ABSTRACT

Title of dissertation: AUTOMATED SOFTWARE DEBUGGING USING HYBRID STATIC/DYNAMIC ANALYSIS

Ethar Elsaka, Doctor of Philosophy, 2016

Dissertation directed by: Professor Atif Memon
Department of Computer Science

With the increasing complexity of today's software, the software development process is becoming highly time and resource consuming. The increasing number of software configurations, input parameters, usage scenarios, supporting platforms, external dependencies, and versions plays an important role in expanding the costs of maintaining and repairing unforeseeable software faults. To repair software faults, developers spend considerable time in identifying the scenarios leading to those faults and root-causing the problems.

While software debugging remains largely manual, it is not the case with software testing and verification. The goal of this research is to improve the software development process in general, and software debugging process in particular, by devising techniques and methods for automated software debugging, which leverage the advances in automatic test case generation and replay.

In this research, novel algorithms are devised to discover faulty execution paths in programs by utilizing already existing software test cases, which can be either automatically or manually generated. The execution traces, or alternatively, the sequence covers of the failing test cases are extracted. Afterwards, commonalities between these test case sequence covers are extracted, processed, analyzed, and then presented to the developers in the form of subsequences that may be causing the fault. The hypothesis is that code sequences that are shared between a number of faulty test cases for the same reason resemble the faulty execution path, and hence, the search space for the faulty execution path can be narrowed down by using a large number of test cases.

To achieve this goal, an efficient algorithm is implemented for finding common subsequences among a set of code sequence covers. Optimization techniques are devised to generate shorter and more logical sequence covers, and to select subsequences with high likelihood of containing the root cause among the set of all possible common subsequences. A hybrid static/dynamic analysis approach is designed to trace back the common subsequences from the end to the root cause.

A debugging tool is created to enable developers to use the approach, and integrate it with an existing Integrated Development Environment. The tool is also integrated with the environment's program editors so that developers can benefit from both the tool suggestions, and their source code counterparts.

Finally, a comparison between the developed approach and the state-of-the-art techniques shows that developers need only to inspect a small number of lines in order to find the root cause of the fault. Furthermore, experimental evaluation shows that the algorithm optimizations lead to better results in terms of both the algorithm running time and the output subsequence length.

AUTOMATED SOFTWARE DEBUGGING USING HYBRID STATIC/DYNAMIC ANALYSIS

by

Ethar Elsaka

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2016

Acknowledgments

I owe my gratitude to all the people who have made this thesis possible and because of whom my graduate experience has been one that I will cherish forever.

First and foremost I'd like to thank my advisor, Professor Atif Memon for giving me an invaluable opportunity to work on challenging and extremely interesting projects over the past five years. He has always helped and gave a lot of valuable advice. He always has believed in me and my ability to face any challenges to complete my thesis.

Words cannot express the gratitude I owe my husband, Walaa Eldin Moustafa, for his support and confidence in my ability to be a mom and a PhD student at the same time. I owe my deepest thanks to my family, my mother Soher Gaber and father Ibrahim Elsaka, who have always stood by me and guided me through my career, and have pulled me through against impossible odds at times. I would also want to thank my brothers, Islam, Ayman, and Atheer, who always pushed me forward by their supporting words and advices. I would also like to thank my kids Yaseen and Maryam for being such loving kids, and for giving me the energy to carry on. I hope that I see their names on their own theses.

I deeply appreciate the support and care of my father in law Moustafa Mohamed. I'm very grateful for all the time that he spent with my kids to give me the opportunity to finish my thesis.

I can't forget all my lovely friends for their support and their standing by me at all times, especially Gannat, Omar, Ingy, Moustafa, Hend, Hossam, Ahmed

Khalil and Mohamed Salem.

Lastly, thank you all and thank God!

Table of Contents

List of Figures

Chapter 1: Introduction

Software debugging is a main activity in the software development process. It is used extensively by software developers to localize faults, find sources of errors and enhance software quality and performance in general. The most popular way to localize faults is by manual debugging, which is hard and time consuming [1]. In order for the developer to manually debug an application that contains an error, she has to first understand the way the application works and determine the root cause of the error by backtracking, navigating through the code dependencies, and 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 the program functionality is very common, as there are many programmers who participate in the development phase. Therefore, the 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 [2]. Even after the developer becomes familiar with the code, figuring out the line(s) of code that produces the error is also a non-trivial task. 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 in automating this step in the literature of software testing [3–9].

The final step, which is determining the source of the error (*fault localization*) is the hardest aspect of debugging [2] because it requires analyzing hundreds of lines to determine the error-causing subset. The developer has to track the program dependencies in the source code, and go through multiple dependency paths to know which are the ones that are exercised by the failing scenario.

Although software debugging remains largely manual, it is not the case with software testing. With recent advances in automatic software test case generation, new approaches use automatically generated test cases to facilitate software testing and detecting software bugs. There are different paradigms in the literature upon which automatic test case generation techniques are based. Some techniques are based on behavioral and interactional UML models [10–26]. Other techniques are based on structure UML models [27–30]. Also, there are some techniques that are based on other models such as Event Flow Graph model [31], Event Interaction Graph model [32], Feature model [33] and the Mathematical model [34]. All these approaches share the common goal of generating a large number of test cases for automatically detecting software bugs.

In this research, the advances achieved in software testing are leveraged to aid the process of automated software debugging. A novel approach for automated software debugging is developed, which is called *Disqover*, or debugging via *code sequence covers*. In this approach, automatically generated test cases are utilized to discover bugs and to help the developer find the source (lines of code) of those bugs. Sequences of lines of code that are executed by these test cases, are recorded, analyzed, and output as a sequence of code statements (with dynamically-observed values of variables) that cause the fault. A series of improvements are developed to the basic algorithm to enhance the output sequence to be more comprehensible, concise, and representative of the error execution scenario so that developers can achieve maximum utilization of the approach's output information.

The advantage of adopting a sequence based approach is that finding error-causing code using the output in the form of code sequences is easier and more convenient for the developer than inspecting the code itself for the following reasons.

- Code sequences are examined in linear order. Developers do not need to track code dependencies and consider different paths through which the code can be executed.

- Code sequences are derived from execution traces, and hence capture runtime behavior as well.

- While generating code sequences, values of program variables are automatically extracted, so that the developer can examine them, and relate the variable assignments to the error.

- Code sequences are enhanced using multiple methods as discussed in Chapter 3, to make them more relevant to the error, and hence, minimize the time needed by developers to root cause and fix the problem.

1.1 Existing Approaches

Several approaches to fault localization involve program slicing [35,36], regression containment [37], and delta debugging [38] and its variants [39, 40]. Slicing identifies all the statements that can affect a variable in a program either statically [35, 36] or dynamically [41, 41–44]. Although slicing techniques reduce the number of lines of code to be examined, the size of the slice can still be large. Regression containment and delta debugging attempt to minimize the difference between working and non-working versions of software programs. Other techniques try to minimize the difference between passed and failed test cases such as *path profiles* [45], *counter examples* [46, 47], *statement coverage* [48, 49], and *predicate values* [50, 51]. These techniques differ from each other based on the information they rely on for analyzing the differences between passed and failed program executions. Despite being useful, many of these approaches are not applicable in all situations. Specifically, they rely on the existence of passed and failed test cases, or working and non-working software versions, which are not always available. They require the application to be run multiple times before localizing the fault which is complex because it is not always possible to generate runnable configurations that contain only specific parts of the code.

Furthermore, compared to the developed approach in this research, inspecting code sequences is more useful than inspecting individual statement suggestions, or parts of the code that are responsible of generating the error. As discussed above, most of the existing approaches for automated debugging provide the developers with a ranked list of statements or blocks of code, sorted by their likelihood of being the root cause of the error. This is usually not useful, as the developer still needs to consider those lines in their source code context to understand how they can be causing the fault, and also the given suggestions may contain multiple code paths, where some of them may be responsible of causing the error, while some are not.

1.2 Motivating Example

In this section, a debugging scenario is demonstrated using a bug in the open-source Java application *Crossword Sage*.[1]

Crossword Sage is a tool for creating professional-looking crosswords with powerful word suggestion capabilities. It can be used to build, load, and save crosswords. It can suggest words for adding to the crosswords, and allows the crosswords builder to give clues for them. Furthermore, in addition to building crosswords, it allows users to load pre-built crosswords and solve them. Crossword Sage project consists of 3072 lines of code, 34 classes and 238 methods.

In order for the user to create a new crossword puzzle, he/she needs to click on the File menu and choose the New Crossword menu item. Then, the application asks the user to input the size of the puzzle through a dialog box. When the user inputs a numeric number between 2 and 20, the application creates an empty grid to allow the user to start building his/her crossword puzzle.

Normally, if the user enters a non-numeric value as the size of the puzzle, an

[1]http://crosswordsage.sourceforge.net

error dialog box should appear that says wrong input value and asks the user to enter another input value. However, in this application when the user enters a non numeric value in the dialog box, the application crashes with a *NumberFormatException*.

Using the most popular debugging method, which is the manual debugging, the developer may follow the following steps in order to locate the source of the error and fix it.

- Initially, the developer locates the line in the source code that is responsible of throwing the exception. In the example, it is line number 33 in the Grid class (setLayout(new GridLayout(Integer.parseInt(cw.getHeight()), Integer.parseInt(cw.getWidth())))).

- Then, the developer should go to this line to investigate the line and try to extract any clues about the reason of the exception.

- From the line, the developer can conclude that the width or the height of the "cw" object might not have been initialized correctly. So, when the Integer.parseInt() function parses the value of any of them, it leads to the NumberFormatException. Now, the user needs to locate where the height and the width of the "cw" object have been firstly initialized.

- Furthermore, the developer may put a break point at this line to figure out the values of the width and height variables to make sure they are not correctly set as numeric values, and run the program once to validate this assumption.

- Using the "Find References" feature available on some IDEs, the developer can find the methods calling the method containing the line 33. There are 4 choices. The developer may need to check each of them to see if they have been on the execution path of that particular error.

- The developer may need to run the program once more to see which of the options is on the execution path. He/she may also need to add more debugging information. It will turn out that line 40 in the CrosswordCompiler class is the one calling the line 33 in the Grid class, and passing the parameter values.
- This line shows that a new instance of the Grid is created with the cw object passed as a parameter which is created in the previous line with the width and height variables that are passed as CrosswordCompiler constructor parameters.
- Also, the developer may again put a break point at this location and run the program again to test the values of the width and height variables.
- The developer also may verify the creation of the cw object by going to the Crossword class which shows that the Crossword constructor initializes the width and the height of the object using the constructor parameters and it assumes that valid values are passed.
- By repeating the procedure above, the developer can find that line 226 in the MainScreen class is the line responsible of creating the CrosswordCompiler object.
- The developer can see that the initialization of the CrosswordCompiler param-

eters is carried out through the "reply" variable. This variable is initialized in the previous line using an input from a dialog box and no check is made regarding the type of the reply variable.

- Again the developer may put a break point at this location and run the program again with different input values to initialize the reply variable and see the effect of these values on the width and height variables.

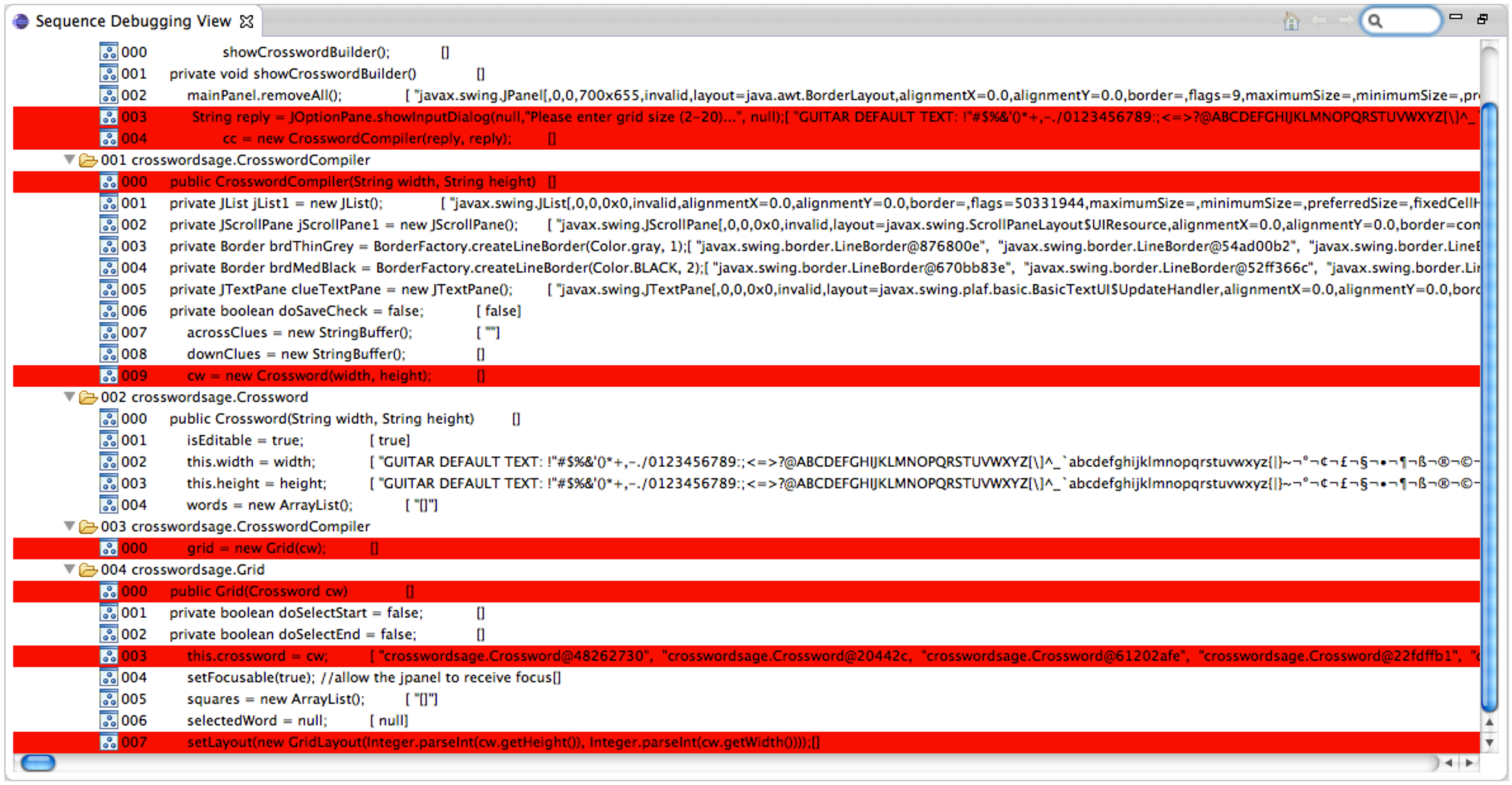
- At this point the developer will discover that a check is needed to verify that the values of the reply string are numeric, or otherwise the program cannot proceed.

As can be seen, using manual debugging, the user has to navigate to the source code many times in order to analyze the code and figure out the cause of the error. Furthermore, the user may run the program again to find out the values that cause the failure.

On the contrary, using Disqover, the user does not need to either navigate to the source code nor run the program again. Figure 1.1 shows the resulting subsequence from applying developed approach to the execution traces of a set of test cases which fail by producing the NumberFormatException at the same line of code.

In order for the developer to find the source of the error using developed approach, the following activities will take place:

- In the Figure 1.1, the last highlighted line is the line (setLayout(new GridLayout(Integer.parseInt(cw.getHeight()), Integer.parseInt(cw.getWidth())));). This



```
000        showCrosswordBuilder();        []
001   private void showCrosswordBuilder()        []
002      mainPanel.removeAll();        [ "javax.swing.JPanel[,0,0,700x655,invalid,layout=java.awt.BorderLayout,alignmentX=0.0,alignmentY=0.0,border=,flags=9,maximumSize=,minimumSize=,pr
003      String reply = JOptionPane.showInputDialog(null,"Please enter grid size (2-20)...", null);[ "GUITAR DEFAULT TEXT: !"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_
004         cc = new CrosswordCompiler(reply, reply);     []
001 crosswordsage.CrosswordCompiler
000   public CrosswordCompiler(String width, String height)  []
001   private JList jList1 = new JList();        [ "javax.swing.JList[,0,0,0x0,invalid,alignmentX=0.0,alignmentY=0.0,border=,flags=50331944,maximumSize=,minimumSize=,preferredSize=,fixedCellH
002   private JScrollPane jScrollPane1 = new JScrollPane();     [ "javax.swing.JScrollPane[,0,0,0x0,invalid,layout=javax.swing.ScrollPaneLayout$UIResource,alignmentX=0.0,alignmentY=0.0,border=con
003   private Border brdThinGrey = BorderFactory.createLineBorder(Color.gray, 1);[ "javax.swing.border.LineBorder@876800e",  "javax.swing.border.LineBorder@54ad00b2",  "javax.swing.border.Line
004   private Border brdMedBlack = BorderFactory.createLineBorder(Color.BLACK, 2);[ "javax.swing.border.LineBorder@670bb83e",  "javax.swing.border.LineBorder@52ff366c",  "javax.swing.border.Li
005   private JTextPane clueTextPane = new JTextPane();     [ "javax.swing.JTextPane[,0,0,0x0,invalid,layout=javax.swing.plaf.basic.BasicTextUI$UpdateHandler,alignmentX=0.0,alignmentY=0.0,bord
006   private boolean doSaveCheck = false;        [ false]
007      acrossClues = new StringBuffer();        [ ""]
008      downClues = new StringBuffer();        []
009      cw = new Crossword(width, height);        []
002 crosswordsage.Crossword
000   public Crossword(String width, String height)        []
001      isEditable = true;        [ true]
002      this.width = width;        [ "GUITAR DEFAULT TEXT: !"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~¬°¬¢¬£¬§¬•¬¶¬ß¬®¬©-
003      this.height = height;        [ "GUITAR DEFAULT TEXT: !"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~¬°¬¢¬£¬§¬•¬¶¬ß¬®¬©-
004      words = new ArrayList();        [ "[]"]
003 crosswordsage.CrosswordCompiler
000      grid = new Grid(cw);        []
004 crosswordsage.Grid
000   public Grid(Crossword cw)        []
001   private boolean doSelectStart = false;        []
002   private boolean doSelectEnd = false;        []
003      this.crossword = cw;        [ "crosswordsage.Crossword@48262730",  "crosswordsage.Crossword@20442c,  "crosswordsage.Crossword@61202afe",  "crosswordsage.Crossword@22fdffb1",  "
004      setFocusable(true); //allow the jpanel to receive focus[]
005      squares = new ArrayList();        [ "[]"]
006      selectedWord = null;        [ null]
007      setLayout(new GridLayout(Integer.parseInt(cw.getHeight()), Integer.parseInt(cw.getWidth())));[]
```

Figure 1.1: A sequence that results from applying developed approach on the seeded fault in Crossword Sage application

line is the line that throws the NumberFormatException.From this line, the developer can conclude that this exception results from applying the Integer.parseInt() function to a non-numeric value.

- This non-numeric value may be assigned to either the *height* or the *width* variables of the *cw* object (because the Integer.parseInt() appears twice in the line).
- Now, the developer can go backwards in the subsequence to figure out where these two variables get assigned.
- The developer can see that the *cw* object comes from the method parameter as shown in line (public Grid(Crossword cw)).
- Since this method represents a constructor, there should be a line in the sequence that creates new object from the Grid class and passes the Crossword object as a parameter as can be seen at the line (grid = new Grid(cw)).
- By going backwards, the developer can see that the Crossword object *cw* is created at the line (cw = new Crossword(width,height);) and the width and the height are passed as parameters.
- These width and height variables are passed to the function through the CrosswordCompiler constructor arguments at line (public CrosswordCompiler(String width, String height)).
- Finally, by going backwards at crosswordsage.MainScreen class, the developer can see that these parameters are passed as arguments when creat-

ing a new instance of crosswordsage.CrosswordCompiler at line (cc = new CrosswordCompiler(reply, reply)) and these arguments are both initialized by the variable *reply* which takes string values in line (String reply = jOptionPane.showInputDialog(null, "Please enter grid size (2 - 20)...", null);).

As it can be seen from the example, the developer does not need to refer to the source code because all the required information exists in the resulted subsequence. Furthermore, the values that are assigned to the variables during the test cases execution are also available for the developer's convenience.

1.3 Challenges

There are a number of challenges involved in achieving the developed approach:

- The problem of software debugging is a hard problem. Even human developers are challenged when addressing software problems. Finding software bug root causes is a tedious effort that consumes significant engineering time.

To address this challenge, The advances in software testing, and the availability of tools that can generate large numbers of test cases are utilized. Test cases are not only rich in their ability to cover large portions of the software execution traces, but also are rich in the information that can be extracted from them, which can be further studied and analyzed to reach the bug root cause.

- Most techniques for software fault localization localize the fault by providing the software developer with a ranked list of statements according to their likelihood of being the source of the problem. Using these techniques still requires developers to spend time to understand the context where those individual lines are executed, and find which faulty execution paths were they actually part of in order to solve the problem.

  To address this challenge, techniques are developed that not only rank the program statements, but also provide the context where those faulty statements took place. This context is in the form of program statement subsequences that get executed when the fault takes place. This is accomplished by obtaining the common subsequences between the execution traces of failing test cases, and designing novel algorithms to decrease the length of those common subsequences and narrow them down to a small number of subsequences to be considered by the developer.

- Finding commonalities between code sequence covers corresponds to the multiple common subsequences problem, which is NP-hard [52]. This problem is also related to the multiple sequence alignment problem that is well studied in the computational biology literature [53–55]. Most approaches for finding multiple common subsequences focus only on finding the longest common subsequence, and hence are not applicable in this case as the common subsequence of lines 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 does not necessarily result in the optimal alignment. Additionally, multiple sequence alignment allows mismatches as a compromise to get longer subsequences, which is not allowed in this case.

To address this challenge, a new efficient algorithm is designed for finding common subsequences between code sequence covers, and a new way is presented to represent those common subsequences as a directed acyclic graph, known as the common subsequences graph. Furthermore, various abstraction techniques are implemented to make the input code sequence covers more concise and developer-friendly.

- Software usually contains more than one fault, making the problem of fault localization more difficult, because of additional dimensionality introduced by every additional fault.

  To address this challenge, the ability of test case generation techniques is utilized to generate a large number of test cases for the same application, and group them by their type. This grouping allows applying the developed approach on every group independently, and providing a separate subsequence to the developer, representing the root cause for each group separately.

- Although knowing which statements have executed and in what order help with identifying the root cause of an error, an effective piece of runtime information that can be utilized as well is the state of program during the execution. Most techniques for automated debugging do not offer program state runtime information as part of their output to assist the developers. Developers must resort to other ways to obtain such information such as manual debugging, printing the values of variables, or digging through program logs.

  To address this challenge, the developed automated debugging approach is integrated with remote debugging tools, which automatically query the program state information during runtime, summarize it, and provide it to the developer as part of the output.

- Finding common subsequences between faulty test cases by itself may not be sufficient to help the developers spot the problem. Common sequences may contain code that is unrelated to the fault, in addition to the root cause subsequence.

  To isolate the root cause subsequence, a hybrid static/dynamic dependency analysis approach is designed to extract the program statements in the common subsequence that have a dependency relationship with the statement at which the fault is discovered. That way, unrelated statements are filtered out, and only relative statements are included in the output subsequence.

1.4 Goals

The goals of this research are the following:

- Enhance the software development in general and the software debugging in particular by doing part of the work the developers are doing in their normal debugging activities automatically, and hence minimize the time spent by the developer on fixing errors, and enhance the overall software quality by spending more time on introducing new and useful features.
- Overcome shortcomings in existing manual debugging approaches, or automated software debugging applications, which require the developer to navigate through source code dependencies and envision runtime behavior.
- Provide developers with quality suggestions by utilizing the advances in the research of automated software testing, and extending them to support automated software debugging.

1.5 Overview

At a high level, the developed approach consists of the following steps to achieve the above goals.

- **Automatic test case generation:** In addition to the manually generated test cases, the existing test case generation systems are utilized to support automated debugging. Automated test case generation tools, such as [56], build models from software applications to generate a large number of test cases. Test cases that fail for the same reason are collected in the same group, in order to extract commonalities between their execution traces as outlined below.

- **Generating code sequence covers:** Once the test cases are generated, the available statement coverage generation tools are extended to generate *code sequence covers*. Code sequence covers are different from regular statement coverage reports in that statement coverage reports just report the statements touched during the code execution along with some statistics such as the number of times they were touched. On the other hand, code sequence covers report a sequence of lines of code that are touched line by line during the execution.

- **Abstracting code sequence covers:** Code sequence covers may be too detailed for developers to inspect as is. There may be blocks that always appear together to perform a certain functionality, which can be abstracted together as a unit. There may be repetitions due to loops and iterative processing. Two sequence cover abstraction techniques are implemented that minimize the effect of these problems, by detecting code execution units (blocks), and eliminating repetition, but with conserving runtime information. The approach for detecting code units (or blocks) is based on finding consecutive lines that appear in the same order, possibly multiple times, in all test cases. Suffix trees is used to detect such sequences. Identifying blocks by searching cross sequences ensures that the common units (or blocks) are still detected and not lost when abstracted one sequence at a time, as there are multiple ways a block can be formed from the same set of lines. Secondly, the approach for detecting repeated lines from loops is by detecting subsequences that are repeated consecutively after each other. Those subsequences are abstracted by listing one occurrence only of the repetition. This process is repeated until no more loops are detected in order to account for nested loops.

- **Finding common subsequences:** The next step after abstracting code sequence covers is to extract their common subsequences. A novel algorithm is presented to identify the common subsequences among a set of sequences (which are sequence covers in this case). The algorithm has the following characteristics:

  1. Unlike most algorithms for finding common subsequences between multiple sequences, the algorithm does not find the longest common subsequence only. Longest common subsequences may be too long for developers to inspect, or worse, the root cause may not exist in the longest subsequence at all.
  2. Unlike most of the algorithms for finding biological sequence alignment, the developed algorithm is not progressive, and hence does not depend on the order of adding new sequences. It also does not stick in a local optima as it considers all the sequences together.
  3. Alternatively, the algorithm constructs a compact representation of all

possible common subsequences called the *common subsequences graph*, which is a directed acyclic graph, and generates common subsequences out of it.

4. The construction of the common subsequence graph is performed using an efficient algorithm that avoids creating unnecessary nodes that do not contribute to the final common subsequences.

- **Ranking the common subsequences:** After constructing the common subsequences graph, common subsequences are generated by traversing paths from that graph. Due to the large number of paths, path selection algorithms are used to select the top-k paths in that graph, which represent the paths with more likelihood of containing the root cause.

- **Dependency information extraction:** The extracted common subsequences may still contain irrelevant lines to the error. To eliminate such irrelevant lines, dependency information is used to automatically extract lines related to the error.

- **Variable values extraction:** Finally, variable values are attached to their corresponding variables in each line in the final subsequence.

Disqover is evaluated to understand whether it helps developers find root causes of failures more effectively, whether diversifying the input test cases or increasing their number reduces the number of statements in the common subsequence, and whether Disqover algorithms lead to a more efficient evaluation of the common subsequence given large software code bases, and a large number of execution traces.

Experiments are performed to measure the number of statements to examine by developers before reaching the root cause in comparison to other techniques such as MUSE [57], Op [49], Tarantula [48] and Fonly [58]. Moreover, the effect of choosing diverse input test cases is measured on the size of the output common subsequence. Furthermore, the abstraction techniques are evaluated to measure their effects on the length of the input sequence covers, and the effect of the number of test cases is evaluated over both the running time and the length of resulting common subsequence, comparing the developed approach to multiple baselines. The results show that Disqover significantly reduces the number of lines needed to discover the source of the fault, and show the effectiveness of the sequence cover abstraction techniques, for reducing the computation time and the length of the output common subsequences, especially, for the computationally intensive ones.

1.6 Broader Impact and Intellectual Merit

This research provides a novel technique for automated software debugging that utilizes code sequences for suggesting code failure paths. The approach suggests concise subsequences for developers which they can trace back to find the source of the error. The approach is based on a novel and efficient algorithm which finds common subsequences among a set of code sequence covers. In addition to the algorithm's applications in automated software debugging, it has a broader impact and relationship to a whole set of other computational applications, such as the alignment of DNA and protein sequences, word alignment for machine translation, and finding optimal matching in optimization problems.

With respect to the impact on the field of software development, a Cambridge research study [59] states that software debugging costs 312 billon annually. Furthermore, the research study states that developers spend 50% of their time debugging software. Additionally, a user study that is performed as part of evaluating the developed approach shows that on average, it saves developers 79% of their debugging time for the applications used in the study. Extrapolating this ratio to the costs reported by the Cambridge study, the developed approach can save significant software development costs and provide developers an additional 40% of their overall software development time to enhance the software quality and introduce additional features. That does not only save software development costs, but also increases software quality, which further increases software value, software organization reputation and credibility, and customer loyalty and appreciation.

1.7 Conclusions

In this chapter, the problem of automated software debugging is motivated, and is briefly reviewed how this problem is being handled in existing systems. The developed approach is presented for solving this problem, which overcomes some of the shortcomings in existing systems. The developed approach utilizes the advances of software testing to aid the downstream activity of *debugging* by finding the commonalities between the executions of failed test cases. An overview of the work that has performed as part of this research is presented. In the next chapter, the related work is discussed in detail, and the differences are highlighted between the developed work and other research in the same area.

Chapter 2: Related Work

In this chapter, the related work to the research in the area of automated software debugging is reviewed. First, the early research that has been done in this field, which is a set of techniques called *program slicing* is presented. Second, *differential debugging* techniques are presented. These techniques narrow down the set of possible statements causing an error using the differences between either the working and non-working software versions, or passed and failed test cases. Third, a technique that is based on failed test cases only is discussed. Fourth, approaches based on machine learning and data mining techniques and model-based approaches are discussed. Finally, some related work about automated performance debugging are presented.

2.1 Slicing

Research in the area of automated debugging started a long time ago by Weiser [35, 36], who proposed *program slicing*. Slicing defines all the statements that can affect a variable in a program. Therefore, given a variable v in a program P, slice S contains all the statement in P that may affect the value of v. The main idea is that if the statement that contains the variable v is erroneous, the source of this error can be in slice S. Slicing is calculated either statically [35,36] by finding the relevant statement according to data and control dependencies, or dynamically [60] by benefiting from information collected during the program execution. Although dynamic slicing and its variations [41–44] potentially reduce the size of the slices and improve debugging, the size of the slices is still fairly large and slicing techniques are rarely used in practice. The slice, which is the output of the slicing algorithm, is a reduced version of the original program. It only contains the statements that affect the value of the output and has the same behavior as the original program. Therefore, the debugger still has to run the reduced program again and manually detect the source of the error. On the contrary, Disqover provides sequences of statements that take place when running the failed test cases with each statement associated with the possible variable values. Therefore, the debugger can easily find the source of the error by backtracking these sequences.

2.2 Differential Techniques

2.2.1 Techniques Based on Working and Non-working Program Versions

This category assumes that there are at least two versions of the program, a working version and a non-working one. This means that running a test case passes in one version and does not in the other one. Therefore, a source of an error can be found by computing the difference between the working and non-working versions. One approach of this category is *regression containment* [37]. It isolates the changes that cause the error by defining the set of changes that have been done between the working and non-working (that is not passing the test) program editions. Since there can be multiple changes, it encapsulates the related changes together in identifiable objects called mods and orders them in chronological order (i.e., by the order they were introduced to the original program). The mods are removed from the non-working edition in reverse chronological order either linearly (one by one) or binary until the test passes. The last mod that has been removed is the mod that is responsible of the test failure. This method works effectively if one change is causing the error, but not a combination of changes. Also, while a change may contain hundred or even thousands of lines of code, only a few lines may be responsible of the error. Finally, the chronological order of changes may not always be available or easily obtained.

Delta debugging [24] is proposed to address the limitations of regression containment. It is similar to the regression containment technique in relying on the existence of working and non-working versions of the software and investigating the set of changes between the two versions. Assuming that a set S contains all the changes that have been done between the passing and failing versions, if S contains one change, this means that the source of the error is in S. Otherwise, S is partitioned into two sets S_1 and S_2 and each set is tested separately. If either of S_1 or S_2 produces the failure, this means that S_1 or S_2, respectively, contains the source of the error and it is subject to more binary partitioning. If both of them pass, this means that the source of the error comes from a combination of them. Therefore, the combination of the two sets is partitioned in different ways until the minimal combination of changes is identified . This divide-and-conquer algorithm takes care of interference (i.e., a combination of changes is responsible of the error), and granularity (a single change may encompass hundreds of lines of code but a small subset of it may be responsible of the error) which are not addressed in the regression containment technique. Delta debugging not only minimizes the difference between two program versions but also the differences between two inputs where one input is correctly run by the program and the other one is failed to be run as proposed in [25].

Other variations of Delta Debugging (DD) have been proposed to overcome its limitations such as *Hierarchical Delta Debugging (HDD)* and *Iterative Delta Debugging (IDD)*[1]. Although, delta debugging effectively identifies the source of the error, it is only applicable with independent change list in which each change in the list is independent of the other changes, e.g., a change containing a for loop and another containing its body form dependent changes, while two for loops are independent. This constraint makes DD works poorly with the data that has hierarchical structure like object oriented programs and XML input files. Therefore, HDD was proposed to overcome this limitation by applying the DD algorithm on each level of the algorithm's input starting from coarsest to the finest levels. IDD finds an older program version, among the existing versions, in which a test case passes but fails in the current version. In some cases, a test case may not be applicable to older versions. Therefore, IDD successively uses DD to apply the necessary changes from the newer version to an older version until it finds an older version that allows a test case to run. IDD starts with the current version P_c (which fails the test case

t). It successively goes back and checks the older versions P_{o^i}. If the output of P_{o^i} for running t matches the output of P_c, IDD skips this version and proceeds with $P_{o^{i-1}}$ version. If the output of P_{o^i} is undetermined, DD is applied between the two versions P_c and P_{o^i} and another version is produced called P'_{o^i} which behaves the same as P_c with the test case t. This process proceeds until either the version that passes t is found or there is no more older versions.

All these techniques rely on the existence of either working and non-working versions of the program under test, or passing and failing inputs. This assumption is not true in most cases because of either the absence of older program versions or the absence of another version that allows the test case to run. Also, extracting the changes between two program versions and applying parts of these changes to the working version are very time consuming because of the execution time required to run multiple combinations of these changes especially for large applications, and it also cannot be done in parallel because the run at one iteration depends on the output of the run at the previous iteration. Furthermore, applying part of the changes to the working version may not always result in an executable version. Moreover, the output of all the previous techniques is a set of lines without any further information. Finally, those approaches operate on static versions of the programs and do not incorporate runtime information back into the debugging output. Conversely, Disqover just relies on the current version of the program. Also, It does not need to worry about changing the source code or generating executable versions of the program. Lastly, it uses the dynamic execution trace of the program and generates sequences of statements that take place when running the failed test cases with each statement associated with the possible variable values.

2.2.2 Techniques Based on Passed and Failed Test Cases

The idea of the second category is based on finding at least one passing test case that is approximately similar to a failing one and extracting the difference between the execution of the test cases. There are multiple types of test-case-based automated debugging techniques such as approaches based on *path profiles* [45], *counter examples* [46,47], *statement coverage* [48,49], *statement mutants* [57], *predicate values* [50,51] and *Program states* [61,62]. These approaches differ from each other in the type of information that they rely on to define the characteristics of the program execution.

Approaches based in path profiles [45] identify the program paths that are explored during the passed and the failed test cases by instrumenting the program during the test case execution. Then it finds the differences between the two sets of paths. In other words, it defines the paths that are present during the execution of the passed test cases and are not present during the execution of the failed test cases S_p and visa-versa S_f. Finally, it calculates the shortest prefixes that appear in all the paths of S_p and do not appear in all the paths of S_f. These prefixes present the critical portions of the code that the programmer should investigate to define the source of the error.

Approaches based on Counter examples [46,47], use the trace reports that result from model checking tools for passing and failing test cases. Then, it takes the differences between the two trace reports. For each error, it compares one error trace with one correct trace.

Approaches based on statement coverage [48, 63] use visualization tools to represent the suspicious code statements. These tools use different techniques to color the program statements according to their participation in the test case execution. [63] uses one color to mark the statements that exist in the dynamic slices of the failed test cases and do not exist in the dynamic slices of the passed test cases. Theses statements most probably contain the source of the fault. At the same time, it uses another color to color the statements that exist in the dynamic slices of the execution of all the test cases. These statements less likely contain the source of the error. [48] colors the statements according to their percentage of participation in running the test cases. So, the statements that participate in the execution of more failed test cases than passed test cases are colored with more red-ish color. On the other hand, the statements that participate in the execution of more passed test cases than failed test cases are colored with more green-ish color. The statements that have the same percentage of participation are colored with yellow color. Also, Wong et al. [64] define the suspiciousness of each statement based on the relationship between its coverage and the execution results (failed/passed) of test cases. This is done by calculating the crosstab of each statement in which the columns represent the coverage information (covered/not covered) and the rows represent the execution results information (failed/passed). Furthermore, Naish et al. [49] calculate the suspiciousness of a statement according to the following formula:

$$O^p = a_{ef} - \frac{a_{ep}}{P+1}$$

, where a_{ef} is the number of failed test cases that execute s, a_ep is the number of passed test cases that execute s, and P is the total number of passed test cases. They expect that buggy statements have high a_{ef} and low a_ep, which leads to a high suspiciousness score. Therefore, statement with the highest suspiciousness score are most likely to be buggy.

One of the most recent and effective techniques is [57]. It creates mutants for each statements according to different characteristics. To create a mutant for a statement, it should be hit by a failed test case. Finally, they calculate the suspiciousness of a statement according to the following formula:

$$suspiciousness(e) = \frac{1}{|mut(s)|} \sum_{m \in mut(s)} (\frac{|f_p(s) \cap p_m|}{|f_p|} - \alpha \frac{|p_p(s) \cap f_m|}{|p_p|})$$

, where $mut(s)$ is the number of mutants that are generated for a statement s, $\frac{|f_p(s) \cap p_m|}{|f_p|}$ is the proportion of tests that failed on P but now pass on a mutant m that mutates s over tests that failed on P, $\frac{|p_p(s) \cap f_m|}{|p_p|}$ is the proportion of tests that passed on P but now fail on a mutant m that mutates s over tests that passed on P, and α as

$$\alpha = \frac{f2p}{|mut(P)| \cdot |f_p|} \cdot \frac{|mut(P)| \cdot |p_p|}{|p2f|}$$

, where $f2p$ and $p2f$ are the number of test result changes from failure to pass and vice versa between before and after all mutants of P, $mut(P)$ is the number of mutants that are generated for all the statements of P. Their hypothesis is that mutating a faulty statement will either keep the program faulty, or fix the program partially. At the same time, mutating a correct statement is more likely introduce a new fault.

Approaches based on predicate values [50, 51] associate bugs in the program with predicates that are instrumented during the execution of the program. The algorithm in [50] computes two probabilities for each predicate P. $Failure(P)$, which is the probability of P being true implies failure, and $Context(P)$, which is the probability of executing P may produce failure. Then, it discards the predicates that have $Failure(P) - Context(P) \leq 0$. Finally, it prioritizes the remaining predicates based on their score. On the other hand, the algorithm in [51] computes the probability of a predicate P is evaluated to true in each run as $\pi(p) = n(t)/(n(t) + n(f))$ where $n(t)$ is the number of times that P is evaluated to true in a specific run and $n(f)$ is the number of times that P is evaluated to false. Then, it correlates a predicate to a bug if its distribution during the failed test cases is significantly different from that in successful test cases.

Zeller [61] and Cleve et al. [62] propose approaches based on the differences between the program states (which consists the variables and their values at particular point during the program execution) during the passed and failed program executions.

The problem with these techniques is that the number of lines of code that they identify for inspection by the developer can still be high because it is not always possible to find close enough failing and passing test cases. Furthermore, the developer still needs to inspect the source code to understand the context of the suggested statements. Also, some techniques like [57] takes too much time in order to calculate the statements suspiciousness.

2.3 Techniques based on Failed Test Cases

Zhang et al. [58] propose a fault localization technique that is based on failing test cases only. Their hypothesis is that the more faulty runs that go through a program entity (e.g. statement), the more likely this entity can lead to the failure. They use the following formula to calculate the suspiciousness of a statement:

$$suspiciousness(e) = \frac{\sum_{c \in D}[c(Y(c) - Y(0))] \times c_{max} / \sum_{c \in D} c^2}{\sqrt{\sum_{c \in D}(Y(c) - Y(0))^2 - (\sum_{c \in D}[c(Y(c) - Y(0))])^2 / \sum_{c \in D} c^2}}$$

, where c is the number of times in which a test case executes e, $Y(c)$ is the number of test cases that executes e c times and $Y(0)$ is the number of test cases that never execute e. This technique still has the same problems of the techniques that are discussed in section 2.2.2 because it provides the developer with a list of ranked statements.

2.4 Machine Learning-based Approaches

A number of research studies propose machine learning and data mining methods for fault localization. Wong et al. [65] propose an approach based on back-propagation (BP) neural networks for fault localization. It utilizes the statement coverage information for passing and failing test cases to train a BP neural network in which the network learns the relationship between the coverage and the success or failure of test cases. Then it computes the suspiciousness of the program statement by including this statement in a virtual test case and using the virtual test case as an input to the BP network. Furthermore, Wong et al. [66] use the same algorithm that is proposed by [65] but using RBF (radial basis function) networks to overcome the limitations of BP neural network such as network paralysis (network learning stops) and local optimization.

Briand et al. [67] analyze test case specifications in terms of their input and output to identify distinct conditions of failure using C4.5 decision trees. Each path in this type of trees represents a rule for distinct failure condition with a probability prediction for distinct failure. The statement coverage for the passed and failed test cases are used for ranking failure conditions. Then the ranking of failure conditions is used for the final ranking of the statements that should be examined to detect the source of a failure.

Brun et al. [68] propose a machine learning approach based on Support Vector Machines or Decision Trees. This approach consists of two phases: training and classification phases. The training phase builds a model using previously known errors and program properties (e.g. program variables). The input to this phase is two program versions, one has one fault and the other does not have that fault. The properties of each version are extracted and classified to fault-revealing, which exist in the faulty version and do not exist in the non-faulty one and non-fault-revealing, which exist in both versions. The classification phase applies the training model to the set of properties that are specified by the user and outputs the set of properties that are more likely related to the error ordered by their likelihood of being fault-revealing.

Nessa et al. [69] compute a set of subsequences of length N (N-grams) from the traces of test cases execution. In order to define these N-grams, they identify the execution blocks by constructing Execution Sequence Graph. In this graph, the vertices represent the lines of code and the edges represent the consecutive relationship between the lines. An edge exists between two vertices if they are executed consecutively in at least one test case. This definition of blocks reduces the size of the trace and helps in defining possible branches. Using these blocks, all possible N-grams of lengths 1 to N are generated. Then, the number of occurrences of each N-gram is calculated in the failed test cases and the ones that are greater than a specified threshold are selected. Finally, the chosen N-gram subsequences are ordered based on their calculated conditional probabilities that a given test case is failed because the appearance of a specific N-gram.

Cellier et al. [70] propose a data mining method to identify rules between the statement execution and test case failure based on association rules and Formal Concept Analysis (FCA). First, they build the trace context in which the objects are the execution traces of the test cases and the attributes are the lines of the program and if the test case pass or fail. Second, they generate the association rules that are strongly related to the failure and specify a minimum threshold. Third, they define the relation between the defined rules using the rule context and rule lattice. Finally, to detect the fault, the rule lattice is investigated bottom up in order to investigate the more specific rules first, then the more general ones.

None of these approaches ensures that the statement that contains the source of the error will exist in the resulting suggested statements. Furthermore, the user still needs to examine the source code to define the source of the error using the clues (list of statements, which are ordered according of their suspiciousness) that are reported by these approaches.

2.5 Model-based Approaches

Some research studies are proposed to analyze the relationship between the failures and the faults or between the source and the failure. Wotawa, et al. [71] propose a model-based approach that exploits the program variable dependencies, the control flow and the whole semantics of the program. The model behavior is extracted from the test cases in terms of their input and output. For searching for the bug locations when a test case contradicts with the model, each program statement is assumed to be correct or incorrect by default, then this assumption is revised during the debugging process until identifying the cause of the error. Mateis, et al. [72] propose a model-based approach for a subset of features of Java programs such as classes, methods, assignments, conditionals and while-loop. The program is statically compiled to a model, which can be divided into two parts: the structural part and the behavioral part. The structural part presents the program components and the connectivity relations between them. The behavioral part, which helps in defining the faulty statements represents the behavior of these components using a logic-based language. Mayer et al., [73, 74], extended this model-based approach to handle the unstructured control flows of Java programs such as exceptions, recursive method calls, return and jump statements.

Furthermore, DeMillo et al. [75] propose a model-based approach that describes the relationship between the failure and the faults. The model consists of failure modes and failure types. Failure modes describe the different symptoms of the program failure. Using these failure modes a program failure is categorized. Failure types describe the nature of the failures. In order to localize the failure, the following steps are followed. First, the failure mode is identified. Then using the model, which describes the relations between the modes and the types, the type of the failure is identified. Finally, using heuristics based on dynamic instrumentation and testing information, the search domain is reduced for predicting possible faulty statements.

Model-based approaches are difficult to apply on real applications because it is extremely hard or impossible to generate a model that accommodates all program behaviors, which makes the model incomplete.

2.6 Performance Debugging

Some approaches have been proposed also for debugging software performance. These approaches focus on defining the system bottlenecks that result from I/O operations, CPU or memory consumption. One of these approaches is [76], which is an approach for summarizing the execution profiles of large systems and identifying overlaps between these summaries. Using a search tool over the summaries, the system bottlenecks can be identified. Also, [77] proposes an approach that uses *thresholding* and *filtering* to define a small set of costly methods invocations. Thresholding chooses only the components that exceed the user-defined threshold and filtering filters out the user-defined components. [78] provides a visualization tool called Jinsight EX that allows the user to choose the most valuable information that should be included during performance analysis. This tool is used to define Java applications bottlenecks. [79] proposes StackMine, a tool for effectively identifying the cause of performance bugs that are reported through the execution traces of a huge number of users. It applies mining algorithms on these execution traces to define the most costly subsequences of function calls that account for a non trivial waiting time, then it identifies the signature that causes this delay. [80] focuses on identifying the cause of the idle time in server applications by analyzing their states of idleness during the execution time using WAIT tool.

2.7 Conclusions

In this chapter, the existing automated debugging approaches are reviewed and the differences between them and the developed approach are discussed. One of the first techniques that have been proposed in this area is the slicing technique, which is proposed by Weiser. Then, other different approaches are discussed that are based on finding the differences between passed and failed either test cases or software versions. Furthermore, other research studies are introduced that are based on machine learning, data mining and model-based techniques. Finally, some studies are presented aimed at automating software performance debugging. In the next chapter, Disqover modeling is discussed in details.

Chapter 3: Modeling Disqover

In this chapter, Disqover, an automated debugging approach is discussed. Disqover is applicable to all types of softwares. The section starts by stating some basic definitions, then it discusses Disqover in detail in the following subsections.

Definition 1 ***(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.*

Definition 2 ***(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 produce an unexpected output during the execution of t.*

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

Definition 3 ***(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, two types of statements that are essential for such type of automated debugging are defined, *failure statement*, and *root cause subsequence.*

Definition 4 ***(Failure statement)*** *is the statement where the unexpected output is detected. The failure can also take the form of an application error.*

It is noticed 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 5 ***(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.*

In this approach, the commonalities between multiple test cases which fail for the same reason (i.e., from the test case group) are extracted. By using more than one failing test case, the subsequence 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). However, this approach is inadequate, as it returns an unordered set of statement with no relationship between them. Therefore, the basic idea of the developed approach is to extract the common subsequence among the failing test cases sequence covers. 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, not only the statements that are just shared between the sequences will be considered, but also 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 a motivating application is presented for using sequence covers for automated debugging, as opposed to using code coverage intersection, but first, these both terms are defined.

Definition 6 ***(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 executed during the execution of* t*.*

Definition 7 ***(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.*

3.1 Motivating Example

Consider the code snippet listed in Figure 3.1, which has the statements s_1, s_2, s_3, s_4 and s_5 (the *if* and the *for* statements are excluded 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 the order of statement execution is utilized, it can be said 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, the execution trace of each test case is generated as a sequence of statements, and the common *ordered* statements between all the test cases are extracted.

```
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;

}
```

Figure 3.1: Example program

$S(t_1) = \text{a b a c}$

$S(t_2) = \text{c a b a}$

$S(t_3) = \text{a b d a}$

Figure 3.2: Sequence covers of three test cases

3.2 The Disqover Approach

Now, Disqover, an automated debugging approach is discussed. Disqover takes as an input the test suite and the source code of the application under test (AUT) and outputs the detected faults with their recommended code subsequence that lead to the source of the sfault.

Disqover consists of 5 steps:

1. The Execution Trace & Logs Extraction, which extracts the test cases execution traces and the test cases execution logs. The execution traces present the order of the statements that are touched during the execution of the test cases. The execution logs present the output of each test case, i.e., whether it

passed or failed.

2. Test cases Partitioning, which groups the test cases according to the type and the location of the faults caught by the test cases execution. It takes as an input the test cases execution logs and outputs test case groups. Each group has the test cases that are failed for the same fault (exception, error, or false assertion) type at the same location in the source code. In addition, it outputs an additional group for all the passed test cases.

3. Common Subsequence Extraction, which extracts a common subsequence of lines found in the trace of the failing test cases.

4. Hybrid Dynamic/Static Analysis, which uses both static information coming from code dependency analysis and dynamic information coming from the common subsequence from the pervious step, to provide the dependency of the failed line within the common subsequence. It takes as input the common subsequence and outputs a final subsequence. This subsequence explains the fault since it contains only the lines that affect the failed line.

5. Remote Debugging, which provides the values of the variables that included in the subsequence that explains the fault.

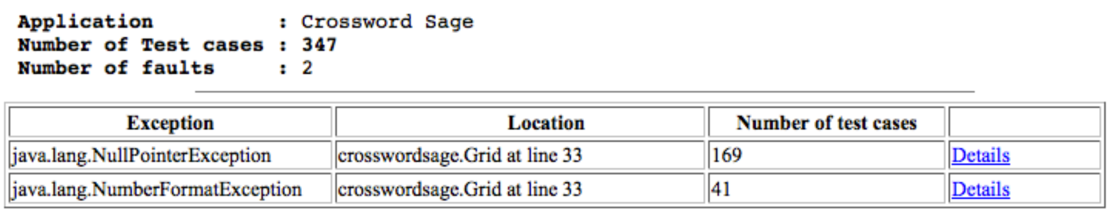
In the following subsections, each step is discussed in more details.

3.2.1 The Execution Trace & Logs Extraction

To analyze the failed test cases, the test case execution trace and execution log are captured. Test case execution trace presents which statements where touched during the test case execution. Since we care about the order of execution of the statements in addition to the statements themselves, this order is captured as well. Test case execution log presents the output of the test case (e.g. whether it passes or fails) .To extract the test cases execution trace, a code instrumentation tool, Cobertura [81], is modified to achieve this task. Cobertura is an open source Java tool that calculates the percentage of code accessed by test cases. It instruments Java byte-code after program compilation. It can generate either HTML or XML reports. Each line is represented by package name, class name, method name, line number and the number of hits during running the test case. In this research, Cobertura source code is modified so that it can output a report of the program execution trace in the form of a sequence of program lines that are touched during replaying the test case.

3.2.2 Test cases Partitioning

In the partitioning step, the test cases are partitioned into groups. Each group has test cases that fail for the same reason. In other words, all the test cases that belong to the same group throw the same type of exception at the same location. This step takes as an input the logs of the test cases that result from executing the test cases. According to the type of the error and the location of this error



```
Application          : Crossword Sage
Number of Test cases : 347
Number of faults     : 2
```

Exception	Location	Number of test cases	
java.lang.NullPointerException	crosswordsage.Grid at line 33	169	Details
java.lang.NumberFormatException	crosswordsage.Grid at line 33	41	Details

Figure 3.3: Partitioning Output

in the source code, the test cases are grouped together. At the same time, all the passed test cases are grouped together to compare them to the test oracle in order to detect more faults. Figure 3.3 shows an example of an HTML-based output of the partitioning step, partitioning the test cases of one of GUI applications, Crossword Sage, into groups according to the two types of errors found by running those test cases: NullPointerException, and NumberFormatException, along with the number of test cases relevant to each error, and a hyperlink to the detail pages explaining the reasons of those failures.

3.2.3 Common Subsequences Extraction

In this section, the algorithm for finding code sequence coverage intersection is discussed in detail. The goal of the 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 the three execution traces in Figure 3.2 are obtained, the algorithm is needed 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 this research 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 the approach for finding all the common subsequences is discussed, and in Section 3.2.4, how to rank the subsequences according to their importance is shown so that the subsequence with the highest rank is outputted according to that criteria.

To enumerate all the possible common subsequences between a set of sequences, the steps outlined below are followed.

3.2.3.1 Applying Code Coverage Intersection

As an initial step, all the code coverage sets of the test cases are intersected to get the set of statements that are common between them. Clearly, the common subsequences must be composed of statements in that intersection only. This set is denoted as $C = C(t_1) \cap C(t_2)...$

3.2.3.2 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, how to model that problem is discussed using the *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, a particular combination of occurrences of a statement is defined in all sequence covers to be an *instance* of that statement, as it can possibly contribute to a common subsequence. For example, in Figure 3.2, 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 the instances of occurrences for each statement are defined, 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, $>$ is defined 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., $inst_i < inst_j$. This approach has a number of disadvantages:

1. It is quadratic in the number of instances, which is exponential in the number of test cases to begin with. So, it is very inefficient.

2. This approach may result in redundant common subsequences. For example, consider the instances $inst_1 = (1, 2, 1)$, representing a, and $inst_2 = (2, 3, 2)$, representing b, and $inst_3 = (3, 4, 4)$, representing another occurrence of a. An approach that blindly constructs common subsequences if the positions are strictly increasing will generate both the common subsequences $(inst_1, inst_2, inst_3)$, i.e., aba, and $(inst_1, inst_3)$, i.e, aa, because both follow the strictly increasing position criteria. However, a wiser approach should generate $(inst_1, inst_2, inst_3)$ only, as $(inst_1, inst_3)$ is already a subset of it.

3. This approach requires enumerating all the possible instances, even if we are not going to use them. For example, once we generate the instance $inst_1 = (1, 2, 1)$ for a, there is no need to generate $inst_2 = (1, 4, 1)$ for a, as $inst_2$ cannot appear with $inst_1$ in any common subsequence, and hence we can save a lot of the exponential time complexity involved in generating *all possible instances*.

As it can be seen, it is inefficient to use an enumeration-based approach. In the experimental evaluation, this approach was evaluated as a baseline; however it failed to find the common subsequences as it resulted in an out of memory exception due to its high memory requirements.

To make this process more scalable, an algorithm that generates the instances *on demand*, and *avoids constructing redundant subsequences* during the common subsequence building time is developed. 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, the least instance for each statement in the code coverage intersection is created, its edges are generated, and recurse. For example, considering the sequence coverage in Figure 3.2, the algorithm starts by defining the least instance for $a : (1, 2, 1)$, and the least instance for $b : (2, 3, 2)$ and adds them to a stack. Then, it picks $(1, 2, 1)$ from the stack, generates its edges by choosing from the *next least possible* instances relative to it, and adds 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, binary search are used 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', the next least position to p_i is found in the sequence of t_i by searching for p_i in that sequence. Consuming nodes from the stack are continued until the stack becomes empty, the point at which a precedence graph is generated 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.

3.2.3.3 Extracting common subsequences

To generate the common subsequences between the execution traces of all test cases, the algorithm starts from the node representing the failure statement in the common subsequences graph, and traverses its neighbors, generates all possible paths. Each of these paths is a common subsequence. Algorithm 1 lists the pseudocode for the common subsequence extraction process.

3.2.4 Algorithm Optimizations

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

3.2.4.1 Test case abstraction

Test case abstraction is achieved using two techniques: loop-based abstraction, and block-based abstraction.

Loop-based abstraction While creating the graph of instances, it is found that the algorithm spent a lot of time and memory in building the graph because of the existence of repeated lines resulting from program loops. Loops result in lines that are repeated multiple times in each sequence cover, and their occurrence in multiple sequence covers results in a number of instances in the graph that is exponentially proportional to the number of test cases. Therefore, this approach is impractical. Furthermore, from the developer's point of view, inspecting a single occurrence of each line in the loop may be more convenient than inspecting all iterations of the loop unrolled. Therefore, before extracting the intersected lines among all the

Algorithm 1 Common subsequences generation algorithm

```
1:  procedure GET_COMMON_SUBSEQUENCE
2:      $C = C(t_1) \cap C(t_2) \cap \ldots C(t_n)$
3:      for statement $s \in C$ do
4:          for test case $\in t_i$ do
5:              $pos[s, t_i]$ = all the positions of stmt $s \in S(t_i)$
6:          end for
7:      end for
8:      for statement $s \in C$ do
9:          for test case $\in t_i$ do
10:             $min\_instance[s, i]$ = -1
11:         end for
12:     end for
13:     for statement $s \in C$ do
14:         for test case $\in t_i$ do
15:             $min\_instance[s, i] =$
16:             $min(pos[s, t_i])$ s.t. $pos[s, t_i] >$
17:             $min\_instance[s, i]$
18:         end for
19:         $instances = instances \cup min\_instance[s]$
20:     end for
21:     while instances is not empty do
22:         $inst$ = pop(instances)
23:         for statement $s \in C$ do
24:             $inst_c$ = least instance $i$ of $s$ such that $i > inst$
25:             if $inst_c \,! = null$ then
26:                 $pred[inst_c] = pred[inst_c] \cup inst$
27:                 if $inst_c$ does not exist in instances && $inst_c$ was not processed
    before then
28:                     $instances = instances \cup inst_c$
29:                 end if
30:             end if
31:         end for
32:     end while
33:     starting from the failure statements in the graph, generate all possible paths
34: end procedure
```

test cases, each loop in each test case sequence is compressed to appear as one iteration. Loops are identified as any subsequence of program lines in the execution trace that is consecutively repeated more than once. Note that this does not affect the variable values reported to the developer as part of the tool output, as those values are extracted anyway for each iteration of the loop as part of the variable value extraction technique discussed in Section 3.2.6. Therefore, although the line appears once in the results, all possible variable values are still preserved.

Block-based abstraction Furthermore, another observation is that there are program lines in the sequence cover of each test case which always appear consecutively either within the same test case or across all the test cases. So, there is no need to build graph instances for each line individually while one instance can be created for all of them together representing a single block, and hence, save a lot of time and memory. Therefore, an efficient technique is developed for identifying consecutive lines of code that are shared between all the test cases. After extracting these common consecutive lines and representing them as individual blocks in the instance graph, those blocks are mapped back to their corresponding lines while outputting the results to the developer.

The approach for extracting common consecutive subsequences among all test cases is non-trivial. The first approach is constructing the graph containing all instances, and extracting paths whose instances are before each other by *exactly one position*, which guarantees that the resulting subsequences are consecutive in each test case. However, this approach does not work for tests with large sequences as building the graph is still very time-consuming, which defeats the purpose of efficiently building and processing the graph. On the other hand, the graph with a subset of the instances that are only consecutive to each other could not be built, as this still requires to consider all possible graph instances in order to find out whether they have consecutive neighbors or not.

In order to extract consecutive common subsequences efficiently, the developed approach starts from common subsequences within each test case separately, which significantly reduces the search space. For each test case, the suffix tree algorithm is used to extract all repeated consecutive subsequences in that test case, where the entire test case sequence is treated as a string, and each line in that sequence as a character in the suffix tree string. The output of that process is a set of subsequences which are repeated multiple times within the test case. Note that those subsequences vary in their length and number of repetitions, which affects their abstraction power, where subsequences with higher length and number of repetitions have more abstraction power over those with less. Those subsequences may also overlap, where applying one of them (i.e., using it to compress the test case into individual blocks) may invalidate the possibility of applying others. Therefore, to address these two issues, each subsequence is assigned a score, that is the result of multiplying its length by its number of repetitions, and hence, each subsequence is associated with a measure of its importance or abstraction power. This consecutive repeated subsequence detection process is applied for each sequence cover, and after extracting the repeated consecutive subsequences for each test case, the set of those repeated subsequences are intersected across all sequence covers to identify the blocks that appear in *all* sequences multiple times. The score of each subsequence in the intersection is updated to be the sum of its scores in the individual sequences to reflect its abstraction power relative to all sequences. Finally, the subsequences are sorted according to their scores and apply them in order. Note that we cannot substitute subsequences locally in each test case sequence without ensuring that the subsequence exists in other test case sequences as well, as this will hide lines underlying each block, which can be shared across all sequences, but cannot be seen when represented as a single block that does not necessarily appear in all test cases. Therefore, by making sure the blocks appear in all sequences, we know that the underlying lines match among those blocks in all sequences as well, and hence, are still be seen in the final output.

One final optimization is related to the suffix tree construction algorithm, which may not scale to very large test case sequences with tens of thousands of lines of code. The test case sequence is partitioned to a number of partitions, each with a smaller number of lines in the sequence, apply the suffix tree algorithm to each partition, and finally union the resulting subsequences from each partition and update their scores accordingly.

3.2.4.2 Extracting the most important subsequences

Traversing all the paths starting from the throwing exception node in large graphs is time consuming, results in a large number of paths, and may cause out of memory exceptions, while we are only interested in just one sequence to present to the user, which should highly likely contain the trace back from the throwing line to the source line. Therefore, in the developed algorithm, instead of traversing all paths, scores are assigned to the nodes in the graph according to their degrees, which indicate the likelihood of those nodes participating in faulty sequences, and then the path that passes along the nodes with the highest scores is generated.

3.2.5 Hybrid Dynamic/Static Analysis

Although the number of the statements in the output common subsequence can be small after applying the abstraction techniques discussed above, they can still include some statements that do not have any effect on failing statement. These statements may be a source of distraction to the developer while backtracking the common subsequence to find the source of the error.

To backtrack the lines, call and data dependency information of the program are employed. Obtaining the common subsequences between the execution traces has an advantage. The existence of those common subsequences enables us to avoid expanding the entire dependency graph of the entire program. Therefore, in order to extract these dependencies efficiently, the common subsequences is utilized to restrict the search space of the dependency graph. This is done by generating the call graph for only the subset of the classes and the methods that appear in the common statement subsequence. For each method, the def-use graph is built of their statements. This graph contains a node for each statement and there is an edge between two nodes if a control can flow from one node to the other one. At the same time, the dependency of the failure statement is extracted and the dependency is restricted to the subset of statements appearing in the common subsequence. There are three main algorithms that are used to achieve this type of hybrid dynamic/static analysis:

1) In the first algorithm, the call graph of the methods that the statements of the common subsequence belong to is generated. This procedure is stated in Algorithm 2 .

Algorithm 2 Call Graph Generation Algorithm

```
1: procedure GET_CALL_GRAPH(Sequence $q$)
2:     for statement $s \in q$ do
3:         $classes = classes \cup s.class\_name$
4:     end for
5:     $g = Generate\_Call\_Graph(classes)$
6:     return $g$
7: end procedure
```

2) To get the def/use *chain* of an individual statement, the variables and the methods that are referenced in that statement are extracted. Then, for the referenced variables, the statements that assign these variables are added to the *chain*. The variables could be passed as method parameters, in which case the assigning statement is found in the calling method which passes the parameter value, and hence step 1 is used to obtain the calling-callee information. For the referenced methods, their return statements are added(if any) to the chain, in addition to any statements that change non-local variables. This procedure is listed in Algorithm 3.

3) Now, starting from the failure statement, its dependencies are got as outlined in the previous step and for each dependency (element of the *chain*), the algorithm recurses on it *only if* it is part of the common subsequence obtained in Section 3.2.4.

This procedure is listed in Algorithm 4. The inputs for that procedure are the call graph, the failure statement, the *in* sequence, which is initialized to the common subsequence generated as discussed in Section 3.2.4, and the *out* sequence, which is initialized to ϕ for the first call.

Algorithm 3 Def/Use Chain Extraction Algorithm

```
procedure GET_DEFUSE_CHAIN(Statement $s$, Sequence $q$)
    $g = Get\_Call\_Graph(q)$
    $uses = get\_referenced\_methods\_and\_vars(s)$
    for $e \in uses$ do                         ▷ $e$ could be a variable or method
        if $e$ is a variable then
            if $e$ is passed as the enclosing method $m$ parameter then
                $m' = g.get\_calling(m)$
                $e'$ = variable corresponding to $e$ in $m'$
                $s' = get\_assigning\_stmt(e', m')$
            else
                $s' = get\_assigning\_stmt(e, m)$
            end if
            $chain = chain \cup s'$
        else                                    ▷ $e$ is a method
            $chain = chain \cup return\_stmts(e)$
            $chain = chain \cup$
                    $stmts\_assigning\_non\_local\_vars(e)$
        end if
    end for
    return $chain$
end procedure
```

To implement this hybrid analysis, Soot [82] is used. Soot is a software engineering tool for analyzing and optimizing Java programs. It provides program call graph and intra-procedural data flow analysis. For the intra-procedural data flow analysis, it operates on a control-flow graph called UnitGraph.

Algorithm 4 Dependency Extraction Algorithm

```
1: procedure GET_DEPENDENCIES(Stmt $s$, Call_Graph $g$, Sequence $in$, Sequence $out$)
2:     $out' = out \cup s$
3:     $chain = Get\_DefUse\_Chain(s)$
4:     for $s' \in chain$ do
5:         if $s' \in in$ then
6:             $Get\_Dependencies(s, g, in, out')$
7:         end if
8:     end for
9:     return $out'$
10: end procedure
```

3.2.6 Remote Debugging

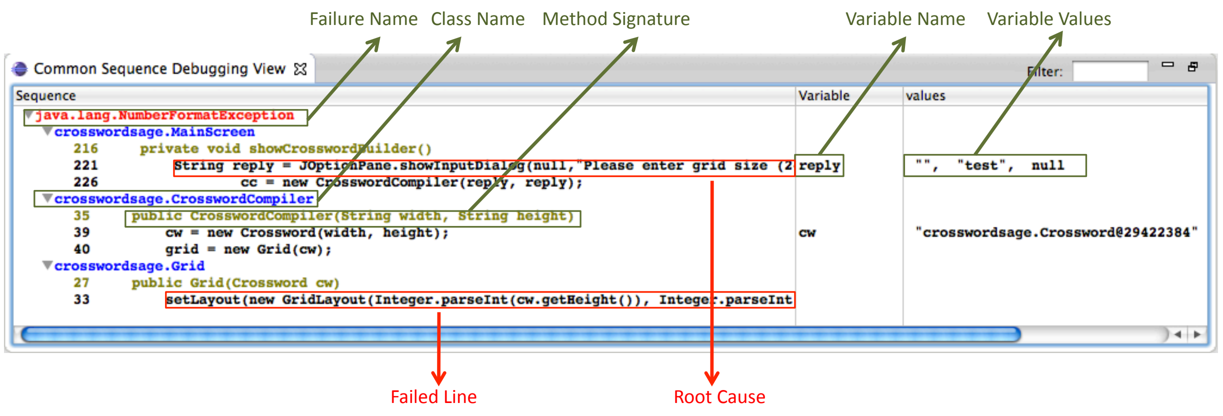
This step is responsible of extracting the variable values. It takes the common subsequence and the application source code as inputs and outputs each line attached with each variable values. Since in this research Java applications are used, the Java Debugger (JDB) [83] command line debugging tool is used to automate extracting the variable values. JDB is a full-fledged Java debugging tool that is based on Java Platform Debugger Architecture that provides inspection and debugging of a local or remote Java virtual machines. It allows setting breakpoints, stepping, suspending on exceptions, all through a command line interface. A script that automatically sets debugging breakpoints at the lines of the program constituting the common subsequence, steps over those breakpoints, and dumps the values of the variables appearing in those lines is written. Variable values in a line can only be retrieved after the line has been fully executed, including any methods that it may call. If those methods have breakpoints too, which is usually the case, we keep track of the method call stack, in order to remember a line when we return back to its method after its execution, as JDB does not simply return to the same line after it exhausts the entire call stack, and at the same time, does not necessarily return to the line next to it in cases like if statements or loops. Therefore, there is no built-in way in this case to know at which point variables values can be already extracted so that they express the state of the program directly after executing a particular line, and hence, a new approach is implemented on top of JDB.

3.3 Implementation

A tool is implemented to enable developers to use the developed approach. The implementation of this tool is discussed in this section.

3.3.1 Sequence Debugging View

In this plugin, a new menu item is added to the package explorer called "Sequence Debugging". This menu item is displayed when the user right clicks a project in the package explorer. It has a sub menu called "Show Sequence Debugging View". When the user clicks this submenu, the "Common Sequence Debugging View" is opened if the project has failures. The view contains all the project failures along with with the common subsequence for each failure. If the project does not contain failures, a dialog message appears stating that there are no failures in this project. As can be seen in Figure 3.4 the subsequence presented to the developer ends with the failed line. The view organizes the information as a hierarchy, in which the top level is the failure name, followed by the class names at the second level, and the



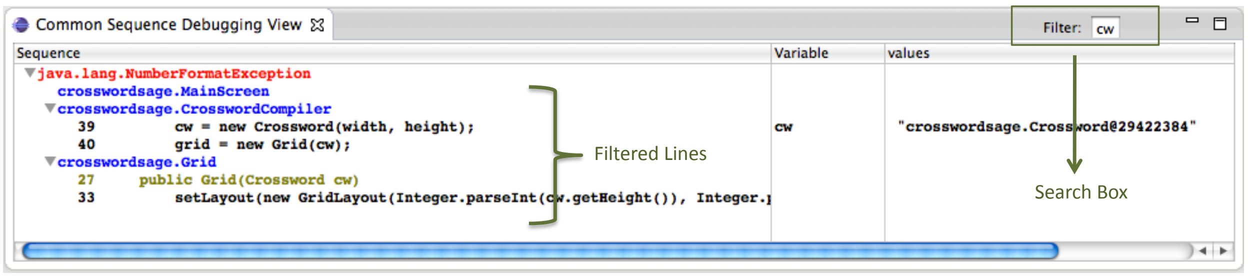
```
Sequence                                                                          Variable   values
java.lang.NumberFormatException
  crosswordsage.MainScreen
    216    private void showCrosswordBuilder()
    221        String reply = JOptionPane.showInputDialog(null,"Please enter grid size (2  reply   "",  "test",  null
    226            cc = new CrosswordCompiler(reply, reply);
  crosswordsage.CrosswordCompiler
    35     public CrosswordCompiler(String width, String height)
    39         cw = new Crossword(width, height);                                          cw      "crosswordsage.Crossword@29422384"
    40         grid = new Grid(cw);
  crosswordsage.Grid
    27     public Grid(Crossword cw)
    33         setLayout(new GridLayout(Integer.parseInt(cw.getHeight()), Integer.parseInt
```

Figure 3.4: Disqover shows the sequence of lines for Crossword Sage NumberFormatException Exception

individual lines at the bottom level. This individual line may be a method signature with dark yellow color or a line code with black color. The user can expand or collapse the subsequence at any level. Furthermore, If any of the lines that are at the bottom level is an assignment statement, the line is associated with all the values that the assigned variable took during the execution of all the failed test cases, so that the developer can correlate those values with the failure.

3.3.2 Search Box

The view contains a search box, which allows the developer to search for any keyword in the sequence, so that the developer can conveniently navigate through the sequence and quickly see where variables are defined/used. If the developer writes any keyword in the search box, only the lines that contains the keyword will stay and all the other lines will disappear. Figure 3.5 shows the results of looking for “cw” keywords in the sequence in Figure 3.4.



```
Sequence                                                                      Variable   values
▼java.lang.NumberFormatException
    crosswordsage.MainScreen
  ▼crosswordsage.CrosswordCompiler
      39          cw = new Crossword(width, height);                           cw         "crosswordsage.Crossword@29422384"
      40          grid = new Grid(cw);
  ▼crosswordsage.Grid
      27      public Grid(Crossword cw)
      33          setLayout(new GridLayout(Integer.parseInt(cw.getHeight()), Integer.
```

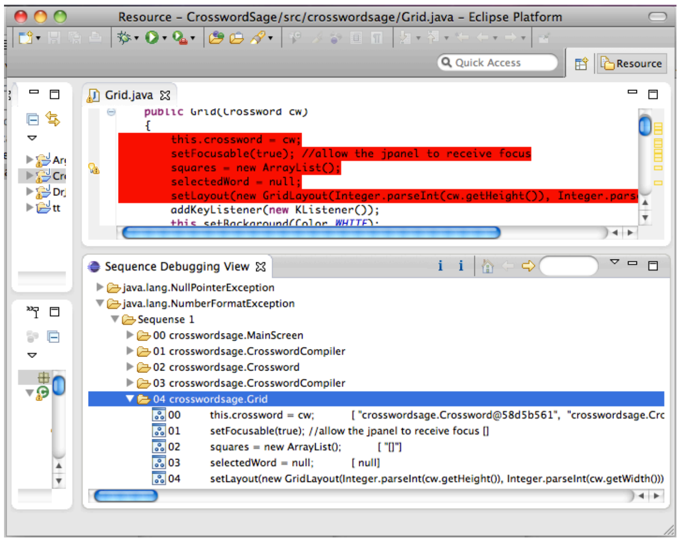
Figure 3.5: The results of searching for "cw" keyword in the sequence

3.3.3 Source Code Highlighting

Since the plugin is part of the Eclipse IDE, the user can navigate or run the source code at any time. Also, the user can see an individual line or a group of lines that are under one class in the subsequence in their actual location in the source code by double clicking the line or the class name (respectively) in the view. If the user double clicks an individual line, the plugin opens the file containing this line in the editor, sets the cursor position at this line, and highlights that line with a green color. If the user double clicks a class, the plugin opens the file containing this class, highlights all the lines that are under this class in the sequence with a red color and sets the cursor position at the first line as can be seen in Figure 3.6.

3.4 Conclusions

In this chapter, Disqover approach is discussed in detail. Disqover contains of 5 main steps: The Execution Trace & Logs Extraction, which extracts test case execution traces and test case execution logs; Test cases Partitioning, which groups the test cases according to the type and the location of the faults caught by the test


Resource – CrosswordSage/src/crosswordsage/Grid.java – Eclipse Platform
Quick Access
Resource
Grid.java
Sequence Debugging View

```
public Grid(Crossword cw)
{
        this.crossword = cw;
        setFocusable(true); //allow the jpanel to receive focus
        squares = new ArrayList();
        selectedWord = null;
        setLayout(new GridLayout(Integer.parseInt(cw.getHeight()), Integer.pars
        addKeyListener(new KListener());
        this.setBackground(Color.WHITE);
```

```
▶ java.lang.NullPointerException
▼ java.lang.NumberFormatException
  ▼ Sequense 1
    ▶ 00 crosswordsage.MainScreen
    ▶ 01 crosswordsage.CrosswordCompiler
    ▶ 02 crosswordsage.Crossword
    ▶ 03 crosswordsage.CrosswordCompiler
    ▼ 04 crosswordsage.Grid
        00      this.crossword = cw;          [ "crosswordsage.Crossword@58d5b561",  "crosswordsage.Crc
        01      setFocusable(true); //allow the jpanel to receive focus []
        02      squares = new ArrayList();          [ "[]"]
        03      selectedWord = null;          [ null]
        04      setLayout(new GridLayout(Integer.parseInt(cw.getHeight()), Integer.parseInt(cw.getWidth()))
```

Figure 3.6: Highlighted lines with red in the class that appear in the sequence

cases execution; Common Subsequence Extraction, which extracts a common subsequence of lines found in the trace of the failing test cases; Hybrid Dynamic/Static Analysis, which uses both static information coming from code dependency analysis and dynamic information coming from the common subsequence from the pervious step to provide the dependency of the failed line within the common subsequence; Remote Debugging, which provides the values of the variables that included in the subsequence that explains the fault. The chapter concludes by describing the tool that is generated to facilitate using the resulting common subsequence to the user. In the next chapter, 3 case studies are presented to show how a developer can use the resulting common subsequence to find the source of the error.

Chapter 4: Using Disqover

In this chapter, three case studies of three errors in three different applications are discussed. The applications are Crossword Sage, ArgoUML, and Freemind. The application sizes vary from thousands of lines of code to hundreds of thousands of lines of code. Throughout the case studies, concrete examples of the developed approach's capability are shown to find and identify root causes of bugs, and ways of showing them to the developer in a self-explained manner are presented. Also, the final output of the developed approach for each error is shown, along with the number of lines to inspect in that output. Since all the applications that are used here are GUI-applications, an automated test case generation called GUITAR [56] is utilized to generate a large number of test cases. Furthermore, the developed common subsequence algorithm is not applied to the test case traces only, but also to the test cases (which contain a sequence of events).

4.1 Case Study 1: Crossword Sage

In order for the user to create a new crossword puzzle, he/she needs to click on the File menu and choose the New Crossword menu item. Then, the application asks the user to input the size of the puzzle through a dialog box. When the user

```
java.lang.NumberFormatException:  For input string:  ""

at java.lang.NumberFormatException.forInputString

(NumberFormatException.java:48)

at java.lang.Integer.parseInt(Integer.java:470)

at java.lang.Integer.parseInt(Integer.java:499)

at crosswordsage.Grid.<init>(Grid.java:33)

at crosswordsage.CrosswordCompiler.<init>

(CrosswordCompiler.java:40)

at crosswordsage.MainScreen.showCrosswordBuilder

(MainScreen.java:226)

at crosswordsage.MainScreen.access$3(MainScreen.java:216)

at crosswordsage.MainScreen$MenuListene

.actionPerformed(MainScreen.java:423)
```

Figure 4.1: Crossword Sage NumberFormatException

inputs a numeric number between 2 and 20, the application creates an empty grid to allow the user to start building his/her crossword puzzle.

Normally, if the user enters a non-numeric value as the size of the puzzle, an error dialog box should appear warning the user about the wrong input format and asks the user to enter another input value. However, in this application when the user enters a non numeric value in the dialog box, the application crashes with a NumberFormatException as can be seen in Figure 4.1.

Now, we discuss how using Disqover, the developer can get the concise sequence of statements explaining the error as shown in Figure 4.2. The steps performed by Disqover are listed below. All of those steps are performed automatically.

Step 1: The process starts by generating and running the application test suite using the automated testing framework GUITAR [56]. The output of this step is 347 test cases and their execution logs.

Step 2: At the same time, the test case execution traces are extracted using the modified Cobertura during the test cases execution.

Step 3: Then, the test cases that reveal the NumberFormatException are grouped together using the technique discussed in Section 3.2.2. This step detects 41 test cases that fail because of the exception that is shown in Figure 4.1.

Step 4: Next, the common subsequence algorithm that is discussed in Section 3.2.3 is applied to the 41 test cases. This step returns the common events that cause the NumberFormatException, which are File $\rightarrow$ New Crossword $\rightarrow$ Cancel.

Step 5: Then, for each event in the common events, the common subsequence algorithm is applied again for the event code to get the common statement subsequence.

Step 6: Then, the hybrid dynamic/static analysis discussed in Section 3.2.5 gets the dependency of the line that throws the exception. The output of this step is shown in Figure 4.2.

Step 7: Finally, the remote debugger that is explained in Section 3.2.6 is applied to the final output to get the variable values of each assignment statement.

In order for the developer to find the source of the error using the developed

```
1 private void showCrosswordBuilder()

2 String reply = JOptionPane.showInputDialog(null,"Please enter grid size (2-20)...", null);

3 cc = new CrosswordCompiler(reply, reply);



4 public CrosswordCompiler(String width, String height)

5 cw = new Crossword(width, height);



6 public Crossword(String width, String height)

7 isEditable = true;

8 this.width = width;

9 this.height = height;

10 words = new ArrayList();



11 public CrosswordCompiler(String width, String height)

12 grid = new Grid(cw);



13 void Grid(Crossword cw)

14 setLayout(new GridLayout(Integer.parseInt(cw.getHeight()), Integer.parseInt(cw.getWidth())));
```

Figure 4.2: Sequence Explaining Fault for Crossword Sage

approach, only the following activities will take place:

- The last line in the sequence is line 14 (setLayout(new GridLayout(Integer.parseInt(cw.getHeight()), Integer.parseInt(cw.getWidth())));). This line is the line that throws the NumberFormatException. From this line, the developer can conclude that this exception results from applying the Integer.parseInt() function to a non-numeric value.

- This non-numeric value may be assigned to either the height or the width variables of the cw object (because the Integer.parseInt() appears twice in the line).

- Now, the developer can go backwards in the subsequence and see that the cw object comes from the method parameter as shown in line 13 (public Grid(Crossword cw)).

- Going backward, there is a line in the sequence that creates new object from the Grid class and passes the Crossword object as a parameter as can be seen at the line 12 (grid = new Grid(cw)).

- By going backwards further, the developer can see that the Crossword object cw is created at the line 5 (cw = new Crossword(width,height);) and the width and the height are passed as parameters.

- These width and height variables are passed to the function through the Crossword- Compiler constructor arguments at line 4(public CrosswordCompiler(String width, String height)).

- Finally, by going backwards at crosswordsage.MainScreen class, the developer can see that these parameters are passed as arguments when creating a new instance of cross- wordsage.CrosswordCompiler at line 3 (cc = new CrosswordCompiler(reply, reply)) and these arguments are both initialized by the variable reply which takes string values in line 2 (String reply = jOptionPane.showInputDialog(null, Please enter grid size (2 - 20)..., null);).

As can be seen, the developer needs only to inspect 6 lines to find the root cause of the bug. Those 6 lines are self-contained, and do not require prior knowledge of the code, as the problem can be seen by just inspecting those lines.

4.2 Case Study 2: ArgoUML

When the user exports the graphics using the Export All Graphics menu item, and saves them to a file, if the user enters a directory location that does not exist on disk, the application throws a FileNotFoundException as can be seen in Figure 4.3, and exits the Save dialog without notifying the user of the problem. The error is thrown when the application is actually trying to save the file, while it is originated when the user chooses the improper directory.

The output of Disqover after being applied to this exception is shown in Figure 4.4. To obtain that output, Disqover, performs all the following steps automatically.

Step 1: The process starts by applying GUITAR framework to ArgoUML. This step generates and runs 6317 test cases.

Step 2: At the same time, the trace execution extraction in Section 3.2.1

```
java.io.FileNotFoundException:  /crash/crash/ClassDiagram.png (No such file or directory)

at java.io.FileOutputStream.open(Native Method)

at java.io.FileOutputStream.<init>(FileOutputStream.java:179)

at java.io.FileOutputStream.<init>(FileOutputStream.java:131)

at org.argouml.uml.ui.ActionSaveAllGraphics.saveGraphicsToFile

(ActionSaveAllGraphics.java:230)

at org.argouml.uml.ui.ActionSaveAllGraphics.trySaveDiagram

(ActionSaveAllGraphics.java:161)

at org.argouml.uml.ui.ActionSaveAllGraphics.trySave

(ActionSaveAllGraphics.java:130)

at org.argouml.uml.ui.ActionSaveAllGraphics.trySave

(ActionSaveAllGraphics.java:106)

at org.argouml.uml.ui.ActionSaveAllGraphics.actionPerformed

(ActionSaveAllGraphics.java:98)
```

Figure 4.3: ArgoUML FileNotFoundException

finds out that the average number of lines per test case trace is 221795 lines. This large number of lines makes the manual debugging impractical.

Step 3: From the 6317 test cases, the partitioning step in Section 3.2.2 finds out that only 122 test cases reveal the FileNotFoundException exception that is shown in Figure 4.3.

Step 4: Now, after applying the common subsequence algorithm that is discussed in Section 3.2.3 on the 122 failed test cases, it detects that the common events that cause the exception are File → Export All Graphics... → Save As: → Save.

Step 5: Then, for each event in the common events, the common subsequence algorithm is applied again for the event code to get the common statement subsequence. This step reduces the number of lines that need to be inspected to 234 lines.

Step 6: Then, the hybrid dynamic/static analysis in Section 3.2.5 gets the final common statement subsequence. The number of lines to be inspected is reduced again to be 31 lines. A relevant subset of those lines is shown in Figure 4.4.

Step 7: Finally, the remote debugging that is explained in Section 3.2.6 gets the variable values of each assignment statement in the final sequence.

In order for the developer to find the source of the error using our developed approach, only the following activities will take place:

- The last line in the sequence is line 15 (fo = new FileOutputStream(theFile)). This line is the line that throws the FileNotFoundException. From this

```
public void actionPerformed( ActionEvent ae )

trySave( false );



public boolean trySave(boolean canOverwrite)

return trySave(canOverwrite, null);



public boolean trySave(boolean canOverwrite, File directory)

Project p = ProjectManager.getManager().getCurrentProject();

File saveDir = (directory != null) ?  directory :  getSaveDir(p);

for (ArgoDiagram d :  p.getDiagramList())

okSoFar = trySaveDiagram(d, saveDir);



protected boolean trySaveDiagram(Object target, File saveDir)

File theFile = new File(saveDir, defaultName + "." + SaveGraphicsManager.getInstance().getDefaultSuffix());

SaveGraphicsAction cmd = SaveGraphicsManager.getInstance().

getSaveActionBySuffix(SaveGraphicsManager.getInstance()

.getDefaultSuffix());

boolean result = saveGraphicsToFile(theFile, cmd);



private boolean saveGraphicsToFile(File theFile, SaveGraphicsAction cmd)

fo = new FileOutputStream( theFile );
```

Figure 4.4: Sequence Explaining Fault for ArgoUML

line, the developer can conclude that this exception results from an attempt to output stream to a file ”theFile” and this file does not exist.

- Now, the developer can go backwards in the subsequence and see that the “theFile” variable comes from the method parameter as shown in line 14 (private boolean saveGraphicsToFile(File theFile, SaveGraphicsAction cmd)).
- Going backward, there is a line in the sequence that calls the saveGraphicsToFile function as can be seen at the line 13 (boolean result = saveGraphicsToFile(theFile, cmd)).
- Since the developer is investigating the variable ”theFile”, we can see that this variable is defined at line 11 (File theFile = new File(saveDir, defaultName + ”.” + SaveGraphicsManager.getInstance().getDefaultSuffix())).
- This line uses a ”saveDir” variable that is passed as the method parameter as can be seen at line 10 (protected boolean trySaveDiagram(Object target, File saveDir)).
- By going backwards further, the developer can see that the function ”trySaveDiagram” is called at line 9 (okSoFar = trySaveDiagram(d, saveDir)).
- Finally, by going backwards at ”trySave” function, the developer can see that the ”saveDir” variable is set at line 7 (File saveDir = (directory != null) ? directory : getSaveDir(p)) to non existing location ”/crash/crash”.

As it can be seen, the total number of lines that are needed to be inspected are only 7.

```
java.lang.NullPointerException at freemind.modes.ControllerAdapter.remove

(ControllerAdapter.java:339)

at freemind.modes.ControllerAdapter.delete

(ControllerAdapter.java:297)

at freemind.modes.ControllerAdapter$RemoveAction

.actionPerformed(ControllerAdapter.java:671)
```

Figure 4.5: Freemind NullPointerException

4.3 Case Study 3: Freemind

The exception shown in Figure 4.5 is thrown by the application when the user attempts to remove a node from the Freemind graph. This is because the selected node is set to null, which causes the application to throw the NullPointerException.

To get the output that is shown in Figure 4.6, the same steps that are used in the previous case studies are followed.

The output of Disqover for this exception is shown in Figure 4.6. To obtain that output, the following steps take place automatically.

Step 1: The process starts by applying GUITAR framework to Freemind, which generates and runs 3055 test cases.

Step 2: At the same time, the trace execution extraction in Section 3.2.1 finds out that the average number of lines per test case trace is 14806 lines.

Step 3: The partitioning in Section 3.2.2 finds out that only 417 test cases are able to reveal NullPointerException exception that is shown in Figure 4.5.

Step 4: Now, after applying the common subsequence algorithm that is discussed in Section 3.2.3, it detects that the common events that cause the exception are Edit → Node → Remove Node.

Step 5: Then, for each event in the common events, the common subsequence algorithm is applied again for the event code to get the common statement subsequence. This step reduces the number of lines to be inspected to 56 lines.

Step 6: Then, the hybrid dynamic/static analysis in Section 3.2.5 gets the final common statement subsequence. The number of lines to be inspected is shrunk down to 6 lines only as can be seen in Figure 4.6.

Step 7: Finally, the remote debugging that is explained in Section 3.2.6 gets the variable values of each assignment statement in the final sequence.

Only the following activities are performed by the developer:

- The last line in the sequence is line 11 (if (!node.isRoot())). This line is the line that throws the NullPointerException. From this line, the developer can conclude that this exception results from an attempt to access the isRoot() function using null object, which is the "node" object.

- Now, the developer can go backward in the subsequence and see that the "node" object is passed as the method parameter as shown in line 10 (public void remove(NodeView node)).

- Going backward, the developer can figure out that there is a line in the sequence that calls the "remove" function as can be seen at line 5 (get-

```
1 public void actionPerformed(ActionEvent e)

2 NodeView selected = null;

3 delete(selected);



4 void delete(NodeView node)

5 getMode().getModeController().remove(node);



6 protected Mode getMode()

7 return mode;



8 public ModeController getModeController()

9 return modecontroller;



10 public void remove(NodeView node)

11 if (!node.isRoot())
```

Figure 4.6: Sequence Explaining Fault for Freemind

Mode().getModeController().remove(node)).

- Again, by going backwards in the sequence, the developer can figure out that the "node" object that is passed in line 5 is passed as the method parameter as can be seen at line 4 (void delete(NodeView node)).
- By going backwards further, the developer can see that the "delete" function is called by line 3 (delete(selected)).
- Finally, by going backwards, the developer can detect that the "selected" variable that is passed in line 3 is set to null at line 2 (NodeView selected = null) in the sequence.

The total number of lines that are needed to be inspected are 6.

4.4 Conclusion

This chapter discusses three case studies for three different errors in three different GUI applications. Each case study starts by describing the error. Then, it lists the steps that are done by Disqover to automatically generate the common subsequence that leads to the root cause of the error. Finally, it discusses how using the output a developer can get the source of the error. In the next chapter, the experiments that are performed to evaluate the developed approach are discussed.

Chapter 5: Evaluation

To evaluate Disqover, a set of experiments are performed to 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 the developed algorithms lead to a more efficient evaluation of the common subsequence?

RQ2 How does hybrid static/dynamic analysis affect the length of the output common subsequence?

This section starts by describing the subject applications and stating some of their code complexity metrics. Then the types of faults in those applications, and the scenarios that result in those faults are presented. Finally, the experiments are described in detail. In those experiments, the above questions are answered using a variety of metrics such as the number of lines to examine, algorithm execution time, and the number of lines of code in the output subsequence.

5.1 Subject Applications and Faults

To evaluate Disqover, 7 open source applications, which consist of 4 GUI applications, and 3 non-GUI applications are used. The GUI applications are ArgoUML [84], Crossword Sage [85], Buddi [86], and Freemind [87]. The non-GUI applications are Commons Math [88], Joda-Time [89], and Commons Lang [90]. All 3 applications are part of the defects4j suite [91]. Defects4j is a database of reproducible and isolated real software faults, and features a framework to enable controlled studies in software testing research. Table 5.1 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 it can be seen, 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).

Both seeded and real faults are used to evaluate Disqover. Below, the faults used with each of the applications are described. Table 5.2 summarizes them. Below, the faults used with each of the applications are described.

ArgoUML In this application, a real fault is used. When the user exports the graphics using the "Export All Graphics" menu item, and saves them to a file, if the user enters a directory location that does not exist on disk, the application throws a FileNotFoundException, and exits the Save dialog without notifying the user of the problem. The error is thrown when the application is actually trying to

App	LOC	# Classes	# Methods
ArgoUML	152513	1787	13117
Crossword Sage	3072	34	238
Buddi	20922	257	1580
Freemind	7702	136	788
Commons Math	85000	678	5441
Joda-Time	28000	208	3501
Commons Lang	22000	150	1358

Table 5.1: Application code complexity metrics

save the file, while it is originated when the user chooses the improper directory.

Crossword Sage In this application, two seeded errors are used. For the first error, when the user creates a new crosswords puzzle, the application asks the user to input the size of the puzzle through a dialog box. Normally, if the user inputs a non-numeric value, the application tries to parse that value, and catches the resulting NumberFormatException and informs the user with the error. To seed the first fault, the code is modified by removing the try/catch block, and letting the application store the return value of the dialog box in a String, which allows the application to proceed normally for a while, until it crashes when trying to actually construct the new puzzle. For the second error, when the user selects the menu item to load a previously saved crossword from a file, a dialog box is shown for the user to select the crossword file. If the user presses Cancel, the dialog box disappears and the user returns to the main screen. The code is modified so that if Cancel is pressed, the method returning the crossword object returns a null value. At some point, the application crashes because it tries to construct a null crossword.

Buddi In this application, a real fault is used. If the user selects the "Save As" menu item, and then enters a directory name that does not exist, the application throws a FileNotFoundException and continues quietly, instead of informing the user of the problem and that it did not actually save the file.

Freemind In this application, two real faults are used. If the user selects the "Save As" or "Open" menu items, and then enters a directory name that does not exist, the application throws a FileNotFoundException and continues quietly, instead of informing the user of the problem and that it did not actually save the file. Furthermore, a new fault is seeded in the application. When a user removes a selected node, the application throws NullPointerException. This is done by setting a selected node to Null.

All the defects4j bugs are assertions that fail because the program execution behaves in an unexpected way.

Table 5.3 summarizes the number of faulty versions used for each application and the number of passing and failing test cases for each fault. The non-GUI application faults are randomly selected from the defects4j repository of each appli-

Application	Fault	Event	Exception	**Seeded or real?**
ArgoUML		Export	FileNotFoundException	real
Crosswordsage	Crosswordsage1 Crosswordsage2	Load crossword New crossword	NullPointerException NumberFormatException	seeded seeded
Buddi		Save As	FileNotFoundException	real
Freemind	Freemind1 Freemind2 Freemind3	Save As Open Remove Node	FileNotFoundException FileNotFoundException NullPointerException	real real seeded
Commons Math	Bug 16 Bug 35_A Bug 35_B Bug 36		assertion assertion assertion assertion	real real real real
Joda-Time	Bug 5 Bug 7 Bug 10 Bug 14		assertion assertion assertion assertion	real real real real
Commons Lang	Bug 8 Bug 30_A Bug 30_B Bug 30_C Bug 34 Bug 57 Bug 57 Bug 61		assertion assertion assertion assertion assertion assertion assertion assertion	real real real real real real real real

Table 5.2: Application faults

Application	Fault	# Passing TCs	# Failing TCs
ArgoUML	Export All Graphics	100	22
Crosswordsage	Load crossword To Edit *	265	83
	New crosswords *	330	38
Buddi	Save As	100	44
Freemind	Save As	301	418
	Open	301	339
	Remove Node *	301	339
Commons Math	Bug 16	2	2
	Bug 35_A	1	2
	Bug 35_B	1	2
	Bug 36	2	4
Joda-Time	Bug 5	2	3
	Bug 7	2	2
	Bug 10	2	2
	Bug 14	2	6
Commons Lang	Bug 8	2	2
	Bug 30_A	2	2
	Bug 30_B	2	4
	Bug 30_C	2	2
	Bug 34	2	2
	Bug 57	2	11
	Bug 57	2	2
	Bug 61	2	2

Table 5.3: Application fault test cases

cation. For all the applications from the defects4j repository, the test cases reported in defects4j are used as well, while for the other applications (GUI application), GUITAR [32] is used to generate their test cases. As it can be noticed, the number of test cases that are used for defects4j faults is small as they are not automatically generated as it is the case with the GUI applications.

In the following subsections, the experiments that are performed to evaluate Disqover are discussed. An experiment is performed to measure the number of statements to examine by developers before reaching the root cause in comparison to other techniques such as MUSE [57], Op [49], Tarantula [48] and Fonly [58]. Moreover, an experiment is performed to measure the effect of choosing diverse input test cases on the size of the output common subsequence. Furthermore, the developed abstraction techniques are evaluated to measure their effect on the length of the input sequence covers. Moreover, the effect of the number of test cases over both the running time and the length of resulting common subsequence is evaluated by comparing the developed approach to multiple baselines. The results show that Disqover significantly reduces the number of lines needed to discover the source of the fault, and they also show that the effectiveness of the developed sequence cover abstraction techniques on reducing the computation time and the length of the output common subsequences, especially, for the computationally intensive ones.

5.1.0.1 Comparison with other approaches

In this experiment, RQ1 is addressed by comparing Disqover with four statement ranking techniques, MUSE [57], Op [49], Tarantula [48] and Fonly [58]. MUSE [57] and Op [49] are the most recent state-of-the-art approaches. Tarantula is chosen because Jones et al. [48] show that Tarantula outperforms many other ranking techniques in fault localization. Furthermore, Fonly is chosen, as it is the only technique that uses failed test cases only in fault localization like the developed technique. All techniques rank the program statements according to their suspiciousness of being the root cause using a scoring formula that assigns a score to each statement.

MUSE creates mutants for each statement according to different characteristics and calculates the statement suspiciousness as:

$$suspiciousness(s) =$$
$$\frac{1}{|mut(s)|} \sum_{m \in mut(s)} (\frac{|f_p(s) \cap p_m|}{|f_p|} - \alpha \frac{|p_p(s) \cap f_m|}{|p_p|})$$

, where $mut(s)$ is the number of mutants that are generated for a statement s, $\frac{|f_p(s) \cap p_m|}{|f_p|}$ is the proportion of tests that failed on P but pass on a mutant m that mutates s over tests that failed on P, $\frac{|p_p(s) \cap f_m|}{|p_p|}$ is the proportion of tests that passed on P but fail on a mutant m that mutates s over tests that passed on P. α is defined as

$$\alpha = \frac{f2p}{|mut(P)| \cdot |f_p|} \cdot \frac{|mut(P)| \cdot |p_p|}{|p2f|}$$

, where $f2p$ and $p2f$ are the number of test result changes from failure to pass and vice versa between before and after all mutants of P, $mut(P)$ is the number of mutants that are generated for all the statements of P. Op calculates the statement suspiciousness as:

$$O^p = a_{ef} - \frac{a_{ep}}{P+1}$$

, where a_{ef} is the number of failed test cases that execute s, $a_e p$ is the number of passed test cases that execute s, and P is the total number of passed test cases. The suspiciousness of a statement e according to the Tarantula method is calculated as:

$$suspiciousness(s) = \frac{\frac{failed(s)}{totalfailed}}{\frac{passed(s)}{totalpassed} + \frac{failed(s)}{totalfailed}}$$

, where $failed(s)$ is the number of failed test cases that execute s and $passed(s)$ is the number of passed test cases that execute s. On the other hand, suspiciousness of statements according to the Fonly technique is calculated as:

$$suspiciousness(s) = \frac{\sum_{c \in D}[c(Y(c) - Y(0))] \times c_{max} / \sum_{c \in D} c^2}{\sqrt{\sum_{c \in D}(Y(c) - Y(0))^2 - (\sum_{c \in D}[c(Y(c) - Y(0))])^2 / \sum_{c \in D} c^2}}$$

, where c is the number of times in which a test case executes s, $Y(c)$ is the number of test cases that executes s c times and $Y(0)$ is the number of test cases that never execute s.

The four approaches are implemented and their formulas are used to rank statements of the applications under test. Regarding the mutants, μJava [92], which is a mutation system for Java programs is used. It automatically generates mutants for both operator mutation testing and class-level mutation testing. It is modified to generate operator mutants for the statements that are executed by the failed test cases.

To compare the developed system to other systems, a metric that quantifies the "number of inspected statements" until the source is found is used. For the developed 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 the other approaches, 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, half of them (to express the expectation of inspecting the root cause statement if statements with equal score are randomly ordered) are counted. We note that even with this type of comparison, developed approach 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 four approaches, the developer will probably have to carry out the task of understanding the sequence causing the bug of *each* suggested statement on his/her own. Therefore, the developed approach produces a number of statements that explain an individual root cause, while other systems produce a number of disconnected statements that are missing their explanation.

In this experiment, all the bugs that are listed in Table 5.3, which also shows the number of passing and failing test cases are used with every application. Results are presented in Table 5.4. The results of MUSE are omitted because we tried it on a number of faults, and found that $f2p$ value is always either zero, or only a very small fraction of all the test cases, which leads to zero or extremely small scores of all the statements. Furthermore, after spending several hours (5 on average) processing 517 mutants per application on average, the root cause was either nonexistent in the output or existent with a very poor score. Compared to other approaches, it is found that the developed system leads to a significantly smaller number of statements. For example, on average the number of statements that need to be inspected by the developed 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

Application	Fault	Disqover	Op	Tarantula	FOnly
ArgoUML	Export All Graphics	12	6498	121	5156
Crosswordsage	Load crossword To Edit	11	231	12	230
	New crosswords	10	5	27	83
Buddi	Save As	5	73	5937	817
Freemind	Save As	7	10	32	487
	Open	6	13	104	1059
	Remove Node	5	2	49	428
Commons Math	Bug 16	3	151	108	75
	Bug 35_A	1	4	9	5
	Bug 35_B	1	4	12	5
	Bug 36	1	178	156	178
Joda-Time	Bug 5	11	577	273	557
	Bug 7	118	781	799	666
	Bug 10	32	879	913	829
	Bug 14	9	10	103	873
Commons Lang	Bug 8	3	63	64	169
	Bug 30_A	2	1	5	11
	Bug 30_B	2	4	35	2
	Bug 30_C	3	15	21	24
	Bug 34	1	38	106	168
	Bug 57	1	3	80	3
	Bug 61	2	7	17	29

Table 5.4: Comparison results

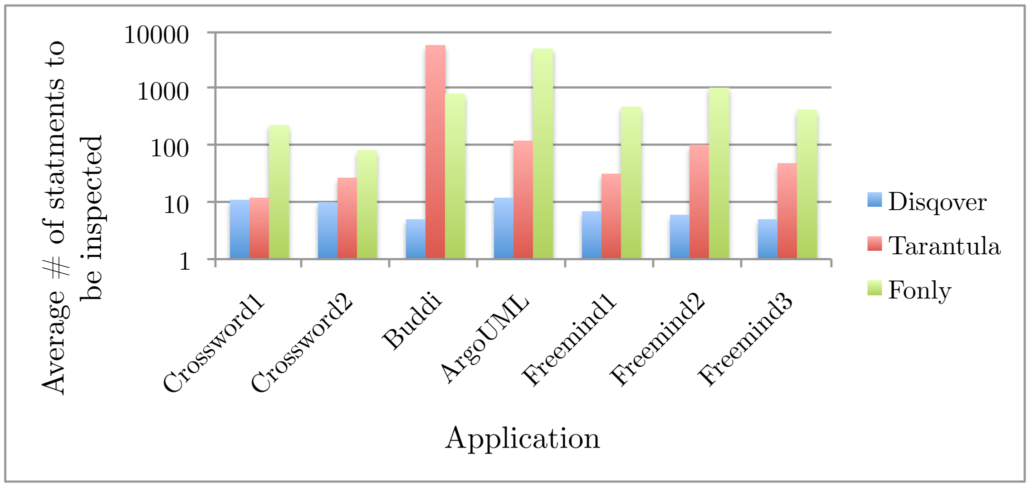
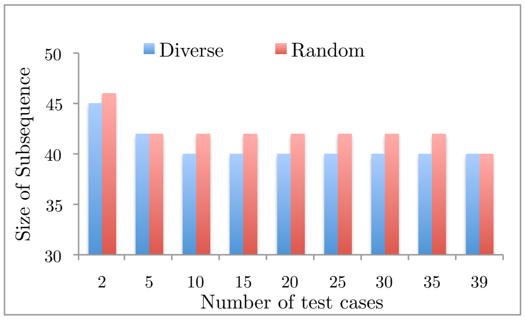


Figure 5.1: Comparison with Tarantula and Fonly

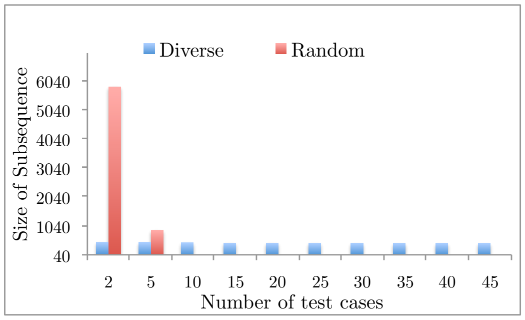
inspected. On the other hand, in the case of Buddi bug, the developed system needs to inspect a number of statements that is 1250 *times* shorter than Tarantula, and in ArgoUML bug is 4291 *times* shorter than Fonly. It is also noticed that Op is the approach that performed best among all the four baselines.

5.1.0.2 Test case diversity experiment

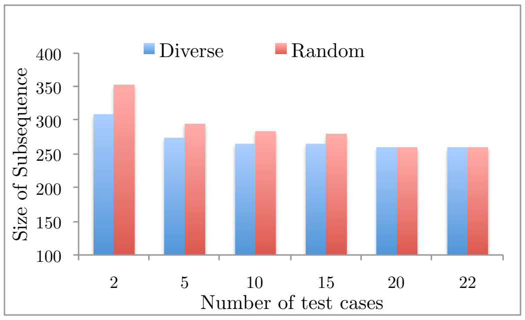
In this experiment, RQ2 is addressed by studying the effect of the diversity of the input test cases on the size of the output common subsequence. As expected, the more diverse the input test cases are, the smaller the size of the output subsequence is. To capture this type of performance, two approaches for selecting the input test cases are compared. 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, the size of the intersection of their code covers is used. 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, the experiment starts 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 all n test cases are added. Although this approach is greedy and may lead to a local optima, it is adopted because of its efficiency. An observation that is seen 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, 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 the figures is chosen. The results of the comparison are shown in Figures 5.2 (a), (b), (c) for the faults of CrossWord Sage (NumberFormatException), Buddi, and ArgoUML, respectively. The number of test cases is varied, and the output common subsequence length for both random and diverse selections is measured. As it can be seen, 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 the same set of test cases is used as input to both approaches. Also, as it can be seen from Figure 5.2 (b), Buddi with the random selection approach always times out after using 5 test cases.



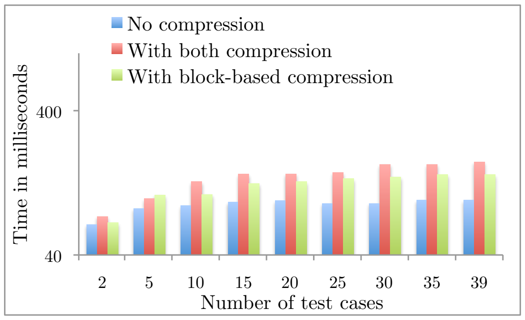
(a) Crossword Sage



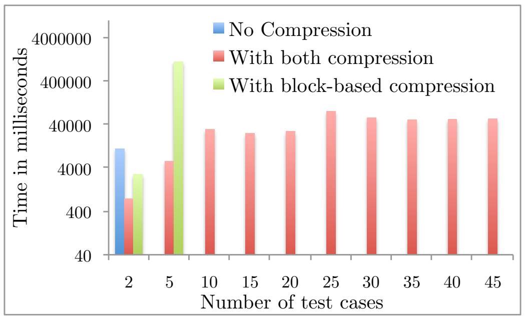
(b) Buddi



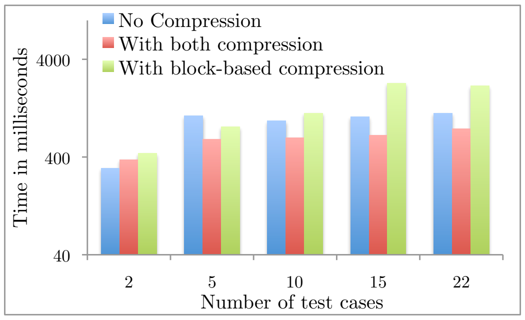
(c) ArgoUML



(d) Crossword Sage



(e) Buddi



(f) ArgoUML

Figure 5.2: (a-c) Test case diversity experiments and (d-f) Running time experiments

5.1.0.3 Sequence Cover Length Experiments

In this experiment, RQ3 is addressed by evaluating the effect of the designed sequence cover abstraction techniques over the average size of sequence cover. Four faults are used. For each fault, the length of the sequence covers is evaluated 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 the 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 5.3, where the average length of the sequence cover using each approach for each fault on log scale is plotted. As it can be seen, 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,

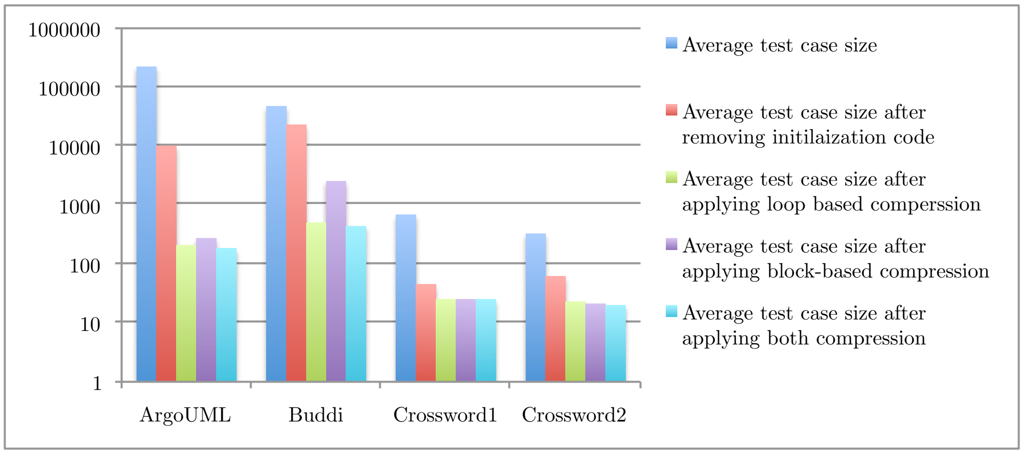


Figure 5.3: Sequence Cover Length Experiments

the effect of block-based abstraction is not as high as if it is applied by itself, but as it can be seen, applying them together is still beneficial. That effect is more obvious when considering the running time of the algorithm using both abstraction techniques, as discussed in Section 5.1.0.4.

5.1.0.4 Running Time Experiments

In these experiments, RQ3 is addressed by evaluating the effect of the number of test cases on the algorithm running time using Disqover, and a number of baselines. The running time using the developed approach is compared to the running time using 1) block-based abstraction only, 2) loop-based abstraction only, and 3) none of the abstraction techniques. The initialization code is removed in all cases. It is also compared against the naive approach for constructing the common subsequences graph which discussed in Section 3.2.3. However, the results of that approach from the discussion, as it does not scale, and causes out of memory exceptions in all cases are omitted. The results are shown in Figures 5.2 (d), (e), (f), for the faults: Crossword Sage (NumberFormatException), ArgoUML (FileNotFoundException), Buddi (FileNotFoundException), respectively. As it can be seen, the baselines outperform the developed 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, the developed 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 the developed approach are much more obvious. The highest running time using the developed approach is 1.4 minutes (from Buddi), while all other approaches could only run for one or two data points, and broke the timeout limit which is 5 minutes for these experiments in all other cases. Just for the purpose of illustration, the timeout constraint is removed 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 the developed 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 the developed approach being the most stable to increasing the number of test cases, which shows that the developed abstraction approaches play an important role in keeping the developed approach scalable.

5.1.0.5 Common Subsequence Length Experiments

In this experiment, RQ4 is addressed by evaluating the effect of the dependency analysis technique over the size of the common subsequence. All defects4j faults are used. For each fault, the length of the common subsequence before and after applying the dependency analysis is reported. The results of this experiment are shown in Table 5.5, where the first column represents the length of the common subsequence before applying the dependency analysis and the second column represents the length after applying dependency analysis. As it can be seen, applying the dependency analysis significantly reduces the number of lines to be inspected by the developer and in some cases, it directly points to the source of the error.

5.2 Conclusions

In this chapter, the experimental evaluation of the developed approach is presented. The experimental evaluation shows the effectiveness of the developed approach in terms of minimizing the developer's debugging time and minimizing the common subsequences algorithm output size and running time. In the next chapter, the future research directions are discussed.

Application	Fault	Subsequence length before	Subsequence length after
ArgoUML	Export All Graphics	235	12
Crosswordsage	Load crossword To Edit	43	5
	New crosswords	45	5
Buddi	Save As	435	5
Freemind	Save As	217	52
	Open	164	44
	Remove Node	56	5
Commons Math	Bug 16	149	4
	Bug 35_A	8	1
	Bug 35_B	8	1
	Bug 36	930	1
Joda-Time	Bug 5	4750	67
	Bug 7	5394	245
	Bug 10	7191	47
	Bug 14	3432	16
Commons Lang	Bug 8	638	16
	Bug 30_A	34	10
	Bug 30_B	8	8
	Bug 30_C	31	5
	Bug 34	427	1
	Bug 57	15	1
	Bug 61	44	10

Table 5.5: Effect of applying dependency analysis

Chapter 6: Future Research Directions

The future research directions are discussed in this section.

1. This work focuses on identifying erroneous code paths and recommending them to software developers. An equally important area of research is profiling of resource consumption. For example, when faced with Out of Memory errors, they have to optimize the software's memory usage, but at the same time, it is quite challenging to figure out which parts of the code consume the most memory, so that they can be further improved. Ideas from this research can be extended to debug memory using code instrumentation and finding commonalities between execution traces which lead to high memory consumption.

2. As discussed in the above point, the memory footprint of a software is one area that can be improved using automated debugging. Garbage collection is another bottleneck that challenges many programmers, but there is a lack of helpful analytical software and tools that profile garbage collection performance. Another line of research is automatically recommending code changes in order to reduce garbage collection overhead. This can be performed using instrumentation methods which extract garbage collection information from the runtime.

3. Along the lines of static analysis and call graph construction, research can be performed to extract the call graph or def/use chains of a software to analyze them to quantify the quality of the code and whether good software engineer practices are being followed. Further analysis can be performed to suggest improvements to the code base to achieve more isolation and modularity between different software components.

4. On the front of dynamic analysis, visualization techniques can be helpful to summarize the coverage of test cases, and help developers quickly identify problems. For example, execution traces can be depicted on top of a visualization of the program call graph, along with a color coding for the program statements representing how frequently they are part of passing versus failing test cases. Such visualization can reveal many important observations such as which parts of the program are not covered by test cases, and which participate in more failing test cases than others.

5. Semantic program profiling: profiling is using a software tool to study the program performance at different levels (e.g., each class or line of code) at run time, and reporting the time (or space) consumption at those levels (e.g., a specific function call consumes 90% of the entire program runtime). Most profilers job ends after reporting the statistics given a specific input configuration. It is up to the developer to understand the relationship between the input and output. A new way of performing profiling is to understand the context of input parameters, and build a model which correlates a set of in-

put configurations and an output of the profiling experiments, which can be referred to as *semantic profiling*. For example, in this new type of profiling, a profiler should be able to reason that with specific inputs a function call takes 80% of the overall runtime, and with other inputs it takes only 10%. Semantic profiling can enable the developers to see insights regarding the factors that make the performance degrade at a higher level.

6.1 Conclusions

In this chapter, the future research directions were discussed. The aim is to discuss other ways of utilizing the ideas developed in this thesis to enhance the software development process along many other dimensions, including improving resource consumption and utilization, enabling visualization of the debugging process, building frameworks to understand the impact of input configurations on software behavior, and analyzing programs to improve code quality and structure.

Bibliography

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

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

[3] Patrice Godefroid, Jonathan de Halleux, Aditya V. Nori, Sriram K. Rajamani, Wolfram Schulte, Nikolai Tillmann, and Michael Y. Levin. Automating software testing using program analysis. *IEEE Software*, 25(5):30–37, 2008.

[4] Qing Xie and Atif M. Memon. Using a pilot study to derive a gui model for automated testing. *ACM Trans. Softw. Eng. Methodol.*, 18(2), 2008.

[5] Atif M. Memon and Qing Xie. Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software. *IEEE Trans. Softw. Eng.*, 31(10):884–896, 2005.

[6] Fevzi Belli. Finite-state testing and analysis of graphical user interfaces. In *ISSRE*, pages 34–43, 2001.

[7] Fevzi Belli, Christof J. Budnik, and Lee White. Event-based modelling, analysis and testing of user interactions: approach and case study. *Softw. Test., Verif. Reliab.*, 16(1):3–32, 2006.

[8] Ana C. R. Paiva, Nikolai Tillmann, João C. P. Faria, and Raul F. A. M. Vidal. Modeling and testing hierarchical guis. In *Abstract State Machines*, pages 329–344, 2005.

[9] Richard K. Shehady and Daniel P. Siewiorek. A method to automate user interface testing using variable finite state machines. In *Proceedings of the 27th International Symposium on Fault-Tolerant Computing (FTCS '97)*, FTCS '97, pages 80–, Washington, DC, USA, 1997.

[10] Linzhang Wang, Jiesong Yuan, Xiaofeng Yu, Jun Hu, Xuandong Li, and Guoliang Zheng. Generating test cases from uml activity diagram based on gray-box method. In *APSEC*, pages 284–291, 2004.

[11] Hyungchoul Kim, Sungwon Kang, Jongmoon Baik, and In-Young Ko. Test cases generation from uml activity diagrams. In *SNPD (3)*, pages 556–561, 2007.

[12] Mingsong Chen, Prabhat Mishra, and Dhrubajyoti Kalita. Coverage-driven automatic test generation for uml activity diagrams. In *ACM Great Lakes Symposium on VLSI*, pages 139–142, 2008.

[13] Mingsong Chen, Xiaokang Qiu, and Xuandong Li. Automatic test case generation for uml activity diagrams. In *AST*, pages 2–8, 2006.

[14] Mingsong Chen, Xiaokang Qiu, Wei Xu, Linzhang Wang, Jianhua Zhao, and Xuandong Li. Uml activity diagram-based automatic test case generation for java programs. *Comput. J.*, 52(5):545–556, 2009.

[15] Chin yu Huang, Jung hua Lo, Sy yen Kuo, and Michael R. Lyu. Software reliability modeling and cost estimation incorporating testing-effort and efficiency. In *Proceedings of the 10th International Symposium on Software Reliability Engineering (ISSRE'99*, pages 62–72, 1999.

[16] Trung Dinh-Trong. A systematic approach to testing uml design models. In *In Doctorial Symposium, 7th International Conference on the Unified Modeling Language: Lisbon, Portugal*, 2004.

[17] Philip Samuel and Anju Teresa Joseph. Test sequence generation from uml sequence diagrams. In *SNPD*, pages 879–887, 2008.

[18] Monalisa Sarma, Debasish Kundu, and Rajib Mall. Automatic test case generation from uml sequence diagrams. In *Proceedings of the 15th International Conference on Advanced Computing and Communications*, pages 60–67. IEEE Computer Society, 2007.

[19] Abu Zafer Javed, Paul A. Strooper, and Geoffrey Watson. Automated generation of test cases using model-driven architecture. In *AST*, pages 3–9, 2007.

[20] Noraida Ismail, Rosziati Ibrahim, and Noraini Ibrahim. Automatic generation of test cases from use-case diagram. In *Proceedings of the International Conference on Electrical Engineering and Informatics Institut Teknologi: Bandung, Indonesia*, 2007.

[21] Santosh Kumar Swain and Durga Prasad Mohapatra. Article:test case generation from behavioral uml models. *International Journal of Computer Applications*, 6(8):5–11, September 2010. Published By Foundation of Computer Science.

[22] Matthias Riebisch, Ilka Philippow, and Marco Götze. Uml-based statistical test case generation. In *NetObjectDays*, pages 394–411, 2002.

[23] Shaukat Ali, Lionel C. Briand, Muhammad Jaffar-Ur Rehman, Hajra Asghar, Muhammad Zohaib Z. Iqbal, and Aamer Nadeem. A state-based approach to integration testing based on uml models. *Information & Software Technology*, 49(11-12):1087–1106, 2007.

[24] Shinpei Ogata and Saeko Matsuura. Towards the reliable integration testing: Uml-based scenario analysis using an automatic prototype generation tool. In *Proceedings of the 9th WSEAS International Conference on Software Engineering, Parallel and Distributed Systems*, SEPADS'10, pages 151–159, Stevens Point, Wisconsin, USA, 2010. World Scientific and Engineering Academy and Society (WSEAS).

[25] Shinpei Ogata, Saeko Matsuura, Shinpei Ogata, and Saeko Matsuura. A method of automatic integration test case generation from uml-based scenario.

[26] Christian Pfaller. Requirements-based test case specification by using information from model construction. In *AST*, pages 7–16, 2008.

[27] Alessandra Cavarra, Charles Crichton, Jim Davies, Alan Hartman, and Laurent Mounier. Using uml for automatic test generation. In *IN INTERNATIONAL SYMPOSIUM ON SOFTWARE TESTING AND ANALYSIS ISSTA*. Springer-Verlag, 2002.

[28] Qurat ul-ann Farooq, Muhammad Zohaib Z. Iqbal, Zafar I. Malik, and Aamer Nadeem. An approach for selective state machine based regression testing. In *A-MOST*, pages 44–52, 2007.

[29] Vahid Garousi, Lionel C. Briand, and Yvan Labiche. Control flow analysis of uml 2.0 sequence diagrams. In *ECMDA-FA*, pages 160–174, 2005.

[30] Aysh Alhroob, Keshav P. Dahal, and M. Alamgir Hossain. Automatic test cases generation from software specifications. *e-Informatica*, 4(1):109–121, 2010.

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

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

[33] Gilles Perrouin, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves Le Traon. Automated and scalable t-wise test case generation strategies for software product lines. In *ICST*, pages 459–468, 2010.

[34] Stefania Gnesi, Diego Latella, Mieke Massink, Via 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.

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

[36] Mark Weiser. Programmers use slices when debugging. *Commun. ACM*, 25(7):446–452, 1982.

[37] Brian Ness and Viet 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.

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

[39] Ghassan Misherghi and Zhendong Su. Hdd: hierarchical delta debugging. In *ICSE*, pages 142–151, 2006.

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

[41] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. Pruning dynamic slices with confidence. In *PLDI*, pages 169–180, 2006.

[42] Richard A. DeMillo, Hsin Pan, and Eugene H. Spafford. Critical slicing for software fault localization. In *ISSTA*, pages 121–134, 1996.

[43] Tibor Gyimothy, Arpad Beszedes, and Istvan Forgacs. An efficient relevant slicing method for debugging. In *ESEC / SIGSOFT FSE*, pages 303–321, 1999.

[44] Xiangyu Zhang, Rajiv Gupta, and Youtao Zhang. Precise dynamic slicing algorithms. In *ICSE*, pages 319–329, 2003.

[45] Thomas W. Reps, Thomas Ball, Manuvir Das, and James R. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *ESEC / SIGSOFT FSE*, pages 432–449, 1997.

[46] Thomas Ball, Mayur Naik, and Sriram K. Rajamani. From symptom to cause: localizing errors in counterexample traces. In *POPL*, pages 97–105, 2003.

[47] Alex Groce, Daniel Kroening, and Flavio Lerda. Understanding counterexamples with explain. In *CAV*, pages 453–456, 2004.

[48] James A. Jones, Mary Jean Harrold, and John T. Stasko. Visualization of test information to assist fault localization. In *ICSE*, pages 467–477, 2002.

[49] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. A model for spectra-based software diagnosis. *ACM Trans. Softw. Eng. Methodol.*, 20(3):11, 2011.

[50] Ben Liblit, Mayur Naik, Alice X. Zheng, Alexander Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *PLDI*, pages 15–26, 2005.

[51] Chao Liu, Xifeng Yan, Long Fei, Jiawei Han, and Samuel P. Midkiff. Sober: statistical model-based bug localization. In *ESEC/SIGSOFT FSE*, pages 286–295, 2005.

[52] Qingguo Wang, Dmitry Korkin, and Yi Shang. A fast multiple longest common subsequence (mlcs) algorithm. *IEEE Trans. Knowl. Data Eng.*, 23(3):321–334, 2011.

[53] Christopher Lee, Catherine Grasso, and Mark F. Sharlow. Multiple sequence alignment using partial order graphs. *Bioinformatics*, 18(3):452–464, 2002.

[54] Robert C. Edgar. Muscle: Multiple sequence alignment with improved accuracy and speed. In *CSB*, pages 728–729, 2004.

[55] Michael Brudno and Burkhard Morgenstern. Fast and sensitive alignment of large genomic sequences. In *CSB*, pages 138–, 2002.

[56] Bao Nguyen, Bryan Robbins, Ishan Banerjee, and Atif Memon. Guitar: an innovative tool for automated testing of gui-driven software. *Automated Software Engineering*, pages 1–41, 2013.

[57] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. Ask the mutants: Mutating faulty programs for fault localization. In *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014-April 4, 2014, Cleveland, Ohio, USA*, pages 153–162, 2014.

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

[59] T. Britton, G. Carver L. Jeng, P. Cheak, and T. Katzenellenbogen. Reversible debugging software. Technical report, University of Cambridge, Judge Business School, 2013.

[60] Hiralal Agrawal and Joseph Robert Horgan. Dynamic program slicing. In *PLDI*, pages 246–256, 1990.

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

[62] Holger Cleve and Andreas Zeller. Locating causes of program failures. In *ICSE*, pages 342–351, 2005.

[63] Joseph Ruthruff, Eugene R. Creswick, Margaret Burnett, Curtis Cook, Shreenivasarao Prabhakararao, Marc F. Ii, and Martin Main. End-user software visualizations for fault localization. In *SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization*, pages 123–132, San Diego, CA, USA, June 2003. ACM Press.

[64] Eric Wong, Tingting Wei, Yu Qi, and Lei Zhao. A crosstab-based statistical method for effective fault localization. In *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, ICST '08, pages 42–51, Washington, DC, USA, 2008. IEEE Computer Society.

[65] W. Eric Wong and Yu Qi. Bp neural network-based effective fault localization. *International Journal of Software Engineering and Knowledge Engineering*, 19(4):573–597, 2009.

[66] W. Eric Wong, Yan Shi, Yu Qi, and Richard Golden. Using an rbf neural network to locate program bugs. In *ISSRE*, pages 27–36, 2008.

[67] Lionel C. Briand, Yvan Labiche, and Xuetao Liu. Using machine learning to support debugging with tarantula. In *ISSRE*, pages 137–146, 2007.

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

[69] Syeda Nessa, Muhammad Abedin, W. Eric Wong, Latifur Khan, and Yu Qi. Software fault localization using n-gram analysis. In *WASA*, pages 548–559, 2008.

[70] Peggy Cellier, Mireille Ducassé, Sébastien Ferré, and Olivier Ridoux. Formal concept analysis enhances fault localization in software. In *ICFCA*, pages 273–288, 2008.

[71] Franz Wotawa, Markus Stumptner, and Wolfgang Mayer. Model-based debugging or how to diagnose programs automatically. In *IEA/AIE*, pages 746–757, 2002.

[72] Cristinel Mateis, Markus Stumptner, and Franz Wotawa. Modeling java programs for diagnosis. In *In Proceedings of the European Conference on Artificial Intelligence (ECAI*, page 2000. Press, 2000.

[73] Wolfgang Mayer and Markus Stumptner. Modeling programs with unstructured control flow for debugging. In *Australian Joint Conference on Artificial Intelligence*, pages 107–118, 2002.

[74] Wolfgang Mayer, Markus Stumptner, and Franz Wotawa. Debugging program exceptions. In *IN PROC. DX03 WORKSHOP*, pages 119–124, 2003.

[75] Richard A. DeMillo, Hsin Pan, and Eugene H. Spafford. Failure and fault analysis for software debugging. In *COMPSAC*, pages 515–521, 1997.

[76] Glenn Ammons, Jong deok Choi, Manish Gupta, and Nikhil Swamy. Finding and removing performance bottlenecks in large systems. In *In Proceedings of ECOOP*. Springer, 2004.

[77] Kavitha Srinivas and Harini Srinivasan. Summarizing application performance from a components perspective. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 136–145, New York, NY, USA, 2005. ACM.

[78] Gary Sevitsky, Wim De Pauw, and Ravi Konuru. An information exploration tool for performance analysis of java programs. In *TOOLS (38)*, pages 85–101. IEEE Computer Society, 2001.

[79] Shi Han, Yingnong Dang, Song Ge, Dongmei Zhang, and Tao Xie. Performance debugging in the large via mining millions of stack traces. In *ICSE*, pages 145–155, 2012.

[80] Erik R. Altman, Matthew Arnold, Stephen Fink, and Nick Mitchell. Performance analysis of idle programs. In *OOPSLA*, pages 739–753, 2010.

[81] Cobertura (A code coverage utility for Java). `http://cobertura.github.io/cobertura/`.

[82] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative Research, November 8-11, 1999, Mississauga, Ontario, Canada*, page 13, 1999.

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

[84] ArgoUML. `http://argouml.tigris.org/`.

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

[86] Buddi. `http://buddi.digitalcave.ca/`.

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

[88] Commons Math. `https://commons.apache.org/proper/commons-math/`.

[89] JodaTime. `http://www.joda.org/joda-time/`.

[90] Commons Lang. `https://commons.apache.org/proper/commons-lang/`.

[91] René Just, Darioush Jalali, and Michael D. Ernst. Defects4j: a database of existing faults to enable controlled testing studies for java programs. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, pages 437–440, 2014.

[92] μJava . `https://cs.gmu.edu/~offutt/mujava/`.