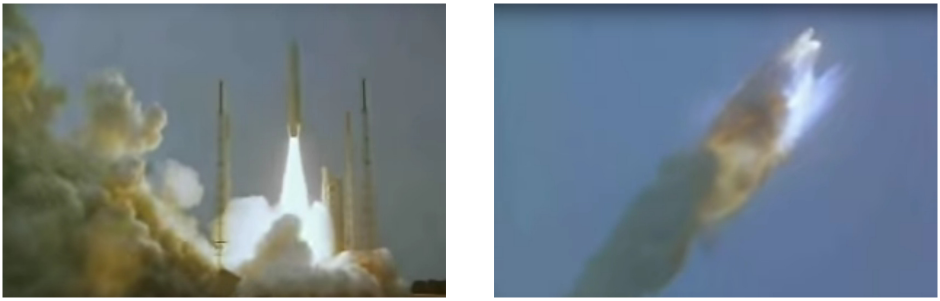
Exploring means to facilitate software debugging

SOLTANI, M.S.

1 Introduction

Today computer systems have profound impact on every aspect of our lives. Software applications have transformed education systems, healthcare, military systems, the way businesses operate, the way individuals gather information and communicate, and so on. Thus, in numerous ways, we have been relying on computer systems to the extent it is difficult to imagine daily life without them. Despite the advantages of using computer systems, over the last decades, catastrophic cases of their failures have been reported.



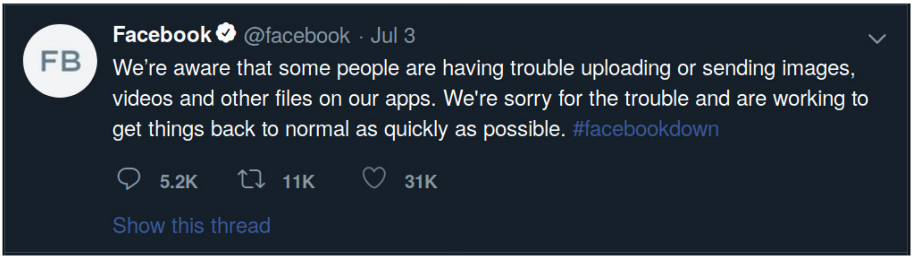
(a) Lifting off

(b) Exploding

Figure 1.1: The lift-off and explosion of the Ariane 5 rocket in 1996.

As Figure 1.1 shows, the explosion of the Ariane 5 rocket in 1996 is among the older and well-known failures which was caused due a single integer overflow. A more recent case was reported in 2015 in which a robot killed a worker at one of the production plants of Volkswagen in Germany [174]. In this case the robot took the

Figure 1.2: Grounding the entire American Airline fleet [39].



Facebook @facebook · Jul 3

We're aware that some people are having trouble uploading or sending images, videos and other files on our apps. We're sorry for the trouble and are working to get things back to normal as quickly as possible. #facebookdown

Figure 1.3: The twit from Facebook confirming the outages of Facebook, Instagram, and WhatsApp applications.

worker and crushed him against a metal plate. Even more recent cases from 2018 and 2019 are reported [39, 131] which range from the outages of Facebook, Instagram, and WhatsApp (shown in Figure 1.3), separating Amazon users from their cloud storage, grounding the entire American Airline fleet (shown in Figure 1.2), canceling over 100 flights by British Airways, pushing the investment firm Knight Capital into bankruptcy, and so on. These failures disrupted lives on a large scale, caused deaths, and cost billions of dollars.

In order to prevent such failures and their disastrous consequences, a large number of quality and functional requirements must be met. Such requirements, in addition to complexity of software products, and short software development cycles place major importance on activities in the areas of software verification, testing, and debugging [116]. In a typical commercial development organization, the cost of software verification, testing and debugging activities can easily range from 50 to 75 percent of the total development cost [116]. In addition, these activities are error-prone and labor-intensive. Therefore, much research effort has been put into automating as many activities in these areas as possible [105]. In what follows, we present a background about the automated techniques which are researched in the areas of test generation, and software debugging. Thereafter, we present the contributions in this thesis, and conclude the chapter with a list of peer-reviewed publications associated with each of the following chapters in this thesis.

1.1 Automated Test Generation Techniques

Various automated techniques for unit test generation have been proposed in the research literature. DART [112] is an older technique which combines directed testing with random testing to automatically generate test cases. PEX [206] is also an older white-box test generation tool which uses dynamic symbolic execution to generate tests for .Net programs.

Recent test generation approaches include SAPIENZ [159], EvoSuite [103], and JTExpert [196]. SAPIENZ [159] is an approach to Android testing that uses multi-objective search to automatically optimize test sequences, by minimizing test lengths, while simultaneously maximizing coverage and fault revelation. EvoSuite [103] generates test suites for Java classes. To this end, Evosuite applies a hybrid approach that generates and optimizes whole test suites towards satisfying a coverage criterion. For the produced test suites, EvoSuite suggests possible oracles by adding small and effective sets of assertions that concisely summarize the current behavior. JTExpert [196], on the other hand, uses a static analysis approach to extract the methods or constructors which change the state of the class under test or that may reach a test target. Then JTExpert [196] uses a generator to make instances of classes by using subclasses and external factory methods. JTExpert also uses a seeding strategy and a diversification strategy to increase the likelihood to reach a test target.

Moreover, research literature also contains the Monkey [18], Dynodroid [156], T3 [185], and RANDOOP [177] techniques which apply various random test generation approaches. Monkey is a command line tool that can be run on any instance of Android emulator or device to stress-test applications. This tool generates pseudo-random streams of user events such as clicks, touches, or gestures, as well as a number of system-level events [18]. Dynodroid [156] views an Android application as an event-driven program that interacts with its environment by means of a sequence of events through the Android framework. By instrumenting the framework once, Dynodroid monitors the reaction of an application upon each event to guide the generation of the next events. Dynodroid also allows interleaving events from machines, which are better at generating a large number of simple inputs, with events from humans, who are better at providing intelligent inputs.

Pacheco et al. [177] propose RANDOOP, which incorporates feedback from executing test inputs as they are created. This technique builds inputs incrementally by randomly selecting a method call to apply and finding arguments from previously constructed inputs. As soon as an input is built, it is executed and checked against a set of contracts and filters. The result of the execution determines whether the input is redundant, illegal, contract-violating, or useful for generating more inputs. The technique outputs a test suite consisting of unit tests for the classes under test.

Finally, T3 [185] is the next generation of the testing tool T2 [186]. T3 is implemented in Java 8, and can be used to automatically generate random tests for Java classes. A test sequence against a Class Under Test (CUT) starts with the creation of an object which is an instance of the CUT, followed by calls to the methods of the object, or updates to the fields. T3 randomly generates a large amount of such test sequences to trigger faulty behavior, and thus finding a bug.

1.2 Automated Debugging Techniques

Andreas Zeller proposes the delta debugging algorithm [220] which systematically narrows the state difference between a passing run and a failing run, to determine whether a change in the program state makes a difference in the test outcome. Applying the delta debugging algorithm to multiple states of the program automatically reveals the cause-effect chain of the failure, which are, the variables and values that caused the failure [220]. Slicing programs is another automated technique described by Andreas Zeller [221], in which, based on statement dependencies, one can focus on specific subsets of the program, which are called slices. These subsets may have influenced a specific statement or be influenced by a specific statement. Therefore, in this approach, depending on the direction of computing the dependencies to or from a statement, forward or backward slicing operations are performed.

Parsa et al. [182] propose an approach to defect localization using elastic net. The proposed approach finds the smallest effective subset of program predicates known as bug predictors. After selecting bug predictors, the main causes of faults are detected by using existing program slicing techniques. Eichinger et al. [96] propose another approach to automated defect localization using dataflow-enabled call graphs that incorporate abstractions of the dataflow. In this approach, defect localization is essentially formulated as a data mining problem, making use of discretisation, frequent subgraph mining and feature selection [96]. Moreover, another approach to fault localization is proposed by Naish et al. [168]. This approach is based on ranking program statements or blocks according to how likely they are to be buggy.

Chandra et al. [80] propose angelic debugging. This technique is based on locating expressions that are likely to be bugs. After locating the likely-buggy expressions, angelic debugging searches the space of all edits to the program for one that repairs the failing test without breaking any passing test. Furthermore, Wei et al. [210] propose the AutoFix-E approach which automatically generates and validates fixes for software faults. AutoFix-E relies on contracts present in the software to ensure that the proposed fixes are semantically sound. On the other hand, Le Goues et al. [148] propose GenProg, which uses an extended form of genetic programming to repair defects in off-the-shelf, legacy programs without formal specifications, program annotations, or special coding practices. GenProg evolves program variants which keep required functionality but are not susceptible to given defects, using existing test suites.

Another approach to assist with software debugging is proposed by Zhang et al. [223] to automatically identify breakpoints. This technique combines the nearest neighbor queries method, dynamic program slicing, and memory graph comparison to detect suspicious program statements and states. Based on this information, breakpoints are generated and divided into two groups, where the primary group contains conditional breakpoints and the secondary group contains unconditional ones.

Moreover, a body of research is done on automated crash reproduction. Earlier research in this field [58,83,169,205] use record-replay techniques to reproduce crashes. However, due to the execution overhead and privacy-related issues, recent research work [81,137,150,151,171,194,215,217,219] take a post-failure approach by using the crash data that is generated after the crash occurs. The crash data is often in the form of crash stack traces which are recorded in software execution logs.

1.3 Contributions

In this section, first we present an overview of the overarching goal of the thesis and an outline of the research questions we aimed to answer. Thereafter, we provide our motivation for each of the contributions in this thesis, followed by their outcomes.

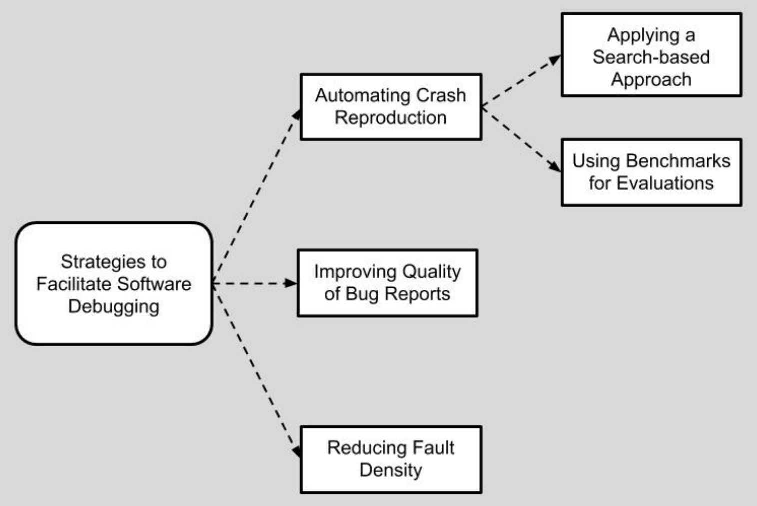

Applying a Search-based Approach
Automating Crash Reproduction
Using Benchmarks for Evaluations
Strategies to Facilitate Software Debugging
Improving Quality of Bug Reports
Reducing Fault Density

Figure 1.4: The overview of the strategies addressed in this thesis to facilitate software debugging.

1.3.1 Overview

As Figure 1.4 shows, the overarching goal of this thesis is to investigate various means to facilitate automated software debugging. We start with investigating a search-based solution for automated crash reproduction to support developers by producing information which help identify the faults faster. While investigating the effectiveness of this solution for developers, we detect additional needs to address:

- an extensible benchmark for comparing automated crash reproduction techniques, and a systemic way to run evaluations using this benchmark,
- investigating techniques to improve the fitness function in the proposed solution for search-based automated crash reproduction,
- investigating the information developers need from bug reports, which make software debugging more efficient, and
- investigating whether the Design by Contract (DBC) software development approach can reduce defect density in software, so that debugging effort is reduced by trying to prevent faults in software products in the first place.

To address the above concerns, we define the following research questions:

- **RQ$_1$:** How can search-based algorithms be used to automate crash reproduction?

- **RQ2:** How to create a benchmark of representative crashes to use for evaluating crash reproduction research prototypes?
- **RQ3:** What is the impact of multi-objectivization on evolutionary crash reproduction?
- **RQ4:** Do different elements of bug reports impact the time it takes to fix bugs?
- **RQ5:** Can program contracts be used to reduce the occurrence of bugs?

1.3.2 Motivation for the Contributions and their Outcomes

As mentioned in Section 1.2 previously, a body of research was done on automated crash reproduction. Despite the previous research in this area, application of evolutionary search-based algorithms remained underexplored. This is while search-based algorithms have proven to be efficient techniques in other areas such as automated test generation.

To evaluate the application of search-based algorithms for automated crash reproduction and answer **RQ1**, we devised a problem representation, a new fitness function, and new genetic algorithm operators, in the first study we performed. We named this approach EvoCrash. To implement EvoCrash, we extended the search based test generating tool, named EvoSuite [103]. In the first study we did, we compared EvoCrash with the state-of-the-art crash reproduction approaches. Our evaluation indicated EvoCrash outperformed the state-of-the-art techniques by reproducing more crashes from real-world projects.

Despite the early evaluations we performed, we noticed the number of projects and available crash stack traces are rather limited, which makes comparison of the research prototypes challenging. Therefore, in the second study we answer **RQ2** by proposing JCrashPack, an extensible benchmark for Java crash reproduction. JCrashPack contains 200 stack traces from various Java projects, including industrial open source ones, on which we run an extensive evaluation of EvoCrash. EvoCrash successfully reproduced 43% of the crashes. Furthermore, we observed that reproducing NullPointerException, IllegalArgumentException, and IllegalStateException is relatively easier than reproducing ClassCastException, ArrayIndexOutOfBoundsException and StringIndexOutOfBoundsException.

To answer **RQ3**, in the third study, we continue with further evaluating the EvoCrash approach. In this study, we investigate the alternatives to the original fitness function we defined in EvoCrash. The original fitness function in EvoCrash uses single weighted sum scalarization function to optimize test generation. Therefore, in the third study, we assessed whether there is any difference if we use a simple sum scalarized function without any weights. In addition, we assessed whether multi-objectivization, which is the process of turning a single-objective optimization to multi-objective optimization, impacts crash reproduction in any way. Our results indicate that for complex crashes the weighted sum function reduces the test case generation time, compared to the simple sum function, while for simpler crashes the effect is the opposite. Similarly, for complex crashes, multi-objectivization reduces test generation time compared to optimizing with the weighted sum function.

When it comes to debugging, the quality of the information that is available to developers can vary. Too little information can influence the priority that is assigned to bug reports. To answer **RQ4**, the fourth study we performed focuses on the quality of bug reports and the significance of bug report elements on bug resolution times. In this study, we interviewed 35 developers and surveyed 305 developers to understand developers' perception on the significance of different bug report elements. According to the results, developers find it highly important that bug reports include crash description, reproducing steps, and crash stack traces. In addition, to evaluate the quality of currently available bug reports, we mined issue repositories of 250 most popular projects on Github. Statistical analysis on the mined issues shows that crash reproducing steps, stack traces, fix suggestions, and user contents, have statistically significant impact on bug resolution times, for ∼70%, ∼76%, ∼55%, and ∼33% of the projects. However, on average, over 70% of bug reports lack these elements.

When it comes to software testing, verification, and debugging with the intention to meet the quality and functional requirements, one aspect to consider is identifying means by which the need for software debugging is reduced. Yuan et al. [218] show in their study that simple testing can prevent most critical failures. Similarly, the studies by Kochhar and Lo [143] and Casalnuovo et al. [77] show there is a negative correlation between the use of program contracts and the frequency of bug occurrence for a given method in the program.

In the fifth study we perform, we extend the studies done by Kochhar and Lo [143] and Casalnuovo et al. [77] to answer **RQ5**. In this study, we report results of an empirical evaluation on the use of contracts in 148 open source projects, written in Java, C++, and Python. Our findings show that the average use of different types of contracts differ depending on the program language. Furthermore, the results of regression analysis shows there is a negative relation between the number of contracts and frequency of defect occurrence in a method. These results are statistically significant for all Java, C++, and Python projects.

1.4 Publications

The different chapters of this thesis are based on the following publications:

- Mozhan Soltani, Annibale Panichella, and Arie Van Deursen. "A guided genetic algorithm for automated crash reproduction." In 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), pp. 209-220. IEEE, (2017). (Chapter 2)

- Mozhan Soltani, Annibale Panichella, and Arie Van Deursen. "Search-based crash reproduction and its impact on debugging." IEEE Transactions on Software Engineering (2018). (Chapter 2)

- Mozhan Soltani, Pouria Derakhshanfar, Xavier Devroey, and Arie Van Deursen. "A benchmark-based evaluation of search-based crash reproduction." Empirical Software Engineering 25, no. 1, pp. 96-138. Springer, (2020). (*The first two authors agreed to share this paper in their PhD theses.*) (Chapter 3)

- Mozhan Soltani, Pouria Derakhshanfar, Annibale Panichella, Xavier Devroey, Andy Zaidman, and Arie van Deursen. "Single-objective Versus Multi-objectivized Optimization for Evolutionary Crash Reproduction." In International Symposium on Search Based Software Engineering, pp. 325-340. Springer, Cham, (2018). (Chapter 4)

- Mozhan Soltani, Felienne Hermans, and Thomas Bäck. "The significance of bug report elements." Empirical Software Engineering. Springer, (2020). (*This paper is under revision at the time of writing the thesis.*) (Chapter 5)