta Debugging algorithm for minimizing failure-inducing inputs. *Predicate Switching* [34] attempts to isolate erroneous code by identifying predicates whose outcomes can be altered during a failing run to cause it to become passing. *Value Replacement* [17] involves searching for the suspicious statements at which values can be replaced during the execution of a failing run to cause the run to become passing. *Suppression* [18] can be used to isolate the root causes of memory bugs by iteratively suppressing the effects of known corrupted memory locations during program execution.

`Statistical approaches`. Approaches based on statistical analysis [20, 21, 24, 25] use dynamic information obtained from test case executions to rank program statements according to likelihood of being faulty. Jiang and Su [19] proposed a context-aware approach that constructs faulty control flow paths linking bug predictors together, to help explain bugs. *Nearest Neighbor* [29] searches for a correct execution that is most similar to an incorrect execution, compares the spectra for these two executions, and identifies the most suspicious parts of the program.

Revealing bugs. *Check 'n' Crash* [7] derives error conditions statically and then generates concrete test cases to dynamically verify whether a bug truly exists. *Eclat* [28] infers an operational model of the correct behavior of a program and identifies inputs whose operational execution patterns differ from the model in particular ways; these inputs are likely to be fault-revealing. The *Extended Static Checker for Java* [9] looks for common programming errors at compile-time by way of an annotation language in which a developer can formally express design decisions. *Daikon* [8] and *DIDUCE* [11] can be used to find bugs through invariant detection. *Valgrind* [27] and *Purify* [12] can be used to detect certain kinds of memory bugs. The `FindBugs` tool [14] automatically detects bug patterns in Java programs.

6. Conclusions and Future Work

We have presented a learning tool called `BugFix` that can automatically assist developers in fixing bugs, which identifies a prioritized list of bug-fix suggestions that are relevant to a given debugging situation. Through a machine learning technique, the tool learns about new debugging situations and their corresponding bug fixes as they are encountered, thus increasing the effectiveness of the tool over time. We also presented a case study illustrating the effectiveness and potential of our tool. Our next step is to conduct a detailed empirical evaluation of `BugFix` to analyze its performance and effectiveness in helping programmers fix buggy software. We also plan to improve the characterization of debugging situations by incorporating information about (1) the failure manifested by the system; and (2) the context of the faulty statement being analyzed, such as the block of code containing the faulty statement.

Acknowledgements. We would like to thank the anonymous reviewers for their valuable feedback. This research is supported by NSF grants CNS-0751961, CNS-0751949, CNS-0810906, and CCF-0753470 to UC Riverside.

References

[1] R. Abraham and M. Erwig. Goal-directed debugging of spreadsheets. *Symposium on Visual Languages and Human-Centric Computing*, pages 37–44, September 2005.

[2] H. Agrawal and J. R. Horgan. Dynamic program slicing. *Conference on Programming Language Design and Implementation*, pages 246–256, June 1990.

[3] H. Agrawal, J. R. Horgan, E. W. Krauser, and S. London. Incremental regression testing. *International Conference on Software Maintenance*, pages 348–357, September 1993.

[4] R. Agrawal, T. Imieliński, and A. Swami. Mining association rules between sets of items in large databases. *Proc. of the 1993 ACM SIGMOD International Conference on Management of Data*, 22(2):207–216, 1993.

[5] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. *International Conference on Very Large Data Bases*, pages 487–499, 1994.

[6] H. Cleve and A. Zeller. Locating causes of program failures. *27th International Conference on Software Engineering*, pages 342–351, May 2005.

[7] C. Csallner and Y. Smaragdakis. Check 'n' Crash: Combining static checking and testing. *International Conference on Software Engineering*, pages 422–431, May 2005.

[8] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, February 2001.

[9] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. *Conference on Programming Language Design and Implementation*, pages 234–245, June 2002.

[10] T. Gyimothy, A. Beszedes, and I. Forgacs. An efficient relevant slicing method for debugging. *Foundations of Software Engineering*, pages 303–321, September 1999.

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

[12] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. *Proc. of the USENIX Winter Technical Conference*, pages 125–136, 1992.

[13] H. He and N. Gupta. Automated debugging using path-based weakest preconditions. *Fundamental Approaches to Software Engineering*, pages 267–280, March 2004.

[14] D. Hovemeyer and W. Pugh. Finding bugs is easy. *ACM SIGPLAN Notices*, 39(12):92–106, December 2004.

[15] `http://www.borgelt.net/apriori.html`.

[16] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow and controlflow-based test adequacy criteria. *International Conference on Software Engineering*, pages 191–200, May 1994.

[17] D. Jeffrey, N. Gupta, and R. Gupta. Fault localization using value replacement. *International Symposium on Software Testing and Analysis*, pages 167–178, July 2008.

[18] D. Jeffrey, N. Gupta, and R. Gupta. Identifying the root causes of memory bugs using corrupted memory location suppression. *IEEE International Conference on Software Maintenance*, pages 356–365, September 2008.

[19] L. Jiang and Z. Su. Context-aware statistical debugging: From bug predictors to faulty control flow paths. *Proc. of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, pages 184–193, November 2007.

[20] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. *Proc. of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 273–282, November 2005.

[21] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. *International Conference on Software Engineering*, pages 467–477, May 2002.

[22] A. Ko and B. Myers. Debugging reinvented: Asking and answering why and why not questions about program behavior. *Proc. of the 30th International Conference on Software Engineering*, pages 301–310, May 2008.

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

[24] B. Liblit, M. Naik, A. Zheng, A. Aiken, and M. Jordan. Scalable statistical bug isolation. *Conference on Programming Language Design and Impl.*, pages 15–26, June 2005.

[25] C. Liu, X. Yan, L. Fei, J. Han, and S. Midkiff. SOBER: Statistical model-based bug localization. *European Software Engineering Conf. held jointly with Intl. Symposium on Foundations of Software Engineering*, pages 286–295, September 2005.

[26] G. Misherghi and Z. Su. HDD: Hierarchical delta debugging. *Proc. of the 28th International Conference on Software Engineering*, pages 142–151, May 2006.

[27] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *Conference on Programming Language Design and Implementation*, pages 89–100, June 2007.

[28] C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. *Object-Oriented Programming, 19th European Conference*, pages 504–527, July 2005.

[29] M. Renieris and S. Reiss. Fault localization with nearest neighbor queries. *International Conference on Automated Software Engineering*, pages 30–39, October 2003.

[30] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, September 1995.

[31] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.

[32] A. Zeller. Isolating cause-effect chains from computer programs. *10th International Symposium on the Foundations of Software Engineering*, pages 1–10, November 2002.

[33] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, February 2002.

[34] X. Zhang, N. Gupta, and R. Gupta. Locating faults through automated predicate switching. *International Conference on Software Engineering*, pages 272–281, May 2006.

[35] X. Zhang, N. Gupta, and R. Gupta. Pruning dynamic slices with confidence. *Conference on Programming Language Design and Implementation*, pages 169–180, June 2006.

[36] X. Zhang and R. Gupta. Cost effective dynamic program slicing. *Conference on Programming Language Design and Implementation*, pages 94–106, June 2004.