Disqover: Debugging via Code Sequence Covers

Ethar Elsaka and Atif Memon
Department of Computer Science
University of Maryland
College Park, MD, USA
Email: {ethar, atif}@cs.umd.edu

***Abstract*—Automated model-based test generation has seen an undeniable trend towards obtaining large numbers of test cases. However, the full benefits of this trend have not yet percolated to downstream activities, such as debugging. We present *Disqover* for automated software debugging based on code sequence covers that leverages execution traces, or alternatively, sequence covers of large numbers of failing test cases to quickly identify causes of test failures, thereby aiding debugging. We develop a new algorithm that efficiently extracts commonalities between sequence covers in the form of ordered subsequences and values of variables contained in these subsequences that contribute to each failure. The results of our experimental evaluation suggest that users of Disqover need only 30% of the time needed to identify faults compared to the baseline in a user study. Furthermore, we show that the number of inspected statements using our approach is smaller than that of other state-of-the-art systems by *multiple orders of magnitude*. Additionally, we show that increasing the number and diversity of test cases improves our results by further decreasing the length of output subsequences to be examined.**

I. Introduction

Software debugging is one of the main activities in the software development process. It is used extensively by software developers to localize faults. Manual debugging is by far the most popular, but difficult and time consuming approach [25]. In order for the developer to manually debug an application that contains an error, first she has to understand the way the application works then determine the root cause of the error by backtracking, navigating through the code dependencies, possibly running the code multiple times, and parsing the program logs in order to collect clues about the reasons of the error, so that the developer can finally identify the source of the error and fix it. The need to understand program functionality has become central to debugging, as there are many programmers who participate in the development phase. A developer who works on fixing a specific bug may not necessarily have written the code, and thus, has to understand unfamiliar program parts. This task takes a considerable amount of effort and time [21]. Even after the developer becomes familiar with the code, figuring out the statements in the code that that produce the error is non-trivial. The developer has to envision multiple scenarios (by exploring different possibilities of the input space) to check all the potentially error-causing execution paths. There has been some work on automating this step in the literature [13]. The final step, which is determining the source of the error (*fault localization*) is the hardest aspect of debugging [21] because it requires analyzing hundreds of lines of code to determine the root cause of the error.

Although to date software debugging remains largely manual, it is not the case with software testing. Advances in software test generation have yielded new techniques that can automatically produce a large numbers of test cases, and execute them to validate the correctness of the program. There are different paradigms in the literature upon which automatic test case generation techniques are based, such as behavioral and interactional UML models [10], event flow graphs [19], event interaction graphs [28], feature models [24], and mathematical models [12].

In this paper, we leverage the execution results of such large numbers of test cases to aid in the downstream activity of *debugging*, i.e., to help the developer find the source of an error. We analyze the sequences of statements executed by failing test cases (sequence covers [1]), and output a subsequence of statements (with dynamically-observed values of variables) that is common between them. The developer uses our output to find the exact root cause of the error. Finding error-causing code using our output in the form of code subsequences is easier and more convenient for the developer than inspecting the code itself. Firstly, code subsequences can be examined in linear order. Developers do not need to track code dependencies and consider different ways the code can be executed. Second, code subsequences are derived from execution traces, and hence capture runtime information as well, such as the code execution order, and variable values at runtime. This allows developers to relate the variable values to the error.

Finding commonalities between code sequence covers corresponds to the multiple common subsequences problem, which is NP-hard [26]. This problem is also related to the multiple sequence alignment problem that is well studied in the computational biology literature [17], [9]. Most approaches for finding multiple common subsequences focus only on finding the longest common subsequence, and hence are not applicable to our case as the common subsequence of statements that contains the faulty path is not necessarily the longest. Furthermore, approaches for multiple sequence alignment are mostly iterative, i.e., they only align one sequence at a time with the current set of sequences, and hence are dependent on the evaluation order, and do not necessarily result in optimal alignment. Additionally, multiple sequence alignment allows mismatches as a compromise to get longer subsequences, which is not allowed in our case. Therefore, we propose a new efficient algorithm for finding common subsequences between code sequence covers, and propose a new way to represent those common subsequences as a directed acyclic graph, known as the common subsequences graph. Furthermore, we propose various abstraction techniques to make the input code

[1] In the rest of the paper, we use *execution traces* and *sequence covers* alternatively.

978-1-5090-0066-5/15/$31.00 ©2015 IEEE

sequence covers more concise and developer-friendly. In our experimental evaluation, we show that using our algorithm, along with the abstraction techniques, we efficiently construct the common subsequences graph in less than a second for multiple sequences with average length of 10,000 statements.

II. Related Work

Weiser [27] propose *program slicing* as a method for aiding debugging. Program slicing identifies all the statements that can affect a variable in a program either statically [27], or dynamically [31]. Dynamic slicing and its variations [11], [14], [32] potentially reduces the size of the slices. However, the slices are fairly large making them undesirable for debugging [30]. Furthermore, the developer still has to run the reduced program (slice) again and manually detect the source of the error. On the contrary, our approach outputs a subsequence of statements (and variable values) covered by the failed test cases, through which the developer can backtrace to quickly find the root cause.

Two notable techniques (*regression containment* [22] and *Delta debugging* [29]) rely on working and non-working versions of the program under test. Regression containment isolates changes that cause the error between the working and non-working (that are not passing the test) program versions. *Delta debugging* [29] iteratively binary-partitions the set of differences between the working and non-working versions until it obtains the minimal failure-inducing set. Other variations of Delta Debugging (DD) have been proposed to overcome its limitations such as *Hierarchical Delta Debugging (HDD)* [20] and *Iterative Delta Debugging (IDD)*[7]. As can be imagined, finding working and non-working versions of the program can be challenging. Also, extracting the changes between two program versions and applying parts of these changes to the working version can be very time consuming because of the execution time required to run multiple combinations of these changes. The runs cannot be performed in parallel because a run at one iteration depends on the output of the run at the previous iteration. Finally, applying part of the changes to the working version may not always result in an executable version. Conversely, our approach just relies on the current version of the program. Also, it does not involve changing the source code or generating different versions of the program.

Several other automated debugging techniques use *statement coverage* [16], their frequencies [33], *predicate values* [18], *program states* [30], and combinations of program elements [8]. The problem with these techniques is that the number of statements they identify for inspection can be very large. Furthermore, they do not output *subsequences* of statements, only sets of statements, which the developer has to inspect and reason about individually.

III. Modeling Disqover

We present our approach in this section. We begin by stating some basic definitions, then discuss how we efficiently extract commonalities between test case execution traces in detail in the following subsections.

Definition: (Test case) Given a software S, a test case is a set of inputs $i_1, i_2, \ldots i_n$ that satisfy a set of preconditions, along with a set of expected outputs $o_1, o_2, \ldots o_m$ that satisfy a set of postconditions. When $i_1, i_2, \ldots i_n$ are given to software S, S should produce $o_1, o_2, \ldots o_m$ in order for the test case to pass.

Disqover is based on the existence of a large number of test cases. Running a number of test cases causes some of them to pass (produce a correct output) and the others to fail (produce a wrong output, throw an error or crash).

Definition: (Passing/failing test case) Given a software S and a test case t, t passes if S runs t to completion correctly, producing the expected output, and t fails if t causes S to throw an error or produce an unexpected output during the execution of t.

In our approach, we group the failing test cases that fail for the same reason (i.e. that throw the same type of error or the unexpected output from the same statement) together under the same *test cases group*. Test case groups enable debugging applications that have multiple errors at the same time.

Definition: (Test cases group) A test cases group is a set that contains one or more test cases that fail at the same location, producing the same type of error or unexpected output.

Furthermore, we define two types of statements that are essential for such type of automated debugging, *failure statement*, and *root cause subsequence*.

Definition: (Failure statement) is the statement where the unexpected output is detected. The failure can also take the form of an application error.

We note that neither the failure statement nor its function call stack trace are necessarily responsible for the unexpected output, and hence, the need for identifying the root cause becomes apparent, which is the bulk of the software debugging process, and the objective of Disqover.

Definition: (Root cause subsequence) is a subsequence of statements that is the main reason for the unexpected output. This subsequence may consist of a single or multiple statements. Fixing this subsequence prevents the unexpected output from being produced.

Since running a test case causes specific parts of the source code to be executed, if a test case fails, the root cause of the failure must be in the statements that are executed during that run. Furthermore, in the vast majority of scenarios, when test cases fail at the same location on the same error or unexpected output, this indicates that their execution traces share the root cause as well.[2] Therefore, by increasing the number of test cases which fail for the same reason (i.e., from the test case group), the subsequence possibly responsible for the error is narrowed down, by eliminating irrelevant statements that are not shared between the execution traces of all the test cases.

A straightforward way for implementing the above observation is by finding a simple set intersection of the statements shared by test cases in a group (code coverage intersection). This approach is inadequate, as it returns an unordered set of statement with no relationship between them.

[2]Although this is true for the vast majority of errors, simple extensions to our techniques can relax this assumption.

Therefore, the basic idea of our approach is to extract the common subsequence among the failing test cases sequence covers. *The root cause can be found in this common subsequence, and can be reached by tracing back from the failure statement.* Using sequence covers as opposed to code coverage intersection has a number of advantages. Tracing the common subsequence back starting from the failure statement makes the debugging process as simple as a linear scan, as opposed to exploring the highly interconnected program dependency graph to trace back an application error. Furthermore, exploiting the fact that the program statements execute in sequence can reduce the number of statements reported, because in this case, we will not only consider the statements that are just shared between the sequences, but also consider that these statements must be executed in the same order. The existence of this additional restriction further decreases the number of the resulting statements that the developer needs to consider at a time.

Below we present a motivating application for using sequence covers for automated debugging, as opposed to using code coverage intersection, but first, we define both terms.

Definition: (Test case code coverage $C(t)$) Given a test case t, the test case code coverage $C(t)$ is a set of statements that are excuted during the execution of t.

Definition: (Test case sequence cover $S(t)$) Given a test case t, a sequence cover, $S(t)$, is the *ordered list* of statements that are executed during the execution of ***t*** according to their execution order.

A. Motivating Example

Consider the code snippet listed in Figure 1, which has the statements s_1, s_2, s_3, s_4 and s_5 (we exclude the *if* and the *for* statements from the sequence for simplicity). It can also execute two test cases t_1 and t_2, which are from the same test case group. t_1 executes the statements s_1, s_3, s_4, s_5 in the following order $s_1 \rightarrow s_3 \rightarrow s_4 \rightarrow s_5$ and t_2 executes the statements s_2, s_3, s_4, s_5 in the following order $s_2 \rightarrow s_5 \rightarrow s_3 \rightarrow s_4$. The two test cases execute each statement only once. In this case, the code coverage set $C(t_1)$ is $\{s_1, s_3, s_4, s_5\}$ and the code coverage set $C(t_2)$ is $\{s_2, s_3, s_4, s_5\}$. Therefore, the statements that result from applying the code-coverage intersection technique are (s_3, s_4, s_5). On the other hand, if we utilize the order of statement execution, we can tell that either the subsequence s_3, s_4 or the subsequence s_5 is responsible of the error because in the first test case s_5 appears before s_3, s_4 and in the second it appears after them. Therefore, s_5 can be inspected in isolation of s_3 and s_4 by the developer, which minimizes the number of statements to consider at a time, and minimizes the number of interactions and dependencies that the developer needs to keep track of while tracing back the statements. In this case, we generate the execution trace of each test case as a sequence of statements, and get the common *ordered* statements between all the test cases.

In the previous example, the sequence cover $S(t_1)$ contains the ordered list (s_1, s_3, s_4, s_5) and the sequence cover $S(t_2)$ contains the ordered list (s_2, s_5, s_3, s_4). Using our sequence coverage intersection technique, the statements that result from applying it are (s_3, s_4) *or* (s_5), i.e., each subsequence has a smaller number of statements than those in the unordered list of code coverage intersection.

```
if (t1) s1;
if (t2) s2;
for (i in 0,1) {
  if(t1 && i=0 || t2 && i=1) {
   s3; s4; }
  if(t1 && i=1 || t2 && i=0) s5;
}
```

Fig. 1: Example program

$S(t_1)$ = a b a c
$S(t_2)$ = c a b a
$S(t_3)$ = a b d a

Fig. 2: Sequence covers of three test cases

B. Common Subsequence Coverage Algorithm

In this section, we discuss our algorithm for finding code sequence coverage intersection in detail. The goal of our algorithm is to detect subsequences of statements that appear in all the test cases in the same order, and at the same time, not necessarily consecutively, i.e., they can have arbitrary gaps between them. For example, assuming we obtain the three execution traces in Figure 2, we want the algorithm to detect that the subsequence (a, b, a) is the one that is common between them. Applying the Longest Common Subsequences, LCS, algorithm is not suitable in our case as it outputs the longest common subsequence only, which may not contain the root cause of the error, as it is just one of the possible subsequences among all the common subsequences. In this section we discuss our approach for finding all the common subsequences, and in Section III-C, we show how to rank the subsequences according to their importance so that we output the subsequence with the highest rank according to that criteria.

To enumerate all the possible common subsequences between a set of sequences (potentially a large number of them, since we may have the output of the execution of a large number of test cases over large programs), we follow the steps outlined below.

1) Applying Code Coverage Intersection: As an initial step, we intersect all the code coverage sets of the test cases to get the set of statements that are common between them. Clearly, the common subsequences must be composed of statements in that intersection only. We denote this as $C = C(t_1) \cap C(t_2)...$

2) Excluding Initialization Code: To minimize the number of statements in the intersection, we remove the initialization code, which is the code responsible of starting up the application. Since the initialization code is shared by all the test cases, the intersection of their code coverage will contain these initialization statements, which are not necessarily related to the error. To help reducing the code coverage intersection size, we assume that the initialization code is bug-free, and calculate the intersection between the test cases by excluding the initialization code. $C = C - C_0$, where C_0, is the application initialization code.

3) Constructing the Common Subsequences Graph: The problem of generating all common subsequences among a set of sequences is difficult because there is an exponential number of combinations that can be considered in order to construct the common subsequence. If a statement appears multiple times in each sequence, say $n_1, \ldots, n_m$ times, then there are $O(\prod_i n_i)$ ways to construct smaller subsequences recursively out of the original ones to continue finding the common subsequences among them and so on. In this subsection, we discuss how to model that problem using our *common subsequences graph*, and how to compute the the common subsequences efficiently by only considering meaningful combinations, because not all of the possible combinations can make it to the final common subsequences.

Since each statement can occur multiple times in each sequence cover, we define a particular combination of occurrences of a statement in all sequence covers to be an *instance* of that statement as it can possibly contribute to a common subsequence. For example, in Figure 2, b has only one possible instance of occurrence: $(2, 3, 2)$, which means that b occurs at position 2 in $S(t_1)$, position 3 at $S(t_2)$, and position 2 at $S(t_3)$. However, a has *eight* possible instances, since it occurs in $S(t_1)$ at positions $1, 3$, in $S(t_2)$ at positions $2, 4$, and in $S(t_3)$ at positions $1, 4$. Therefore, a's possible combinations are $(1, 2, 1), (1, 2, 4), (1, 4, 1), \ldots$ etc.

Now that we have defined instances of occurrences for each statement, a common subsequence is a sequence of instances ($inst_1$, $inst_2$, .., $inst_n$) such that all positions in $inst_i$ are *strictly less than* their corresponding positions in $inst_{i+1}$, for all $1 \leq i < n$.

Definition: (Operator $<$) Given two instances $inst_i$ and $inst_j$, $inst_i < inst_j$ if and only if all the positions in $inst_i$ are less than their corresponding positions in $inst_j$.

Likewise, we define $>$ over pairs of instances, $inst_i$ and $inst_j$ using their corresponding positions.

Example: consider the instance of a's occurrence $inst_1 = (1, 2, 4)$ and the instance of b's occurrence $inst_2 = (2, 3, 2)$. A common subsequence *cannot* consist of $inst_1$ followed by $inst_2$, because $inst_1 \not< inst_2$, because at the third place, a occurs at position 4 while b occurs at position 2, which means that a precedes b in all the test case sequences, but not in the third, where b precedes a, which means that $(inst_1, inst_2)$ is not a valid common subsequence. On the other hand, if we consider $inst_1$ as the instance $(1, 2, 1)$, then $(inst_1, inst_2)$ becomes a valid common subsequence, because $inst_1 < inst_2$, where for every position in $inst_1$, its corresponding position in $inst_2$ is strictly greater than it, which means that a precedes b in all test cases.

The naive way for generating the common subsequences using the instances is by generating all possible instances $(inst_1, inst_2, \ldots)$ for all statements and finding which of them follows the others, i.e., $inst_i < inst_j$. To make this process more scalable, we propose an algorithm that generates the instances *on demand*, and *avoids constructing redundant subsequences* during the common subsequence building time. The algorithm is based on constructing a graph of instances, where nodes of the graph represent instances, and an edge from instance $inst_i$ to $inst_j$ means that $inst_i > inst_j$ *and* there is no other $inst_k$ such that $inst_i > inst_k$ and $inst_k > inst_j$, i.e., there is no intermediate instance that can appear in the common subsequence between $inst_i$ and $inst_j$, and hence, edges of the graph are constructed between nodes that represent instances that *directly* follow each other.

In order to generate the instances on demand, we only create the least instance for each statement in the code coverage intersection, generate its edges, and recurse. For example, considering the sequence coverage in Figure 2, we start by the least instance for $a : (1, 2, 1)$, and the least instance for $b : (2, 3, 2)$ and add them to a stack. We then pick $(1, 2, 1)$ from the stack, generate its edges by choosing from the *next least possible* instances relative to it, and add those next least possible instances back to the stack if they do not already exist or if they have not been already processed. To generate the next least possible instances efficiently, we use binary search by constructing an array $pos[s, t_i]$ storing the positions of each statement s in each sequence cover of t_i in sorted order. Given an instance $(p_1, \ldots, p_m)$ of a statement s', we find the next least position to p_i in the sequence of t_i by searching for p_i in that sequence. We continue consuming nodes from the stack until it becomes empty, the point at which we have generated a precedence graph on the instances, where any path in that graph represents a common subsequence between the execution traces of all test cases.

Definition: (Common subsequences graph) a common subsequences graph is a directed acyclic graph whose nodes represent instances of occurrence of statements in the sequence cover of all test cases, and its edges represent the direct $<$ relationship between those instances. Any path in this graph represents a common subsequence of the execution traces of all test cases.

4) Extracting common subsequences: To generate the common subsequences between the execution traces of all test cases, we start from the node representing the failure statement in the common subsequences graph, and traverse its neighbors, generating all possible paths. Each of these paths is a common subsequence.

C. Algorithm Optimizations

We enhance our algorithm by 1) abstracting the test cases, and 2) extracting the most important subsequences only. Test case abstractions transform the sequence covers to more abstract, shorter versions. Most important common subsequence extraction selects a subsequence from the common subsequences graph that is most likely to contain the root cause.

1) Test case abstraction: We achieve test case abstraction using two techniques: loop-based abstraction, and block-based abstraction. In loop-based abstraction, we compress each loop in each test case sequence to appear as one iteration. We identify loops as any subsequence of program statements in the execution trace that is consecutively repeated more than once. In block-based abstraction, we compress program statements always appear consecutively in all test case sequences and substitute them with one node in the common subsequence graph. We use a variant of the suffix tree algorithm to discover those patterns and substitute them.

2) Extracting the most important common subsequences: Reporting all possible common subsequences to the developers

App	LOC	# Classes	# Methods
ArgoUML	152513	1787	13117
Crossword Sage	3072	34	238
Buddi	20922	257	1580
Freemind	7702	136	788

TABLE I: Application code complexity metrics

Fault	Event	Exception	Seeded or real?
ArgoUML	Export All Graphics	FileNotFoundException	real
Crosswordsage1	Load crossword To Edit	NullPointerException	seeded
Crosswordsage2	New crosswords	NumberFormatException	seeded
Buddi	Save As	FileNotFoundException	real
Freemind1	Save As	FileNotFoundException	real
Freemind2	Open	FileNotFoundException	real
Freemind3	Remove Node	NullPointerException	seeded

TABLE II: Application faults

may be impractical, especially, if one subsequence can point out where the root cause of the error is. Furthermore, traversing all the paths starting from the failure statement node in large graphs is time consuming and results in a large number of paths, while we are only interested in just one sequence to present to the user, which is likely to contain the trace back from the failure statement to the root cause. Therefore, in our algorithm instead of traversing all paths, we assign scores to the nodes in the graph according to their degrees,[3] which indicate the likelihood of those nodes participating in faulty sequences, and then generate the path that passes along the nodes with the highest scores.

IV. Experimental Evaluation

To evaluate Disqover, we perform a set of experiments that answer the following research questions:

RQ1 Does Disqover help developers find root causes of failures more effectively?

RQ2 Does diversifying the input test cases or increasing their number reduces the number of statements in the common subsequence?

RQ3 Do our algorithms lead to a more efficient evaluation of the common subsequence?

A. Subject Applications and Faults

We evaluate Disqover using 4 open source Java GUI applications: ArgoUML [1], Crossword Sage[3], Buddi [2], and Freemind [4]. Table I lists some code complexity metrics of the subject applications such as the number of lines of code (LOC) in each application, the number of classes, and the number of methods. As we can see, the number of lines of code of those applications vary from thousands of lines of code (e.g., Crossword Sage) to hundreds of thousands of lines of code (e.g., ArgoUML).

We use both seeded and real faults to evaluate our approach. We summarize the faults used with each of the applications in Table III. Although all the faults are exception-type faults, the same approach can be used for programs producing unexpected results as long as failing test cases are available for those unexpected results.

We implement a tool as an Eclipse plugin to enable developers to use our approach. We use GUITAR [23] tools to automatically generate the test cases and replay them. We use Cobertura [6] code coverage tool to obtain the code coverage of each test case. We modify it so that it outputs code sequence covers instead of statement coverage reports. We use the *Java Debugger (JDB)* [5] command line debugging tool to automate extracting the variable values.

[3]We study other metrics in future work.

B. Experiments

In this subsection, we describe the experiments performed to evaluate Disqover. We perform experiments to measure the impact of using our approach by real developers to debug the faults in the subject applications, in terms of both the saving in debugging time and the number of statements to examine before finding the root cause. We also perform experiments to measure the number of statements to examine by developers before reaching the root cause in comparison to other techniques such as Tarantula [16] and Fonly [33]. Moreover, we measure the effect of choosing diverse input test cases on the size of the output common subsequence. Furthermore, we evaluate our abstraction techniques effect on the length of our input sequence covers, and evaluate the effect of the number of test cases over both the running time and the length of resulting common subsequence, comparing our approach to multiple baselines. We show that our approach significantly reduces the amount of time needed to discover the source of the fault, and we also show the effectiveness of our sequence cover abstraction techniques, for reducing the computation time and the length of the output common subsequences, especially, for the computationally intensive ones.

1) User Study: We address RQ1 in this experiment by evaluating our techniques and tool under real-world usage scenarios. We present the software bugs to human developers that have no prior knowledge of the software and ask them to identify the root cause. We worked with a group of six competent developers, all pursuing PhD degrees in Computer Science and have solid background in Software Engineering. Three developers were asked to locate the root cause using our tool only, and the other three were asked to use whichever technique they were comfortable with, including command line debugging or Eclipse tools. The assignment was performed randomly. Each developer was asked to identify the root cause of the three faults: ArgoUML (FileNotFoundException), CrossWord Sage (NullPointerException and NumberFormatException). We measured the time from when the developer started the debugging process until he/she identified the root cause. The developers who used our tool took an average of 2:30, 1:26, 1:14 minutes to perform the tasks, respectively. They accurately identified the root causes, and one of them needed to check the source code once, while the others did not check the source code at all. Furthermore, our developers found the search tool very useful in finding relevant statements to the statements they wanted to trace back from. They thought the tool helped them a lot finding the bug source quickly, and the variable values gave them hints about which variables to focus their backtracking process on. The developers needed to

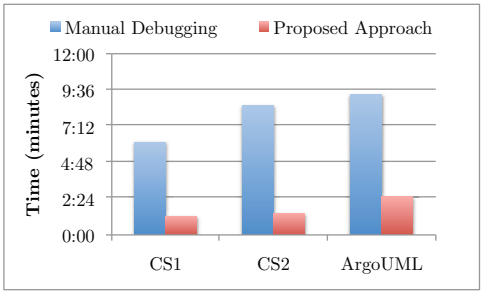

Fig. 3: Developer debugging time experiment

examine 12, 11, and 10 statements only in the subsequence to find the root cause of the faults above, respectively. On the other hand, developers who did not use our tool either identified the source incorrectly, or took longer time to identify it. For example, one developer just used the output exception stack trace to identify the root cause, and only chose from the lines that were referred to by the stack trace. That approach is not correct, because the root cause of the problems does not necessarily exist in the exception stack trace. That approach led to incorrectly identifying two out of the three faults by that developer. Disregarding the incorrect results, the developers not using our tool took the following times on average to complete the tasks: 9:18, 8:36, and 6:07 minutes, respectively. All the results are shown in Figure 3. As we can see, the ratios between the times taken by the developers using our approach to the these of the developers using the other approach are 27%, 17%, and 20% for the three applications, respectively. Therefore, on average, our approach saved developers 79% of their debugging time in this experiment.

2) Comparison with other approaches: In this experiment, we address RQ1 by comparing Disqover with two statement ranking techniques, Tarantula [16] and Fonly [33]. We choose Tarantula because Jones et al. [15] show that Tarantula outperforms many other ranking techniques in fault localization. Furthermore, we choose Fonly, as it is the only technique that uses failed test cases only in fault localization like our technique. Both techniques rank the program statements according to the their suspiciousness of being the root cause using a scoring formula that assigns a score to each statement.

To compare our system to both systems, we use a metric that quantifies the "number of inspected statements" until the source is found. For our system, this metric is simply the number of statements that a developer traces back in order to identify the root cause starting with the failure statement. For both Tarantula and Fonly, this metric is defined as the number of statements whose score is greater than or equal to the score of the root cause statement. To express an average case, instead of counting all the statements whose score is equal to the score of the root cause statement if many of them share the same score, we just count half of them (to express the expectation of inspecting the root cause statement if statements with equal score are randomly ordered). We note that even with this type of comparison, our system still has an advantage, which is that the statements being inspected are not disconnected, or parts of unrelated methods or classes. They actually form a sequence as one statement leads to the other, and helps the developer understand the execution sequence that leads to the error, while with the other two approaches, the developer will probably

Fault	No. of Failed TC	No. of Passed TC
ArgoUML	22	100
Crosswordsage1	261	86
Crosswordsage2	321	45
Buddi	143	108
Freemind1	338	300
Freemind2	337	300
Freemind3	417	300

TABLE III: Application faults

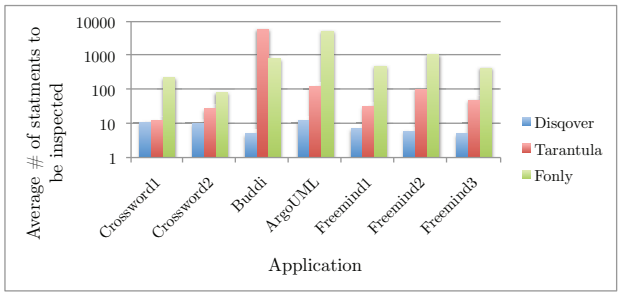

Fig. 4: Comparison with Tarantula and Fonly

have to carry out the task of understanding the sequence causing the bug of *each* suggested statement on his/her own. Therefore, our system produces many statements that explain an individual root cause, while other systems produce many statements that are missing their explanation.

In this experiment, we use all of the 7 bugs discussed in Section IV-A. We implement the formulas of both techniques and obtain the scores. Table III shows the number of passing and failing test cases we used with every application. Results are presented in Figure 4 in log scale. We find that our system leads to a significantly smaller number of statements. For example, on average the number of statements that need to be inspected by our approach is 112 *times* smaller than Tarantula's number of statements to be inspected, and 147 *times* smaller than Fonly's number of statements to be inspected. On the other hand, in the case of Buddi bug, our system needs to inspect a number of statements that is 1250 *times* shorter than Tarantula, and in ArgoUML bug is 4291 *times* shorter than Fonly. Finally, we use a significantly smaller number of test cases than those reported in [33] and [15].

3) Test case diversity experiment: In this experiment, we address RQ2 by studying the effect of the diversity of the input test cases on the size of the output common subsequence. As expected, the more diverse the input test cases are, the smaller the size of the output subsequence is. To capture this type of performance, we compare two approaches for selecting the input test cases. The first approach selects sufficiently diverse subset of test cases among the set of all input test cases, and the other approach selects a random subset. To measure diversity between two test cases, we use the size of the intersection of their code covers. The smaller the number of the intersection, the more diverse the two test cases are. Therefore, to select a diverse subset of test cases of size n, we start with the the two test cases with the highest diversity according to the definition above, and incrementally add one test case that will maximize the diversity, until we have n test cases. Although this approach is greedy and may lead to a local optima, we adopt it because of its efficiency. An

observation that we saw during implementing this experiment is that many attempts of running random caused an out of memory exception to occur, and took a very long time to evaluate the common subsequence before finally timing out (after hours of letting it run). To enable the comparison, we choose an example attempt of running the random approach that did not cause an out of memory exception and did not time out, and show its results in our figures. The results of our comparison are shown in Figures 5 (a), (b), (c) for the faults of CrossWord Sage (NumberFormatException), Buddi, and ArgoUML, respectively. We vary the number of test cases, and measure the output common subsequence length for both random and diverse selections. As we can see, for the same size of input test cases, the test cases that are more diverse lead to a shorter common subsequence size than that resulting from the random selection approach. Note that the last point in each figure has the same value for both approaches because we use the same set of test cases as input to both approaches. Also, as we can see from Figure 5 (b), Buddi with the random selection approach always times out after using 5 test cases.

4) Sequence Cover Length Experiments: In this experiment, we address RQ3 by evaluating the effect of our sequence cover abstraction techniques over the average size of sequence cover. We use four faults. For each fault, we evaluate the length of the sequence covers in the following cases 1) no initialization code removal and no sequence abstraction techniques, 2) with removing the initialization code but without applying any of our abstraction techniques, 3) with removing the initialization code and applying block-based abstraction only, 4) with removing the initialization code and applying loop-based abstraction only, and 5) with removing initialization code and applying both abstraction techniques. The results of this experiment are shown in Figure 6, where we plot the average length of the sequence cover using each approach for each fault on log scale. As we can see, removing the initialization code results in a significant reduction of the average sequence cover length relative to the original length, averaging a length that is 20% of the original average sequence cover length. After applying the loop-based abstraction, the average length drops to 3% of the original length, which is significantly lower than the reduction ratio without that abstraction technique, illustrating the benefit of that approach. On the other hand, both removing initialization code and block-based abstraction only lead to 4% average length, which is slightly higher than removing initialization code and applying loop-based abstraction only, but still has a significant effect. The overall length after applying all techniques together is 2.7% of the original length. The reason is that when applying both loop-based and block-based abstraction together, the effect of block-based abstraction is not as high as if it is applied by itself, but as we can see, applying them together is still beneficial. That effect is more obvious when considering the running time of the algorithm using both abstraction techniques, as we discuss in Section IV-B5.

5) Running Time Experiments: In these experiments, we address RQ3 by evaluating the effect of the number of test cases on the algorithm running time using Disqover, and a number of baselines. We compare the running time using our approach to the running time using 1) block-based abstraction only, 2) loop-based abstraction only, and 3) none of the abstraction techniques. We remove the initialization code

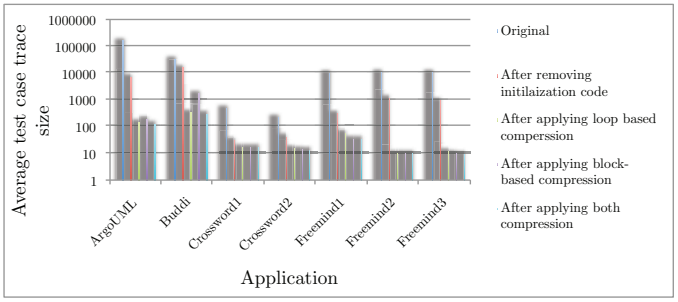

Fig. 6: Sequence Cover Length Experiments

in all cases. We also compared against the naive approach for constructing the common subsequences graph which we discussed in Section III-B. However, we omit the results of that approach from the discussion, as it does not scale, and causes out of memory exceptions in all cases. The results are shown in Figures 5 (d), (e), (f), for the faults: Crossword Sage (NumberFormatException), ArgoUML (FileNotFoundException), Buddi (FileNotFoundException), respectively. As we can see, the baselines outperform our approach only in the case of Crossword Sage, because the length of sequence covers is already very small. Therefore, the overhead introduced by applying the abstraction techniques does not lead to much overall computation reduction over the case without abstraction. However, in the other two cases, ArgoUML and Buddi, our approach evaluates the common subsequences in much less time than the baselines, especially in the case of Buddi, where the average sequence cover length is very high, the advantages of our approach are much more obvious. The highest running time using our approach is 1.4 minutes (from Buddi), while all other approaches could only run for one or two data points, and broke our timeout limit which is 5 minutes for these experiments in all other cases. Just for the purpose of illustration, we removed the timeout constraint on the data point with 5 test cases in the case of Buddi using block-based abstraction only, and the common subsequence evaluation took 19 minutes, which is **2456 times** slower than our optimized approach. Another observation is related to the relationship between the number of test cases and the running time. As expected, the running time increases with the increase of the number of test cases, with our approach being the most stable to increasing the number of test cases, which shows that our abstraction approaches play an important role in keeping our approach scalable.

V. Conclusions

In this paper we introduced Disqover, a new method for automated software debugging via sequence covers. Disqover exploits automated test case generation tools to generate a large number of test cases, and implements a novel algorithm to find commonalities between the failing test cases by automatically replaying them, and extracting their sequence covers, i.e., execution traces. We propose an efficient algorithm for finding the common subsequences between multiple sequence covers, and use that algorithm to present a suggested subsequence to the developer for each error separately, which he/she can trace back to find the root cause of a fault. Our experimental evaluation shows the effectiveness of our approach in terms of minimizing the developer's debugging time and minimizing the common subsequences algorithm output size and running

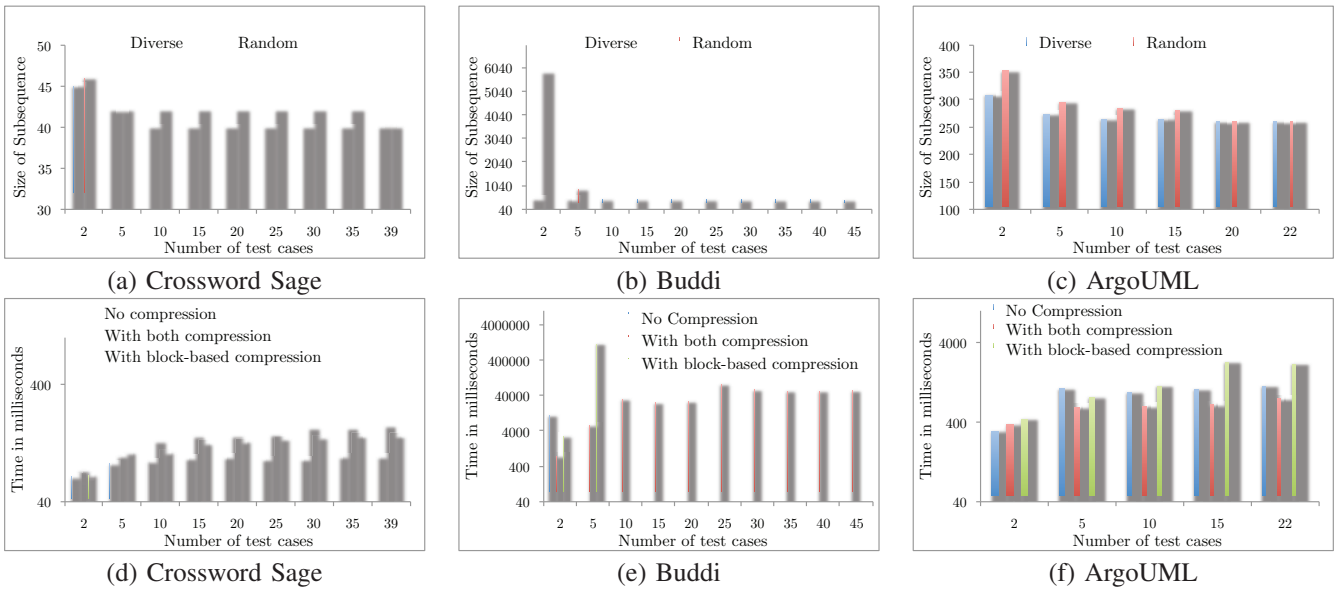

Fig. 5: (a-c) Test case diversity experiments and (d-f) Running time experiments

time. Finally, we will extend our approach to accommodate other ranking functions for the common subsequences than the function maximizing the sum of the node degrees in the common subsequences graph.

REFERENCES

[1] ArgoUML. http://argouml.tigris.org/.

[2] Buddi. http://buddi.digitalcave.ca/.

[3] Crossword Sage. http://sourceforge.net/projects/crosswordsage/.

[4] FreeMind. http://freemind.sourceforge.net/wiki/index.php/Main_Page.

[5] jdb - The Java Debugger. http://docs.oracle.com/javase/7/docs/technotes/tools/windows/jdb.html, 1993.

[6] Cobertura (A code coverage utility for Java). http://cobertura.github.io/cobertura/, 2001.

[7] C. Artho. Iterative delta debugging. *STTT*, 13(3):223–246, 2011.

[8] R. A. Assi and W. Masri. Identifying failure-correlated dependence chains. In *Fourth International IEEE Conference on Software Testing, Verification and Validation, ICST 2012, Berlin, Germany, 21-25 March, 2011, Workshop Proceedings*, pages 607–616, 2011.

[9] M. Brudno and B. Morgenstern. Fast and sensitive alignment of large genomic sequences. In *CSB*, pages 138–, 2002.

[10] M. Chen, X. Qiu, and X. Li. Automatic test case generation for uml activity diagrams. In *AST*, pages 2–8, 2006.

[11] R. A. DeMillo, H. Pan, and E. H. Spafford. Critical slicing for software fault localization. In *ISSTA*, pages 121–134, 1996.

[12] S. Gnesi, D. Latella, M. Massink, V. Moruzzi, and I. Pisa. Formal test-case generation for uml statecharts. In *Proc. 9th IEEE Int. Conf. on Engineering of Complex Computer Systems*, pages 75–84. IEEE Computer Society, 2004.

[13] P. Godefroid, J. de Halleux, A. V. Nori, S. K. Rajamani, W. Schulte, N. Tillmann, and M. Y. Levin. Automating software testing using program analysis. *IEEE Software*, 25(5):30–37, 2008.

[14] T. Gyimothy, A. Beszedes, and I. Forgacs. An efficient relevant slicing method for debugging. In *ESEC / SIGSOFT FSE*, pages 303–321, 1999.

[15] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), November 7-11, 2005, Long Beach, CA, USA*, pages 273–282, 2005.

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

[17] C. Lee, C. Grasso, and M. F. Sharlow. Multiple sequence alignment using partial order graphs. *Bioinformatics*, 18(3):452–464, 2002.

[18] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *PLDI*, pages 15–26, 2005.

[19] A. M. Memon. An event-flow model of gui-based applications for testing. *Software Testing, Verification and Reliability*, 17(3):137–157, 2007.

[20] G. Misherghi and Z. Su. Hdd: hierarchical delta debugging. In *ICSE*, pages 142–151, 2006.

[21] G. J. Myers. *The art of software testing (2. ed.)*. Wiley, 2004.

[22] B. Ness and V. Ngo. Regression containment through source change isolation. In *COMPSAC '97: Proceedings of the 21st International Computer Software and Applications Conference*, pages 616–621, Washington, DC, USA, 1997. IEEE Computer Society.

[23] B. Nguyen, B. Robbins, I. Banerjee, and A. Memon. Guitar: an innovative tool for automated testing of gui-driven software. *Automated Software Engineering*, pages 1–41, 2013.

[24] G. Perrouin, S. Sen, J. Klein, B. Baudry, and Y. L. Traon. Automated and scalable t-wise test case generation strategies for software product lines. In *ICST*, pages 459–468, 2010.

[25] I. Vessey. Expertise in debugging computer programs: A process analysis. *International Journal of Man-Machine Studies*, 23(5):459–494, 1985.

[26] Q. Wang, D. Korkin, and Y. Shang. A fast multiple longest common subsequence (mlcs) algorithm. *IEEE Trans. Knowl. Data Eng.*, 23(3):321–334, 2011.

[27] M. Weiser. Program slicing. *IEEE Trans. Software Eng.*, 10(4):352–357, 1984.

[28] X. Yuan and A. M. Memon. Generating event sequence-based test cases using gui runtime state feedback. *IEEE Trans. Softw. Eng.*, 36(1):81–95, Jan. 2010.

[29] A. Zeller. Yesterday, my program worked. today, it does not. Why? In *ESEC / SIGSOFT FSE*, pages 253–267, 1999.

[30] A. Zeller. Isolating cause-effect chains from computer programs. In *SIGSOFT FSE*, pages 1–10, 2002.

[31] X. Zhang, N. Gupta, and R. Gupta. Pruning dynamic slices with confidence. In *PLDI*, pages 169–180, 2006.

[32] X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. In *ICSE*, pages 319–329, 2003.

[33] Z. Zhang, W. K. Chan, and T. H. Tse. Fault localization based only on failed runs. *IEEE Computer*, 45(6):64–71, 2012.