Context-Aware Statistical Debugging: From Bug Predictors to Faulty Control Flow Paths*

Lingxiao Jiang Zhendong Su
University of California at Davis
{jiangl,su}@cs.ucdavis.edu

ABSTRACT

Effective bug localization is important for realizing automated debugging. One attractive approach is to apply statistical techniques on a collection of evaluation profiles of program properties to help localize bugs. Previous research has proposed various specialized techniques to isolate certain program predicates as bug predictors. However, because many bugs may not be directly associated with these predicates, these techniques are often ineffective in localizing bugs. Relevant control flow paths that may contain bug locations are more informative than stand-alone predicates for discovering and understanding bugs. In this paper, we propose an approach to automatically generate such faulty control flow paths that link many bug predictors together for revealing bugs. Our approach combines *feature selection* (to accurately select failure-related predicates as bug predictors), *clustering* (to group correlated predicates), and control flow graph traversal in a novel way to help generate the paths. We have evaluated our approach on code including the Siemens test suite and rhythmbox (a large music management application for GNOME). Our experiments show that the faulty control flow paths are accurate, useful for localizing many bugs, and helped to discover previously unknown errors in rhythmbox.

Categories and Subject Descriptors: D.2.4/D.2.5 [**Software Engineering**]: Testing and Debugging, Software/Program Verification—*Reliability, Statistical methods, Debugging aids*

General Terms: Experimentation, Reliability

Keywords: Bug localization, Statistical debugging, control flow analysis, machine learning

1. INTRODUCTION

Debugging is an important part of the software development process because developers spend significant fraction of their time on debugging. Traditionally, debugging is a manual process and often done in two steps: (1) under testing, an application exhibits unexpected behavior, and (2) the developer examines the execution states looking for causes of the problem. Such a manual task can be tedious, challenging, and error-prone because the state space is typically very large and may not even be completely available (*e.g.*, in the case of a failed user run). It is desirable to automate the debugging process as much as possible.

Bug localization is a step towards automated debugging: much code unrelated to bugs is filtered out and only the remaining code needs further debugging. Effective bug localization techniques can potentially save much developer time by not only pinpointing bug locations in code but also providing useful contextual information for understanding the bug causes.

In recent years, much research has been devoted to this area. One general, attractive approach is to rely on feedback data from the large number of users of deployed software, as shown in Figures 1(a)–(d): (1) an application is first (lightly) instrumented to profile certain program properties; (2) users of the instrumented application send execution profiles to a central database; and (3) postmortem analyses, such as *statistical debugging*, are performed on the gathered profiles to identify bug predictors that may refer to actual bug locations. Statistical debugging [2, 38, 39, 41, 59] is based on a low-overhead, privacy-safe instrumentation infrastructure within the context of Cooperative Bug Isolation (CBI) [36]. In this infrastructure, program predicates, such as the number of times a branch condition is taken, are recorded, then statistical models are applied to rank the predicates in terms of how closely they relate to bugs. Developers can then inspect highly ranked predicates (in isolation) for actual bugs. If users are willing to tolerate more performance overhead, more heavyweight instrumentation mechanisms may be deployed to gather more information for identifying bug predictors. For example, Tarantula [30, 31] instruments almost every statement in a program, and ranks and visualizes the statements according to their potential relations with bugs.

These techniques may be effective at locating *where* bugs may lie, but often do not provide sufficient information for debugging, which may require additional contextual information (*e.g.*, control and data dependency, or concrete execution traces) to understand *how* highly ranked predicates lead to program failures or *why* they are related to real bugs. In this paper, we aim at answering both *where* and *how* program failures happen and thus improving existing statistical debugging techniques. In particular, we introduce a *context-aware* approach that considers not only individual bug predictors but also predicate correlations and control flow paths that connect the bug predictors and correlated predicates for better diagnosis of bugs. We extend the general approach shown in Figures 1(a)–(d) in two aspects: (1) we exploit the profiles gathered from users further to discover correlations among program predicates (Figure 1(e)), and (2) we propose an efficient algorithm for constructing such faulty control flow paths to help diagnose potential bugs (Figure 1(f)).

*This research was supported in part by NSF NeTS-NBD Grant No. 0520320, NSF CAREER Grant No. 0546844, NSF CyberTrust Grant No. 0627749, and a generous gift from Intel. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

ASE'07, November 5–9, 2007, Atlanta, Georgia, USA.

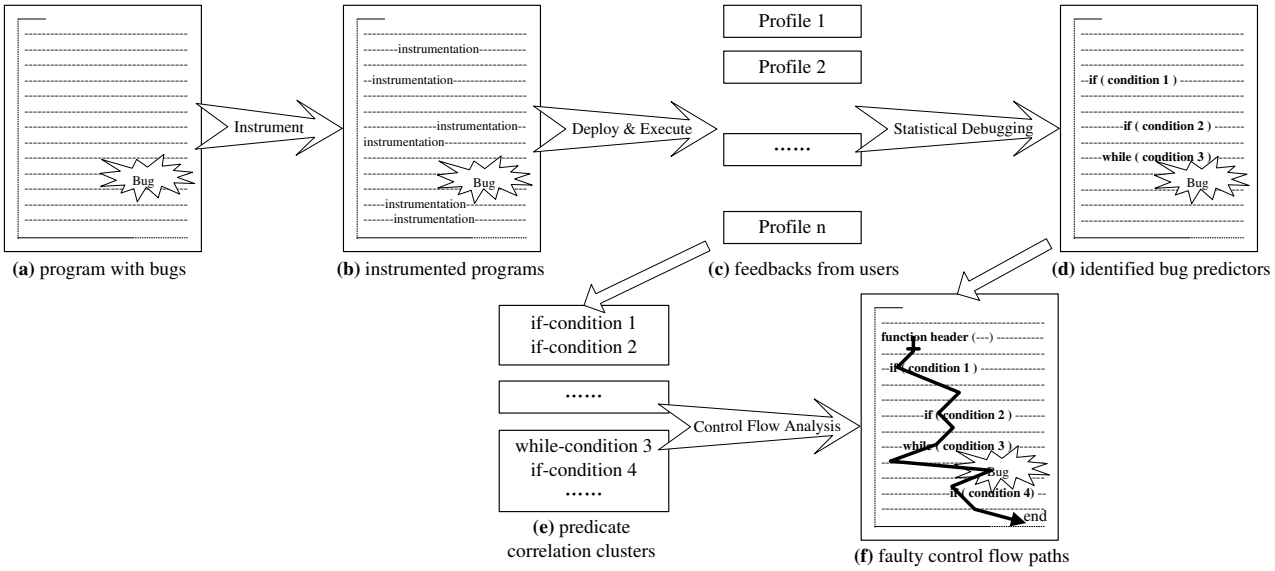

Figure 1: The framework of a general bug localization approach.

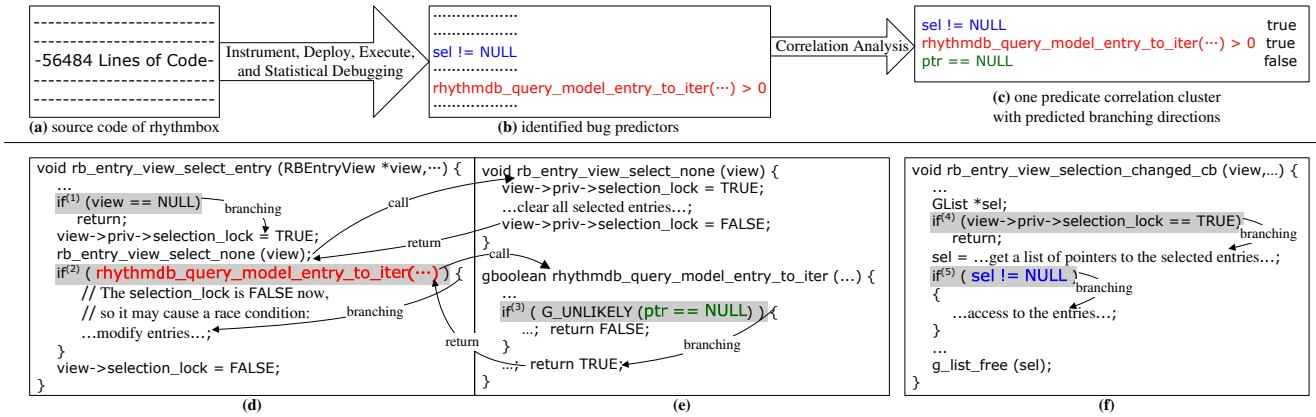

Figure 2: Sample code fragment extracted from rhythmbox 0.6.4 (solid arrows indicate control flow transfers).

We use a real-world example to illustrate how our extended approach can be more useful for localizing bugs than existing statistical debugging. Figures 2(d)–(f) show code extracted from `rhythmbox` 0.6.4, a music management application for GNOME with a total of 56484 lines of code [39]. In fact, the code in Figures 2(d) and 2(e) can be executed concurrently with the code in Figure 2(f), and races may occur because all the code snippets write to the unprotected variable `view->priv->selection_lock`.

Without considering bug context, our approach can predict many program predicates, including the seemingly unrelated two shown in Figure 2(b), as bug predictors. Although the predicates are close to actual bug locations, presenting developers such predicates in isolation is not very helpful for them to understand the bug. To improve the situation, (1) our approach further discovers predicate correlations and predicts branching directions based on execution profiles. E.g., the two predicates in Figure 2(b) and more related predicates are grouped into the same correlation cluster (Figure 2(c)), indicating that they may be all responsible for the bug under the predicted branching directions; (2) Control flow paths that connect these predicates are constructed to help reveal the bug cause (Figures 2(d)–(f)).

The paths in Figures 2(d)–(e) involve three functions and approximately 20 lines of code (excluding blank lines). By inspecting this control flow path, one can see that `view->priv->selection_lock` is always `FALSE` on the line of $\texttt{if}^{(2)}$ and the condition for $\texttt{if}^{(2)}$ is `TRUE`. Thus, the modification in the true-branch may easily cause races if different threads in rhythmbox run the code at the same time. In addition, the path in Figure 2(f) (involving approx. 10 lines of code) shows that `view->priv->selection_lock` is never set to `TRUE` after $\texttt{if}^{(4)}$ and the operations in the true-branch of $\texttt{if}^{(5)}$ may cause another race.[1] If we had isolated the bug predictors without predicate clustering and control flow paths, it would have been more difficult to understand the bug.

Our approach is based on a novel combination of the CBI instrumentation infrastructure [36], feature selection and clustering in machine learning, and control flow graph analysis. Figure 3 shows the organization of our approach. First, a program is instrumented and execution profiles for certain program predicates are collected (Section 2.1). Second, the data are preprocessed and fed into machine learning algorithms to produce two kinds of information. One is about which program predicates are most likely bug predictors. We obtain this information via *feature selection* (Section 3.1) using two well-known classification algorithms: support vector machines (SVMs) and random forests (RFs) (Section 2.2). The other is about which predicates are correlated in terms of similar evaluation histories via *clustering* (Section 3.2) using a variant of the k-means clustering algorithm. Next, we utilize this information to heuristi-

[1]The latest version of rhythmbox has been modified to use the programming models (real locks and critical sections) provided by `GTK+`, instead of the naive `selection_lock`, to avoid such races.

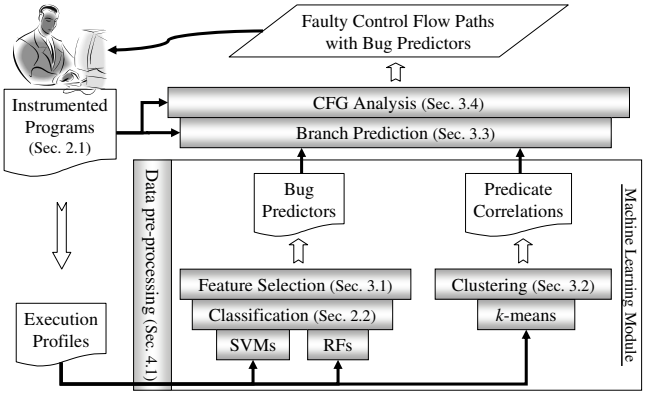

Figure 3: The organization of our approach.

cally predict the directions of conditional branches (Section 3.3). Then, guided by such branch predictions, our approach traverses the program's control flow graph (CFG) to identify faulty control flow paths that connect the bug predictors and correlated predicates (Section 3.4). Finally, the developer can inspect the identified paths to locate actual bugs.

We also present an empirical evaluation of our approach and discuss potential threats to its validity (Section 4). The results show that our context-aware approach helps developers more easily diagnose program errors, in terms of numbers of revealed bugs and the amount of code examined, than existing bug localization techniques which provide only stand-alone bug predictors. In particular, we located 79 bugs out of 132 in the Siemens test suite [24] by inspecting the constructed faulty control flow paths, 38 of which were located by inspecting at most 1% of the code in each program. Also, five real bugs in rhythmbox 0.6.4 (a real-world, multi-threaded music management application [51] for GNOME) with two previously unknown were discovered; each bug required inspecting tens to hundreds of lines of code (out of 56484 lines).

2. TECHNICAL BACKGROUND

2.1 Program Instrumentation Infrastructure

Our approach uses execution profiles collected by the CBI infrastructure [36]. CBI lightly instruments a program with statically fixed n predicates, and an execution of the instrumented program is recorded as an n-dimensional vector, where the i-th value of the vector counts the number of times that the i-th predicate is observed to be true during the execution. The vector for each execution also has a label, indicating failure or success of the execution. A key observation from CBI and *statistical debugging* [2, 38, 39, 41, 59] is that although it is practically impossible to recover program behavior (or user specific information) from one vector, a large collection of such vectors from many users can be useful for understanding its (mis)behavior. In this paper, we use the following three kinds of predicates instrumented by CBI:

branches: Given a branch condition C, two predicates $C == true$ (*abbr.* C_T) and $C == false$ (*abbr.* C_F) may be instrumented to count how many times an execution takes the true branch and the false branch respectively.

returns: Given a function call site with return value R, three predicates $R < 0$, $R == 0$, and $R > 0$ may be instrumented to track the sign of the return value.

scalar-pairs: Given two program variables A and B (B can also be 0 instead) of the same scalar or pointer type, three predicates $A < B$, $A == B$, and $A > B$ may be instrumented to track the arithmetic relationship between A and B.

Different predicates may indicate various types of bugs: wrong branch conditions may indicate that a program enters abnormal paths; wrong return values may indicate failures of function invocations; and scalar-pair relations may help reveal null-pointer dereferences and out-of-boundary issues.

2.2 Machine Learning

We now introduce several machine learning concepts [43] relevant to our approach.

2.2.1 Definitions

In machine learning, a data set U is usually given: each data point $u \in U$ (1) is a value vector $\langle v_1, \ldots, v_n \rangle$ for a set of pre-selected features $P = \{p_1, \ldots, p_n\}$, and (2) has an associated label L_u that indicates the class to which u belongs. Vectors collected by CBI correspond naturally to such a data set U: each instrumented predicate corresponds to a feature p_i, each execution profile corresponds to a data point u, and the execution result (*success* or *failure*) corresponds to the class label L_u.

Classification, *feature selection*, and *clustering* are three common learning tasks performed on U:

Classification: Given U and P, establish a classification function $M(u)$ to map each $u \in U$ to a class, maximizing *classification accuracy*, *i.e.*, maximizing the number of correct mappings (compared with u's own label), even if U contains noise data (that are irrelevant for the final function or that have wrong labels) or data with missing values.

Feature Selection: Given U and P, select a subset of features $P_k = \{p_{s_1}, \ldots, p_{s_k}\} \subseteq P$, such that the classification function based on U_k and P_k is still accurate enough compared with the classification function based on U and P, where U_k is U projected onto P_k, *i.e.*, $U_k = \{\langle v_{s_1}, \ldots, v_{s_k} \rangle \mid \langle v_1, v_2, \ldots, v_n \rangle \in U\}$. Usually k is restricted to a constant, and it is referred to as *k-feature selection*. The selected features have more impact on the classification function than all other features, and thus they can be the best choice as bug predictors for the purpose of bug localization.

Clustering: Given U and P without labels, divide U into subsets (*clusters*) such that data points in each subset are similar w.r.t. certain distance measures [4]. In this paper, we apply clustering techniques to discover correlated predicates that are similar w.r.t. evaluation histories (Section 3.2).

2.2.2 Machine Learning Algorithms

To identify bug predictors (*i.e.*, to perform feature selection), we utilize two machine learning algorithms in this paper:

Support Vector Machines (SVMs): SVMs [9] are a family of machine learning algorithms. Briefly speaking, a SVM is a classification function (*e.g.*, a linear function $u \cdot \omega$, where ω is an adjustable parameter), and it iteratively adjusts the parameters to maximize classification accuracy while minimizing certain inevitable errors in machine learning.

Random Forests (RFs): An RF [7] is basically a collection of *decision trees* (DTs) [43]. Each decision tree is built on a different subset of U; it computes ranks for every feature and decides the class label of a data point u based on the ranks. The final label of u is then decided by the majority votes of all the trees in the forest.

In this paper, we use both SVMs and RFs. For large data sets, RFs are usually more efficient than others, while SVMs are usually more accurate, especially for small data sets.

3. BUG LOCALIZATION FRAMEWORK

In this section, we describe more details about the main components of our bug localization approach (as shown in Figure 3).

3.1 Feature Selection

Feature selection (*i.e.*, select critical program predicates that can be accurate bug predictors) is a fundamental step in our approach. In this paper, we apply classification algorithms to perform k-feature selection: we assign an *importance score* to each feature based on its impact on the classification function (*i.e.*, each predicate is associated with a score indicating how likely it may reveal bugs); then we choose the top-k features with the highest scores as the bug predictors. The following describes how we compute such scores based on SVMs and RFs.

In SVMs with a linear classification function $M(u) = u \cdot \omega$, where $\omega = \langle \omega_1, \ldots, \omega_n \rangle$, larger ω_i intuitively has bigger impact on the computed class. Thus, it is natural to define the i-th feature's importance score $S_M(p_i)$ as $|\omega_i|$ or ω_i^2 [25]. In our setting for bug localization, importance scores are slightly different because we need predicates that have not only the biggest impact but also the biggest *positive* impact (*i.e.*, causing an execution to fail). Therefore, we use ω_i directly as the importance score for the predicate p_i if we assign a positive value (typically 1) to the label for failure and a negative value (typically -1) to the label for success during the learning of $M(u)$. Generally, in SVMs with non-linear functions, the importance score for a predicate $S_M(p_i)$ can be defined as the partial derivative of $M(u)$ w.r.t. the i-th predicate:

$$S_M(p_i) \triangleq \frac{\partial M(u)}{\partial u_i}$$

RFs use another interesting heuristic to compute importance scores: if a predicate has a big impact on the classification function, then when the data values for this predicate change, the class label for this data point is very likely to change too; otherwise, the class label is likely to remain the same. After the classification function $M(u)$ is established, RFs randomly permute the data values for each predicate p_i and use $M(u)$ to classify the permuted data points. Then, the difference between the classification accuracies before and after the permutation can be used as the importance score for p_i. The larger the difference, the bigger impact of p_i on $M(u)$.

In addition, each individual machine learning algorithm is not a panacea for any learning problem. It would be better to "combine" results from different algorithms. So, we propose two simple strategies to "combine" different k-feature selection results.

Halving: Given two lists of ranked features of size$\geq \lfloor \frac{k}{2} \rfloor$, choose at least the top-$\lfloor \frac{k}{2} \rfloor$ features from each list.

Rank Mediation: Select the final top-k features according to the average ranks of all feature ranks from different algorithms.

For example, suppose a set of predicates $\langle p_1, \ldots, p_{10} \rangle$ is ranked by an SVM as $\langle 1, 2, 2, 4, 4, 4, 7, 8, 9, 9 \rangle$ and ranked by an RF as $\langle 5, 5, 4, 1, 2, 2, 7, 7, 7, 7 \rangle$. If we want top-5 predicates, a halving strategy would return $\langle p_1, p_2, p_3, p_4, p_5 \rangle$ as the selected predicates, while a rank mediation strategy would assign new ranks to the predicates as $\langle 3, 3.5, 3, 2.5, 3, 3, 7, 7.5, 8, 8 \rangle$ and return $\langle p_4, p_1, p_3, p_5, p_6 \rangle$ as the final five predicates.[2]

3.2 Clustering

Previous work has focused on discovering how different executions relate and how predicates and executions relate. Essentially, such work is based on a *vertical* view of evaluation profiles, *i.e.*, the profile for one execution is viewed as one unit for comparison. Little has been done to discover how predicates relate across different executions. This can be achieved with a new *horizontal* view of the profiles: the profiles for each predicate across all executions are viewed as one unit, and predicate correlations can be discovered by looking for "similar" horizontal units. For example, suppose we have three executions of a program with three instrumented predicates p_1, p_2, and p_3, and the three profiles are represented under the traditional vertical view as: $e_1 = \langle 5, 10, 5 \rangle$, $e_2 = \langle 2, 8, 2 \rangle$, and $e_3 = \langle 4, 6, 4 \rangle$. Under the horizontal view, the profiles are represented as: $p_1 = \langle 5, 2, 4 \rangle$, $p_2 = \langle 10, 8, 6 \rangle$, and $p_3 = \langle 5, 2, 4 \rangle$, and it becomes clear that p_1 has the same evaluation history as p_3 and this may indicate that p_1 has correlationship with p_3.

Predicate correlations that can be discovered based on the horizontal view are interesting because program failures may be caused by or influence many predicates in an execution and correlations among those predicates can provide additional contextual information for debugging. For example, let the branch conditions for $\texttt{if}^{(2)}$, $\texttt{if}^{(3)}$, and $\texttt{if}^{(5)}$ in Figure 2(d)-(f) be A, B, and C respectively, then the clustering result in Figure 2(c) can tell us that the predicates $A == true$, $B == false$, and $C == true$ had similar evaluation histories in failed executions. Such a correlation helped (1) reveal more information about the state of a program when it fails, (2) disclose a more accurate execution path the failure may take, and (3) provide additional contextual information for us to understand the failure.

We apply a variant of *k-means clustering* [4] to discover predicate correlations: all predicates are partitioned into clusters such that (1) the *distance* between any two predicates in the same cluster is less than a specified parameter ϵ, and (2) the distance between *mass centers* (the arithmetic average of all data points in a cluster) of any two clusters is larger than ϵ. The distance can be defined in many ways. We use a metric based on normalized Manhattan distance. Suppose under the horizontal view, two predicates p_1 and p_2 are characterized by $p_1 = \langle v_{11}, \ldots, v_{n1} \rangle$ and $p_2 = \langle v_{12}, \ldots, v_{n2} \rangle$, the distance between p_1 and p_2 is defined as:

$$D(p_1, p_2) \triangleq \sum_{i=1}^{n} \frac{|r_{i1} - r_{i2}|}{n}$$

where r_{i1} and r_{i2} are v_{i1} and v_{i2} linearly scaled to $[0, 1]$ respectively, *i.e.*,

$$r_i = \begin{cases} 0 & \text{if} \quad \max_j(v_j) = \min_j(v_j) = 0 \\ 1 & \text{if} \quad \max_j(v_j) = \min_j(v_j) \neq 0 \\ \frac{v_i - \min_j(v_j)}{\max_j(v_j) - \min_j(v_j)} & \text{otherwise} \end{cases}$$

3.3 Branch Prediction

In this and the following subsections, we present an algorithm to construct faulty control flow paths based on the predicates identified as bug predictors and their correlated predicates, aiming to provide more contextual information to help developers understand better *how* the predicates relate to bugs.

The basic idea is to use the predicates to guide the traversal of control flow graphs since the locations of the predicates tell us where the constructed paths should traverse. Also, we observed that much information about branch directions taken in failed runs can be inferred from the bug predictors and the execution profiles. Such knowledge thus helps to prune unlikely faulty paths during the traversal and make our algorithm efficient. This subsection presents the heuristics that we use to predict branch directions, and the following subsection will describe how to utilize these predictions to efficiently construct faulty control flow paths.

For each branch condition C, we have two profiled predicates C_T and C_F (*cf.* Section 2.1). We can decide the truth values of these predicates in *failed* runs as follows:

- The predicates identified by feature selection are the ones that are most likely related with failures, and so are the pred-

[2] It would be interesting to investigate how to "combine" different machine learning algorithms in general and what effects different "combinations" may have. The two simple strategies showed little impact on our evaluation results and we only show results based on the halving strategy in this paper.

icates identified by clustering. We let the truth values of these predicates to be true.

- The truth values of other predicates may also be heuristically decided by analyzing the horizontal views of the execution profiles. Suppose the horizontal view of a predicate P is projected to *failed* runs (*i.e.*, values from successful runs in the view are filtered out), and recall that each scalar value in the view is the number of times P is observed to be true. Then we may infer that the truth value of P was *never* true in failed runs if all values in the projected view are zeroes. Also, if most values ($> 50\%$) in the projected view are non-zeroes, then we decide the truth value of P is true; otherwise, we decide it to be unknown.[3]

Now we can predict which branch of C to take based on the truth values of C_T and C_F:

$$(C_T == false \wedge C_F == false) \Rightarrow C == \mathsf{neither} \quad (1)$$
$$(C_T == false \wedge C_F \neq false\ ) \Rightarrow C == \mathsf{false} \quad (2)$$
$$(\ C_T \neq false \wedge C_F == false) \Rightarrow C == \mathsf{true} \quad (3)$$
$$(\ C_T \neq false \wedge C_F \neq false\ ) \Rightarrow C == \mathsf{both} \quad (4)$$

Case (1): If both C_T and C_F are false, the condition C is likely not executed at all in failed runs, and all paths after C should be pruned during the CFG traversal.

Case (2): If C_T is false and C_F is true or unknown, the true branch of C is likely not taken in failed runs and should be pruned.

Case (3): If C_T is true or unknown and C_F is false, C's false branch is likely not taken in failed runs and should be pruned.

Case (4): If both C_T and C_F are true or unknown, then both branches may be taken and should be traversed.

3.4 Faulty Control Flow Path Construction

Based on the branch predictions, we can now easily traverse control flow graphs to greedily find paths that connect as many bug predictors as possible. Algorithm 1 describes how we traverse a CFG and find the paths we want. It is essentially a depth-first search except that we use heuristics to reduce backtracking used in the standard depth-first search algorithms and prune unlikely faulty paths. The algorithm chooses a bug predictor closest (in terms of the length of its shortest path) to the main entry of a program (line 2) and starts the traversal from the function containing this predicate (lines 4 and 5), then it repeatedly selects a next node to extend the path (lines 6–35) until there is no more next node for the path (lines 11, 26, 30, and 35). Such a process is repeated until all correlated predicates are covered (lines 1–36).

The main heuristic used in PATHGEN is to decide which branch to take during the CFG traversal (line 16). For each branch condition C, (1) if we predict neither branch should be visited, then we backtrack to the last visited branching node or start a new iteration to search for a new path; (2) if we predict that the false (true) branch should be pruned, then we take the true (false) branch to continue the traversal; (3) otherwise, we pick a random branch to follow; and (4) to avoid traversing a branch twice, we choose the other branch if this one has already been visited.

Our second heuristic is to make the constructed paths always begin at function entries and end at function exits to provide relatively complete paths. The two calls to shortestPath (lines 4 and 8) are added for this purpose (and may be disabled).

Algorithm 2 post-processes all the constructed faulty control flow paths, to remove unnecessary portions of the paths and order them for inspection. The step at line 3 in PATHPOST is used to prune

[3] One may exploit more information provided by the profiles, such as the total number of times a predicate is executed in one run, to further refine the cases.

Algorithm 1 Construct Faulty Control Flow Paths: PATHGEN

Input: $P = \{p_1, \ldots, p_n\}$ (a predicate cluster), G (a CFG)
Output: PATHS (a set of generated paths)
Notation: $p \in c$ (predicate p is contained in path c);
FC (a stack storing function calling contexts);
Flag $= \{f_1, \ldots, f_n\}$, and
$f_i = true$ iff p_i is contained in some path in PATHS
Initialization: PATHS $\leftarrow \emptyset$; $\forall f_i \in$ Flag $: f_i \leftarrow false$

```
1: repeat
2:     Pick p_i ∈ P with f_i = false and closest to main
3:     FC ← ∅; f_i ← true
4:     c ← shortestPath(entry of p_i's function, p_i)
5:     Traverse G, starting from the node for p_i:
6:     repeat  given the current node n, do:
7:         if all successors of n have been visited then
8:             c_0 ← shortestPath(n, exit of n's function)
9:             ∀p_j ∈ c_0 : f_j ← true
10:            PATHS ← {concatPath(c, c_0)} ∪ PATHS
11:            break
12:        else  /* visit n and find the next node */
13:            c ← concatPath(c, {n})
14:            ∀p_j ∈ n : f_j ← true
15:            if kindOf(n) = branch then
16:                Select a branch (Sec. 3.3)
17:            else if kindOf(n) = function-call then
18:                Push the current node n onto FC
19:                Let the next node be the entry of the callee
20:            else if kindOf(n) = function-exit then
21:                if ¬isEmpty(FC) then
22:                    Pop a node fn from FC
23:                    Let the next node be fn's successor
24:                else  /* finish all paths starting from p_i */
25:                    PATHS ← {c} ∪ PATHS
26:                    break
27:                end if
28:            else if kindOf(n) = program-exit then
29:                PATHS ← {c} ∪ PATHS
30:                break
31:            else
32:                Let the next node be n's successor
33:            end if
34:        end if
35:    until no more next node
36: until ∀i : f_i = true
```

likely irrelevant portions of the faulty control flow paths. Sometimes a path may pass through a function that contains no instrumented predicates, and the subpaths within such a function can be pruned to reduce inspection burden because the developers are unlikely interested in a function they do not instrument. However, it is also possible that a function does not have any instrumented predicate simply because it has been overlooked. Thus, when we cannot locate a bug in the pruned paths, this pruning step can be optionally disabled to present longer paths for inspection. Such an interactive feature helps to better balance the length of constructed paths and their information content for localizing bugs.

Our algorithm for constructing faulty control flow paths is efficient. For a fixed number of execution profiles, the branch prediction takes linear time w.r.t. the number of branch predicates. The CFG traversal takes worst case linear time w.r.t. the size of the CFG. In practice, the traversal is more efficient than linear because many branches can be pruned by the branch prediction.

Algorithm 2 Post-process Paths: PATHPOST

Input: $C = \{P_1, \ldots, P_m\}$ (predicate clusters), G (a CFG)
Output: PATHS in ascending order of path lengths
Initialization: PATHS $\leftarrow \emptyset$

1: $\forall P_i \in C$: PATHS $\leftarrow$ PATHS $\cup$ PATHGEN(P_i, G)
2: Prune duplicated portions of the paths in PATHS
3: Prune intra-procedural subpaths with no predicates
4: Sort all paths $p \in$ PATHS in ascending order of p's length

4. EMPIRICAL EVALUATION

In this section, we present evaluation results of our approach on different programs. The main question we hope to answer is how effective the generated faulty control flow paths by our approach can help localize bugs. Our own experience clearly indicates that these faulty control paths are more informative than isolated predicates because they provide useful contexts in understanding the bugs. We also performed a quantitative evaluation using concrete test cases and based on the following metrics: (1) how many bugs our approach can localize, and (2) how much manual code inspection is required to localize the bugs.

4.1 Experimental Setup

Subject Programs. We use the HR variants [24] of the Siemens test suite [26]. The suite contains 132 faulty versions of seven programs; each has hundreds of lines of code. Each program also has thousands of test cases and from zero to hundreds of failed runs. Some statistics on the source code can be found in Graves *et al.*'s study [19]. We instrumented the suite and collected its execution profiles using CBI [36].

CBI has also accumulated large data sets for many applications, including rhythmbox. These data sets have previously been used to discover interesting bugs [39]. Ben Liblit generously provided his collection of execution profiles for rhythmbox 0.6.4 to help us evaluate our approach.

Machine Learning Tools and Platforms. For experiments with SVMs, we used LIBSVM [11] on a Linux with a 2.4GHz Intel Xeon processor and 1GB of RAM. For experiments with RFs, we used an evaluation version of RandomForests [52] on a Windows XP with a 2GHz Intel P4-M processor and 512MB of RAM. Also, we implemented the variant of k-means clustering for predicate correlations (Section 3.2).

Data Preprocessing. The original data set for rhythmbox contains about 32000 executions with a total of 432335 instrumented predicates. Previous work [39] showed that hundreds to thousands of runs containing tens to hundreds of failed ones are often sufficient to find useful predicates as bug predictors. Thus, we randomly choose small subsets of the whole data set for our experiments. In addition, we separate each subset into three smaller subsets corresponding to the three kinds of instrumented predicates (branches, returns, and scalar-pairs, *cf.* Section 2.1) in order to assess the effectiveness of different kinds of predicates on bug localization.

Parameters for Machine Learning. Many factors may affect the accuracy of machine learning and are subject to change for different applications. *Trial-and-comparison* on small sample data sets is usually effective in choosing optimal parameters. We now present some parameters we used.

First, to account for *unbalanced* data (*i.e.,* the number of successful runs is much larger than the number of failed runs), we adjusted the weights for both successful and failed runs—the ratio of the two weights is the reverse ratio of the numbers of the two kinds of runs—so that profiles from successful runs will not overwhelm profiles from failed runs.

Second, we use 3-feature selection ($k = 3$) for Siemens programs (small programs) and 10-feature selection for rhythmbox (a large program). Previous work [41] suggested that the effectiveness of different k's may only differ slightly, especially when programmers are only willing to inspect less than 10% of the code and when our clustering strategy can provide additional correlated predicates.

Third, our clustering algorithm requires ϵ (*cf.* Section 3.2), which may lead to different correlation clusters. Smaller ϵ means more clusters and more "isolated" predicates; larger ϵ means more correlated predicates and possibly longer paths for inspection. Because we scale all profile data to the range of $[0, 1]$, it is intuitive to use $\epsilon = 0.01$ (1% of the range). In fact, trial-and-error showed that $\epsilon \in [0.005, 0.02]$ has only negligible effects on the resulting clusters in our experiments.

4.2 Evaluations On Siemens Test Suite

We first present a summarized bug localization results for the Siemens test suite based on branch predicates only. Interested readers can refer to our technical report [28] for more detailed results on the programs.

All constructed faulty control flow paths for the suite amount to about 4000 lines of code out of a total of about 43400 lines in the 132 versions of the seven programs. In total, 79 bugs were localized in the paths, meaning that programmers can discover 79 bugs by accumulatively examining about 9.2% of the code in all programs. Table 1 shows more quantitative measures on the effectiveness of our approach in terms of the number of bugs localized and the code inspection burden.[4] Our results show that a developer can discover 38 bugs by inspecting no more than 1% of the code in each version of the programs.

In terms of such quantitative measures, previous work localized less bugs for the suite within 1% code limit (*e.g.,* 17 for Tarantula [30] and 11 for SOBER [41]), while they localized more bugs within 20% code limit (*e.g.,* 75 for Tarantula and 96 for SOBER). However, several factors should be considered in such direct quantitative comparisons: (1) these existing techniques focus on finding only stand-alone bug predictors; (2) they use different instrumentation mechanisms (Tarantula instruments almost every statement in a program; SOBER uses a different implementation of CBI and instruments certain different program predicates); and (3) they computed the quantitative measures by a breadth-first search of the program dependency graph of a program, instead of following the faulty control flow paths as we did. We believe the fault control flow paths can provide more meaningful contextual information to help developers understand localized bugs than previous work. It would be interesting future work, however, to perform more systematic user studies.

4.3 Evaluations on Rhythmbox

As a brief summary, we are able to localize five bugs in rhythmbox 0.6.4, inspecting tens to hundreds of lines of code for each of the bugs, out of a total of 56484 lines of code [39]. Three of the bugs are due to similar causes as that for the bugs discovered by Liblit [34, 35]; two of the bugs were previously unknown.

Our experiments also show that branch predicates are more effective for bug localization than the other two kinds of predicates. This seems to agree with the hypothesis that many defects can be revealed by certain abnormal paths which are usually determined by branch conditions.

Our experiments were mainly performed with 1711 runs including 247 failed ones. Although the data set is small compared with

[4] In order to compute such quantitative measures, we have to omit certain subjective factors. So, we assume that a developer can determine whether a line of code has a bug whenever he sees it, as is assumed in other work [13, 41].

% code examined (for each version)	$\leq$ 1 line	$\leq$ 2 lines	$\leq$ 5 lines	$\leq$ 10 lines	$\leq$ 1%	$\leq$ 2%	$\leq$ 4%	$\leq$ 10%	$\leq$ 20%	Total % code examined in *all* versions: 3967 / 43433
# of Bugs Found	11	31	52	64	38	45	54	67	73	79 out of 132 bugs

Table 1: Summary of our results for the Siemens test suite. Each column shows how many bugs can be discovered by inspecting up to so much code in each version of the programs in the suite. **Each line of code is counted as inspected if (1) it is contained in the faulty control flow paths, and (2) it is located before the actual bug in the paths or there is no bug localized in the paths.**

#	branches	returns	scalar-pairs
1	`global_gconf_client==NULL`	`g_ptr_array_free>0`	`i==2`
2	`i<impl_array->len`	`g_type_check_instance_cast>0`	`...` (several predicates of form `i=constant`
3	`monkey_media_is_alive()==FALSE`	`monkey_media_is_alive>0`	`alive==0`
4	`selected_entry!=view->priv->selected_entry`	`g_strdup>0`	`...`
5	`!player->priv->url`	`rb_entry_view_get_entry_contained>0`	`global_gconf_client==0`
6	`rb_entry_view_get_entry_contained()`	`rhythmdb_query_model_entry_to_iter>0`	`...`
7	`g_threads_got_initialized`	`rb_entry_view_get_playing_entry>0`	`data->shell->priv->play_queued<1287`
8	`gdk_threads_mutex`	`rb_source_get_entry_view>0`	`cc>cc`
9	`view->priv->change_sig_queued`	`g_utf8_validate>0`	`...`
10	`monkey_media_player_get_uri`	`monkey_media_player_get_uri>0`	`changed==callback_runs`

Table 2: Top 10 failure-related predicates identified by LIBSVM for rhythmbox.

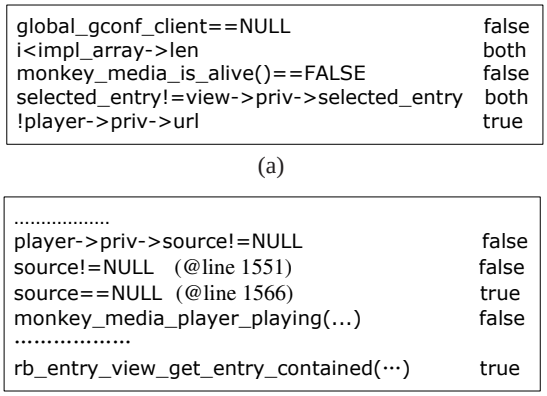

Figure 4: Sample predicate clusters for rhythmbox.

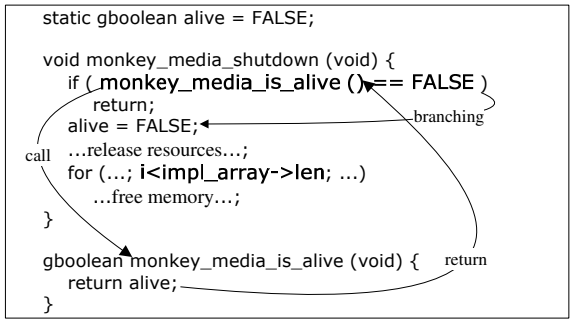

Figure 5: Sample faulty control flow path for the predicate cluster in Figure 4(a).

the whole data set, it is already helpful for identifying many bug-related predicates. Table 2 shows the the three kinds of top 10 predicates identified by LIBSVM as bug predictors. The subsequent predicate clustering and faulty control flow path construction show that these predicates provide insightful information for debugging.

4.3.1 Sample Faulty Control Flow Paths

Sample predicate clusters and faulty control flow paths are shown in Figures 4 and 5. In the following, we explain more about how they helped discover bugs:

- Figure 4(a) shows a cluster containing branch predicates 1, 2, 3, 4, and 5 from Table 2. Also, the heuristic branch prediction for each condition is shown, which can be used to direct our CFG traversal. Several faulty control flow paths are constructed for the cluster using Algorithm 1. Figure 5 shows one of the paths. A simple inspection of the path tells us that there is a race on the global variable `alive`: different threads could call `monkey_media_shutdown` at the same time and cause rhythmbox to crash on exit.
- Figure 4(b) shows (partially) a larger cluster containing 36 predicates, including the 6-th branch predicate in Table 2. It involves 20 functions from eight files. Because the code heavily uses function pointers which we do not yet model in Algorithm 1, the constructed faulty control flow paths for the cluster were mainly restricted within procedures. Even so, they still provided hints at bugs: the path segment between line 1551 and 1566 in `rb-shell-player.c` implies that a null `source` could be operated on by a callback function connected to the `monkey media player` in rhythmbox 0.6.4 through `g_signal_connect` provided by GLib signal system, due to races similar to the ones discovered by Liblit *et al.* [34, 35]. Newer versions of rhythmbox no longer use the `monkey media player` to play music; its threading mechanism has also been rewritten for better reliability. All of these five bugs we localized have thus been eliminated.

4.3.2 Confirming the Bugs

As an additional part of our evaluation, we constructed concrete test cases to confirm that the cases we localized are actual bugs. It is difficult to construct tests directly for the original rhythmbox code. Instead, we use the constructed faulty control flow paths with identified bug predictors to simplify the code first. We treat functions containing a bug predictor as top-level functions for testing, and automatically prune unrelated branches and functions along the paths. Then, we generate simple definitions for library and external (undefined) functions. Finally, we use a specialized test driver generator for multi-threaded programs to perform random testing on the top-level functions for exhibiting their (mis)behavior. The localized bugs are all manifested as segmentation faults in this way.

4.3.3 Bug Localization Cost

We also measured the time cost of our experiments on rhythmbox with LIBSVM. Table 3 shows the results. The time in the table includes all "machine" time, excluding the time spent on code inspection. We were new to rhythmbox, GTK+, and GNOME related programming, and it took us minutes to hours to follow each constructed faulty control flow path and look for potential bugs. During path inspection, we also manually performed simple data-flow analyses to help us understand the code behavior; in particular, we performed alias analysis for function pointers to help link "segmented" paths together. As interesting future work, these anal-

Instrumentation Predicates	Number of Predicates	Time (minutes) of Feature Selection	Clustering	Path-Gen	Classification Accuracy
branches	6,863	41	30	< 1	99%
returns	25,287	45	770	< 1	99%
scalar-pairs	400,185	654	n/a	n/a	96%

Table 3: Performance of LIBSVM-based experiments on rhythmbox 0.6.4.

#	Branches
1	`g_threads_got_initialized`
2	`children`
3	`gdk_threads_mutex`
4	`requisition->height>child_requisition.height`
5	`size != -1`
6	`monkey_media_is_alive()==FALSE`
7	`global_gconf_client==NULL`
8	`(child->widget->flags&256U)!=0`
9	`i<impl_array->len`
10	`rb_entry_view_get_entry_contained()`

Table 4: Top 10 predicates identified by RandomForests as bug predictors on rhythmbox 0.6.4.

yses can be automated and help improve the effectiveness of our context-aware approach. We believe that the cost on constructing and inspecting faulty control flow paths is worthwhile, compared with many hours spent on traditional testing, especially for a real-world large-scale application that we are not familiar with and that contains unknown bugs. In addition, together with Table 2, Table 3 again implies that branches are more effective for bug localization than other kinds of predicates.

4.3.4 Experiments with RFs

We have also experimented with RandomForests on rhythmbox. The experiments were done with a smaller data set including 219 successful and 87 failed runs since the evaluation version of RandomForests we used only handles small data sets. Table 4 shows the top 10 branch predicates identified by RandomForests as bug predictors. Despite of the smaller data set used, many useful bug predictors were still identified, more than half of which overlap with the predicates identified by LIBSVM. This fact further supports previous experiences [39] that hundreds to thousands of runs including tens to hundreds of failed ones are often sufficient to find useful bug predictors.

4.4 Discussions

Herein, we further discuss some aspects of our approach.

4.4.1 Benefits of Predicate Clustering

An alternative for constructing faulty control flow paths is to use only those predicates identified by feature selection, excluding the clustered predicates. Compared with this alternative, our clustering strategy can provide additional information for debugging: (1) predicates are separated into different clusters—different clusters may indicate different bugs in programs, while predicates in a same cluster are more likely related to the same bug; (2) additional low-rank predicates can be included for path construction and help produce more informative faulty control flow paths.

The code in Figure 2 illustrated that clustering helps link the code shown in Figures 2(a) and 2(b) with the code in Figure 2(c), and thus helps identify the bugs related to the same data field. There are also 14 cases in the Siemens test suite that can only be localized with the additional clustered predicates in our approach [28].

Lal *et al.* [32] also present an algorithm for constructing a *shortest* control flow path that contains the maximum number of some given bug predictors. However, they only used stand-alone predicates. We believe our clustering strategy can also help improve the effectiveness of their algorithm on bug localization.

4.4.2 Threats to Validity

From the evaluation, we also notice that our approach cannot effectively localize all bugs. There are several factors impacting the effectiveness of our approach, besides the machine learning parameters and the parameter specifying how long a path (*i.e.*, the amount of code) that the developer is willing to manually inspect.

The most crucial one is what kind of and how many predicates should be instrumented. As we have mentioned, different kinds of predicates have different effects on bug localization. For example, branch predicates are not good at localizing bugs related to data definition errors (*e.g.*, a variable is assigned a wrong constant in Version 7 of `tcas` in the Siemens suite). Also, previous work using different instrumentation achieved different bug localization results. Choosing the most appropriate and adequate predicates for bug localization remains a fundamental, interesting research problem. Certain coverage criteria used to generate adequate test cases [12] may be applicable to predicate instrumentation as well. Also, program analysis and filtering techniques, such as [33, 44] may help to reduce irrelevant predicates and decrease instrumentation overhead but retain effectiveness on bug localization.

The second factor concerns the nature of different types of bugs. Certain program locations are not directly instrumentable (*e.g.*, the erroneous macro definition in Version 36 of `tcas` in the Siemens suite); certain bugs involve many locations through implicit data flows (*e.g.*, callback functions heavily used in rhythmbox). Such bugs cannot be understood if only explicit control flows are presented. It would require incorporating certain data-flow analysis and alias analysis into path construction to improve the effectiveness of our approach. In practice, there may also be many types of bugs requiring special localization techniques. This paper did not investigate into these special techniques; it will also be an interesting and challenging research topic for our future work.

The third factor concerns the adequacy of the data set, and particularly the number of failed runs which may affect the results of feature selection. When there are not enough failed runs (*e.g.*, Version 8 of `tcas` in the Siemens suite), our approach cannot identify bug predictors or construct faulty control flow paths. However, we believe that it is not a major concern for an infrastructure such as CBI, since in its real deployment, data can be easily gathered from large number of users.

5. RELATED WORK

In this section, we survey additional research related to automated debugging. We roughly classify the related work into the following categories:

Program-behavior clustering. It is useful to cluster similar program runs together for tasks such as failure identification and test-case filtering. Haran *et al.* [23] applied random forests to predict whether a run succeeds or fails. Dickinson *et al.* [14] utilized several cluster filtering strategies to group similar runs. Podgurski *et al.* [48] classified and prioritized program failure reports for diagnosis using pattern classification and multivariate visualization. Bowring *et al.* [5, 6] classified execution profiles with iterative, agglomerative hierarchical clustering, and the profiles are unconventionally represented as Markov models of event and value transitions. Liu *et al.* [40] regarded profiles of two failed runs as similar if they suggest roughly the same bug locations. Similar to us, they all expect that differences between failed and successful runs are good indications of bugs. However, these techniques are mainly for selecting appropriate failed test cases or profiles for further analysis, while ours is to select bug predictors and construct faulty control flow paths from the profiles.

Bug-predictor identification. Identifying bug predictors helps discover actual bugs, just like symptoms help diagnose diseases. In

general, bug predictors can be anything, including program states, execution counts of statements, branches, functions, event transitions, etc. Brun and Ernst [8] identified bug-revealing program properties by applying decision trees and support vector machines on execution profiles of different versions of a program. They used Daikon, a dynamic invariant detection tool [16], to detect properties, instead of direct instrumentation; they also assumed one of the versions is free of bugs and used it to determine whether the runs of other versions failed or not. Jones *et al.* [30, 31] ranked statements in a program based on coverage information of each statement in failed and successful runs. Both they and Orso *et al.* [45] utilized coloring schemes to visualize all statements and highlight "dangerous" ones to assist code inspection. They instrumented almost all statements in a program and provided hierarchical, interactive views of source code, while we use a different instrumentation and provide additional contextual information besides identified bug predictors. Renieris *et al.* [49] looked for a succeeded run that is most similar to a given failed run from a large set of profiles, and flagged the difference between the two profiles as bug predictors. Their definition of similarity among profiles is similar to ours for clustering, but for different purposes: they look for the nearest neighbor of a failed run (based on the vertical view), and we look for neighbors of predicates to discover how they relate (based on the horizontal view). More recently, Arumuga Nainar *et al.* [2] used combinations of simple, atomic predicates as bug predictors and showed informative bug localization results. Also, Jones *et al.* [29] proposed a parallel debugging technique and methodology that enables multiple developers to simultaneously debug multiple bugs in the same program. Our approach, the machine learning module and the fault control flow path construction in particular, may also be extended for such compound predicates and multiple existence for better effectiveness on bug localization.

Feature-correlation discovery. Branch prediction in computer architecture has explored predicate correlations for instruction-level optimization [47, 55]. Within the context of dynamic detection of program invariants, Dodoo *et al.* [15] introduced several strategies, including clustering, to select representative predicates for detecting implications, and applied the predicate implications to improve the performance of a theorem prover and separate faulty from correct executions of erroneous programs. Liblit *et al.* [39] also considered predicate correlations based on execution profiles, instead of actual executions, but treated such correlated predicates as logically redundant and eliminated them for just feature selection purpose, while we utilize such correlated predicates to efficiently construct context-aware faulty control flow paths.

If we view a *slice* as a set of program elements related to a certain behavior, many studies on slicing-based debugging can be classified as correlation discovery. Agrawal *et al.* [1] assumed the differences (called *dices*) between the slices of a failed and a successful run may contain bugs and visualized the dices to aid debugging. Pan *et al.* [46] suggested a family of heuristics based on program slices to reduce the burden of code inspection. Gyimóthy *et al.* [22] augmented dynamic slices with potentially bug-relevant statements. Wong *et al.* [54] defined execution slices as features and defined distances among the slices based on statement counts and then looked for useful features. Gupta *et al.* [21] used the intersection of forward and backward dynamic slices started from failure points for debugging. Zhang *et al.* [58] empirically evaluated the effectiveness of different dynamic slicing techniques on bug localization. To the best of our knowledge, no slicing algorithm has utilized the information provided by identified bug predictors, and it will be interesting to investigate how to use the bug predictors to "guide" slicing and improve the accuracy of slices.

Bug-revealing path generation. Such work aims to find understandable paths in programs, either static flows or dynamic traces, that may reveal bugs. If we view program slices as paths, the aforementioned slicing techniques can also be classified into this category. Also, Reps *et al.* [50] highlighted paths in a program that may lead to abnormal behaviors by identifying divergences between failed and successful runs. Gotlieb *et al.* [18] generated test cases which pass through particular program points by solving constraint systems that correspond to the program. Liblit *et al.* [37] proposed a family of analyses to build time lines of possible program actions that lead to failures based on certain information such as failure points, stack traces and event logs. Delta debugging [13, 56, 57] also provided automated ways to simplify failure-inducing inputs; it locates not only failure-related states, but also *cause transitions*—moments when failure causes are transited from one relevant variable to another. It needs detailed program states and executes the program itself. Ball *et al.* [3] found possible bug causes by exploiting the differences between correct and faulty traces generated by model checkers. Manevich *et al.* [42] used postmortem symbolic evaluation to produce a set of execution traces along which the program may be driven to one *given* failure point. More recently, Jhala *et al.* [27] used path slicing to reduce counterexamples from model checkers. Groce *et al.* [20] reduced a program to a smaller one which can produce executions consistent with a given (partial) trace of events. Lal *et al.* [32] constructed a shortest control flow path that contains the maximum number of given predicates. Their algorithm is based on weighted push-down systems and utilizes data dependencies to prune infeasible paths; it runs in linear time w.r.t. program sizes but exponential w.r.t. the number of given predicates. Also, symbolic and concrete executions are being combined in in-house testing phase, including DART [17], CUTE [53], and EXE [10], to generate test cases that can effectively drive programs along particular erroneous paths.

6. CONCLUSIONS

In this paper, we have presented a novel context-aware approach for bug localization, which not only identifies accurate bug predictors but also constructs faulty control flow paths. We imposed two different views on execution profiles and combined several machine learning algorithms to accurately identify bug predictors and discover predicate correlations. Also, we developed an efficient algorithm based on branch prediction and control flow graph traversal to construct faulty control flow paths that connect bug predictors and provide more contextual information for revealing actual bugs. We have evaluated our approach on the Siemens test suite and rhythmbox within the CBI instrumentation infrastructure. Evaluation results showed that our approach is able to localize more bugs than previous techniques with less code inspection burden; and more importantly, it can provide informative control flow paths to help understand and debug the code. We believe this is a promising direction for bug localization and an important step towards realizing automated debugging.

Acknowledgments

We are grateful to Ben Liblit for the CBI infrastructure and the data. We also thank Alex Aiken, Earl Barr, Christian Bird, Mark Gabel, Matthew Roper, Gary Wassermann, and all anonymous reviewers for valuable feedback on earlier drafts of this paper.

7. REFERENCES

[1] H. Agrawal, J. R. Horgan, S. London, and W. E. Wong. Fault localization using execution slices and dataflow tests. In *International Symposium on Software Reliability Engineering (ISSRE)*, pages 143–151, 1995.

[2] P. Arumuga Nainar, T. Chen, J. Rosin, and B. Liblit. Statistical debugging using compound boolean predicates. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 5–15, 2007.

[3] T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: localizing errors in counterexample traces. In *Symposium on Principles of Programming Languages (POPL)*, pages 97–105, 2003.

[4] P. Berkhin. Survey of clustering data mining techniques. Technical report, Accrue Software, San Jose, CA, 2002.

[5] J. F. Bowring, M. J. Harrold, and J. M. Rehg. Improving the classification of software behaviors using ensembles. Technical report, Georgia Institute of Technology, 2005.

[6] J. F. Bowring, J. M. Rehg, and M. J. Harrold. Active learning for automatic classification of software behavior. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 195–205, 2004.

[7] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, Oct. 2001.

[8] Y. Brun and M. D. Ernst. Finding latent code errors via machine learning over program executions. In *ICSE*, pages 480–490, 2004.

[9] C. J. C. Burges. A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery*, 2(2):121–167, 1998.

[10] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. In *Conference on Computer and Communications Security (CCS)*, 2006.

[11] C.-C. Chang and C.-J. Lin. *LIBSVM: a library for support vector machines*. `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[12] L. A. Clarke, A. Podgurski, D. J. Richardson, and S. J. Zeil. A formal evaluation of data flow path selection criteria. *Trans. on Software Engineering (TSE)*, 15(11):1318–1332, 1989.

[13] H. Cleve and A. Zeller. Locating causes of program failures. In *ICSE*, pages 342–351, 2005.

[14] W. Dickinson, D. Leon, and A. Podgurski. Finding failures by cluster analysis of execution profiles. In *ICSE*, pages 339–348, 2001.

[15] N. Dodoo, L. Lin, and M. D. Ernst. Selecting, refining, and evaluating predicates for program analysis. Technical Report MIT-LCS-TR-914, MIT Laboratory for Computer Science, Cambridge, MA, July 21, 2003.

[16] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *ICSE*, pages 449–458, 2000.

[17] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 213–223, 2005.

[18] A. Gotlieb, B. Botella, and M. Rueher. Automatic test data generation using constraint solving techniques. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 53–62, 1998.

[19] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. *Trans. on Software Engineering and Methodology (TOSEM)*, 10(2):184–208, 2001.

[20] A. Groce and R. Joshi. Exploiting traces in program analysis. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 3920 of *Lecture Notes in Computer Science (LNCS)*, pages 379–393. Springer, 2006.

[21] N. Gupta, H. He, X. Zhang, and R. Gupta. Locating faulty code using failure-inducing chops. In *ASE*, pages 263–272, 2005.

[22] T. Gyimóthy, Á. Beszédes, and I. Forgács. An efficient relevant slicing method for debugging. In *joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 303–321, 1999.

[23] M. Haran, A. F. Karr, A. Orso, A. A. Porter, and A. P. Sanil. Applying classification techniques to remotely-collected program execution data. In *joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 146–155, 2005.

[24] M. J. Harrold and G. Rothermel. *Aristotle Analysis System – Siemens Programs, HR Variants*. http://www.cc.gatech.edu/aristotle/Tools/subjects/.

[25] M. Heiler, D. Cremers, and C. Schnörr. Efficient feature subset selection for support vector machines. Technical Report 21/2001, Computer Science Series, University of Mannheim.

[26] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *ICSE*, pages 191–200, 1994.

[27] R. Jhala and R. Majumdar. Path slicing. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 38–47, 2005.

[28] L. Jiang and Z. Su. Automatic isolation of cause-effect chains with machine learning. Technical Report CSE-2005-32, UC Davis, 2005.

[29] J. A. Jones, J. F. Bowring, and M. J. Harrold. Debugging in parallel. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 16–26, 2007.

[30] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *ASE*, pages 273–282, 2005.

[31] J. A. Jones, M. J. Harrold, and J. T. Stasko. Visualization of test information to assist fault localization. In *ICSE*, pages 467–477, 2002.

[32] A. Lal, J. Lim, M. Polishchuk, and B. Liblit. Path optimization in programs and its application to debugging. In *European Symposium on Programming (ESOP)*, volume 3924 of *Lecture Notes in Computer Science (LNCS)*, pages 246–263. SpringerESE, 2006.

[33] D. Leon, W. Masri, and A. Podgurski. An empirical evaluation of test case filtering techniques based on exercising complex information flows. In *ICSE*, pages 412–421, 2005.

[34] B. Liblit. *Bug 137460: dangling timeout event source ID causes crashes*. `http://bugzilla.gnome.org/show_bug.cgi?id=137460`.

[35] B. Liblit. *Bug 137834: dangling RBShellPlayer callbacks cause crashes*. `http://bugzilla.gnome.org/show_bug.cgi?id=137834`.

[36] B. Liblit. *The Cooperative Bug Isolation Project*. `http://www.cs.wisc.edu/cbi/`.

[37] B. Liblit and A. Aiken. Building a better backtrace: Techniques for postmortem program analysis. Technical Report CSD-02-1203, UC Berkeley, 2002.

[38] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 141–154, 2003.

[39] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 15–26, 2005.

[40] C. Liu and J. Han. Failure proximity: A fault localization-based approach. In *Symposium on Foundations of Software Engineering (FSE)*, pages 46–56, 2006.

[41] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. SOBER: statistical model-based bug localization. In *joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 286–295, 2005.

[42] R. Manevich, M. Sridharan, S. Adams, M. Das, and Z. Yang. PSE: Explaining program failures via postmortem static analysis. In *Symposium on Foundations of Software Engineering (FSE)*, pages 46–56, 2004.

[43] T. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[44] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy software. *Trans. on Programming Languages and Systems (TOPLAS)*, 27(3):477–526, 2005.

[45] A. Orso, J. A. Jones, and M. J. Harrold. Visualization of program-execution data for deployed software. In *Symposium on Software Visualization (SOFTVIS)*, pages 67–76, 2003.

[46] H. Pan and E. H. Spafford. Heuristics for automatic localization of software faults. Technical Report SERC-TR-116-P, Purdue University, 1992.

[47] S.-T. Pan, K. So, and J. T. Rahmeh. Improving the accuracy of dynamic branch prediction using branch correlation. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 76–84, 1992.

[48] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang. Automated support for classifying software failure reports. In *ICSE*, pages 465–477, 2003.

[49] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *ASE*, 2003.

[50] T. W. Reps, T. Ball, M. Das, and J. R. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 432–449, 1997.

[51] Rhythmbox. Music management application for GNOME. `http://www.rhythmbox.org`.

[52] Salford Systems Inc. *RandomForests*^{TM}. `http://www.salford-systems.com/`.

[53] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 263–272. ACM, 2005.

[54] W. E. Wong, S. S. Gokhale, and J. R. Horgan. Measuring distance between program features. In *International Conference on Computer Software and Applications (COMPSAC)*, pages 307–312, 2002.

[55] C. Young and M. D. Smith. Static correlated branch prediction. *Trans. on Programming Languages and Systems (TOPLAS)*, 21(5):1028–1075, 1999.

[56] A. Zeller. Isolating cause-effect chains from computer programs. In *Symposium on Foundations of Software Engineering (FSE)*, pages 1–10, 2002.

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

[58] X. Zhang, H. He, N. Gupta, and R. Gupta. Experimental evaluation of using dynamic slices for fault location. In *Automated and Algorithmic Debugging (AADEBUG)*, pages 33–42, 2005.

[59] A. X. Zheng, M. I. Jordan, B. Liblit, and A. Aiken. Statistical debugging of sampled programs. In *Neural Information Processing Systems (NIPS)*. 2003.