Formal Automated Program Repair

Bat-Chen Rothenberg

Formal Automated Program Repair

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

Bat-Chen Rothenberg

Submitted to the Senate
of the Technion — Israel Institute of Technology
Cheshvan 5781 Haifa October 2020

This research was carried out under the supervision of Prof. Orna Grumberg, in the Faculty of Computer Science.

Some results in this thesis have been published as articles by the author and research collaborators in conferences and journals during the course of the author's doctoral research period, the most up-to-date versions of which being:

Bat-Chen Rothenberg, Daniel Dietsch, and Matthias Heizmann. Incremental verification using trace abstraction. In *International Static Analysis Symposium*, pages 364–382. Springer, 2018.

Bat-Chen Rothenberg and Orna Grumberg. Sound and complete mutation-based program repair. In *International Symposium on Formal Methods*, pages 593–611. Springer, 2016.

Bat-Chen Rothenberg and Orna Grumberg. Poster: Program repair that learns from mistakes. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 218–219. IEEE, 2018.

Bat-Chen Rothenberg and Orna Grumberg. Must fault localization for program repair. In *International Conference on Computer Aided Verification*, pages 658–680. Springer, 2020.

Acknowledgements

First and foremost, I would like to thank my advisor, Orna Grumberg. Thank you for letting me follow my passion, for always being there to help and give good advice, and for teaching me up close that the most important thing is to enjoy the road.

I am very grateful for all the kind people who took part in my research, one way or another. To my co-authors, Matthias and Daniel - thank you for a productive collaboration which I have learned a lot from. To the committee members of my candidacy and thesis exams - thank you for the helpful comments, guidance and fruitful discussions. To my various friends and office partners over the years - thank you for being there for me during difficult times, for always helping with any question or request, and for making me feel most at home in the world.

Huge thanks to my parents, Avi and Naomi, who always believed in me and never pushed me to be anything but who I am. Thank you to my mother-in-law Elaine, to my brothers Nadav and Shana, and to the whole extended family for always being there for me unconditionally. I love you all very much.

Special thanks to my children, Lior and Gali, who always remind me of what is important in life, and put me in proportion. Everything I do is for you.

And a last and unique thank you to my one and only, my best friend, and my life partner, my husband Dani. Thank you for accompanying me all the way, for better or worse. I could not have done it without you.

The generous financial help of the Technion, the Technion Hiroshi Fujiwara cyber security research center and the Israel cyber bureau is gratefully acknowledged.

Contents

List of Figures

Abstract

Nowadays, software systems can be found not only on our laptops, but all around us: in phones, TVs, cars, airplanes and many more. Such systems are complex and hard to design, and therefore they often reach the client while still containing many hidden bugs. When discovered by the client, these bugs damage companies reputation, sometimes irreversibly. Therefore, companies make tremendous efforts to detect and repair bugs, devoting a significant percentage of the development process to these tasks.

Yet even though these efforts lead to the discovery of many bugs, still, companies are forced to prioritize their repair, and many bugs remain uncorrected. This is because human resources are limited, and manual repair of bugs is a notoriously difficult task, which often requires expertise and close acquaintance with the code under inspection.

Hence, automating program repair has always been a research challenge of much interest. In recent years, however, research in this area has seen significant progress, with the development of methods that have been shown to be useful for repairing real bugs in large-scale programs.

This thesis is concerned with automated program repair from a formal point of view. This means that programs are repaired with respect to a formal specification. Also, formal methods from the world of program verification are used for the repair process.

We mostly focus on search-based repair, where programs are iteratively sampled from within a search space and then checked to see if they meet the specification. We present three published papers, describing several algorithms. The common goal of all algorithms is to make search-based repair more efficient, by pruning the search space whenever possible and by making the repeated correctness checks more efficient.

Chapter 1

Introduction

Nowadays, computers are everywhere. And where there are computers, there are computer bugs. Most of the time, these bugs merely disturb the user. For example, when an application suddenly crashes on the user's phone. But, when found in safety-critical systems, such as autonomous cars, bugs can have disastrous outcomes. The manual detection, examination and repair of computer bugs are all notoriously difficult tasks that programmers face daily. These tasks are hard because they usually require expertise and close acquaintance with the code under inspection.

Automated program repair receives a specification and a program that violates it, and outputs a different program, called a repair, which meets the specification. The repair needs to be as similar to the original buggy program as possible, where similarity can be measured syntactically or semantically: syntactic similarity means that their code should be mostly identical, while semantic similarity means that their behavior should remain the same for most inputs.

Research on automated program repair has grown intensively in recent years [TYPR16, DSS16, ABS17, GPKS17, XWY+17, MNN+18, JXZ+18, SGZL18, WCW+18, HZWK18b, vTG18]. The techniques used for automated repair are varied, and include search-based algorithms [LNFW12, TYPR16, WCW+18], formal synthesis [NQRC13, MYR15, DSS16, MNN+18] and machine learning [LR16, GPKS17].

While many existing techniques use a finite set of tests as a specification, this thesis focuses on repair using a formal specification. When a program is repaired with respect to a formal specification, it is guaranteed to meet it, which means that it is correct for all inputs and not just a selected set. Thus, formal repair allows for greater confidence in the given program. On the other hand, the repair problem is more challenging, and therefore algorithms are expected to be less efficient in general.

Another difference of this work from much of the previous work on repair is in the methods we have used. The emphasis in our work was on the use of formal methods, which make it possible to prove the correctness of formal specifications. The main methods we have used are techniques for software model checking, solvers of boolean formulas (SAT solvers), and solvers of formulas over first-order theories (SMT solvers).

All program repair algorithms we have designed follow a search-based repair approach (as opposed to a synthesis-based one). Techniques following this approach look for a repair to a program P by making changes to P according to a predefined set of rules. We refer to the set of programs created in this way as the *search space* of the technique, and to any program in it as a *patched program*.

Many search-based techniques [LNFW12, QML+13b, RHJ+12, LR16] follow a *generate and validate* working scheme. This means that each time a patched program is sampled according to a certain policy, the correctness of this program is examined. If it is correct - it is returned as a repair, otherwise - another patched program is chosen, and so on.

This thesis includes three papers whose common objective is to enable efficient generate and validate algorithms for formal program repair. The first paper presents a mutation-based algorithm for program repair following the generate and validate scheme. Its efficiency stems from pruning non-minimal repairs from the search space each time a patched program is found to be correct during the validate stage. Its efficiency also stems from using incremental SAT and SMT solving. The second paper presents two automata-based algorithms for incremental verification, which is the problem of efficiently verifying many similar revisions of a program in a row. Though this paper is not concerned directly with program repair, one of the major applications for the algorithms presented in it is to be used for the validate stage in a generate and validate loop. The third paper improves the repair algorithm of the first by using fault localization and pruning the search space also in the case where a patched program is found to be buggy during the validate stage.

Next, we present a short abstract for each of the papers.

Mutation-Based Repair This work presents a novel approach for automatically repairing an erroneous program with respect to a given set of assertions. Programs are repaired using a predefined set of mutations. We refer to a bounded notion of correctness, even though, for a large enough bound all returned programs are fully correct. To ensure no changes are made to the original program unless necessary, if a program can be repaired by applying a set of mutations Mut, then no superset of Mut is later considered. Programs are checked in increasing number of mutations, and every minimal repaired program is returned as soon as found.

We impose no assumptions on the number of erroneous locations in the program, yet we are able to guarantee soundness and completeness. That is, we assure that a program is returned iff it is minimal and bounded correct.

Searching the space of mutated programs is reduced to searching unsatisfiable sets of constraints, which is performed efficiently using a sophisticated cooperation between SAT and SMT solvers. Similarities between mutated programs are exploited in a new way, by using both the SAT and the SMT solvers incrementally.

We implemented a prototype of our algorithm, compared it with a state-of-the-art

repair tool and got very encouraging results.

Incremental Verification Modern software projects often consist of thousands of lines of code and are updated on a daily basis. Despite the increasing effectiveness of model checking tools, automatically re-verifying the entire program whenever a new revision is created is often not feasible using existing tools. Incremental verification aims at facilitating this re-verification, by reusing partial results from previous revisions in order to focus the analysis on parts of the program that were semantically affected by the change.

In this paper, we propose a novel approach for incremental verification that is based on trace abstraction. Trace abstraction is an automata-based verification technique that follows the counter-example guided abstraction refinement (CEGAR) scheme. The abstraction itself is a sequence of automata, and it is refined by repeatedly adding an automaton to the sequence. When a program is proven correct, this sequence forms a proof of its correctness. When a bug is found in a program, the sequence obtained up to the point of finding the bug forms a proof of correctness for some subset of program traces.

We present two algorithms that reuse the obtained sequence, one eagerly and one lazily. We demonstrate their effectiveness in an extensive experimental evaluation on a previously established benchmark set for incremental verification based on different revisions of device drivers from the Linux kernel. Our algorithm is able to achieve significant speedups on this set, compared to stand-alone verification.

Fault Localization This work is concerned with fault localization for automated program repair.

We define a novel concept of a *must* location set. Intuitively, such a set includes at least one program location from every repair for a bug. Thus, it is impossible to fix the bug without changing at least one location from this set. A fault localization technique is considered a *must* algorithm if it returns a must location set for every buggy program and every bug in the program. We show that some traditional fault localization techniques are not must.

We observe that the notion of must fault localization depends on the chosen repair scheme, which identifies the changes that can be applied to program statements as part of a repair. We develop a new algorithm for fault localization and prove that it is *must* with respect to commonly used schemes in automated program repair.

We incorporate the new fault localization technique into an existing mutation-based program repair algorithm. We exploit it in order to prune the search space when a buggy mutated program has been generated. Our experiments show that must fault localization is able to significantly speed-up the repair process, without losing any of the potential repairs.

1.1 Related Work

Program Repair Several repair methods follow a test-based generate and validate approach. GenProg [LNFW12] uses a genetic programming algorithm to go over a search space of programs created using deletions, replacements and insertions (only of preexisting code). This approach has been proved very effective for real-life bugs [LDVFW12]. The success of GenProg led to the development of several related tools, trying to improve it in different aspects. TrpAutoRepair [QML13a] and AE [WFF13] aim at reducing the cost of running the entire test suite during the validate stage, and RSRepair [QML+13b] suggests to use random search instead of genetic programming.

SPR [LR15] uses parametrized transformation schemes, each representing a class of useful program transformations. Then, target value search is used to reject schemes that can not produce a repair for any parameter value. Finally, condition synthesis is used in attempt to find appropriate parameter values for each of the remaining schemes, thus replacing it with a concrete condition that constitutes a repair. Prophet [LR16] improves SPR by better prioritizing the programs in SPR's search space, in an attempt to find "good" repairs (i.e. repairs that will be accepted by the user) faster. The priorities are determined using a probabilistic model learned automatically from a large code database containing successful human patches. PAR [KNSK13] and Monperrus and Martinez [MM15] also use successful human patches to learn schemes to be used for repair, but the learning is done by human observation rather than machine learning. Also, the resulting templates are fixed (i.e. contain no parameters) and only use existing code in the surroundings of the altered statement. CodePhage [SDLLR15] exploits the knowledge embedded in correct applications by directly transferring peaces of code from correct donor applications to buggy recipient ones. In this way it is able to eliminate out of bounds access, integer overflow, and divide by zero errors in the recipient.

In the literature there is also a wide range of techniques for automated program repair using formal methods [NQRC13, MYR16, ABS17, JGB05, VJ15, KKK15, DSS16, NWKF17]. SemFix [NQRC13], DirectFix [MYR15] and Angelix [MYR16] are test-based repair tools, but instead of iterating through the search space in an attempt to find a successful repair, they infer semantic information from the program and use it to synthesize a repair. All three tools assume the problem lies in faulty expressions and use controlled symbolic execution to infer a repair constraint. Next, they use component-based synthesis to synthesize new expressions satisfying this constraint. SemFix relies on the assumption that the program contains a single faulty expression. DirectFix manages to get rid of this assumption, at the cost of having a large repair constraint, resulting in low scalability. Angelix combines the best of both, having no assumption on the number of faulty expressions and at the same time a compact repair constraint.

Both [DW10] and [RHJ+12] use fault localization in order to produce a suspiciousness ranking, and then programs are mutated according to it. In [DW10] the Tarantula spectrum-based fault localization is used, and in [DW10] dynamic slicing is used. The

tool MUT-APR [AB18] fixes binary operator faults in C programs, but only targets faults that require one line modification. The tools FoREnSiC[BDF+12] and Maple[NTC19] repair C programs with respect to a formal specification by replacing expressions with templates, which are then patched and analysed. SemGraft[MNN+18] conducts repair with respect to a reference implementation, and relies on tests for SBFL fault localization of the original program.

Incremental Verification The validation of evolving software has been the subject of extensive research over the years (see the book by Chockler et al. [CKMS15]). Several different problems have been studied in this context, e.g., analyzing the semantic difference between successive revisions [TGK17] or determining which revision is responsible for a bug [MVP15, ABV16]. Here, we will focus on the problem of formally verifying all program revisions.

A dominant approach to solve this problem is to only verify the first revision, and then prove that every pair of successive revisions is equivalent. It was suggested by Godlin and Strichman in [SG08], where they gave it the name *regression verification* and introduced an algorithm that is based on the theory of uninterpreted functions. Papers about regression verification are concerned with improving equivalence checking and increasing its applicability. In [BPRT13], a summary of program behaviors impacted by the change is computed for both programs, and then equivalence is checked on summaries alone. Similarly, in [BOR13], checking equivalence is done gradually by partitioning the common input space of programs and checking equivalence separately for each set in the partition. In [FGK+14], a reduction is made from equivalence checking to Horn constraint solving. In [SVBV16] applicability is extended to pairs of recursive functions that are not in lock-step, and in [CGS12] to multi-threaded concurrent programs. The work of [BUVHW15] is focused on Programmable Logic Controllers, which are computing devices that control production in many safety-critical systems. Finally, [BV16] proposes a different notion of equivalence, which on top of the usual functional equivalence also considers runtime equivalence.

Another approach towards efficiently verifying all program revisions, which is the one we follow in our work, is to use during each revision verification partial results obtained from previous revisions, in order to limit necessary analysis. Work in this field vary based on the underlying non-incremental verification technique used, which determines what information can be reused and how efficiently so.

The work of Beyer et al. [BLN+13] suggests to reuse the abstraction precision in predicate abstraction. Other techniques for reuse of verification results include reuse of function summaries for bounded model checking [CGS12], contextual assumptions for assume-guarantee reasoning [HMW16], parts of a proof or counter-example obtained through ic3 [CIM+11] and inductive invariants [FGS14].

Also, incremental techniques for runtime verification of probabilistic systems modeled as Markov decision processes are developed in [FKP+12]. For the special case of

component-based systems, [JCK13] uses algebraic representations to minimize the number of individual components that need to be reverified. Last, the tool Green [VGD12] facilitates reuse of SMT solver results for general purposes, and authors demonstrate how this could be beneficial for incremental program analysis.

Fault Localization Approaches for fault localization include spectrum-based (SBFL) [JHS01, AZVG06, EDC10, NLR11, WDGL14], mutation-based (MBFL)[MKKY14, PT15, HLK+15, GZL15] and formula-based (FBFL) [JM11, ESW12, SSNW13, HSNB+16, CHM+19] techniques. Both SBFL and MBFL techniques compute the suspiciousness of a statement using coverage information from failing and passing test executions. MBFL uses, in addition, information on how test results change after applying different mutations to the program. A comprehensive survey of fault localization techniques can be found in [WGL+16].

FBFL techniques represent information about the bug using a logical formula, and analyze it to find suspicious locations. Analysis is done using error invariants [ESW12, CESW13, SSNW13, HSNB+16], maximum satisfiability solvers [JM11, LNH15, LN16], and weakest preconditions [CHM+19]. In contrast to SBFL and MBFL techniques, FBFL techniques are usually accompanied with a formal proof that the returned set of locations satisfies some property of interest, with respect to the formula.

The techniques of [JM11] and [ESW12] use a formula called the *extended trace formula* to encode the error trace of the bug. This formula encodes three things: a) that the input causing the bug remains the same, b) that the computation remains the same, i.e., that the same sequence of statements is executed in the same order, and, c) that the property holds at the end. Therefore, the formula is unsatisfiable. Both methods intuitively look for explanations of its unsatisfiability, from which they learn which parts of the computation were relevant for the bug to appear. The method of [JM11] finds a maximal-satisfiable-subset (MSS) of the formula, and returns its complement, a Co-MSS as the explanation. The method of [ESW12] use error invariants instead. An error invariant for a position in an error trace is a formula over program variables that over-approximates the reachable states at the given position while only capturing states that will still produce the error, if execution of the trace is continued from that position. If two successive positions along the trace have the same error invariant, then the execution of the statement between them had no effect on this set of states, and therefore the statement can be deemed irrelevant for the bug in question. Flow-sensitive fault localization [CESW13] enhances the traditional trace formula with information on the control-flow of the program. This new information is important since a possible fix could be to avoid reaching certain locations by altering a control-flow statement before them.

The relation between fault localization and program repair has also been studied in the past. In [YQM17], an empirical evaluation is made to compare the performance of the automated repair tool Nopol [DXLM14] using two fault localization strategies: rank-

first, which is based on suspiciousness rankings of statements, and suspiciousness-first, which is based on suspiciousness value of statements. The work of [LKB+19] studied the impact of different fault localization configurations on the performance of automated program repair tools. An interesting observation they make is that many bugs in the commonly used Defects4J benchmark cannot be fixed using any of the examined fault localization tools, because neither of them correctly identifies the location of the bug. Another important conclusion the authors draw is that success rates of repair tools are highly influenced by the fine-tuning of fault localization parameters, and therefore it is very important that repair tools are compared with emphasis on using the same parameters for fault localization (which is not the current practice). Finally, [LGL+20] proposes to incorporate fault localization and repair also in the other direction: feedback from the repair process can be used to improve the ranking returned by fault localization.

1.2 Thesis Structure

The thesis is constructed as follows: first, we present an overview of the methods used in our work, in chapter 2. Then, chapters 3,4 and 5 contain three of our papers, as published in [RG16, RDH18, RG20], respectively. These papers lay out the body of our work, including description of algorithms and proofs, and presentation of experimental results. They are arranged in chronological order, according to their publication time. Finally, chapter 6 concludes and discusses interesting directions for future work.

Chapter 2

Research Methods

The emphasis of this work is on **formal** program repair. Therefore, we have mostly used formal methods, taken from the world of verification and program analysis. In this chapter, we describe these methods in detail in order to provide the reader with necessary background for what comes next.

2.1 Incremental SAT and SMT Solving

A SAT solver is a decision procedure for deciding the satisfiability of a propositional formula. Formulas are usually in conjunction normal form (CNF) and can also be seen as a set of clauses.

Incremental SAT solving is a general name for a set of techniques aimed at improving the SAT solver's performance when called repeatedly for similar formulas (i.e., similar sets of clauses). The basic principal behind these techniques is to save running time by retaining information learned by the SAT solver between calls.

One of the common methods for incremental SAT solving is incremental SAT solving under assumptions, introduced by Minisat [ES04]. According to this method, at each step the SAT-solver is given a CNF formula φ_i and a set of assumption literals A_i, and solves $\varphi_i \wedge A_i$. The given formula must increase in every step, i.e the set of clauses in φ_i must be a subset of the set of clauses in φ_{i+1}, for all i. The set of assumptions, on the other hand, may change arbitrarily in each step.

An SMT solver (where SMT stands for satisfiability modulo theories), is another kind of decision procedure of much recent interest. It decides the satisfiability of a formula expressed in first order logic (FOL), where the interpretation of some symbols is constrained by a background theory (for more details see [DB09]). Examples of commonly used theories are the theory of linear arithmetic over integers and the theory of arrays. An SMT formula can be seen as a set of constraints in the theory (referred to as *SMT constraints*).

Similarly to SAT solving, incremental techniques can be applied to SMT solving as well. For this to be useful, an SMT formula φ is usually instrumented with boolean

variables called *guard variables*. The instrumentation of a formula φ is done as follows: each constraint $c_i \in \varphi$ is replaced by the constraint $x_i \rightarrow c_i$, where x_i is a fresh boolean variable. As a result, the new constraint can easily be satisfied by setting x_i to false. Guard variables are conjuncted with φ and are used as *assumptions*, passed to an incremental SMT solver. They have the effect of canceling out a subset of constraints. For example, if $\varphi = c_1 \wedge c_2$, after instrumentation we get the formula $\varphi' = (x_1 \rightarrow c_1) \wedge (x_2 \rightarrow c_2)$. Calling an incremental SMT solver on φ' with the set of assumptions $\{x_1\}$ causes the SMT solver to check the satisfiability of $\varphi' \wedge x_1$, which essentially disables the constraint c_2. That is, because nothing prevents x_2 from being set to false, and x_1 must be set to true, checking satisfiability of φ' is reduced to checking satisfiability of c_1.

2.2 Software Model Checking

Software model checking is the problem of checking whether a model of a software system meets its formal specification. In this thesis we focus on programs that are sequential and imperative. Also, we focus on safety properties, which assert that the system should never reach an error state (or states).

A common way to model a program in order to check a safety property is using its control flow graph (CFG). The set of vertices of the CFG is the set of program locations L, which contains a distinguished initial location, l_i, and a subset of distinguished error locations, L_e. Edges of the CFG are labeled with statements of the program. An edge (l_j, s, l_k) appears in the graph iff when the control of the program reaches location l_j, it is possible to continue to location l_k upon execution of the statement s.

A trace is an *error trace* of the program if it labels a path from l_i to some error location $l_e \in L_e$ in this graph. An error trace is *feasible* if there exists an execution of the program following it. The program is *correct* iff no error trace is feasible.

Bounded Model Checking

Bounded model checking is the problem of determining for a program P and an integer bound b, whether there exists a bug in P along executions of length at most b. Put in terms of the CFG, this means checking whether there exists a feasible error trace of length at most b. If there does not exist such a trace, then the program is said to be b-correct.

In our work, we have used the SMT-based bounded verification approach of [CKY03], implemented in the tool CBMC [CKL04]. This approach gets both the input program P and the bound b and converts P into a set of SMT constraints s.t. the program is b-correct iff the set of constraints is unsatisfiable (i.e the conjunction of all constraints in it is unsatisfiable). Then, an SMT solver is used to determine the satisfiability of this set. For more details on the conversion process, see section 3.3.1.

Trace-Abstraction-Based Verification

Another verification method that we rely on is the one proposed by Heizmann and Podelski in [HHP13]. At the basis of this verification method is the idea of looking at the basic statements of the program, i.e., its assignments and conditions, as letters of a finite alphabet. Following this point of view, the program can be seen as a finite automaton whose states are the program locations, and whose language is the set of error traces of the program.

Formally, a program P is modeled using an automaton $\mathcal{P} = (Q, \Sigma, q_0, \delta, F)$ where:

1. Q, the (finite) set of automaton states, is the set of all program locations Loc.
2. Σ, the alphabet of the automaton, is the set of all statements that appear in the program.
3. q_0, the initial state of the automaton, is the initial location l_i.
4. δ, the transition relation, is a subset of $Loc \times \Sigma \times Loc$ containing exactly those triples that are edges of the CFG.
5. F, the set of final states, is the set of error locations, L_e.

Note that the language of this automaton $L(\mathcal{P})$, is indeed the set of error traces of the program.

The way the verification method works is by constructing an abstraction $\mathcal{A}$ of the set of feasible program traces, called a *trace abstraction*, which is a sequence of automata over the alphabet of statements.

The algorithm consists of a counter-example-guided-abstraction-refinement loop, where $\mathcal{A}$ is initially empty (i.e., contains no automata). In each iteration of the loop, an uncovered error trace π is sampled, by checking the emptiness of the language $L(\mathcal{P}) - \left(\bigcup_{\mathcal{A}_i \in \mathcal{A}} L(\mathcal{A}_i)\right)$. Next, the feasibility of π is examined: if it is feasible- then the program is not correct; if it isn't feasible- then an automaton $\mathcal{A}_i$ is constructed using the proof of its unfeasibility. By construction, $\mathcal{A}_i$ accepts π along with other traces that are, intuitively, unfeasible for the same reasons as π. $\mathcal{A}_i$ is then added to $\mathcal{A}$ and the loop is repeated.

Chapter 3

Sound and Complete Mutation-Based Program Repair

Abstract

This work presents a novel approach for automatically repairing an erroneous program with respect to a given set of assertions. Programs are repaired using a predefined set of mutations. We refer to a bounded notion of correctness, even though, for a large enough bound all returned programs are fully correct. To ensure no changes are made to the original program unless necessary, if a program can be repaired by applying a set of mutations Mut, then no superset of Mut is later considered. Programs are checked in increasing number of mutations, and every minimal repaired program is returned as soon as found.

We impose no assumptions on the number of erroneous locations in the program, yet we are able to guarantee soundness and completeness. That is, we assure that a program is returned iff it is minimal and bounded correct.

Searching the space of mutated programs is reduced to searching unsatisfiable sets of constraints, which is performed efficiently using a sophisticated cooperation between SAT and SMT solvers. Similarities between mutated programs are exploited in a new way, by using both the SAT and the SMT solvers incrementally.

We implemented a prototype of our algorithm, compared it with a state-of-the-art repair tool and got very encouraging results.

3.1 Introduction

In the process of software production and maintenance, much effort and many resources are invested in order to ensure that the product is as bug free as possible. Manual bug repair is time-consuming and requires close acquaintance with the checked program. Therefore, there is a great need for tools performing automated program repair. In recent years, there has been much progress in this field (e.g., [LNFW12, NQRC13, MYR16, DXLM14, LR15, KB13]).

In previous work, the presented motivation for the development of program repair tools is to enable the automatic repair of real-world bugs found in large-scale software

projects. As a result, existing tools for automated repair aim at being scalable and are targeted for the type of bugs found in deployed software.

We have designed our algorithm with a different goal in mind. In our opinion, automatic repair can be equally or even more useful when applied in the earlier stages of development, before any manual effort was invested in debugging at all. This is because, in our view, it is precisely the initial debugging work that could benefit the most from this automation, since it involves relatively simple bugs being fixed manually by the programmer. For these early development stages, as well as for millions of independent programmers working on small pieces of code, even a non-scalable automatic repair method can help save a lot of time and avoid much frustration.

Our vision is to have a fast, easy-to-use program repair tool, which programmers can run routinely. Ideally, programmers will run the tool immediately after making changes to the program, before any manual effort was invested in debugging at all. Then, if the program contains an assertion violation, the chosen course of action will be determined by the tool's result. If the tool returns one or more possible repairs, those are guaranteed to suppress all assertion violations and thus may be safely applied to the program. If the tool does not return any possible repairs, the programmer can be sure that the problem can not be solved using changes within the search space of the tool. In the later case, though manual debugging will still be needed, knowing what will not solve the problem might give the programmer a head start.

In this work, we take a step forward towards accomplishing this vision, presenting a novel algorithm for automatically repairing a program with respect to a given set of assertions. We use a bounded notion of correctness. That is, for a given bound b, we consider only *bounded computations*, along which each loop in the program is performed at most b times and each recursive call is in-lined at most b times. We say that a program is *repaired* if whenever a bounded computation reaches an assertion, the assertion is evaluated to true. Our repair method is *sound*, meaning that every returned program is repaired (i.e., no violation occurs in it up to the given bound). Just like Bounded Model Checking, this increases our confidence in the returned program.

Our programs are repaired using a predefined set of mutations, applied to expressions in conditionals and assignments (e.g. replacing a $+$ operator by a $-$), as was shown useful in previous work [DW10, RHJ$^+$12, DW14]. We impose no assumptions on the number of mutations needed to repair the program and are able to produce repairs involving multiple buggy locations, possibly co-dependent. To make sure that our suggested repairs are as close to the original program as possible, the repaired programs are examined and returned in increasing number of mutations. In addition, only *minimal* sets of mutations are taken into account. That is, if a program can be repaired by applying a set of mutations Mut, then no superset of Mut is later considered. Intuitively, this is our way to make sure all changes made to the program by a certain repair are indeed necessary. Our method is *complete* in the sense of returning *all* minimal sets of mutations that create a repaired program. Specifically, if no repair is found, one can conclude that the

given set of mutations is not enough to repair the program. Furthermore, we show that for large enough bound, all returned programs are (unbounded) fully correct.

Note that, the choice to use mutations for repair makes the search space small enough to enable us to have completeness at an affordable cost, yet it is expressive enough to repair meaningful bugs (especially those present in earlier stages of development).

Our algorithm is based on the translation of the program into a set of SMT constraints which is satisfiable (i.e., the conjunction of constraints in it is satisfiable) iff the program contains an assertion violation. This was originally done for the purpose of bounded model checking in [CKL04][1]. Our key observation is that mutating an expression in the program corresponds to replacing a constraint in the set of constraints encoding the program. Thus, searching the space of mutated programs is reduced to searching unsatisfiable sets of constraints. The latter can be performed efficiently using a sophisticated cooperation between SAT and SMT solvers, as was done in [LPMMS16] for the purpose of finding minimal unsatisfiable cores.

The SAT solver is used to restrict the search space of mutated programs to only those obtained by a minimal mutation set and the SMT solver verifies whether a mutated program is indeed correct. Both the SAT solver and the SMT solver are used incrementally, which means that learned information is passed between successive calls, resulting in big savings in terms of resources used. Using an SMT solver incrementally constitutes a novel way to exploit information learned while checking the correctness of one program for the process of checking correctness of another program. Note, that if the programs are similar, their encoding as sets of SMT constraints will also be similar (due to our observation presented above), resulting in bigger savings when using incremental SMT. This is another important contribution of this paper.

We implemented a prototype of our algorithm for C programs, compared it with the methods of [KB11, KB13] and got very encouraging results.

To summarize, the main contributions of our work are:

- We propose a novel *sound and complete* algorithm which returns *all* minimal repaired programs.

- The returned programs are proved to be bounded correct. However, we show that for a large enough bound, all returned programs are fully correct and all minimal fully correct programs are returned.

- We develop an efficient implementation of the algorithm, based on sophisticated cooperation between SAT and SMT solvers, both used incrementally.

[1] To be precise, [CKL04] first translates the program into a bit-vector formula and then further translates it into a propositional formula. Here, we only use the first part of the translation

3.1.1 Related Work

Several repair methods follow a test-based "generate and validate" approach. They iteratively select a candidate from the repair search space and check its validity by running all tests in the test suite against it. Examples are GenProg [LNFW12, LDVFW12], TrpAutoRepair [QML13a], AE [WFF13], RSRepair [QML+13b] and the more recent SPR [LR15]. PAR [KNSK13], Monperrus and Martinez [MM15] and Prophet [LR16] suggest to use information learned from successful human repairs to extract and prioritize repair actions suitable for the suspected location of the error. Similarily, CodePhage [SDLLR15] directly transfers pieces of code from correct donor applications to buggy recipient ones. AutoFix-E [WPF+10] and AutoFix [PFN+14] also use location based repair actions, but require programs to be equipped with contracts.

SemFix [NQRC13], DirectFix [MYR15] and Angelix [MYR16] use symbolic execution to infer a repair constraint and synthesize a repair based on it. Nopol [DXLM14] also uses synthesis, but only deals with buggy if conditions and missing pre conditions. [KKK15] uses deductive synthesis and is based on pre and post conditions, rather than tests alone. [JGB05] and [VJ15] describe systems using automata and use LTL specifications for repair.

Mutation based program repair (where the term "mutation" has the same meaning as in this work) was previously done in [RHJ+12] and [DW14]. Both use a test suite as the only specification and focus their efforts on efficient error localization. We, on the other hand, use a formal specification and have no use of localization, since we have to consider all locations in order to guarantee completeness. Also, we allow the repair of multiple expressions, whereas both methods assume a single fault ([DW14] mentions a possible extension to multiple faults, but this is not a part of the described method).

Finally, he methods of [KB11, KB13] are similar to ours in that they work on C programs equipped with assertions (or test suites) and assume faulty expressions. The differences are that they use program analysis based on a finite number of inputs each time, while we use incremental SMT solving that allows reuse of information. Also, they use templates (e.g. a linear combination of variables) for repair, while we use mutations and are able to guarantee completeness. We provide a comparison of performance results between our method and theirs in sec. 3.6.

3.2 Preliminaries

Program Correctness

For our purposes, a *program* is a sequential program composed of standard commands: assignments, conditionals, loops and function calls. Each command is located at a certain *program location* l_i, and all commands are defined over the set of program variables X.

In addition to the standard commands, a program may contain assumptions and

assertions, which are commands that help the user specify the desired behavior. *Assumptions* (resp., *assertions*) are commands of the form assume(e) (resp., assert(e)), where e is a boolean expression over X. An assertion assert(e) at location l_i, specifies that the user expects e to evaluate to true whenever control reaches l_i, in all program runs. If e evaluates to true every time control reaches l_i during a run r, we say the assertion *holds* for r. Otherwise, the assertion is *violated*. Once an assertion in the program is violated, the program terminates (this early termination indicates an error has occurred and is usually preceded by an error message explaining what went wrong). An assumption assume(e) at location l_i, specifies that every run reaching l_i with e evaluated to false is terminated. Unlike before, this early termination is not an indication that something went wrong, but simply that the user does not want to consider the rest of this run when checking correctness. For example, if a function f gets as input an integer n, but the user assumes it will only be called with $n \geq 2$, an assumption $assume(n \geq 2)$ can be inserted at the beginning of the function to make sure all runs in which this function is called inappropriately will be truncated.

Definition 3.2.1 (correct program)**.** A program is correct if all assertions in it hold in all runs.

For a program P and an integer b, a *b-run* of P is a run of P that goes through each loop at most b times and has a recursion depth of at most b (i.e., the depth of the call stack is at most b during the entire run).

Definition 3.2.2 (b-correct program)**.** Let b be an integer. A program is b-correct if all assertions in it hold in all b-runs.

Our repair method aims at finding programs which are b-correct, therefore we use the term *repaired program* as a notation for a b-correct program.

3.2.1 Incremental SAT and SMT Solving

A SAT solver is a decision procedure for deciding the satisfiability of a propositional formula. Formulas are usually in conjunction normal form (CNF) and can also be seen as a set of clauses. *Incremental SAT solving* is a general name for a set of techniques aimed at improving the SAT solver's performance when called repeatedly for similar formulas (i.e., similar sets of clauses). The basic principal behind these techniques is to save running time by retaining information learned by the SAT solver between calls.

An SMT solver (where SMT stands for satisfiability modulo theories), is another kind of decision procedure of much recent interest. It decides the satisfiability of a formula expressed in first order logic (FOL), where the interpretation of some symbols is constrained by a background theory (for more details see [DB09]). Examples of commonly used theories are the theory of linear arithmetic over integers and the theory of arrays. Just like a CNF formula can be seen as a set of clauses, an SMT formula can be seen as a set of constraints in the theory (referred to as *SMT constraints*).

Similarly to SAT solving, incremental techniques can be applied to SMT solving as well. For this to be useful, an SMT formula φ is usually instrumented with boolean variables called *guard variables*. The instrumentation of a formula φ is done as follows: each constraint $c_i \in \varphi$ is replaced by the constraint $x_i \rightarrow c_i$, where x_i is a fresh boolean variable. As a result, the new constraint can easily be satisfied by setting x_i to false. Guard variables are conjuncted with φ and are used as *assumptions*, passed to an incremental SMT solver. They have the effect of canceling out a subset of constraints. For example, if $\varphi = c_1 \wedge c_2$, after instrumentation we get the formula $\varphi' = (x_1 \rightarrow c_1) \wedge (x_2 \rightarrow c_2)$. Calling an incremental SMT solver on φ' with the set of assumptions $\{x_1\}$ causes the SMT solver to check the satisfiability of $\varphi' \wedge x_1$, which essentially disables the constraint c_2. That is, because nothing prevents x_2 from being set to false, and x_1 must be set to true, checking satisfiability of φ' is reduced to checking satisfiability of c_1.

Boolean Cardinality Constraints

Boolean cardinality constraints are constraints of the form $\sum_{i=1}^{n} l_i \leq k$, where l_i is a literal assigned the value 1 if true and 0 if false, and k is an integer constant. For readability, we will refer to these constraints using the notation AtMost$(\{l_1, .., l_n\}, k)$, also used in [LS08], in order to remind the reader of their intuitive meaning: require that at most k of these literals get the value true. Similarly, the notation AtLeast$(\{l_1, .., l_n\}, k)$, denotes the constraint $\sum_{i=1}^{n} l_i \geq k$. For our implementation we used Minicard [LM12], which is a SAT-solver designed to perform well on instances containing cardinality constraints.

3.3 Our Approach

In this section we fix a bound b and refer to repaired programs which are b-correct. Figure 3.1 presents an overview of our repair system. It is composed of three units: the translation unit, the mutation unit and the repair unit.

The initial processing is done in the translation unit. The translation unit translates the input program into two sets of SMT constraints: S_{hard}, encoding parts of the program which cannot be changed (e.g. assertions), and S_{soft}. Then, the mutation unit constructs for each constraint c_i in S_{soft} a set of alternative constraints S_i, by applying mutations to c_i. Finally, the repair unit searches for all sets of constraints encoding minimal repaired programs (where minimality will be defined with respect to the set of mutations used). In the rest of the section we explain in detail how each unit works.

3.3.1 The Translation Unit

The translation unit is the first step of the process. It gets an input program and an integer bound b and converts the input program into a set of SMT constraints s.t. the

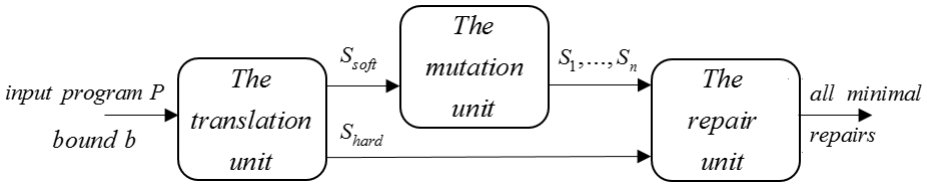


Figure 3.1: Overview of the repair system

program is b-correct iff the set of constraints is unsatisfiable (i.e the conjunction of all constraints in it is unsatisfiable).

Before the set of constraints is constructed, the program undergoes three transformations: simplification, unwinding, and conversion to static single assignment (SSA) form. This transformations are taken from [CKL04], but we present them here because the details are important in order to understand our method.

To explain the different transformations we will use the example presented in Figure 3.2. Figure 3.2a presents a C function named sum, which gets as input an integer n and is supposed to return $\sum_{1}^{n} i$. But, being used to 0-based counting, the programmer made a mistake in line 3, by initializing i to 0 instead of 1 and checking $i < n$ instead of $i <= n$. The assertion in line 6 specifies that the result should always be calculated according to the formula $\frac{n \cdot (n+1)}{2}$, which is the correct sum calculated using the formula for a sum of an arithmetic progression.

We will now go over each transformation and explain its role shortly, using the described example.

Simplification Figure 3.2b shows the result of applying simplification to the program in Figure 3.2a. Complex constructs are replaced with simpler ones (for example, the for loop was replaced with a while loop). More importantly, all conditions are assigned to auxiliary boolean variables (g in the example). Note that after this step, all original program expressions are right-hand-sides of assignments.

Unwinding Figure 3.2c shows the result of applying unwinding for $b = 2$ to the program in Figure 3.2b. The loop is unwound b times by duplicating the loop body b times, where each copy is guarded using an if statement that uses the same condition as the loop statement (lines 5-15). Inside the innermost copy, an assume statement is inserted with the negation of the condition (line 13), to specify we do not want to consider runs going through the loop more than b times.[2] Function calls are inlined, with recursive calls treated similarly to loops (inserted up to a depth of b).

Conversion to SSA Form The program is converted to SSA form (which means each variable is assigned only once). Figure 3.2d shows the result of converting the program in Figure 3.2c to SSA form. All variables are replaced with indexed variables, and whenever a variable appears as the left-hand-side of an assignment, its index is

[2] In [CKL04] an assertion was inserted and not an assume. Since we fix the program with respect to all assertions in it, we need this to be an assume and not an assert, because we do not want to refer to unbounded runs as bugs.

increased by 1. If a variable x is assigned inside a conditional statement and is used after the statement, an assignment is inserted straight after the conditional statement to determine which copy of x should be used. For example, lines 16-17 determine the updated value of *sum* after the nested if statements, according to $g1$ and $g2$. We refer to this type of assignments as Φ-assignments.

After the above transformations, conversion to a set of SMT constraints S is straightforward. An assignment $x = e$ is converted to the constraint $x = e$, an $assume(e)$ is converted to the constraint e and an $assert(e)$ is converted to the constraint $\neg e$.[3] Shortly, we say that a constraint *encodes* a statement.

In the next step, the mutation unit will apply mutations independently to every constraint passed to it. The problem is that, due to unwinding, all statements which are part of a loop (as the loop condition or in the loop body) are encoded using more than one constraint in S. This is of course undesirable, because we do not want constraints encoding the same statement to be mutated using different mutations. To avoid this, if a statement s is encoded using the constraints $c_1, ..., c_t \in S$ (where $t > 1$), we remove $c_1, ..., c_t$ from S, and add instead one complex constraint, $\bigwedge_{i=1}^{t} c_i$. Note that this has no effect on the satisfiability of S (which is determined by the conjunction of all constraints in S anyway).

As a final step, the modified set S is partitioned into two sets: S_{soft}, containing all constraints encoding statements subject to repair (i.e statements containing original program expressions), and S_{hard}, containing the rest (constraints encoding negated assertions, assumptions and Φ-assignments). Note that since we made sure all original program expressions are right-hand-sides of assignments using simplification, we can be sure all constraints in S_{soft} are of the form $(x = e)$ (where x is an SSA variable and e is an expression), or of the form $(c_1 \wedge c_2, ..., \wedge c_n)$ where each c_i is of the form $(x = e)$. Furthermore, we can be sure all program statements which are subject to repair are encoded using a single constraint and vice versa, and thus the size of S_{soft} will always be the same as the number of original program expressions (regardless of the bound b).

3.3.2 The Mutation Unit

We assume the program is incorrect because it contains one or more faulty expressions, and we try to repair it by applying mutations to program expressions. A *mutation* can be any function mapping a program expression to another program expression of the same type. Examples of mutations include replacing an operator by a similar one (e.g., $\leq$ by $<$) and applying constant manipulations (e.g., replacing a constant by 0). The mutation unit is the component in charge of applying the mutations. In fact, as described in Fig. 3.1, the mutations are not applied directly on the program, but on constraints encoding the program, received from the translation unit.

[3] Assertions are negated because we want a satisfying assignment to the set of constraints to represent a violation of the assertion. If multiple assertions exist in the code, the disjunction of their negations is added as a constraint.

```
int sum(int n){
1.    assume(n>=1);
2.    int sum = 0;
3.    for (i=0;i<n;i++){
4.        sum += i;
5.    }
6.    assert(sum=n*(n+1)/2);
7.    return sum;
}
```

(a) Original program

```
int sum(int n){
1.    assume(n>=1);
2.    int sum = 0;
3.    int i = 0;
4.    bool g = i<n;
5.    while (g){
6.        sum = sum+i;
7.        i = i+1;
8.        g = i<n;
9.    }
10.    assert(sum=n*(n+1)/2);
11.    return sum;
}
```

(b) Program after simplification

```
int sum(int n){
1.    assume(n>=1);
2.    int sum = 0;
3.    int i = 0;
4.    bool g = i<n;
5.    if (g){
6.        sum = sum+i;
7.        i = i+1;
8.        g = i<n;
9.        if (g){
10.            sum = sum+i;
11.            i = i+1;
12.            g = i<n;
13.            assume(!g)
14.        }
15.    }
16.    assert(sum=n*(n+1)/2);
17.    return sum;
}
```

(c) Program after unwinding for $b = 2$

```
int sum(int n_1){
1.    assume(n_1>=1);
2.    int sum_1 = 0;
3.    int i_1 = 0;
4.    bool g_1 = i_1<n_1;
5.    if (g_1){
6.        sum_2 = sum_1+i_1;
7.        i_2 = i_1+1;
8.        g_2 = i_2<n_1;
9.        if (g_2){
10.            sum_3 = sum_2+i_2;
11.            i_3 = i_2+1;
12.            g_3 = i_3<n_1;
13.            assume(!g_3)
14.        }
15.    }
16.    sum_4 = g_2 ? sum_3 : sum_2;
17.    sum_5 = g_1 ? sum4 : sum_1;
18.    assert(sum_5=n_1*(n_1+1)/2);
19.    return sum_5;
}
```

(d) Program after conversion to SSA

Figure 3.2: Example of program transformations during translation

As explained in Section 3.3.1, the constraints in the input set, S_{soft}, can be single assignment constraints or multiple assignments constraints. Formally, given a mutation M, and a single assignment constraint $(x = e)$, $M(x = e)$ is the constraint $(x = M(e))$. For a multiple assignment constraint $c = (c_1 \wedge c_2 \wedge ... \wedge c_t)$, $M(c)$ is the constraint $(M(c_1) \wedge M(c_2) \wedge ... \wedge M(c_t))$.

The mutation unit maintains a fixed list of possible mutations, $M_1, M_2, ..., M_m$. For each $c_i \in S_{soft}$ $(1 \leq i \leq n)$ all the mutations are applied and the set $S_i = \{c_i, M_1(c_i), ..., M_m(c_i)\}$ is created.[4] Note that the set S_i contains the original constraint c_i, so leaving a statement intact is always an option. Finally, the sets $S_1, ..., S_n$ are passed on to the repair unit, which uses them to search for a repair.

3.3.3 The Repair Unit

Basic Terms and Definitions The input to the repair unit is a set of "hard constraints", S_{hard}, encoding the parts of the program which can not be changed, and n disjoint sets of "soft constraints", $S_1, ..., S_n$, corresponding to n program locations where a possible fault may occur. Every set S_i contains one special constraint, c_o^i, encoding the original statement in line i, referred to as the *original constraint*. The rest of the constraints in S_i encode possible replacements for line i, obtained by applying mutations to the expression in the original statement.

Intuitively, the goal of the repair unit is to construct a repaired program by choosing one constraint from each S_i. Formally, we define a *selection vector* (**sv**) $[c_1, ..., c_n]$ as a vector of constraints where c_i is taken from S_i for all $1 \leq i \leq n$. Recall that constraints in S_i encode different statements for line i, therefore choosing a specific constraint from each S_i can be seen as choosing a statement to appear in each line, i.e choosing a mutated program. Thus, each selection vector *encodes* a program. We are interested in selection vectors encoding repaired or correct programs. This leads to the following definitions.

Definition 3.3.1 (**Rsv**,**Csv**)**.** A selection vector is repaired, denoted **Rsv**, if it encodes a repaired program. A selection vector is correct, denoted **Csv**, if it encodes a correct program.

Though (bounded) correctness is essential for repair, it is not enough. We would also like for the repair to be "minimal", in the sense that no changes are made unless necessary. For example, if a program can be repaired by applying a certain mutation to line number 2, we are not interested in a repair suggesting to additionally mutate line number 3, even if it makes the program repaired. To capture this intuition we define a partial order between constraints and between selection vectors.

[4]This is a simplification made for ease of presentation. In practice, we might not be able to (or not want to) apply all mutations to all constraints. The choice of mutations to use may depend on the expression's type and/or its complexity.

Definition 3.3.2 ($\sqsubseteq$ partial order between constraints)**.** Let $c_i^1, c_i^2 \in S_i$. $c_i^1 \sqsubseteq c_i^2$ if $c_i^1 = c_o^i$ and $c_i^2 \neq c_o^i$ (i.e., only c_i^2 encodes a change to line i), or if $c_i^1 = c_i^2$ (i.e., both encode the same statement for line i).

Definition 3.3.3 ($\sqsubseteq$ partial order between **sv**s)**.** Let $v_1 = [c_1^1, ..., c_n^1], v_2 = [c_1^2, ..., c_n^2]$ be selection vectors. $v_1 \sqsubseteq v_2$ if for all $1 \leq i \leq n$ $c_i^1 \sqsubseteq c_i^2$.

Definition 3.3.4 (**mRsv**,**mCsv**)**.** A repaired selection vector v is minimal repaired, denoted **mRsv**, if there is no v' s.t. $v' \neq v$, v' is a repaired selection vector and $v' \sqsubseteq v$.

A correct selection vector v is minimal correct, denoted **mCsv**, if there is no v' s.t. $v' \neq v$, v' is a correct selection vector and $v' \sqsubseteq v$.

Finally, it makes sense to prefer repairs involving as few statements as possible, because those are more likely to satisfy the user. For example, if the program can be repaired by mutating line 1 and also by mutating lines 2 and 3, the first repair is preferable. This intuition is formalized using the following definition:

Definition 3.3.5 (size)**.** Let v be a selection vector. The size of v, denoted size(v), is $|\{i | 1 \leq i \leq n, v[i] \neq c_o^i\}|$.

In other words, size(v) is the number of mutated lines in the program encoded by v. Thus, the repair unit should only look for minimal repaired selection vectors, and amongst them prefer those with smaller size. In what follows, we present an algorithm that computes *all* minimal repaired selection vectors (**mRsv**s), and produces results in increasing size over time.

3.4 Algorithm AllRepair for the Repair Unit

3.4.1 Outline of the Algorithm

Figure 3.3 presents the general outline of our algorithm. Overall, the algorithm goes over the search space of all **sv**s, in increasing size order. This order is enforced using the variable k, which limits the allowed size of the searched **sv**s (k is initially 1 and grows over time)[5]. Once the search reaches an **sv** v, we say v has been *explored* (until then, v is *unexplored*). The algorithm is divided into two repeating phases:

[5] k is not to be confused with the unwinding bound b, which is fixed at this point.

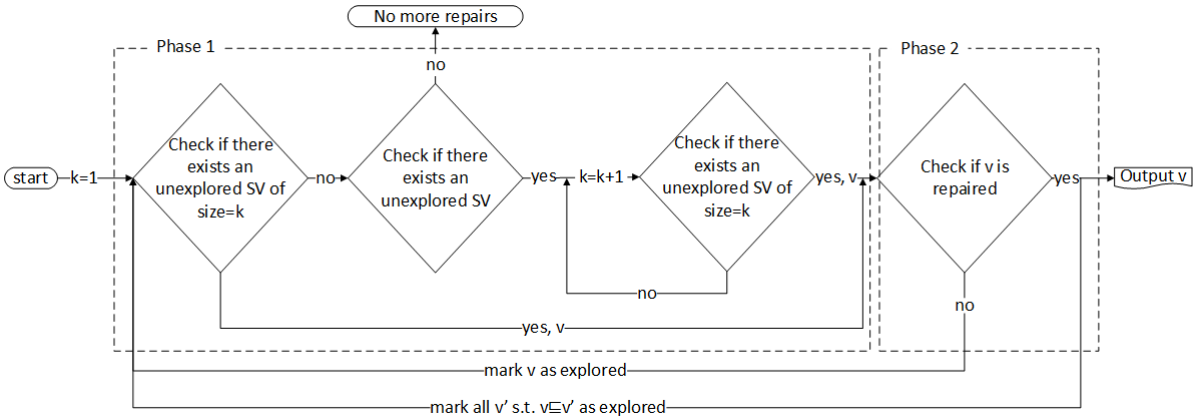


Figure 3.3: Outline of Algorithm **AllRepair**

Phase 1 is responsible for finding the next unexplored **sv**. First, it looks for an unexplored **sv** of size k. If one exists, it is passed on to Phase 2. Otherwise, it checks if there exist any unexplored **sv**s left at all. If not, the search is over and the procedure ends. Otherwise, k is repeatedly increased by one until an unexplored **sv** v of size k is found (v must be found for some k since we know an unexplored **sv** exists). Once found, v is passed on to Phase 2.

Phase 2 gets as input an unexplored **sv** v. First, it checks if v is repaired, that is, if v is b-correct. If it is, v is returned as a possible repair. In addition, if v is repaired, Phase 2 marks not only v as explored, but also every **sv** v' s.t. $v \sqsubseteq v'$. This is done in order to make sure that we will not waste time exploring v' in the future, since it is necessarily not minimal. If v is not repaired, then only v is marked as explored.

3.4.2 Algorithm AllRepair in Detail

The pseudo-code of algorithm **AllRepair** is presented in Figure 3.4. This algorithm follows the general outline presented before, where an incremental SAT-solver with cardinality constraints is used for the implementation of Phase 1, and an incremental SMT-solver is used for the implementation of Phase 2. Note that, we are interested in the *satisfying assignments* returned by the SAT solver and in the *unsatisfiable instances* returned by the SMT solver. The former represent **sv**s of desired sizes while the latter represent repaired programs.

The description below is strongly based on the background given in Section 3.2.1. The first step is to instrument all constraints in $S_1, \ldots, S_n$ with guard variables. This is done using a function call in line 2, and the results are the sets of instrumented constraints, $S'_1, \ldots, S'_n$ (where $S'_i = \{x_j \rightarrow c_j \mid \text{for every } c_j \in S_i\}$) and the sets of fresh guard variables used to guard the constraints in each set, $V_1, \ldots, V_n$ (where V_i contains the variables x_j used to guard constraints in S_i). This instrumentation serves us in building both the SMT formula τ and the boolean formula φ (passed to the SAT-solver).

```
1: function AllRepair(Input: S_hard, S_1, ..., S_n,  Output: All mRsvs)
2:     S'_1, ..., S'_n, V_1, ..., V_n ← AddSelVars(S_1, ..., S_n)
3:     τ ← true                                          ▷ initialization of SMT formula
4:     for c ∈ S_hard ∪ S'_1 ∪ ... ∪ S'_n do
5:         τ ← τ ∧ c
6:     φ ← true                                          ▷ initialization of boolean formula
7:     for 1 ≤ i ≤ n do
8:         φ ← φ∧AtMost(V_i,1)                           ▷ choose at most one statement per line
9:         φ ← φ∧(⋁_{v∈V_i} v)                           ▷ choose at least one statement per line
10:    V_o ← GetSelVarsOfOriginal(V_1, ..., V_n)
11:    k ← 1
12:    while true do
13:        φ_k ← φ ∧ AtLeast(V_o, n − k)
14:        satRes, V ← SAT(φ_k)
15:        if satRes is unsat then
16:            if ¬SAT(φ) then                           ▷ No more svs to explore
17:                return
18:            repeat
19:                k ← k + 1
20:                φ_k ← φ ∧ AtLeast(V_o, n − k)
21:                satRes, V ← SAT(φ_k)
22:            until satRes is sat
23:        smtRes ← IncrementalSMT(τ, V)                 ▷ at this point V has been assigned
24:        if smtRes is SAT then
25:            φ_Block ← BlockUnrepairedsv(V)
26:        else
27:            output Getsv(V, S'_1, ..., S'_n)
28:            φ_Block ← BlockRepairedsv(V)
29:        φ ← φ ∧ φ_Block                               ▷ φ includes new blocking; k is not changed
```

Figure 3.4: Algorithm **AllRepair** for finding all **mRsv**s

Next, in lines 3-6, τ is initialized to the conjunction of all constraints in S_{hard} and all the instrumented constraints. Notice that this will enable us to determine which of the soft constraints will be considered in each call to the SMT solver, by using their guard variables as assumptions (while hard constraints will be considered in all calls, regardless of the assumptions).

The boolean formula φ is initialized in lines 7-11. The boolean variables composing this formula are the guard variables $V_1, ..., V_n$, and therefore every satisfying assignment of it can be seen as a subset of guard variables (those assigned true by the assignment). We would like every satisfying assignment to be not just any subset of guard variables, but one consistent with the definition of an **sv**, i.e., a subset that contains exactly one selector variable from each V_i. Lines 9-10 add to φ the necessary constraints to enforce this. From now on, we will say that satisfying assignments returned by the SAT-solver *represent* **sv**s.

Next, we would like to be able to add an upper bound on the size of represented **sv**s. For this purpose, we define an additional formula, φ_k. In order to construct φ_k, we first need to identify which guard variables guard the original constraints. This is done in the function call in line 12, and the result is stored in V_o.

Lines 13-26 essentially implement Phase 1 of the outline in Figure 3.3. k is initialized to 1 (line 13) and the iterative repetition of the two phases begins. First, φ_k is set to the conjunction of φ and the clause AtLeast$(V_o, n-k)$ (line 15). That is, in φ_k we additionally require that at least $n-k$ variables from V_o get the value true. This essentially means that every satisfying assignment to φ_k now represents an **sv** of size at

most k.

Next, we check whether there exists an unexplored **sv** of size at most k by sending φ_k to the SAT solver (line 16). The satisfiability result (sat/unsat) is saved into $satRes$, and if the result is sat, V gets the set of all variables assigned true by the satisfying assignment. If the result is unsat, we check whether there exists an unexplored **sv** (without limitation on size) by sending φ to the SAT solver (line 18). If the result is unsat, the algorithm ends (line 19). Otherwise, we repeatedly increase k by one and resend φ_k to the SAT solver, until the result is sat (lines 21-25).

Phase 2 begins in line 27, by calling the function IncrementalSMT(τ, V), which checks the satisfiability of τ with all variables in V passed as assumptions. This is in fact equivalent to checking the satisfiabillity of the conjunction of all constraints in S_{hard} and all soft constraints guarded by variables in V (since all other constraints can be easily satisfied by setting their guard variables to false). Note that this formula is unsat iff the **sv** represented by V (i.e., the constraints guarded by variables in V) is an **Rsv**. Therefore, if the result is sat, we create a blocking clause φ_{Block} for the case in which V represents an **sv** that is not repaired (line 29). The blocking clause in this case is simply $\bigvee_{v \in V} \neg v$ (i.e only V is blocked). If the result is unsat, we translate V into the represented **sv** and return it as a possible repair (line 31). The blocking clause we add in this case (line 32) is $\bigvee_{v \in V \setminus V_o} \neg v$, which requires that the same set of mutations will never appear as a subset of any future set of mutations. This way we block not only V but also every V' for which $v \sqsubseteq v'$ (where v, v' are the **sv**s represented by V, V', respectively).

3.5 Soundness and Completeness of Algorithm AllRepair

In this section we analyze our algorithm. We show that it is *sound*, that is, every returned **sv** is minimal repaired, and that it is *complete* in the sense that every minimal repaired **sv** is eventually returned.

Clearly, the algorithm returns all **mRsv**s, because we go over all **sv**s and only mark an **sv** as explored if it is returned (as repaired), if it is not repaired, or if it is not minimal. Also, all **sv**s returned by the algorithm are **mRsv**s, because every returned **sv** is repaired (it is explicitly checked), and is minimal repaired because otherwise it would have been marked as explored by another **sv** in a previous iteration. Thus, the following theorem holds:

Theorem 3.1 (Correctness of **AllRepair**)**.** *Our algorithm is sound and complete. That is, every* **sv** *v returned by our algorithm is an* **mRsv** *and every* **mRsv** *v is returned by our algorithm at some point.*

3.5.1 Extension to Full Correctness

We now analyze the soundness and completeness of our algorithm with respect to full (unbounded) correctness. We show that there is a bound B for which the notion of B-correctness is equivalent to the notion of correctness.

We first notice that since the set of mutations we consider is finite, so is the set of mutated programs PG. For each $P \in PG$, if it is not correct then it has a b-run for some b, along which some assertion is violated. Let b_P be the smallest bound for which such a run exists for P. Then, by definition, P is not b-correct for any b greater than b_p. Let *max-bound* B be defined as follows. $B = 1 + max\{b_P \mid P \in PG$ and P is not correct$\}$. Clearly, for every program P in PG, P is B-correct iff P is correct. The following theorem describes this observation by means of the selection vectors encoding programs in PG.

Theorem 3.2 (Equivalence of B-correctness and Full correctness)**.** *Let B be the max-bound defined above. Then v is an* **Rsv** *for bound B iff v is a* **Csv***. Further, v is an* **mRsv** *for bound B iff v is an* **mCsv***.*

Proof. The first part of the theorem is a direct consequence of the definition of B. The second part of the theorem is a direct consequence of the first part. This is because, by definition, v is an **mRsv** for bound B iff v is an **Rsv** for B and every v' s.t. $v' \sqsubseteq v$ and $v' \neq v$ is not an **Rsv** for B. By the first part, this happens iff v is a **Csv** and every v' s.t. $v' \sqsubseteq v$ and $v' \neq v$ is not a **Csv**, which means v is an **mCsv**.

Theorem 3.2 implies that for a large enough bound, all returned programs are correct and all minimal correct programs are returned.

3.6 Experimental Results

We implemented a prototype of our algorithm on top of two existing tools. The translation unit and the mutation unit were implemented in C++, by modifying version 5.2 of the CBMC model checking tool [CKL04]. The repair unit was implemented in Python, by modifying version 1.1 of the MARCO tool [LPMMS16]. MARCO uses Z3 [DMB08] as an SMT solver and Minicard [LM12] as a SAT solver.

Our current implementation works on C programs and uses a basic set of mutations, which is a subset of the set used in [RHJ+12]. We define two *mutation levels*: level 2 contains all possible mutations and level 1 contains only a subset of them. Thus level 1 involves easier computation but may fail more often in finding a repaired program.

Table 3.1 shows the list of mutations used in every mutation level. For example, for the sub-category of arithmetic operator replacement, in mutation level 1, the table specifies two sets: {+,-} and {*,/,%}. This means that a + can be replaced with a - , and vice versa, and that the operators *,/,% can be replaced with each other. Constant manipulation mutations apply to a numeric constant and include increasing its value by

		Level 1	Level 2
Op. replacement	Arithmetic	{+,-},{*,/,%}	{+,-,*,/,%}
	Relational	{>,>=},{<,<=}	{>,>=,<,<=}, {==,!=}
	Logical	{||,&&}	
	Bit-wise	{>>, <<},{&,|,^}	
Constant manipulation			C→C+1,C→C-1, C→-C,C→0

Table 3.1: Partition of mutations to levels

1 (C→C+1), decreasing it by 1 (C→C-1), setting it to 0 (C→0) and changing its sign (C→-C).

We have evaluated our algorithm on the TCAS benchmarks from the Siemens suite [DER05]. The TCAS program implements a traffic collision avoidance system for aircrafts. It has about 180 lines of code and it comes in 41 faulty versions, together with a reference implementation (a test suite is also included but we do not use it).

We compared our results to those obtained by Könighofer and Bloem [KB11, KB13]. The results are summarized in table 3.2. Each row refers to a different faulty version of TCAS (we only include versions for which at least one method was able to produce a repair). The specification used (in both our work and their's) is an assertion requiring equivalence with the correct version[6]. For each method there are two columns: "Fixed?", which contains a + if the method was able to find a repair for that version, and "Time", which specifies the time (in seconds) it took to find a repair (if found). The bottom line specifies for each method the number of repaired versions along with their percentage from the total 41 faulty TCAS versions, and the average time it took to find a repair.

From table 3.2 it is clear that there is a trade-off between repairability and runtime when deciding which mutations to use. When using mutation level 1, our method repairs less faulty versions than [KB11, KB13] (11 vs. 15,16), but is significantly faster (2.3s vs. 38s on average). When using mutation level 2, the number of faulty versions we fix increases to 18, which is better than [KB11, KB13], but the average time to repair increases to 48s.

For all versions that we can not repair (including those that do not appear in the table), we are able to say that they can not be fixed using the given set of mutations. Using mutation level 1 it takes approximately 2 seconds on average to reach the conclusion that the program can not be fixed using mutation sets of size 1, and approximately 7 seconds to reach that conclusion for sets of size 2 (we did not collect information about larger sizes though it is possible). Using mutation level 2 these times increase significantly to 1.5 and 24 minutes, respectively.

Note that the runtime of mutation level 1 for version number 10 is exceptionally large. This is because this version requires applying two mutations in two different locations in order to be repaired. Since we inspect programs with increasing size of

[6]This is implemented by inlining the code of the correct version, saving the results of both versions to variables res1 and res2, and asserting that res1=res2. The code of the correct version is marked so that it will not be mutated (constraints encoding it are hard constraints).

	Method of [KB11]		Method of [KB13]		Our method			
					Mutation level 1		Mutation level 2	
Ver.	Fixed?	Time[s]	Fixed?	Time[s]	Fixed?	Time[s]	Fixed?	Time[s]
1	+	65			+	1.392	+	8.879
2	+	26	+	12				
3					+	1.725	+	68.651
6	+	55	+	79	+	2.056	+	33.762
7	+	11	+	6				
8	+	17	+	38				
9	+	41	+	28	+	1.203	+	17.286
10					+	6.429	+	90.666
12					+	2.157	+	77.852
16	+	9	+	6			+	84.711
17	+	12	+	6			+	55.538
18	+	14	+	40				
19	+	18	+	37				
20	+	85	+	26	+	1.709	+	15.883
25	+	82	+	100	+	2.68	+	16.234
28	+	34	+	35			+	93.678
31					+	1.246	+	4.661
32					+	1.902	+	85.349
35	+	41	+	46			+	92.866
36	+	8	+	6			+	94.599
39	+	82	+	101	+	2.558	+	16.393
40							+	4.829
41							+	4.875
	16 (39%)	38	15 (36.6%)	38	11 (26.83%)	2.278	18 (43.9%)	48.151

Table 3.2: Performance results on TCAS versions

mutation sets, we have to first apply all mutation sets of size 1 before inspecting any mutation sets of size 2. Though our method takes longer to produce this multi-line repair, it succeeds while [KB11, KB13] fail.

Since the TCAS program does not contain any loops or recursive calls, all returned programs are guaranteed to be (fully) correct, and the unwinding bound is insignificant. Therefore, we also evaluated our algorithm on a set of programs with loops. This set contains implementations of commonly known algorithms (e.g., bubble-sort and max-sort) in which we inserted bugs to create different versions (a total of 10 faulty versions). All bugs can be fixed using mutation level 1, but some require multi-line repair (up to 3 mutations at a time). In all the above experiments a correct repair was found for a bound as small as 3. Furthermore, for a bound of 3, all returned programs were found to be correct (and not only bounded correct) by a manual inspection. These results suggest that though our algorithm only guarantees bounded correctness, in many cases the returned programs are correct, even when using a small bound and even in the presence of several bugs.

3.7 Conclusion and Future Work

This work presents a novel approach to program repair. Given an erroneous program, a set of assertions and a predefined set of mutations, our algorithm returns *all* minimal repairs to the program, in increasing number of changes.

Since the number of optional repairs might be huge, it is necessary to prune the

search space whenever possible. Our technique does it by blocking all repairs that are not minimal: Whenever a successful repair is found, all repairs that use a superset of its mutations are blocked. Thus, a significant pruning of the search space is obtained.

Another promising direction is to block sets of mutations that are guaranteed *not* to succeed in repairing, based on previously seen unsuccessful once.

Chapter 4

Incremental Verification Using Trace Abstraction

Abstract

Despite the increasing effectiveness of model checking tools, automatically re-verifying a program whenever a new revision of it is created is often not feasible using existing tools. Incremental verification aims at facilitating this re-verification, by reusing partial results. In this paper, we propose a novel approach for incremental verification that is based on trace abstraction. Trace abstraction is an automata-based verification technique in which the program is proved correct using a sequence of automata. We present two algorithms that reuse this sequence across different revisions, one eagerly and one lazily. We demonstrate their effectiveness in an extensive experimental evaluation on a previously established benchmark set for incremental verification based on different revisions of device drivers from the Linux kernel. Our algorithm is able to achieve significant speedups on this set, compared to stand-alone verification.

4.1 Introduction

Manual detection of bugs in software is extremely time consuming and requires expertise and close acquaintance with the code. Yet, for some applications, delivering a bug-free product is crucial. Using automated program verification tools is a useful means to ease the burden. Despite the increasing effectiveness of such tools, advancements in technology of the past decade have given rise to new challenges. Modern software consists of thousands of lines of code and is developed by dozens of developers at a time. As a result, the software update rate is extremely high and dozens or even hundreds of successive program versions (also called revisions) are created every day. Automatically re-verifying the entire program whenever a new revision is created is often not feasible using existing tools.

Incremental verification is a methodology designed to make re-verification realistic. When a program revision undergoes incremental verification, changes made from the

previous revision are taken into account in an attempt to limit the analysis to only the parts of the program that need to be reanalyzed. Partial verification results obtained from previous revisions can help accomplish this task and can also be used to make analysis more effective.

The development of incremental verification techniques is a long-standing research topic, e.g., [Mes92, BMD00, LBBO01, CPMB07, SFS12, JCK13, HMW16]. The main challenge these techniques face is deciding which information to pass on from the verification of one revision to another, and to find effective ways to reuse this information. The proposed solutions vary, based on the underlying non-incremental verification technique used. For example, the technique of [HMW16] is based on assume-guarantee reasoning, and thus suggests reusing contextual assumptions, whereas the technique of [SFS12] is based on bounded model checking using function summaries, and thus suggests reusing these summaries.

In this paper, we propose a new technique for incremental verification, which is based on the verification method of [HHP09, HHP13]. At the basis of this verification method is the idea of looking at the basic statements of the program, i.e., its assignments and conditions, as letters of a finite alphabet. Following this point of view, the paths of the program can be seen as words over this alphabet; the program itself can be seen as a finite automaton whose states are the program locations, and whose language is a set of paths. The way the method works is by constructing an abstraction of the set of feasible program paths, called a *trace abstraction*, which is a sequence of automata over the alphabet of statements. Our suggestion is to use this trace abstraction for incremental verification. We believe that some of its properties, which we will present in later sections, make it an ideal candidate for reuse.

The paper is organized as follows: In Section 4.2 we will provide notations and formal definitions. Then, in Section 4.3, we will briefly review the work of [HHP09, HHP13] on which our incremental approach is based. Next, in Section 4.4, we will present our approach, and in Section 4.5 we will discuss our implementation details, and present extensive experimental results. Finally, in Section 4.6 we will survey related work, and in Section 4.7 we will conclude.

4.2 Preliminaries

In this section we will present the formal setting of our work. Basic concepts from the world of verification, such as a program and program correctness, will be defined in terms of formal languages and automata.

Traces. Throughout the paper, we assume the existence of a fixed set of statements, ST. The reader should think of this set as the set of all possible statements one can compose in a given programming language. An *alphabet* is a finite non-empty subset of ST. A *trace* over the alphabet Σ, denoted π, is an arbitrary word over Σ (i.e., $\pi \in \Sigma^*$).

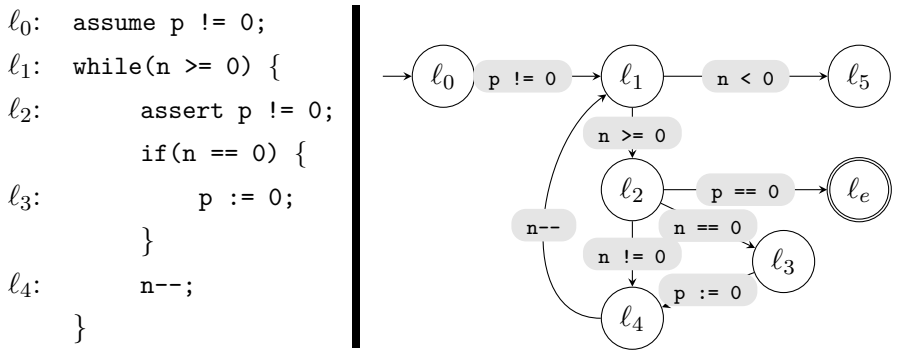


Figure 4.1: Pseudo-code of a program P_{ex1} and its control-flow automaton $\mathcal{A}_{\mathcal{P}_{\text{ex1}}}$

Programs. It is common to represent a program using its control flow graph (CFG). The set of vertices of the CFG is the set of program locations L, which contains a distinguished initial location, l_i, and a subset of distinguished error locations, L_e. Edges of the CFG are labeled with statements of the program. An edge (l_j, s, l_k) appears in the graph iff when the control of the program reaches location l_j, it is possible to continue to location l_k upon execution of the statement s. A trace is an *error trace* of the program if it labels a path from l_i to some error location $l_e \in L_e$ in this graph.

In our setting, we prefer to view the program as an automaton over the alphabet of statements, instead of a graph. Formally, we define a *program* $\mathcal{P}$ as an automaton $(Q, \Sigma, q_0, \delta, F)$, called a *control-flow automaton*, where:

1. Q, the (finite) set of automaton states, is the set of all program locations L.
2. Σ, the alphabet of the automaton, is the set of all statements that appear in the program. Note that this set is indeed an alphabet according to our previous definition (i.e., $\Sigma \subseteq ST$).
3. q_0, the initial state of the automaton, is the initial location l_i.
4. δ, the transition relation, is a subset of $L \times \Sigma \times L$ containing exactly those triples that are edges of the CFG.
5. F, the set of final states, is the set of error locations, L_e.

By construction, the language of this automaton, $\mathcal{L}(\mathcal{P})$, is the set of error traces of the program.

Example 4.2.1. Figure 4.1 presents the pseudo-code of a program P_{ex1}, along with its control-flow automaton, $\mathcal{A}_{\mathcal{P}_{\text{ex1}}}$. The correctness of this program is specified via the assert statement at location ℓ_2: every time this location is reached, the value of the variable `p` must not equal 0. Thus, modeling of the assert statement is done using an edge labeled with the negation of the assertion (here, `p == 0`) to a fresh error location, ℓ_e. The initial state of the automaton is the entering point of the program, ℓ_0, and the only accepting state is ℓ_e.

Correctness. We assume a fixed set of predicates Φ, which comes with a binary entailment relation. If the pair (φ_1, φ_2) belongs to the entailment relation, we say that φ_1 *entails* φ_2 and we write $\varphi_1 \models \varphi_2$. We also assume a fixed set HT of triples of the form $(\varphi_1, s, \varphi_2)$, where $\varphi_1, \varphi_2 \in \Phi$ and $s \in ST$. A triple $(\varphi_1, s, \varphi_2)$ is said to be a *valid Hoare triple* if it belongs to HT. In this case, we write $\{\varphi_1\}s\{\varphi_2\}$. The set of valid Hoare triples with $s \in \Sigma$ is denoted HT_Σ. Given a set $S \subseteq HT$, we denote by Φ_S the set of predicates that appear in S (i.e., all predicates that are the first or the last element of some triple in S).

Next, we extend the notion of validity from statements to traces. Given a trace $\pi = s_1 \cdots s_n$, where $n \geq 1$, the triple $(\varphi_1, \pi, \varphi_{n+1})$ is *valid* (and we write $\{\varphi_1\}\pi\{\varphi_{n+1}\}$), iff there exists a sequence of predicates $\varphi_2 \cdots \varphi_n$ s.t. $\{\varphi_i\}s_i\{\varphi_{i+1}\}$ for all $1 \leq i \leq n$. For an empty trace π (a trace of length 0), the triple (φ, π, φ') is *valid* iff φ entails φ'.

In order to define correctness, we also assume the existence of a pair of specific predicates from Φ, $true$ and $false$. A trace π is *infeasible* if $\{true\}\pi\{false\}$. The set of all infeasible traces over the alphabet Σ is denoted INFEASIBLE_Σ. Finally, a program $\mathcal{P}$ is said to be *correct* if all error traces of it are infeasible. That is, if $\mathcal{L}(\mathcal{P}) \subseteq \text{INFEASIBLE}_\Sigma$, where Σ is the alphabet of the program.

4.3 Verification Using Trace Abstraction

In this section we will review the work of [HHP09] and [HHP13], which presents an automata-based approach for verification, upon which our incremental verification scheme is based. Even though some of the notions had to be adapted to our setting, all relevant theorems remain valid.

4.3.1 Floyd-Hoare Automata

We begin by introducing the notion of a Floyd-Hoare automaton, presented in [HHP13], and describing some of its key properties. Intuitively, a Floyd-Hoare automaton is an automaton over an alphabet Σ whose states can be mapped to predicates from Φ and whose transitions can be mapped to valid Hoare triples. The motivation behind this definition is that we want Floyd-Hoare automata to accept only infeasible traces, by construction. Formally, we use the following definition:

Definition 4.3.1 (Floyd-Hoare automaton)**.** A *Floyd-Hoare automaton* is a tuple

$$\mathcal{A} = (Q, \Sigma, q_0, \delta, F, \theta)$$

where Q is a finite set of states, Σ is an alphabet, $q_0 \in Q$ is the initial state, $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, $F \subseteq Q$ is the set of final states, and $\theta : Q \to \Phi$ is a mapping from states to predicates s.t. the following conditions hold:

1. $\theta(q_0) = true$.

2. For every $q \in F$, $\theta(q) = false$.

3. For every triple $(q_i, s, q_j) \in \delta$, $\{\theta(q_i)\}s\{\theta(q_j)\}$.

The function θ is called the *annotation* of $\mathcal{A}$. The image of θ (i.e., the set of all predicates $\varphi \in \Phi$ s.t. there exists a $q \in Q$ for which $\theta(q) = \varphi$) is called the *predicate set* of $\mathcal{A}$ and is denoted $\Phi_{\mathcal{A}}$.

Theorem 4.1 ([HHP13, page 12])**.** *Every trace accepted by a Floyd-Hoare automaton $\mathcal{A}$ is infeasible. That is, for every Floyd-Hoare automaton $\mathcal{A}$ over Σ,*

$$L(\mathcal{A}) \subseteq \text{INFEASIBLE}_{\Sigma}$$

.

In what follows, we define a mapping from Floyd-Hoare automata to sets of valid Hoare triples, and vice versa, using a pair of functions, α and β. The function α is a function from sets of valid Hoare triples to Floyd-Hoare automata. A set S of valid Hoare triples over Σ is mapped by α to the Floyd-Hoare automaton $\mathcal{A}_S = (Q_S, \Sigma, q_{0S}, \delta_S, F_S, \theta_S)$ where:

- $Q_S = \{q_\varphi | \varphi \in \Phi_S\} \cup \{q_{true}, q_{false}\}$.
- $q_{0S} = q_{true}$
- $\delta_S = \{(q_{\varphi_1}, s, q_{\varphi_2}) | (\varphi_1, s, \varphi_2) \in S\}$
- $F_S = \{q_{false}\}$
- $\forall q_\varphi \in Q_S \quad \theta_S(q_\varphi) = \varphi$

Note that this is indeed a Floyd-Hoare automaton according to definition 4.3.1, since S contains only valid Hoare triples.

The function β is a function from Floyd-Hoare automata to sets of valid Hoare triples. Given a Floyd-Hoare automaton $\mathcal{A} = (Q, \Sigma, q_0, \delta, F, \theta)$, β maps $\mathcal{A}$ to the set $\{(\theta(q_i), s, \theta(q_j)) \,|\, (q_i, s, q_j) \in \delta\}$. By definition 4.3.1 (specifically, by requirement number 3 of θ), this set contains only valid Hoare triples.

4.3.2 Automata-Based Verification

Next, we describe how Floyd-Hoare automata can be used to verify programs via trace abstraction [HHP09]. Formally, a *trace abstraction* is a tuple of Floyd-Hoare automata $(\mathcal{A}_1, \ldots, \mathcal{A}_n)$ over the same alphabet Σ. The alphabet Σ is referred to as *the alphabet of the trace abstraction*. We say that a program $\mathcal{P}$ *is covered by* $(\mathcal{A}_1, \ldots, \mathcal{A}_n)$ if $\mathcal{P}$ and $(\mathcal{A}_1, \ldots, \mathcal{A}_n)$ are over the same alphabet and $\mathcal{L}(\mathcal{P}) \subseteq \mathcal{L}(A_1) \cup \ldots \cup \mathcal{L}(A_n)$.

Theorem 4.2 ([HHP09, page 7])**.** *Given a program $\mathcal{P}$, if there exists a trace abstraction $(\mathcal{A}_1, \ldots, \mathcal{A}_n)$ s.t. $\mathcal{P}$ is covered by $(\mathcal{A}_1, \ldots, \mathcal{A}_n)$, then $\mathcal{P}$ is correct.*

Theorem 4.2 implies a way to verify a program $\mathcal{P}$, namely, by constructing a trace abstraction $(\mathcal{A}_1, \ldots, \mathcal{A}_n)$ s.t. $\mathcal{P}$ is covered by $(\mathcal{A}_1, \ldots, \mathcal{A}_n)$. This is realized in [HHP09] in an algorithm that is based on the counter-example guided abstraction refinement (CEGAR) paradigm (Fig. 4.2). Initially, the trace abstraction is an empty sequence of automata, and then it is iteratively refined by adding automata, until the program is covered by the trace abstraction.

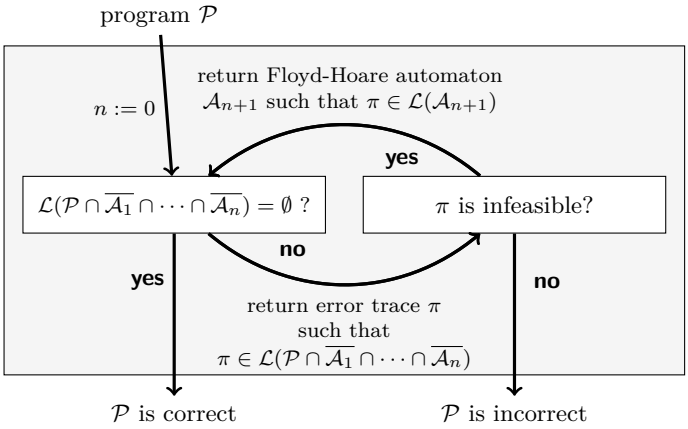
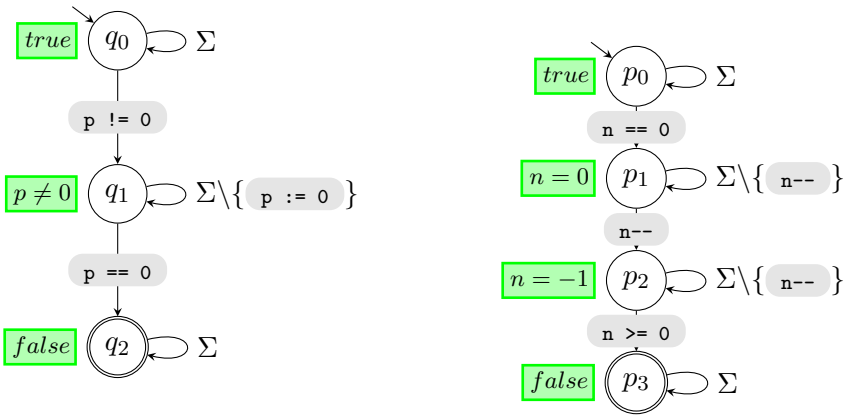


Figure 4.2: [HHP09] CEGAR-based scheme for non-incremental verification using trace abstraction

Each iteration consists of two phases: validation and refinement. During the validation phase, we check whether the equation $\mathcal{L}(\mathcal{P} \cap \overline{\mathcal{A}_1} \cap \cdots \cap \overline{\mathcal{A}_n}) = \emptyset$ holds. The overline notation stands for computing automata complementation and the $\cap$ notation stands for computing automata intersection. Note that complementation, intersection and emptiness checking, can all be done efficiently for finite automata. Checking whether this equation holds is semantically equivalent to checking whether $\mathcal{L}(\mathcal{P}) \subseteq \mathcal{L}(A_1) \cup \ldots \cup \mathcal{L}(A_n)$, so if the answer is "yes", we can state that the program is correct (Theorem 4.2). If the answer is "no", then we get a witness in the form of a trace π s.t. $\pi \in \mathcal{L}(\mathcal{P} \cap \overline{\mathcal{A}_1} \cap \cdots \cap \overline{\mathcal{A}_n})$, which is passed on to the refinement phase.

During the refinement phase, π is semantically analyzed to decide whether it is infeasible or not. If it is not, we can state that the program is incorrect, since π is an error trace of $\mathcal{P}$. If it is, then the proof of its infeasibility can be used to construct a Floyd-Hoare automaton $\mathcal{A}_{n+1}$ that accepts π (in particular, the way this is done in [HHP13], is by obtaining a set of valid Hoare triples from the proof and applying α on it). This automaton is then added to the produced trace abstraction, and the process is repeated.

Example 4.3.2. Recall program P_{ex1} from Figure 4.1. We claim that an assertion violation is not possible in this program. A convincing argument for this claim can be made by considering separately those executions that visit ℓ_3 at least once and those who do not. For the later, `p` is never assigned during the execution, and the assume statement makes sure that initially `p` does not equal 0, so every time the assertion



(a) $\mathcal{A}_1$ (b) $\mathcal{A}_2$

Figure 4.3: Floyd-Hoare automata $\mathcal{A}_1$, $\mathcal{A}_2$, with accepting states q_2, p_3, resp. The gray frames labeling transitions represent letters from Σ, where an edge labeled with $G \subseteq \Sigma$ means a transition reading any letter from G. The green frames labeling states represent predicates assigned to states by the annotation θ.

is reached the condition `p != 0` must hold. For the former, since ℓ_3 is reached, the true branch of the `if` statement was taken during that iteration, so `n` equals 0 at ℓ_4. Therefore, after the execution of `n--`, `n` will equal -1, and thus the loop will be exited and the assertion will not be reached.

Program P_{ex1} is successfully verified using the scheme of Figure 4.2. The trace abstraction obtained is the tuple $(\mathcal{A}_1, \mathcal{A}_2)$, presented in Figure 4.3. Observe that the language of $\mathcal{A}_1$ consists of all traces that contain the statement `p != 0` followed by the statement `p == 0`, without an assignment to `p` in between. The language of $\mathcal{A}_2$ consists of all traces that contain the statement `n == 0` followed by the statement `n--` and the statement `n >= 0`, without an assignment to `n` between any of these three statements. As we have just explained, all error traces of P_{ex1} fall into one of these categories (which one depends on whether or not ℓ_3 is visited), so the inclusion $\mathcal{L}(\mathcal{A}_{\mathcal{P}_{\text{ex1}}}) \subseteq \mathcal{L}(\mathcal{A}_1) \cup \mathcal{L}(\mathcal{A}_2)$ indeed holds.

4.4 Incremental Verification Using Trace Abstraction

In the previous section we saw a CEGAR-based algorithm for verification using trace abstraction. In this section, we show how incremental verification can be done by reusing a trace abstraction. For the incremental setting, in addition to the program $\mathcal{P}$, the algorithm also gets as input a trace abstraction TA^I, and is expected to return the verification result coupled with a trace abstraction TA^O. The alphabet of TA^I may be different than that of the program $\mathcal{P}$. From here on, let us denote the alphabet of the program by Σ^O and the alphabet of the trace abstraction by Σ^I. While there is no restriction on Σ^I, the performance of the algorithm is expected to improve the more similar it is to Σ^O (i.e., the larger the set $\Sigma^I \cap \Sigma^O$ is).

Procedure TranslateAutomaton
Input: A_{Σ^I} over Σ^I, $S_{\Sigma^O} \subseteq HT_{\Sigma^O}$. **Output:** A_{Σ^O} over Σ^O
1. Construct the set $S_1 = \beta(A_{\Sigma^I})$.
2. Construct the set $S_2 = (S_1 \setminus HT_{\Sigma^I \setminus \Sigma^O}) \cup S_{\Sigma^O}$.
3. Return $A_{\Sigma^O} = \alpha(S_2)$.

Figure 4.4: Translation procedure

4.4.1 Translation of Floyd-Hoare Automata

The rationale for reusing a trace abstraction is that each Floyd-Hoare automaton in it forms a proof that the set of traces it accepts is infeasible (Theorem 4.1), and therefore we do not need to analyze any trace in this set. The organization of the information in the form of an automaton, gives us a convenient way to get rid of all error traces of $\mathcal{P}$ that belong to this set: simply by subtracting the automaton from the program (which is also an automaton). The problem is that the above subtraction can not be done straight away, since the program and the trace abstraction are not necessarily over the same alphabet.

Notice that the only alphabet that is of interest to us for the current verification problem is Σ^O (traces that contain statements that are not from Σ^O are definitely not error traces of our program). Therefore, we would like to translate the given trace abstraction from Σ^I to Σ^O. But, first, we must define what it means to "translate". In the context of Floyd-Hoare automata it is easier to define translation first in terms of valid Hoare triples.

Definition 4.4.1 (Translation of a set of valid Hoare triples)**.** Given a set of valid Hoare triples $S_{\Sigma^I} \subseteq HT_{\Sigma^I}$, a set of valid Hoare triples $S_{\Sigma^O} \subseteq HT_{\Sigma^O}$ is a translation of S_{Σ^I} to Σ^O, if all valid Hoare triples over $\Sigma^I \cap \Sigma^O$ in S_{Σ^I} are also in S_{Σ^O}. In other words, if

$$S_{\Sigma^I} \cap HT_{\Sigma^I \cap \Sigma^O} \subseteq S_{\Sigma^O}$$

.

It is easy to extend the definition of translation from sets of valid Hoare triples to Floyd-Hoare automata, using the connection between the two, defined in the previous section (specifically, the function β).

Definition 4.4.2 (Translation of a Floyd-Hoare automaton)**.** Given a Floyd-Hoare automaton A_{Σ^I} over the alphabet Σ^I, a Floyd-Hoare automaton A_{Σ^O} over the alphabet Σ^O is a translation of A_{Σ^I} to Σ^O, if $\beta(A_{\Sigma^O})$ is a translation of $\beta(A_{\Sigma^I})$ to Σ^O.

Given a Floyd-Hoare automaton A_{Σ^I} over the alphabet Σ^I and a set S_{Σ^O} of valid Hoare triples over the alphabet Σ^O, one can translate A_{Σ^I} to A_{Σ^O} over the alphabet Σ^O, using the procedure in Fig. 4.4.

Proposition 4.4.3. *Every Floyd-Hoare automaton A_{Σ^O} that is constructed using the procedure TranslateAutomaton, is a translation of A_{Σ^I} to Σ^O.*

Proof. Since all valid Hoare triples removed from S_1 when creating S_2 were over $\Sigma^I \setminus \Sigma^O$, then $S_1 \cap HT_{\Sigma^I \cap \Sigma^O} \subseteq S_2$. Therefore, by definition 4.4.1, S_2 is a translation of S_1 to Σ^O. Now, $S_1 = \beta(A_{\Sigma^I})$, so we conclude that S_2 is a translation of $\beta(A_{\Sigma^I})$ to Σ^O. Next, we want to claim that $\beta(A_{\Sigma^O}) = S_2$. This is correct because, according to the definitions of β and α, for every set S, $\beta(\alpha(S)) = S$, so in particular $\beta(\alpha(S_2)) = S_2$. Thus, we conclude that $\beta(A_{\Sigma^O})$ is a translation of $\beta(A_{\Sigma^I})$ to Σ^O. By definition 4.4.2, this means that A_{Σ^O} is a translation of A_{Σ^I} to Σ^O. ■

The TranslateAutomaton procedure enables us to translate the trace abstraction into the alphabet of the program, but the question remains which set should we use as S_{Σ^O}. This set can be any subset of HT_{Σ^O} between $\emptyset$ and HT_{Σ^O} itself. Notice that, since $A_{\Sigma^O} = \alpha(S_2)$ and $S_{\Sigma^O} \subseteq S_2$, the number of transitions in A_{Σ^O} is at least the size of S_{Σ^O}. Therefore, the choice of S_{Σ^O} greatly influences the efficiency of the algorithm. The larger S_{Σ^O} is, the larger translated automata will be, thus the more effort will be required during translation and during operations involving these automata in future steps (e.g., complementation and intersection).

On the other hand, translation is our mechanism for adapting the proof of correctness of the old program, which is the input trace abstraction, to the new program. So, certain addition of transitions not only is not a bad thing, but it is crucial for reuse to be effective, as we somehow need to account for the changes made to the program. Thus, in choosing S_{Σ^O} there is a trade-off between how much effort we are willing to spend on building the translated automata and using them, and how useful they will be for proving the new program correct.

In the current implementation we considered three options for S_{Σ^O}:

- $S_{\Sigma^O}^{empty} = \emptyset$
- $S_{\Sigma^O}^{unseen} = \{\{\varphi_1\}s\{\varphi_2\} \mid s \in \Sigma^O \setminus \Sigma^I,\ \varphi_1, \varphi_2 \in \Phi_{A_{\Sigma^I}}\}$
- $S_{\Sigma^O}^{all} = \{\{\varphi_1\}s\{\varphi_2\} \mid s \in \Sigma^O,\ \varphi_1, \varphi_2 \in \Phi_{A_{\Sigma^I}}\}$

Note that all three sets are indeed subsets of HT_{Σ^O}, and all of them only use predicates from $\Phi_{A_{\Sigma^I}}$. As a result, using TranslateAutomaton with either of these sets as S_{Σ^O} yields an automaton A_{Σ^O} whose states were also in A_{Σ^I} (states are only removed and not added). Also, in all three cases, transitions with irrelevant letters (i.e., in $\Sigma^I \setminus \Sigma^O$) are removed from A_{Σ^I}, while transitions with relevant letters (i.e., in Σ^O) remain intact.

The difference between the three lies in which transitions are added to A_{Σ^I} in each case. In the case of $S_{\Sigma^O}^{empty}$, no transitions are added at all. In this case, translated automata are only useful to prove infeasibility of error traces that remained unchanged from the previous version, but translation requires no effort. On the other end of the spectrum there is $S_{\Sigma^O}^{all}$, in which all valid Hoare triples over Σ^O are added as transitions to A_{Σ^O}. Here, any error trace that can be proved infeasible using predicates from $\Phi_{A_{\Sigma^I}}$

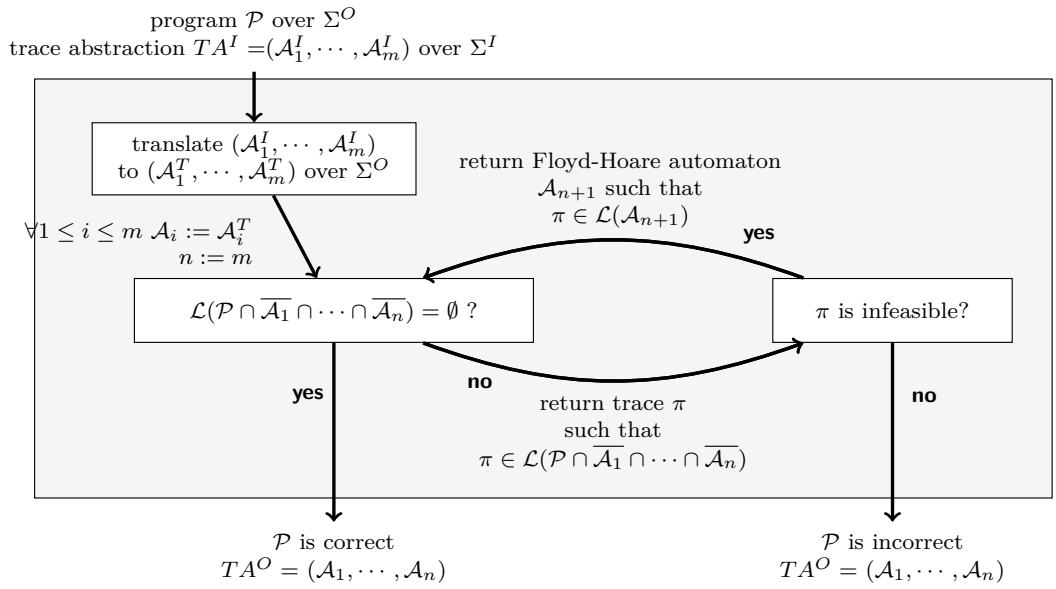


Figure 4.5: **Eager reuse:** Scheme for incremental verification using an eager approach

will be accepted by A_{Σ^O}. On the other hand, translation is expensive and resulting automata are often rather large.

$S^{unseen}_{\Sigma^O}$ suggests an intermediate solution, by considering only valid Hoare triples over $\Sigma^O \setminus \Sigma^I$. The rationale is that all valid Hoare triples over $\Sigma^O \cap \Sigma^I$ that are relevant to prove infeasibility of error traces would have already been considered when A_{Σ^I} was constructed. In practice, in cases where a trace was changed by reordering statements or by adding preexisting statements somewhere in the middle of it, A_{Σ^O} might not have the necessary transitions to prove that trace infeasible.

We have performed our experiments with all three of these options. The set that gave the best overall results in practice, on average, was $S^{all}_{\Sigma^O}$. Thus, the experimental results presented in section 4.5.1 are based on using $S^{all}_{\Sigma^O}$ as S_{Σ^O}. The fact that $S^{all}_{\Sigma^O}$ outperforms $S^{unseen}_{\Sigma^O}$ suggests, perhaps, that changes such as reordering code and adding preexisting code (i.e., copy-pasting), on which $S^{unseen}_{\Sigma^O}$ has bad results, are frequent in software evolution.

4.4.2 Reuse Algorithms

We now present two schemes for incremental verification, that differ in the strategy they use for subtraction of Floyd-Hoare automata from the program. In both schemes, any subtraction $\mathcal{P} - \mathcal{A}$ is replaced with $\mathcal{P} \cap \overline{\mathcal{A}}$, which results in the same language but uses different automata operations that more faithfully represent our implementation.

Eager Reuse. The first scheme, presented in Figure 4.5, suggests an eager approach towards reuse of Floyd-Hoare automata. Here, subtraction of Floyd-Hoare automata is done straight away, and entirely (all Floyd-Hoare automata in the trace abstraction are

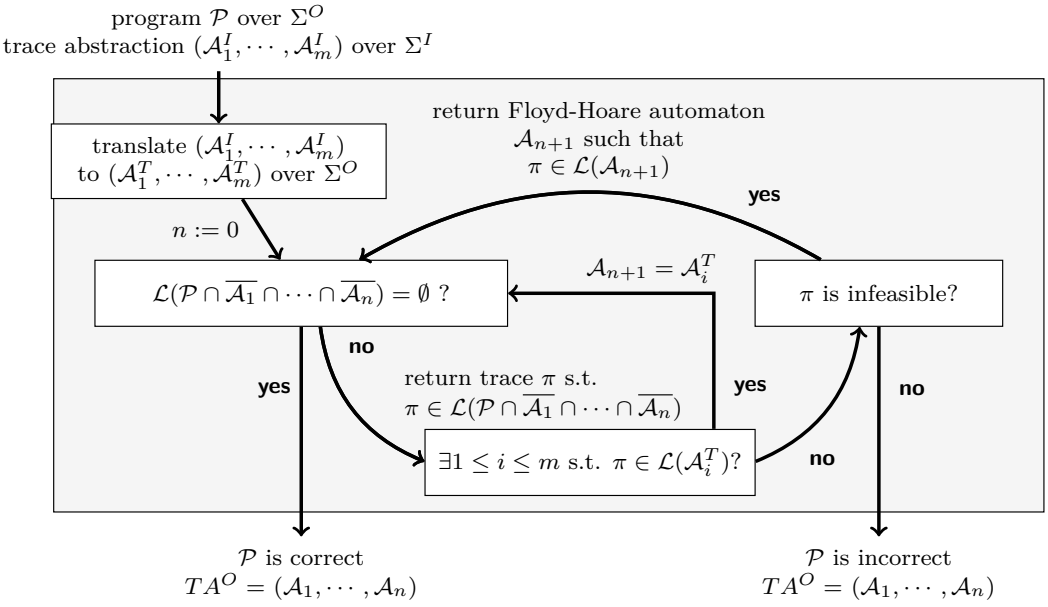


Figure 4.6: **Lazy reuse:** Scheme for incremental verification using a lazy approach

subtracted). Then, the CEGAR-based algorithm continues as in the non-incremental case. The output trace abstraction, TA^O, contains all automata translated from TA^I along with all other automata obtained during the CEGAR loop.

An advantage of this scheme is that all those traces that can be proved infeasible using the Floyd-Hoare automata are excluded to begin with, and are not analyzed during the CEGAR loop. On the other hand, we may have done some subtractions (or, in fact, intersections) that did not change the language at all and hence were not useful. For example, it is possible that for some automaton $\mathcal{A}_i^T$ translated from TA^I, $\mathcal{L}(\mathcal{P} \cap \overline{\mathcal{A}_1^T} \cap \cdots \cap \overline{\mathcal{A}_i^T}) = \mathcal{L}(\mathcal{P} \cap \overline{\mathcal{A}_1^T} \cap \cdots \cap \overline{\mathcal{A}_{i-1}^T})$ and so the computation of the intersection with $\overline{\mathcal{A}_i^T}$ was done in vain. Note that all Floyd-Hoare automata are added to TA^O, regardless of whether they were useful or not (since retrieving this information is prohibitively expensive).

Lazy Reuse. The second scheme, presented in Figure 4.6, suggests a lazy approach towards reuse of Floyd-Hoare automata. A Floyd-Hoare automaton is only subtracted once we know that it is useful, i.e., that its subtraction will remove at least one trace from the set of traces we have not yet proven infeasible.

In this scheme, the initial trace abstraction is the empty sequence, as in the non-incremental case. Then the CEGAR loop begins, but with an additional phase, which we call the *reuse phase*, inserted between the validation and refinement phases (which themselves are not changed). If the validation phase finds a trace π in $\mathcal{L}(\mathcal{P} \cap \overline{\mathcal{A}_1} \cap \cdots \cap \overline{\mathcal{A}_n})$, then the reuse phase first checks whether this trace is accepted by some automaton $\mathcal{A}_i^T$ translated from TA^I. If it is, then $\mathcal{A}_i^T$ is added to the trace abstraction and we return to the validation phase again. If it is not, then we pass π to the refinement phase

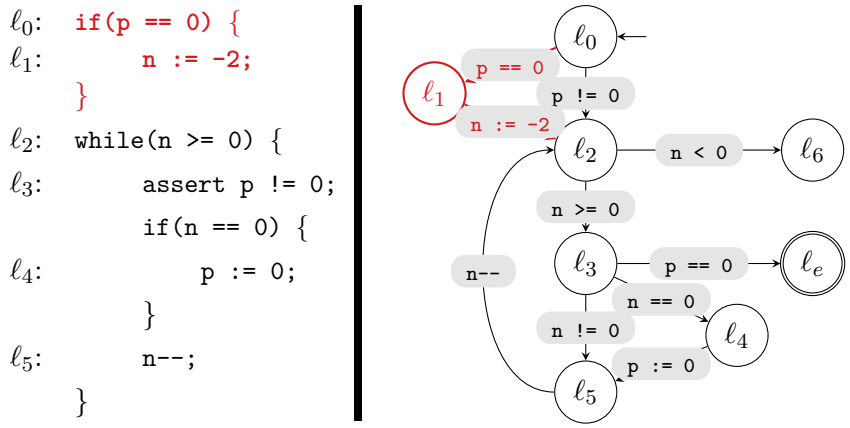


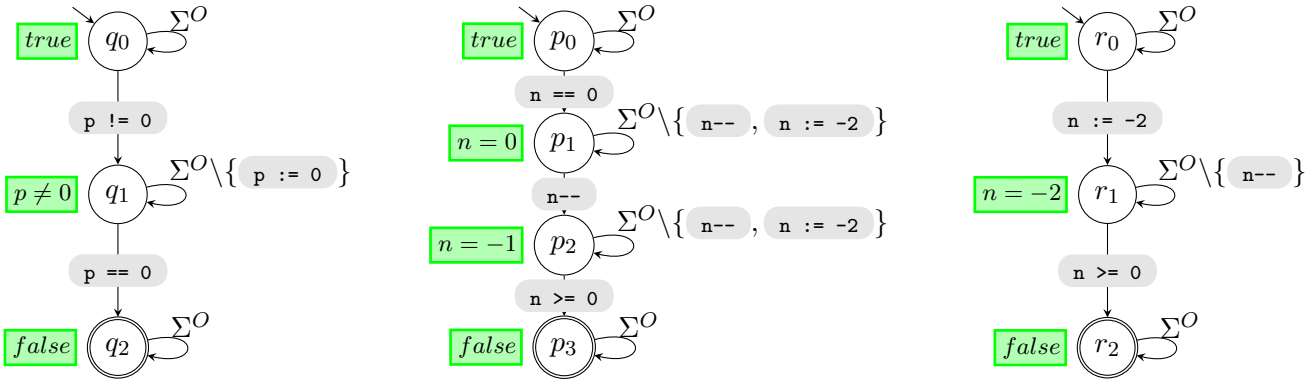
Figure 4.7: Program P_{ex2}, which is a modified version of program P_{ex1}. Changes from P_{ex1} appear in red.

and proceed as before. The output trace abstraction in this case includes only those automata translated from TA^I that were added to it during the reuse phase, in addition to all those created during the refinement phase.

Example 4.4.4. Figure 4.7 presents the source code and the control-flow automaton of a program P_{ex2}. This program is an updated version of P_{ex1} (Figure 4.1), where instead of assuming that `p` is initially different than 0, if `p` equals 0, `n` is set to -2. The alphabet Σ^O of the control-flow automaton $\mathcal{A}_{\mathcal{P}_{\text{ex2}}}$ is the set of P_{ex2}'s statements (i.e., $\Sigma^O = \Sigma^I \cup \{$ `n := -2` $\}$, where Σ^I is the alphabet of $\mathcal{A}_{\mathcal{P}_{\text{ex1}}}$).

You will notice that despite the changes made, the assertion still can not be violated. For executions who visit ℓ_4 (formerly ℓ_3) at least once, we can make the same argument as we did in example 4.3.2. For executions who do not visit ℓ_4, the argument we used in example 4.3.2 relied on `p` being initially different than 0, so now it only applies to those executions beginning in a transition from ℓ_0 to ℓ_2. For executions going from ℓ_0 to ℓ_1, we need a new argument. For them, we can say that the visit in ℓ_1 guarantees `n` will be equal to -2 upon reaching ℓ_2, and thus the loop will not be entered and the assertion will not be reached.

Figure 4.8 presents the output trace abstraction $TA^O = (\mathcal{A}_1^O, \mathcal{A}_2^O, \mathcal{A}_3^O)$ produced by our algorithm, in both the Eager and the Lazy variants, when using the tuple $(\mathcal{A}_1, \mathcal{A}_2)$ from Figure 4.3 as the input trace abstraction TA^I. The first two automata, $\mathcal{A}_1^O$ and $\mathcal{A}_2^O$, are the translations of automata $\mathcal{A}_1$ and $\mathcal{A}_2$ to Σ^O, resp. Translation, in this case, amounts to adding transitions with the new letter, `n := -2`, where appropriate. Specifically, `n := -2` was added to the 3 self-loops in $\mathcal{A}_1$, and to the self loops from p_0 and p_3 in $\mathcal{A}_2$. The third automaton, $\mathcal{A}_3^O$, is a new Floyd-Hoare automaton, obtained during the refinement phase.



(a) $\mathcal{A}_1^O$ (b) $\mathcal{A}_2^O$ (c) $\mathcal{A}_3^O$

Figure 4.8: Trace abstraction ($\mathcal{A}_1^O$,$\mathcal{A}_2^O$,$\mathcal{A}_3^O$), which is the output of our algorithm for P_{ex2}, when using the tuple $(\mathcal{A}_1, \mathcal{A}_2)$ from Figure 4.3 as TA^I.

4.5 Evaluation

We have implemented our incremental verification algorithms on top of the Ultimate Automizer software verification tool, which is part of the Ultimate program analysis framework [1]. The source code is available on Github [2]. We currently support incremental verification of C and Boogie programs with respect to safety properties (e.g., validity of assertions or memory-access safety).

On-the-fly Computation For simplicity of presentation, schemes of our algorithms in figures 4.5 and 4.6 show a stand-alone translation phase that precedes the CEGAR loop. According to these schemes, each automaton $\mathcal{A}_j^I$ in the input trace abstraction is first translated into an automaton $\mathcal{A}_j^T$ over Σ^O. In practice, computing $\mathcal{A}_j^T$ entirely can be quite expensive, depending on S_{Σ^O}, as previously discussed. Also, the computation of certain transitions may turn out to be redundant, as we may not need these transitions at any point during the CEGAR loop. Therefore, our implementation translates automata on-the-fly, adding transitions only as soon as the need for them emerges. On-the-fly translation may happen during the reuse phase in the Lazy reuse algorithm, and during the validation phase in both algorithms. Additionally, creation of Floyd-Hoare automaton $\mathcal{A}_{n+1}$ in case a trace is found infeasible during the refinement phase is already done on-the-fly in the preexisting implementation of Ultimate Automizer. That is, transitions are added to $\mathcal{A}_{n+1}$ only if they are needed during the following validation phase.

[1] `https://ultimate.informatik.uni-freiburg.de`
[2] `https://github.com/ultimate-pa`

4.5.1 Experimental Results

We have performed an extensive experimental evaluation of our approach on a set of benchmarks previously established in [BLN+13], available on-line[3]. This benchmark set is based on industrial source code from the Linux kernel, and contains 4,193 verification tasks from 1,119 revisions of 62 device drivers. A verification task is a combination of driver name, revision number, and specification, where the specification is one of six different rules for correct Linux kernel core API usage (more details can be found in [BLN+13]). We excluded those tasks where Ultimate Automizer was unable to parse the input program successfully, and were left with a total of 2,660 verification tasks.

Our experiments were made on a machine with a 4GHz CPU (Intel Core i7-6700K). We used Ultimate Automizer version 0.1.23-bb20188 with the default configuration, which was also used in SV-COMP'18 [4], [HYFD+18]. In this configuration Ultimate Automizer first uses SMTInterpol[5] with Craig interpolation for the analysis of error traces during the refinement phase, and if this fails, falls back on Z3 [6] with trace interpolation [DHM+17]. Validity of Hoare triples is also checked with Z3. A timeout of 90s was set to all verification tasks and the Java heap size was limited to 6GB.

For each verification task we verified the revision against the specification three times: first, without any reuse, and then with reuse, using both the Eager and the Lazy algorithms. The output trace abstraction of each revision was used as the input trace abstraction of the next revision. The results of these experiments are summarized in Table 4.1.

These results clearly show that our method, both when used with the Eager algorithm and with the Lazy one, manages to save the user a considerable amount of time, for the vast majority of these benchmarks. The difference in performance between the Eager and Lazy algorithms on these benchmarks was quite negligible; both obtain a nontrivial speedup of around $\times 4.7$ in analysis time, and $\times 3.6$ in overall time, on average. When comparing mean analysis speedups of our approach and that of [BLN+13], we get a speedup that is $\times 1.5$ larger. But, what is additionally interesting to note, is that we do not succeed on the same benchmarks as [BLN+13] does; the best 15 series in our work and theirs are completely disjoint. This suggests that the two methods are orthogonal.

Slowdowns are demonstrated for our worst 7 results. On the other hand, our top 7 results all demonstrate speedups of more than an order of magnitude, with an impressive max value of $\times 79.80$. For each pair of successive revisions, we have computed their edit-distance by summing up the number of added, modified and deleted lines, and dividing by the total number of lines in the file. To compute the edit-distance of a series, we have computed the mean edit-distance of all revisions in it. We expected to see a correlation between the edit-distance of a series and the speedup obtained for

[3] `https://www.sosy-lab.org/research/cpa-reuse/regression-benchmarks`
[4] `https://sv-comp.sosy-lab.org/2018/`
[5] `https://ultimate.informatik.uni-freiburg.de/smtinterpol`, version 2.1-441-gf99e49f
[6] `https://github.com/Z3Prover/z3`, version master 450f3c9b

Driver	Spec	Tasks	Default	Eager		Lazy		Speedup Overall	Speedup Analysis	[BLN+13] Speedup
			Overall	Overall	Analysis	Overall	Analysis			
dvb-usb-rtl28xxu	08_1a	10	20.509	0.352	0.187	0.416	0.257	49.30	79.80	3.6
dvb-usb-rtl28xxu	39_7a	10	110.893	4.081	1.992	4.059	2.546	27.32	43.55	6.3
dvb-usb-rtl28xxu	32_7a	10	35.551	1.306	0.725	1.550	0.844	22.93	42.12	4.9
dvb-usb-az6007	08_1a	5	4.620	0.173	0.118	0.187	0.132	24.70	35.00	3.5
dvb-usb-az6007	39_7a	5	17.952	1.378	0.862	1.425	0.989	12.59	18.15	4.9
cx231xx-dvb	08_1a	13	3.330	0.303	0.206	0.323	0.228	10.30	14.60	1.8
panasonic-laptop	08_1a	16	3.466	0.337	0.222	0.384	0.257	9.02	13.48	2.4
spcp8x5	43_1a	13	5.531	0.632	0.437	0.618	0.432	8.94	12.80	1.6
panasonic-laptop	32_1	4	0.623	0.100	0.061	0.072	0.051	8.65	12.21	3.4
panasonic-laptop	39_7a	16	18.961	2.377	1.654	2.617	1.906	7.24	9.94	3.6
leds-bd2802	68_1	4	1.039	0.180	0.112	0.191	0.123	5.43	8.44	4.4
leds-bd2802	32_1	4	0.484	0.089	0.057	0.097	0.064	4.98	7.56	3.9
wm831x-dcdc	32_1	3	0.330	0.063	0.044	0.066	0.047	5.00	7.02	2.1
cx231xx-dvb	39_7a	13	17.536	3.389	2.425	3.464	2.517	5.06	6.96	3.2
ems_usb	08_1a	21	2.334	0.502	0.327	0.543	0.362	4.29	6.44	2.9
... (for full results cf. http://batg.cswp.cs.technion.ac.il/publications/)										
ar7part	32_7a	6	0.071	0.067	0.056	0.074	0.063	0.95	1.12	1.3
metro-usb	08_1a	25	0.394	0.497	0.330	0.518	0.356	0.76	1.10	2.1
rtc-max6902	32_7a	9	0.133	0.124	0.106	0.147	0.126	0.90	1.05	1.1
i2c-algo-pca	43_1a	7	0.012	0.018	0.018	0.019	0.019	1.00	1.00	1.0
dvb-usb-vp7045	43_1a	2	0.001	0.002	0.002	0.027	0.027	1.00	1.00	2.6
cfag12864b	43_1a	2	0.036	0.039	0.036	0.040	0.037	0.90	0.97	1.0
rtc-max6902	43_1a	5	0.278	0.273	0.262	0.303	0.291	0.91	0.95	1.1
magellan	32_7a	2	0.015	0.018	0.016	0.018	0.016	0.83	0.93	0.93
vsxxxaa	43_1a	2	0.030	0.037	0.033	0.036	0.032	0.83	0.93	6.8
ar7part	43_1a	2	0.036	0.043	0.038	0.044	0.039	0.81	0.92	1.2
Sum		1,177	529.258	142.856	107.543	146.275	112.225			
Mean		13	5.881	1.587	1.195	1.625	1.247	3.618	4.716	3.17
Sum (All)		2,660	3,048.373	434.853	334.603	448.424	349.69			
Mean (All)		15	16.749	2.389	1.838	2.464	1.921	6.798	8.717	4.3

Table 4.1: The results of our evaluation. Each row contains the results for a series of revisions of a driver and one type of specification. The table only shows those series where we could parse all files, allowing for a comparison in speedup with [BLN+13]. We also limited the display to the best 15 and the worst 10 series in terms of speedup. The number of tasks specifies the number of files including the first revision. The settings "Eager" and "Lazy" are divided in overall and analysis time, where analysis time is the overall time without the time it took writing the output trace abstraction to file. As the "Default" setting does not write an output trace abstraction, its analysis time is the same as its overall time. All times are given as seconds of wall time and do not include the time for the first revision. The speedup colums compare the relative speedup between the Default setting and the Lazy setting. The rows "Sum" and "Mean" show the sum and mean of all the series where we were able to parse all the tasks, whereas the rows "Sum (All)" and "Mean (All)" show the sum and the mean of all the tasks we could parse. We adjusted the mean speedup of [BLN+13] for our subset by recomputing their speedup relative to our shared subset, but their mean speedup in the "Mean (All)" row refers to the original 4,193 tasks.

it. In general, such a correlation does seems to exist; a speedup of greater than 4 is achieved mostly for revisions where the edit distance is small. But, this correlation is not definitive. For example, we had one series where the mean edit-distance was over 90 percent, but the speedup was over $\times 60$. Also, cases with slowdowns distribute evenly over the mean edit-distance size.

4.6 Related Work

The validation of evolving software has been the subject of extensive research over the years (see the book by Chockler et al. [CKMS15]). Several different problems have been studied in this context, e.g., analyzing the semantic difference between successive revisions [TGK17] or determining which revision is responsible for a bug [MVP15, ABV16]. In this section, we will focus on the problem of formally verifying all program revisions.

A dominant approach to solve this problem is to only verify the first revision, and then prove that every pair of successive revisions is equivalent. It was suggested by Godlin and Strichman in [SG08], where they gave it the name *regression verification* and introduced an algorithm that is based on the theory of uninterpreted functions. Papers about regression verification are concerned with improving equivalence checking and increasing its applicability. In [BPRT13], a summary of program behaviors impacted by the change is computed for both programs, and then equivalence is checked on summaries alone. Similarly, in [BOR13], checking equivalence is done gradually by partitioning the common input space of programs and checking equivalence separately for each set in the partition. In [FGK$^+$14], a reduction is made from equivalence checking to Horn constraint solving. In [SVBV16] applicability is extended to pairs of recursive functions that are not in lock-step, and in [CGS12] to multi-threaded concurrent programs. The work of [BUVHW15] is focused on Programmable Logic Controllers, which are computing devices that control production in many safety-critical systems. Finally, [BV16] proposes a different notion of equivalence, which on top of the usual functional equivalence also considers runtime equivalence.

Another approach towards efficiently verifying all program revisions, which is the one we follow in this paper, is to use during each revision verification partial results obtained from previous revisions, in order to limit necessary analysis. Work in this field vary based on the underlying non-incremental verification technique used, which determines what information can be reused and how efficiently so. The work we find most closely related to ours is that of Beyer et al [BLN$^+$13], which suggests to reuse the abstraction precision in predicate abstraction.

Other techniques for reuse of verification results include reuse of function summaries for bounded model checking [CGS12], contextual assumptions for assume-guarantee reasoning [HMW16], parts of a proof or counter-example obtained through ic3 [CIM$^+$11] and inductive invariants [FGS14]. Also, incremental techniques for runtime verification of

probabilistic systems modeled as Markov decision processes are developed in [FKP+12]. For the special case of component-based systems, [JCK13] uses algebraic representations to minimize the number of individual components that need to be reverified. Last, the tool Green [VGD12] facilitates reuse of SMT solver results for general purposes, and authors demonstrate how this could be beneficial for incremental program analysis.

4.7 Conclusion

We have presented a novel automata-based approach for incremental verification. Our approach relies on the method of [HHP09, HHP13] which uses a trace abstraction as a proof of correctness. Our idea is to reuse a trace abstraction by first translating it to the alphabet of the program under inspection, and then subtracting its automata from the control-flow automaton. We have defined a procedure, TranslateAutomaton, for automata translation, and two algorithms for reuse of trace abstraction that differ in their strategy for automata subtraction. We have evaluated our approach on a set of previously established benchmarks on which we get significant speedups, thus demonstrating the usefulness of trace abstraction reuse.

Chapter 5

Must Fault Localization For Program Repair

Abstract

This work is concerned with fault localization for automated program repair.

We define a novel concept of a *must* location set. Intuitively, such a set includes at least one program location from every repair for a bug. Thus, it is impossible to fix the bug without changing at least one location from this set. A fault localization technique is considered a *must* algorithm if it returns a must location set for every buggy program and every bug in the program. We show that some traditional fault localization techniques are not must.

We observe that the notion of must fault localization depends on the chosen repair scheme, which identifies the changes that can be applied to program statements as part of a repair. We develop a new algorithm for fault localization and prove that it is *must* with respect to commonly used schemes in automated program repair.

We incorporate the new fault localization technique into an existing mutation-based program repair algorithm. We exploit it in order to prune the search space when a buggy mutated program has been generated. Our experiments show that must fault localization is able to significantly speed-up the repair process, without losing any of the potential repairs.

5.1 Introduction

Fault localization and automated program repair have long been combined. Traditionally, given a buggy program, fault localization suggests locations in the program that might be the cause of the bug. Repair then attempts to change those suspicious locations in order to eliminate the bug.

Bad fault localization may cause a miss of potential repairs, if it is too restrictive, or cause an extra work, if it is too permissive. Studies have shown that for test-based repair imprecise fault localizations happen very often in practice [LKB+19]. This identifies the need for fault localization that can narrow down the space of candidates while still promising not to lose potential causes for a bug.

In this work, we define the concept of a *must* location set. Intuitively, such a set includes at least one location from every repair for the bug. Thus, it *must* be used for repair. In other words, **it is impossible to fix the bug using only locations outside this set**. A fault localization technique is considered a *must* algorithm if it returns a must location set for every buggy program and every bug in the program.

To demonstrate the importance of the *must* notion, consider the program in Figure 5.1 for computing the absolute value of a variable `x`. The program is buggy since the assertion in location 4 is violated when intially `x = -1`. Intuitively, a good repair would replace the condition `(x < -1)` in location 2 with condition `x <= -1`. Our must fault localization, defined formally in the paper, will include location 2 in the must location set. In contrast, the fault localization techniques defined for instance in [JM11, ESW12] do not include 2 in their location sets: They are not must and may miss optional repairs.

Our first observation regarding must notions is that their definition should take into account the *repair scheme* under consideration. A repair scheme identifies the changes that can be applied to program statements as part of a repair. A scheme can allow, for instance, certain syntactic changes in a condition (e.g. replacing `<` with `>`) or in the right-hand-side expression of an assignment (e.g. replacing `+` by `-`). A particular location set can be a *must* set using one scheme, but non-*must* using another. We further discuss this observation when presenting our formal definition of a must fault localization.

The setting of our work is as follows. Our approach is formula-based rather than test-based. We handle simple C-programs, with specification given as assertions in the code. Similarly to bounded model checking tools (e.g. [CKL04]), the program and the negated specification are translated to a set of constraints, whose conjunction forms the *program formula*. This formula is satisfiable if and only if the program violates an assertion, in which case a satisfying assignment (also called a *model*) is returned.

We focus on a simple repair scheme of syntactic changes, as described above. We assume that the user prefers repairs that are as close to the original program as possible and will want to get several repair suggestions. Thus, we return *all minimal repairs* (minimal in the number of changes applied to the program code).

Once the notion of must fault localization is defined, we develop a new algorithm for fault localization and prove that it is *must* with respect to syntactic mutation schemes. The input to the algorithm is a program formula φ and a model μ for φ, representing a buggy execution of the program. Our approach is based on a dynamic-slicing-like algorithm that computes dependencies.

For a variable v in φ, its slice F is computed based on dynamic dependencies among variables in φ, whose values influence the value of v in μ. Informally, F is a must location set that contains all assignemts to the variables that v depends upon. Some assignment from F thus must be changed in order to eliminate the bug associated with μ.

esides the promise of being *must*, our fault localization technique has several addi-

tional advantages:

1. The input to our method is exactly the output of bounded model checking tools such as [CKL04]: A program formula and its model. There is no need to run the program and to extract execution information. We also do not require any additional test inputs.

2. Unlike other formula-based approaches, we do not require solving the formula (footnote to clarify that solving was needed to obtain the model in the first place, but other approaches would require additional solving, of the extended formula, while ours does not). Our technique is linear in the size of the program formula.

3. Our technique is susceptible to incrementality under the mutation scheme: When used several times in a row for several mutated programs with different error traces, a significant amount of computational effort can be saved. Using a preliminary computation, which is linear in the size of the program formula, we are then able to compute each fault localization instance at a cost that is linear only in the size of the result (where the size of the result is at most the number of executed statements).

We incorporated the new fault localization technique into an existing mutation-based program repair algorithm [RG16]. In [RG16], the repair scheme is based on a predefined set of mutations. Given a buggy program P, the goal of the algorithm is to return all minimal repairs for P. The algorithm goes through iterations of generate-validate, where the generate part produces a mutated program of P and the validate part checks whether it is bounded-correct. The bottleneck of the algorithm is the size of the search space, consisting of all possible mutated programs of P. In [RG16], the search space has been pruned when the generated mutated program has been successfully validated. No pruning has been applied otherwise.

In this work, we exploit our novel *must* fault localization in order to prune the search space when a buggy mutated program P' has been generated (i.e. validation failed). In this case, we compute the *must* location set F of P'. We can now prune from the search space any mutated program whose F locations are identical to those of P'. This is because, by the property of *must* location set, it is guaranteed that the bug cannot be repaired without changing a location in F. Thus, a large set of buggy mutated programs is pruned, without the need for additional validation and without losing any minimally repaired program. It should be noted that the smaller F is, the larger the pruned set is. Our experimental results confirm the effectiveness of this pruning by showing significant speedups.

To summarize, the contributions of this work are:

1. We define a novel notion of *must* fault localization with respect to a repair scheme. We show that many of the formula-based techniques are not must.

```
   procedure absValue(x)
1: abs := x
2: if x < -1 then
3:     abs := -x
4: assert (abs >= 0)
```

Figure 5.1: A buggy program

2. We present a novel fault localization technique and prove that it is *must* for the scheme of syntactic mutations. Our technique also has other advantages, such as low-complexity and incrementality.

3. We show how our new fault localization technique can be incorporated into an existing mutation-based program repair algorithm for pruning its search space. The technique is applied iteratively, whenever a generated mutated program is found to be incorrect.

4. We implemented the algorithm of repair with fault localization as part of the open source tool AllRepair. Our experimental results show that fault-localization is able to significantly speed-up the repair process, without losing any of the potential repairs.

5.2 Motivating Example

Figure 5.1 presents a simple program for computing the absolute value of a variable `x`. The result is computed in the variable `abs`, and the specification states, using an assertion on line 4, that in the end `abs` should always be non-negative. Unfortunately, the program has a bug. The true branch of the if is intended to flip the sign of `x` whenever `x` is negative, but it accidentally misses the case where `x` is -1. As a result, if `x` is -1, the wrong branch of the if is taken, and the assertion is reached with `abs` $= -1$, which causes a violation.

Clearly, it is desirable that line number 2 be returned when running fault localization on this bug, as a human written repair is likely to change the condition on this line from `x < -1` to `x <= -1` or `x < 0`. But, as we will show next, some of the existing formula-based fault localization techniques do not include this line in their result.

The error trace representing the bug for input $I = \{\texttt{x} \leftarrow -1\}$ is $\pi =< 1, 2, 4 >$ (this is the sequence of program locations visited when executing the program on I). The MAX-SAT-based fault localization technique of [JM11] and the error-invariant-based technique of [ESW12] use a formula called the *extended trace formula* in order to find faulty statements along the error trace. The extended trace formula for the bug in question is

$$\underbrace{(x = -1)}_{\text{Input}} \wedge \underbrace{(abs = x) \wedge (x \geq -1)}_{\text{Computation}} \wedge \underbrace{(abs \geq 0)}_{\text{Assertion}}$$

This formula encodes three things: a) that the input remains I, b) that the computation is as the trace dictates, and, c) that the assertion holds at the end. Therefore, the formula is unsatisfiable. Both [JM11] and [ESW12] intuitively look for explanations of its unsatisfiability, and therefore decide that the statement $(x \geq -1)$ on line 2 is irrelevant; The formula remains unsatisfiable even if the constraint $(x \geq -1)$ is removed.

Even the method of [CESW13], which suggests a flow-sensitive encoding of the extended trace formula, with the goal of including all statements affecting control-flow decisions that are relevant to the bug, classifies the statement on line 2 as irrelevant. This is because the error trace does not include any location from the body of the branch that was taken (in our case it is the `else` branch, which is empty), in which case the flow-sensitive formula remains identical to the traditional formula.

The dynamic slicing method of [AH90, KL88] also fails to include line 2 in its result. This method computes the set of statements influencing the evaluation of the assertion along the trace, using data and control dependency relations. A statement st_1 is data dependent on st_2 iff st_1 uses a variable x, and st_2 is the last to assign a value to x along the trace. In our example, the assertion on line 4 is data dependent only on the statement in line 1, which in itself is not data dependent on any other statement. A statement st_1 is control dependent on a conditional statement st_2 iff st_1 is inside the body of either branch of st_2. None of the statements along our error trace is control dependent on another statement. The slice, which is the set of lines returned, is computed using the transitive closure of these relations. Thus, for our example, only line 1 is part of the slice.

In this example, we have seen how many different fault localization techniques fail to include a statement that is relevant, i.e., where a modification could be made for the bug to be fixed. In contrast, the set of locations returned by our technique for this example is $\{1, 2\}$. The fact that our technique includes line 2 is not a coincidence: We show that, intuitively, whenever a repair can be made by making changes to a single line, this line *must* be included in the result. In general, whenever a repair can be made by making changes to a set of lines, at least one of them must be included in the result.

5.3 Preliminaries

5.3.1 Programs and Error Traces

For our purposes, a *program* is a sequential program composed of standard statements: assignments, conditionals, loops and function calls, all with their standard semantics. Each statement is located at a certain *location* (or *line*) l_i, and all statements are defined over the set of program variables X.

In addition to the standard statements, a program may also contain *assume* statements of the form `assume(bexpr)`, and *assert* statements of the form `assert(bexpr)`. In both cases `bexpr` is a boolean expression over X. If an assume or an assert statement

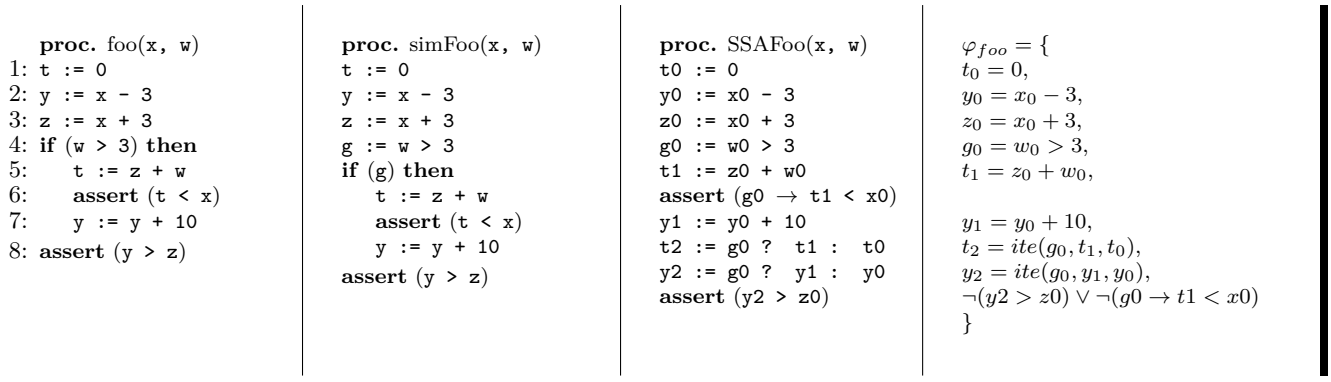


Figure 5.2: Example of the translation process of a simple program

is located in l_i, execution of the program stops whenever location l_i is reached in a state where `bexpr` is evaluated to false. In the case of an assertion, this early termination has the special name *assertion violation*, and it is an indication that an error has occurred.

A program P has a *bug on input* I if an assertion violation occurs during the execution of P on I. Otherwise, the program is *correct for* I.[1] Whenever P has a bug on I, this bug is associated with an *error trace*, which is the sequence of statements visited during the execution of P on I.

5.3.2 From Programs to Program Formulas

In this section we explain how a program is translated into a set of constraints, whose conjunction constitutes the program formula. In addition to constraints representing assignments and conditionals, such a formula includes constraints representing assumptions and a constraint representing the negated conjunction of all assertions. Thus, a satisfying assignment (a *model*) of the program formula represents an execution of the program that satisfies all assumption but violates at least one assertion. Such an execution is a *counterexample*.

The translation, following [CKL04], goes through four stages. We refer to the example in Figure 5.2 to demonstrate certain steps.

1. Simplification: Complex constructs of the language are replaced with equivalent simpler ones. Also, branch conditions are replaced with fresh boolean variables. In the example, the `if` condition `(w > 3)` is assigned to a fresh boolean variable `g`. Branching is then done based on the value of `g`, instead of `(w > 3)`.

2. Unwinding: The body of each loop and each function is inlined wb times. The set of executions of the new program is called the wb-executions of P.

3. Conversion to SSA: The program is converted to static single assignment (SSA) form, which means that each variable in the new program is assigned at most once.

[1] Alternatively, one could assume to know the desired output of the program for I and define a bug on I as a case where the program outputs the wrong value for I.

This is done by replacing all variables with indexed variables, and increasing the index of a variable whenever it appears on the left-hand-side of an assignment. In the example, the first assignment to `t` is replaced by an assignment to `t0` and the second, by an assignment to `t1`. Since `t` is assigned inside a conditional statement and is used after the statement, the if-then-else assignment `t2 := g0?t1:t0` is inserted in order to determine which copy of `t` should be used after the conditional statement. These special if-then-else assignments are called Φ-*assignments*. In the example, there is also a Φ-assignment for `y` (`y2=g0?y1:y0`).

Note that, assertions are also expressed by means of indexed variables. The specific indices in the assertion indicate the location in the execution in which the assertion is checked. In addition, if an assumption or an assertion is located within an `if` statement with branch condition g, then it is implied by g if it is within the `then` part of the `if` and is implied by $\neg g$, if it is within the `else` part. In the example, `assert (t < x)` is encoded by $(g_0 \rightarrow t_1 < x_0)$.

4. Conversion to SMT constraints: Once the program is in SSA form, conversion to SMT is straightforward: An assignment `x:=e` is converted to the constraint $x = e$; A Φ-assignment `x:= b?x1:x2` is converted to the constraint $(x = ite(b, x_1, x_2))$, which is an abbreviation of $((b \wedge x = x_1) \vee (\neg b \wedge x = x_2))$; An assume statement `assume(bexpr)` is converted to the constraint `bexpr`, and an assert statement `assert(bexpr)` is converted to the constraint $\neg$`bexpr` (since a model of the SMT formula should correspond to an assertion violation).

   If the program includes several assertions, then they are converted to one constraint, representing the negation of their conjunction. In the example, the two assertions are converted to the following constraint:

$$\neg(y2 > z0) \vee \neg(g0 \rightarrow t1 < x0).$$

   We say that a constraint *encodes* the statement it came from and we partition constraints into three sets, S_{assign}, S_{phi} and S_{demand}, based on what they encode. S_{assign} contains constraints encoding assignments, including those originated from assigning a fresh boolean variable with a branching condition; S_{phi} - encoding Φ-assignments; and S_{demand} - encoding demands from assert and assume statements. In particular, it encodes the negated conjunction of all assertions.

The triple $(S_{assign}, S_{phi}, S_{demand})$ is called a *program constraint set*. The program constraint set we get from a program P when using wb as an unwinding bound is denoted CS_P^{wb}. The *program formula* φ_P^{wb}, is the conjunction of all constraints in all three sets of CS_P^{wb}:

$$\varphi_P^{wb} = (\bigwedge_{s \in S_{assign}} s) \wedge (\bigwedge_{s \in S_{phi}} s) \wedge (\bigwedge_{s \in S_{demand}} s).$$

[CKY03] 1. A program P is *wb*-violation free iff the formula φ_P^{wb} is unsatisfiable.

For simplicity of notation, in the rest of the paper we omit the superscript *wb*.

Since the program formula is the result of translating an SSA program, the formula is defined over indexed variables. Further, each constraint in S_{assign} corresponds to the single variable, which is assigned in the statement encoded by the constraint.

5.4 Must Fault Localization

In this section, we precisely define when a location should be considered relevant for a bug. This definition is motivated by a repair perspective, taking into account which changes can be made to statements in order to repair a bug.

In order to define the changes allowed, we use repair schemes. A *repair scheme* $\mathcal{S}$ is a function from statements to sets of statements. An $\mathcal{S}$*-patch* for a program P is a set of pairs of location and statement $\{(l_1, st_1^r), \cdots, (l_k, st_k^r)\}$, for which the following holds: for all $1 \leq i \leq k$, let st_i be the statement in location l_i in P, then $st_i^r \in \mathcal{S}(st_i)$. The patch is said to be *defined over* the set of locations $\{l_1, \cdots, l_k\}$. Applying an $\mathcal{S}$*-patch* τ to a program P means replacing for every location l_i in τ, the statement st_i with st_i^r. This results in an $\mathcal{S}$*-patched* program of P. The set of all $\mathcal{S}$*-patched* programs created from a program P is the $\mathcal{S}$*-search space* of P.

Let P be a program with a bug on input I, and $\mathcal{S}$ be a repair scheme. An $\mathcal{S}$*-repair* for I is an $\mathcal{S}$-patched program that is correct for I. An $\mathcal{S}$*-repairable set* is a set of locations F such that there exists an $\mathcal{S}$-repair defined over F. An $\mathcal{S}$-repairable set is *minimal* if removing any location from it makes it no longer an $\mathcal{S}$-repairable set. A location is $\mathcal{S}$*-relevant* if it is a part of a minimal $\mathcal{S}$-repairable set. [2]

In this paper, we focus on two repair schemes that are frequently used for automated program repair: the arbitrary scheme ($\mathcal{S}_{arb}$) and the mutation scheme ($\mathcal{S}_{mut}$). Both schemes only manipulate program expressions, but the mutation scheme is more restrictive than the arbitrary scheme: $\mathcal{S}_{arb}(st)$ is the set of all options to replace the expression of st[3] with an arbitrary expression, while $\mathcal{S}_{mut}(st)$ only contains statements where the expression in st is mutated according to a set of simple syntactic rules. The rules we consider are replacing a `+` operator with a `-` operator, and vice versa, replacing a `<` operator with a `>` operator, and vice versa, and increasing or decreasing a numerical constant by 1.[4]

Example 5.4.1. In this example we demonstrate how different repair schemes define different sets of relevant locations. Consider again the foo program from Figure 5.2.

[2] We sometimes omit $\mathcal{S}$ from notations where $\mathcal{S}$ is clear from context.

[3] If st is an assignment, its expression is its right-hand-side. If st is a conditional statement, its expression is its condition.

[4] This simple definition of the mutation scheme is used only for simplicity of presentation. Our implementation supports a much richer set of mutation rules, as explained in section 5.7.

This program has a bug on input $I = \texttt{x} \leftarrow 0, \texttt{w} \leftarrow 0$. The error trace associated with the bug is $\langle 1, 2, 3, 4, 8 \rangle$ (the assertion on line 8 is violated).

The location set $\{3, 4\}$ is a minimal $\mathcal{S}_{mut}$-repairable set: It is an $\mathcal{S}_{mut}$-repairable set because applying the $\mathcal{S}_{mut}$-patch $\{(3, \texttt{z:=x-3}), (4, \texttt{w<3})\}$, results in an $\mathcal{S}_{mut}$-patched program that is correct for I. This set is also minimal, because none of the $\mathcal{S}_{mut}$-patches defined over $\{3\}$ or $\{4\}$ alone is an $\mathcal{S}_{mut}$-repair for I: Each one of the $\mathcal{S}_{mut}$-patches $\{(3, \texttt{z:=x-3})\}, \{(3, \texttt{z:=x+4})\}, \{(3, \texttt{z:=x+2})\}, \{(4, \texttt{w<3})\}, \{(4, \texttt{w>4})\}, \{(4, \texttt{w>2})\}$ results in an assertion violation for I.

On the other hand, $\{3, 4\}$ is *not* a minimal $\mathcal{S}_{arb}$-repairable set: For example, the $\mathcal{S}_{arb}$-patch $\{(3, \texttt{z:=-6})\}$ is an $\mathcal{S}_{arb}$-repair for I. Note that, the $\mathcal{S}_{arb}$-patch only needs to repair the bug, and not the program. That is, it is sufficient that there is no assertion violation on the specific input I, even though an assertion could be violated in the $\mathcal{S}_{arb}$-patched program on another input.

The set of all minimal $\mathcal{S}_{arb}$*-repairable set*s is $\{\{2\}, \{3\}, \{4, 5\}\}$. Therefore, the set of $\mathcal{S}_{arb}$-relevant statements is $\{2, 3, 4, 5\}$. The set of all minimal $\mathcal{S}_{mut}$*-repairable set*s is $\{\{2, 3\}, \{3, 4\}\}$. Therefore, the set of $\mathcal{S}_{mut}$-relevant statements is $\{2, 3, 4\}$.

Fault localization should focus the programmer's attention on locations that are relevant for the bug. But, returning the exact set of $\mathcal{S}$*-relevant* locations, as defined above, can be computationally hard. In practice, what many fault localization algorithms return is a set of locations that *may* be relevant: The returned locations have a higher chance of being $\mathcal{S}$-relevant than those who are not, but there is no guarantee that all returned locations are $\mathcal{S}$-relevant, nor that all $\mathcal{S}$-relevant locations are returned. We call such an algorithm *may fault localization*. In contrast, we define *must fault localization*, as follows:

$\mathcal{S}$-must location set 1. An *$\mathcal{S}$-must location set* is a set of locations that contains at least one location from each minimal $\mathcal{S}$-repairable set.[5]

$\mathcal{S}$-must fault localization 2. An *$\mathcal{S}$-must fault localization* algorithm is an algorithm that for every program P and every buggy input I, returns an $\mathcal{S}$-must location set.

Note that, an $\mathcal{S}$-must location set is not required to contain all $\mathcal{S}$-relevant locations, but only one location from each minimal $\mathcal{S}$-repairable set. Still, this is a powerful notion since it guarantees that no repair is possible without including at least one element from the set.

Also note, that the set of all locations visited by P during its execution on I is always an $\mathcal{S}$-must location set. This is because any $\mathcal{S}$-patch where none of these locations is included is definitely **not** an $\mathcal{S}$-repair, since the same assertion will be violated along the same path. However, this set of locations may not be minimal. In the sequel, we aim at finding small $\mathcal{S}$-must location sets.

[5] This is, in fact, a hitting set of the set of all minimal $\mathcal{S}$-repairable sets.

Example 5.4.2. Continuing the previous example, the set $\{2, 3, 4\}$ is an $\mathcal{S}_{arb}$-must location set, and also an $\mathcal{S}_{mut}$-must location set. In contrast, the set $\{2, 3\}$ is only an $\mathcal{S}_{mut}$-must location set, but not an $\mathcal{S}_{arb}$-must location set, since it does not contain any location from the $\mathcal{S}_{arb}$-minimal repairable set $\{4, 5\}$. The set $\{2\}$ is neither an $\mathcal{S}_{arb}$-must location set nor an $\mathcal{S}_{mut}$-must location set.

Example 5.4.3. Consider again the `absValue` procedure of Figure 5.1. The set $\{2\}$ is an $\mathcal{S}_{mut}$-minimal repairable set and an $\mathcal{S}_{arb}$-minimal repairable set for the bug in question. Therefore, we can say that all algorithms that were shown in Section 5.2 not to include the location 2 in their result [JM11, ESW12, CESW13, AH90, KL88], are neither $\mathcal{S}_{arb}$-must nor $\mathcal{S}_{mut}$-must fault localization algorithms.

5.5 Fault Localization Using Program Formula Slicing

In this section we formally define the notion of slicing. Based on this, we present an algorithm for computing must fault localization for $\mathcal{S}_{arb}$ and $\mathcal{S}_{mut}$.

5.5.1 Program Formula Slicing

A central building block in our fault localization technique is *slicing*. But, we do not define slicing in terms of the program directly, but in terms of the program formula representing it, instead. The input to the slicing algorithm is a program formula φ, a model μ of it, and a variable v. Recall that φ is a conjunction of constraints from S_{assign}, S_{phi} and S_{demand} (see Section 5.3.2). The goal of the slicing algorithm is to compute the *slice* of the variable v with respect to φ and μ. Intuitively, this slice includes the set of all constraints that influence the value v gets in μ.

Similar to traditional slicing, it is easy to define the slice as the reflexive-transitive closure of a dependency relation. But, unlike traditional slicing, which defines dependencies between statements, our dependency relation is between variables of the formula. These variables are indexed. Each originates from a variable of the underlying SSA program, where it was assigned at most once. We refer to variables never assigned as *input variables*, and denote the set containing them by $InputVars$. A variable v that was assigned once is called a *computed variable*, and the (unique) constraint encoding the assignment to it is denoted $Assign(v)$. The set of all computed variables is denoted $ComputedVars$. We also denote by $vars(e)$ the set of variables that appear in a formula or expression e.

Static Dependency 3. The static dependency relation of a program formula φ is $SD_\varphi \subseteq vars(\varphi) \times vars(\varphi)$ s.t.

$$SD_\varphi = \{(v_1, v_2) \mid \exists e \; s.t. \; (v_1 = e) \in S_{assign}, v_2 \in vars(e)\} \cup$$

$$\{(v, b), (v, v_1), (v, v_2) | \; (v = ite(b, v_1, v_2)) \in S_{phi}\}$$

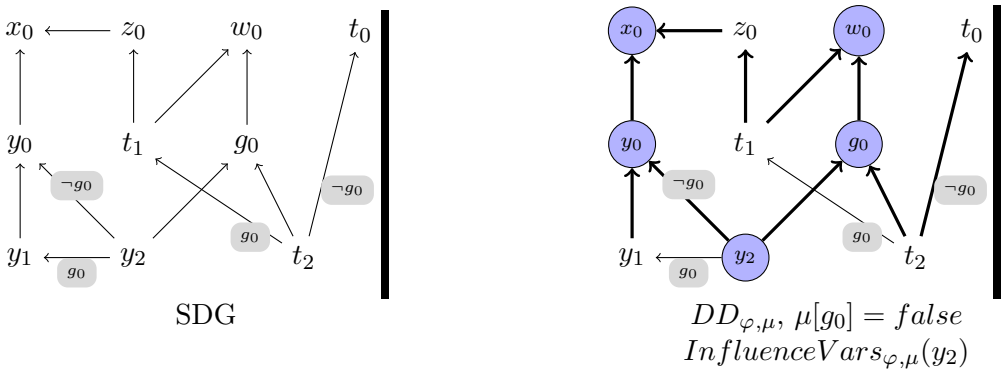


Figure 5.3: Illustration of the static and dynamic dependency relations of the `foo procedure`

.

The left-hand-side of Figure 5.3 presents the graph for the static depency relation of the `foo procedure` of Figure 5.2. The nodes in the graph are (indexed) variables and there is an arrow from v_1 to v_2 iff $(v_1, v_2) \in SD_\varphi$.

Dynamic Dependency 4. The dynamic dependency relation of a program formula φ and a model μ of φ is $DD_{\varphi,\mu} \subseteq vars(\varphi) \times vars(\varphi)$ s.t.

$$DD_{\varphi,\mu} = \{(v, v_1) \mid \exists b, v_2 \; s.t. \; (v = ite(b, v_1, v_2)) \in S_{phi}, \; \mu[b] = true\}$$

$$\cup\{(v, v_2) \mid \exists b, v_1 \; s.t. \; (v = ite(b, v_1, v_2)) \in S_{phi}, \; \mu[b] = false\}$$

$$\cup\{(v, b) \mid \exists v_1, v_2 \; s.t. \; (v = ite(b, v_1, v_2)) \in S_{phi}\}$$

$$\cup\{(v, v_1) \mid \exists e \; s.t. \; (v = e) \in S_{assign}, v_1 \in vars(e)\}$$

Note that, dynamic dependency includes only dependencies that coincide with the specific model μ, which determines whether the `then` or the `else` direction of the `if` is executed. Static dependency, on the other hand, takes both options into account. Thus, $DD_{\varphi,\mu} \subseteq SD_\varphi$ for every model μ.

The bold arrows on the right-hand-side of Figure 5.3 represent the relation $DD_{\varphi,\mu}$ of the `foo procedure`, for any μ where $\mu[g_0] = false$.

Influencing Variables 5. Given a program formula φ, a model μ of it, and a computed variable v, the set of influencing variables of v with respect to φ and μ is:

$$InfluenceVars_{\varphi,\mu}(v) = \{v' \mid (v, v') \in (DD_{\varphi,\mu})^* \}$$

The circled nodes on the right-hand-side of Figure 5.3 represents the variables that belong to $InfluenceVars_{\varphi,\mu}(y_2)$.

Program Formula Slice 6. Given a program formula φ, a model μ of it, and a computed variable v, the program formula slice of v with respect to φ and μ is:

$$Slice_{\varphi,\mu}(v) = \{Assign(v') \mid v' \in (InfluenceVars_{\varphi,\mu}(v) \cap ComputedVars)\}$$

Thus, intuitively, $Slice_{\varphi,\mu}(v)$ includes all constraints (in SSA form) encoding assignments that influence the value of v in μ. More precisely, when considering the conjunction of only the constraints of $Slice_{\varphi,\mu}(v)$, as long as the value of all input variables remains the same as in μ, the value of v will remain the same as well. This is formalized in the following theorem, whose proof can be found in Appendix B.

Theorem 2. For every φ, μ and v, the following holds:

$$\left[\bigwedge_{c \in Slice_{\varphi,\mu}(v)} c \wedge \bigwedge_{v_i \in InputVars} (v_i = \mu[v_i])\right] \implies (v = \mu[v])$$

ecall that, we define a variable v to be dependent on a boolean variable b only if v is defined by means of an assignment in S_{phi}. This in turn indicates that v has been assigned within the body of an `if` statement with condition b. Thus, b is included in the slice of v only if b is indeed relevant to the value of v.
Continuing with our example of `foo procedure`,

$$Slice_{\varphi,\mu}(y_2) = \{\ y_2 = ite(g_0, y_1, y_0),\ y_0 = x_0 - 3,\ g_0 = w_0 > 3\ \}.$$

5.5.2 Computing the Program Formula Slice

The computation of the program formula slice is composed of two steps. In the first step, we build a graph based on the static dependency relation, SD_φ. In the second step, we compute the slice $Slice_{\varphi,\mu}(v)$ by computing the set of nodes reachable from v in this graph, using a customized reachability algorithm, which makes use of the model μ.

The graph built during the first step is called the *Static Dependency Graph (SDG)* of φ. Nodes of this graph are variables of φ and edges are the static dependencies of SD_φ. Edges are annotated using the function ψ, mapping every static dependency (v, v') to a boolean formula such that $(v, v') \in DD_{\varphi,\mu}$ iff $\mu \models \psi[(v, v')]$. Specifically, for every constraint of the form $(v = ite(b, v_1, v_2))$ in S_{phi}, the edge (v, v_1) is annotated with b and the edge (v, v_2) is annotated with $\neg b$. All other edges of the graph are annotated with *true*. See the left-hand-side of Figure 5.3. For simplicity all *true* annotations are omitted.

The algorithm for the second step is presented in Algorithm 5.1. This algorithm gets a program formula φ, its SDG, a model μ of φ, and a variable v, and computes $Slice_{\varphi,\mu}(v)$. First, the set $InfluenceVars_{\varphi,\mu}(v)$ is computed as the set of nodes reachable from v in SDG, except that the reachability algorithm traverses an edge (v, v') only if

$\mu \models \psi[(v, v')]$. Thus, an edge (v, v') is traversed iff $(v, v') \in DD_{\varphi,\mu}$, which means that the set of reachable nodes computed this way is in fact $InfluenceVars_{\varphi,\mu}(v)$. Finally, the slice $Slice_{\varphi,\mu}(v)$ is the set of constraints encoding assignments to variables in $InfluenceVars_{\varphi,\mu}(v)$.

Algorithm 5.1 Compute The Program Formula Slice

Input: a program formula φ, its SDG, a model μ of φ and a variable v.

Output: $Slice_{\varphi,\mu}(v)$.

Procedure $ComputeSlice(\varphi, SDG, \mu, v)$

1: $V := \emptyset$
2: $ModelBasedDFS(SDG, v, \mu, V)$
3: $Slice := \{Assign(v') \mid v' \in V\}$
4: **return** $Slice$

Procedure $ModelBasedDFS(SDG, v, \mu, V)$

1: $V := V \cup \{v\}$
2: **for** $(v, w) \in E$ s.t. $\mu \models \psi[(v, w)]$ **do**
3: **if** $w \notin V$ **then**
4: $ModelBasedDFS(SDG, w, \mu, V)$

Algorithm 5.2 FOrmula-Slicing-Fault-Localization (FOSFL)

Input: A program formula φ of a program P, and a model μ of φ.

Output: A set of statements F of P.

Procedure $FOSFL(\varphi, \mu)$

1: $SDG := ComputeDependencyGraph(\varphi)$
2: demandFormula := $\bigwedge_{c \in S_{demand}} c$
3: $V := ImportantVars$(demandFormula,μ)
4: $S := \emptyset$
5: **for** $v \in V$ **do**
6: $S := S \cup ComputeSlice(\varphi, SDG, \mu, v)$
7: $F := \emptyset$
8: **for** $c \in S \cap S_{assign}$ **do**
9: $F := F \cup \{Origin(c)\}$
10: **return** F

5.5.3 The Fault Localization Algorithm

Our fault localization algorithm is presented in Algorithm 5.2. The input to this algorithm is a program formula φ of a program P, and a model μ of φ. The model μ represents a buggy execution of P on an input I, and the algorithm returns a set of locations, F, that is an $\mathcal{S}_{mut}$-must location set.

As before, we assume to know the origin of constraints in φ, and use the sets S_{assign}, S_{phi} and S_{demand}. Furthermore, here we also assume that for every constraint $c \in S_{assign}$, we know exactly which program statement it came from. We call this statement the *origin* of c, and denote it by $Origin(c)$.

As a first step, the algorithm computes a set of variables V by calling the procedure $ImportantVars$. This procedure receives an SMT formula φ and a model μ of φ, and reduces μ to a partial model of φ. A *partial model* of φ w.r.t. μ is a partial mapping from variables of the formula to values, which is consistent with μ and is sufficient to satisfy the formula. For example, for the formula $\varphi = (a = 0 \vee b = 0)$ and the model $\mu = \{a \mapsto 0, b \mapsto 1\}$, the valuation $\{a \mapsto 0\}$ is a partial model of φ. Procedure $ImportantVars$ will return the set of variables that appear in the partial model ($\{a\}$ in our example). Details of this procedure are presented in Appendix A.

The formula passed to $ImportantVars$ in our case is the conjunction of all demands in S_{demand}. Recall that the set S_{demand} contains constraints encoding all conditions

that need to be met for an assertion violation to happen: Conditions from assumptions appear as is, while conditions from assertions are negated and disjuncted (See Figure 5.2. The last constraint on the right-hand-side represents the disjunction of the negated assertions). Therefore, the set of variables V, returned by $ImportantVars$, is such that as long as their values in μ remain the same, this conjunction will still be satisfied, which means that an assertion violation will still occur.

To make sure that their values do *not* remain the same, we use slicing: The algorithm proceeds by computing the program formula slice for each of the variables in V using Algorithm 5.1. All slices are united into the combined set S. This set represents all constraints that if remain the same, then *all* the variables in V maintain their value. Thus, at least one element from S must be included in any repair.

Note that, by first applying $ImportantVars$, we reduce the number of variables whose value should be preserved in order to maintain the bug. The smaller this number, the smaller F is. We will explain the usefulness of a small F in Section 5.6.

Finally, we need to translate the constraints in S back to statements of P. Because of how the slicing algorithm works, constraints in S may belong to either S_{assign} or S_{phi}. If they belong to S_{phi}, we ignore them, because they encode the control-flow structure of the program, rather than a particular statement. Otherwise, we add the origin of the constraint, which is a statement of the program, to the set of returned locations, F. Note that, several different constraints may have the same origin, for example due to loop unwinding. In such a case, it is sufficient for one constraint encoding the statement st to be included in S, for st to be included in F. A proof for the following theorem can be found in Appendix B.

Theorem 3. Algorithm FOSFL is an $\mathcal{S}_{arb}$-must and also an $\mathcal{S}_{mut}$-must fault localization algorithm.

5.5.4 Incremental Fault Localization

It is often necessary to apply fault localization to several bugs in the same program, or even to several programs with different bugs. Therefore, it is desired that the fault localization algorithm be *incremental*, which means that the computation effort of each fault localization attempt should be proportional to the changes made from the previous attempt. In other words, we should avoid re-computation whenever possible, taking advantage of the fact that the program remains the same, or at least remains similar.

Algorithm FOSFL can be easily made incremental for the case of different bugs of the same program. In this case, several successive calls are made to the algorithm using the same program formula φ, but with different models of it. Since the static dependency relation SD_{φ} depends solely on the program formula, and not on the model, we can avoid re-computing the SDG for each call. Instead, we can compute the SDG once, upfront, and whenever FOSFL is called, simply skip the first line. We call the incremental version of FOSFL Incremental-Formula-Slicing-Fault-Localization

(I-FOSFL).

Note that I-FOSFL is useful not only for fault localization of different bugs of the same program, but also whenever the SDG remains the same during successive fault localization calls. This is the case when considering different mutated programs P' of the same program P, since every change to P' replaces an expression e with an expression e' over the same variables. Thus, the SDG remains the same, since the static dependency relation, in fact, only depends on $vars(e)$, and not on e itself[6].

5.6 Program Repair with Iterative Fault Localization

In [RG16], a mutation-based algorithm for program repair, named AllRepair, was presented. This algorithm uses the mutation scheme in order to repair programs with respect to assertions in the code. Unlike fault localization, where the motivation is repairing a bug for a specific input, program repair aims at repairing the program for *all* inputs. To avoid confusion, we refer to a repair for all inputs as a *full repair*. In [RG16], the notion of a *full repair* is bounded: loops are unwound wb times, and a program is considered *fully repaired* if no assertion is violated along executions with at most wb unwindings. A program that is not fully repaired is said to be *buggy*. For the rest of this section, we refer to an $\mathcal{S}_{mut}$-patch as a patch, and to an $\mathcal{S}_{mut}$-patched program as a mutated program.

As its name implies, the goal of AllRepair is to obtain all *minimal* fully repaired mutated programs, where minimality refers to the patch used in the program. It goes through an iterative generate-validate process. The generate phase chooses a mutated program from the search space, and the validate phase checks whether this program is fully repaired, by solving its program formula. The mutated program is fully repaired iff the formula is unsatisfiable.

The generate-validate process is realized using an interplay between a SAT solver and an SMT solver. The SAT solver is used for the generate stage. For every mutation M and line l, there is a boolean variable $B_M(l)$, which is true if and only if mutation M is applied to line l. A boolean formula is constructed and sent to the SAT solver, where each satisfying assignment corresponds to a program in the search space. The SMT solver is used for the validate stage. The program formula of the mutated program is solved to check if it is buggy or not. To achieve minimality, when a mutated program created using a patch τ is fully repaired, every mutated program created using a patch τ', with $\tau \subseteq \tau'$, is blocked.

Example 5.6.1. Let P^M be a fully repaired mutated program obtained by applying the patch τ, consisting of mutating line l_1 using mutation M_1 and mutating line l_2 using mutation M_2. Then blocking any superset of τ will we done by adding to the boolean

[6]This is true for $\mathcal{S}_{mut}$ but not for $\mathcal{S}_{arb}$, since the latter allows to replace an expression e with an expression e' over different variables.

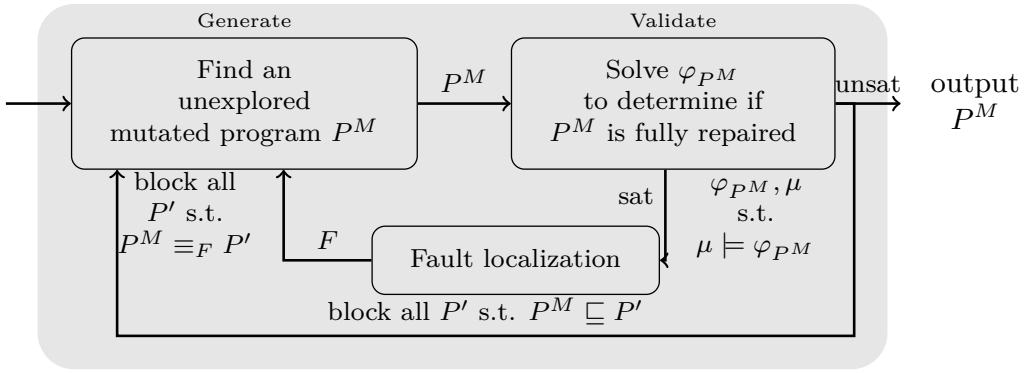


Figure 5.4: Algorithm FL-AllRepair: Mutation-based program repair with iterative fault localization. The notation $P^M \equiv_F P'$ means that P^M and P' agree on the content of all locations in F. The notation $P^M \sqsubseteq P'$ means that the patch used for creating P' is a superset of the patch used for creating P^M.

formula representing the search space, the blocking clause $\neg(B_{M_1}(l_1) \wedge B_{M_2}(l_2))$, which means "either do not apply M_1 to l_1 or do not apply M_2 to l_2". This clause blocks any mutated program with $\tau \subseteq \tau'$.

Blocking such programs prunes the search space, but only in a limited way. No pruning occurs when the mutated program is buggy.

In this paper, we extend the algorithm of [RG16] with a fault localization component. The goal of the new component is to prune the search space by identifying sets of mutated programs that are buggy, without inspecting each of the individual programs in the set.

Figure 5.4 shows the program repair algorithm with the addition of fault localization. In the new algorithm, called FL-AllRepair, whenever a mutated program is found to be buggy during the validation step, its program formula is passed to the fault localization component along with the model obtained when solving the formula. The fault localization component returns a set of locations F, following the I-FOSFL algorithm. Since this set is guaranteed to be an $\mathcal{S}_{mut}$-must location set, at least one of the locations in it should be changed for the bug to be fixed. Consequently, all mutated programs in which all locations from F remain unchanged are blocked from being explored in the future. As before, blocking is done by adding a blocking clause that disallows such programs.

Example 5.6.2. Let P^M be a buggy mutated program for which F consists of $\{l_1, l_2, l_3\}$, where l_1 was mutated with M_1, l_2 was not mutated, and l_3 was mutated with M_3. The blocking clause $\neg B_{M_1}(l_1) \vee \neg B_{Original}(l_2) \vee \neg B_{M_3}(l_3)$ will be added to the boolean formula representing the search space of mutated programs. It restricts the search space to those mutated programs that either do not apply mutation M_1 to l_1, or do mutate l_2 or do not apply M_3 to l_3. This will prune from the search space all mutated programs which are identical to P^M on the locations in F. Note that smaller F will result in a larger set of pruned programs.

Correctness of Algorithm FL-AllRepair In [RG16], the base algorithm was proved to be sound and complete, where sound means that every mutated program returned is minimally fully repaired, and complete means that every minimally fully repaired program will eventually be returned. In the new algorithm, whenever P^M is buggy, not only P^M is blocked, but also every P' that agrees with P^M on the content of all locations in F. Clearly, this blocking does not harm soundness. Moreover, since fault localization returns an $\mathcal{S}_{mut}$-must location set, blocking does not damage completeness either.

Proposition 1. Algorithm FL-AllRepair is sound and complete

5.7 Experimental Results

We have implemented our fault localization technique and its integration with mutated-based program repair in the tool AllRepair, available at `https://github.com/batchenRothenberg/AllRepair`. In this section, we present experiments evaluating the contribution of the new fault localization component to the program repair algorithm. We refer to the algorithm of [RG16], without fault localization, as AllRepair, and to the algorithm presented in this paper as FL-AllRepair. Both algorithms search for minimal *wb*-violation free programs, and both are sound and complete. Thus, for every buggy program and every bound *wb*, both algorithms will eventually produce the same list of repairs.

The difference between the algorithms lies in the repair loop. In case a mutated program is found to be buggy, the AllRepair algorithm will only block the one program, while the FL-AllRepair algorithm might block a set of programs. Therefore, the number of repair iterations required to cover the search space can only decrease using the FL-AllRepair algorithm. On the other hand, the cost of each iteration with fault localization is strictly higher than without it. Our goal in this evaluation is to check if the use of fault localization pays off. That is, to check if repairs are produced faster using FL-AllRepair than using AllRepair.

Benchmarks For our evaluation, we have used programs from two benchmarks: TCAS and Codeflaws. The TCAS benchmark is part of the Siemens suite [DER05], and is frequently used for program repair evaluation [BDF+12, RG16, NTC19]. The TCAS program implements a traffic collision avoidance system for aircrafts, and consists of approximately 180 lines of code. We have used all 41 faulty versions of the benchmark in our experiments.

The Codeflaws benchmark [TYM+17] is also a well-known and widely used benchmark for program repair. Programs in this benchmark are taken from buggy user submissions to the programming contest site Codeforces[7]. In each program, a user

[7] http://codeforces.com/

Level 1	Level 2
{+,-},{/,%}	{+,-,*},{/,%}
{>,>=},{<,<=}	{>,>=,<,<=}, {==,!=}
{||,&&}	
{>>, <<},{&,|,^}	
	C→C+1,C→C-1, C→-C,C→0

Figure 5.5: Partition of mutations to levels

tries to solve a programming problem published as part of a contest on the site. The programming problems are varied, and also the users have a diverse level of expertise. The benchmark also provides correct versions for all buggy versions, which are used to classify bug types by computing the syntactic difference. For our experiments we randomly chose 13 buggy versions classified with bug types that can be fixed using mutations. The size of the chosen programs ranges from 17 to 44 lines of code.

Mutations The mutations used in AllRepair (and accordingly in FL-AllRepair) is a subset of the mutations used in [RHJ+12]. We define two *mutation levels*, where level 1 contains only a subset of the mutations available in level 2. Thus, level 1 involves easier computation but may fail more often in finding repairs.

Table 5.5 shows the list of mutations used in each mutation level. For example, for the category of arithmetic operator replacement, in mutation level 1, the table specifies two sets: {+,-} and {/,%}. This means that a + can be replaced by a - , and vice versa, and that the operators /,% can be replaced with each other. Constant manipulation mutations apply to a numeric constant and include increasing its value by 1 (C→C+1), decreasing it by 1 (C→C-1), setting it to 0 (C→0) and changing its sign (C→-C).

Setting All of our experiments were run on a Linux 64-bit Ubuntu 16.0.4 virtual machine with 1 CPU, 4 GB of RAM and 40 GB of storage, provided using the VMware vRA service[8]. For each of the buggy versions in our benchmarks we have experimented with both mutation levels 1 and 2. For the Codeflaws benchmarks we additionally experimented with different unwinding bounds: 2 (entering the loop once), 5, 8 and 10. This experiment is irrelevant to the TCAS benchmarks since the TCAS program does not contain loops or recursive calls. Overall we had 186 combinations of buggy programs, mutation levels and unwinding bounds. We refer to each such combination as an *input*. For each input, we run both the AllRepair and the FL-AllRepair algorithms with a timeout of 10 minutes and a mutation size limit of 2 (i.e., at most two mutations could be applied at once).

[8]https://www.vmware.com/il/products/vrealize-automation.html

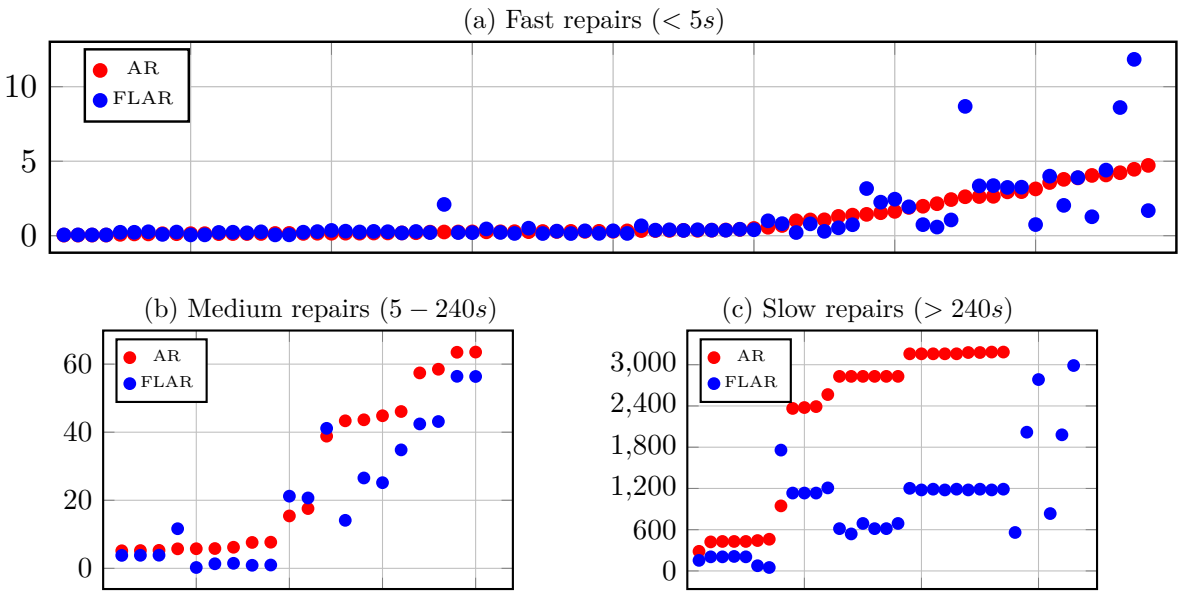


Figure 5.6: Time to find each repair using AllRepair (AR) and FL-AllRepair (FLAR). Each x value represents a single repair, and the corresponding y values represent the time, in seconds, it took to find that repair using both algorithms. Note that the graphs differ in the y axis scale.

5.7.1 Results

In total, 131 different repairs were found during our experiments, for 60 different inputs (for several inputs there was more than one possible repair). In this count, we treat repairs fixing the same program in the same way as different, if they were produced using different mutation levels or unwinding bounds. This is because our evaluation is concerned with the time to find these repairs, and both the mutation level and the unwinding bound greatly influence this time.

Because the time to produce a repair sometimes varied in several orders of magnitude depending on the input, we have chosen to split repairs into three categories: fast, intermediate, and slow, and examine the time difference separately for each category. Splitting repairs to categories was done according to the time it took to find them using the AllRepair algorithm. If that time was under 5 seconds, the repair was considered fast. If it was over 4 minutes, it was considered slow, and otherwise it was considered intermediate.

Figure 5.6 shows a comparison of the time, in seconds, it took to find repairs in both algorithms. There are three graphs, according to our three categories. In all graphs, each x value represents a single repair, where the corresponding blue dot in the y axis represents the time it took to find that repair using AllRepair, and the red square represents the time using FL-AllRepair. So, whenever the blue dot is above the red square, FL-AllRepair was faster in finding that repair, and the y difference represents the time saved.

For the fast category (Figure 5.6a), there is no clear advantage to FL-AllRepair. The majority of the repairs in this category are produced in less than a second using both algorithms. For the remaining repairs, there appears to be as many cases where FL-AllRepair is faster as when it is slower. But, in all cases where there is a time difference, in either direction, it is only of a few seconds.

For the intermediate category (Figure 5.6b), the advantage of FL-AllRepair is starting to become clear. There are now only 4 repairs (out of 20) for which FL-AllRepair is slower. Also, on average, it is slower by 4 seconds, but faster by 10 seconds. Finally, for the slow category (Figure 5.6c), there is an obvious advantage to FL-AllRepair. First, it is able to find 6 repairs *exclusively*, while AllRepair reaches a time-out. Also, for the remaining 27 repairs, FL-AllRepair is faster in all cases but one. The time difference is now also very significant: FL-AllRepair is faster by 1512 seconds (around 25 minutes) on average.

To sum up, the results show that in many cases our algorithm FL-AllRepair is able to save time in finding repairs. The savings are especially significant in cases where it takes a long time to produce the repair using the original AllRepair algorithm, and these are the cases where time savings are most needed.

5.7.2 Comparison with Other Repair Methods

The TCAS benchmark was recently used also in [NTC19], where AllRepair's performance was compared to that of four other automated repair tools: Angelix [MYR16], GenProg [LNFW12], FoRenSiC [BDF+12] and Maple [NTC19]. AllRepair was found to be faster by an order of magnitude than all of the compared tools, taking only 16.9 seconds to find a repair on average, where the other tools take 1540.7, 325.4, 360.1, and 155.3 seconds, respectively. Since in our experiments on TCAS fl-AllRepair was faster than AllRepair on average (and even when it was slower it was only by a few seconds), we conclude that fl-AllRepair also compares favorably to these other tools.

In terms of repairability, the repair scheme used by AllRepair (and fl-AllRepair) is limited compared to the other tools: AllRepair only uses mutations on expressions while Angelix, FoRenSiC and Maple allow replacing an expression with a template (e.g., a linear combination of variables), which is then filled out to create a repair. GenProg allows modifying a statement as well as deleting it or adding a statement after it. Therefore, the other tools are inherently capable of producing repairs in more cases than AllRepair.

In the case of TCAS, the study showed that AllRepair is able to find repairs for 18 versions (a result that we confirm in our experiments as well), while Angelix, GenProg, FoRenSiC and Maple found 32, 11, 23 and 26, respectively. But, what the study also showed, is that in repair methods that are based on tests, in many cases the repair found only adhered to the test-suite, but was not correct when inspected manually. When counting only correct repairs, AllRepair finds repairs for 18 versions

(all of AllRepairs repairs are correct), while Angelix, GenProg, FoRenSiC and Maple find 9, 0, 15 and 26, respectively. Since fl-AllRepair is able to find all repairs found by AllRepair, the same results also apply to fl-AllRepair.

5.8 Related Work

Dynamic slicing has been widely used for fault localization in the past [ZGG07a, ZGG07b, QX08, Wot10, HW12, WPP+14]. But, as we have seen, traditional notations of dynamic slicing [AH90, KL88] are not must (with respect to neither of the presented schemes), and thus, the above techniques may fail to include relevant locations in their results.

Other approaches for fault localization include spectrum-based (SBFL) [JHS01, AZVG06, EDC10, NLR11, WDGL14], mutation-based (MBFL)[MKKY14, PT15, HLK+15, GZL15] and formula-based (FBFL) [JM11, ESW12, SSNW13, HSNB+16, CHM+19]. Both SBFL and MBFL techniques compute the suspiciousness of a statement using coverage information from failing and passing test executions. MBFL uses, in addition, information on how test results change after applying different mutations to the program. Both SBFL and MBFL techniques can be seen as may fault localization techniques, in nature: they return locations that *are likely* to be relevant to the failing execution, based on all executions. We see may fault localization techniques as orthogonal to ours (and to must fault localization techniques in general), since in the trade-off between returning a small set of locations, and returning one that is guaranteed to contain all relevant statements, may techniques prefer the first, while must techniques prefer the second. In the context of repair, there are interesting applications for both.

FBFL techniques represent an error trace using an SMT formula and analyze it to find suspicious locations. These techniques include using error invariants [ESW12, CESW13, SSNW13, HSNB+16], maximum satisfiability [JM11, LNH15, LN16], and weakest preconditions [CHM+19]. What we were able to show in this paper, is that the methods of [JM11, ESW12, CESW13] are not must. In contrast, we believe (though we do not prove it) that the methods of [LNH15, LN16, CHM+19] are must. But, what [LNH15, LN16, CHM+19] have in common is that they use the semantics of the error trace or the program. Though semantic information can help to further minimize the number of suspicious locations, retrieving it involves using expensive solving-based procedures. Our approach, on the other hand, uses only syntactic information, which makes the fault localization computation relatively cheap; No SMT solving is needed. Thus, these approaches can be seen as complementary to ours.

In the literature there is also a wide range of techniques for automated program repair using formal methods [NQRC13, MYR16, ABS17, JGB05, VJ15, KKK15, DSS16, NWKF17]. Both [DW10] and [RHJ+12] also use fault localization followed by applying mutations for repair. But, unlike this work, fault localization is applied only for the original program. Also, neither the Tarantula fault localization used in [DW10] nor the dynamic slicing used in [RHJ+12] carries the guarantee of being a must fault localization.

The tool MUT-APR [AB18] fixes binary operator faults in C programs, but only targets faults that require one line modification. The tools FoREnSiC [BDF+12] and Maple [NTC19] repair C programs with respect to a formal specification, but they do so by replacing expressions with templates, which are then patched and analysed. SemGraft [MNN+18] conducts repair with respect to a reference implementation, but relies on tests for SBFL fault localization of the original program.

5.9 Conclusion

In this work we define a novel notion of *must* fault localization, that carefully identifies program locations that are relevant for a bug, so that the set is sufficiently small but is guaranteed not to miss desired repairs. We also show that the notion of *must* fault localization should be defined with respect to the repair scheme in use. We show that our notion of must fault localization is particularly useful in pruning the search space of a specific mutation-based repair algorithm.

To the best of our knowledge, we are the first to investigate the widely-used notion of fault localization and to suggest criteria for evaluating its different implementation.

e have presented a slicing technique for programs in SSA form, and a fault localization algorithm that relies on it. This fault localization algorithm was integrated into a mutation-based program repair algorithm, where it was applied not only to the original program, but whenever a buggy mutated program was found during the search. The set of constraints returned by fault localization was used for efficient pruning of the search space: instead of blocking the one buggy program, all mutated programs in which this set remains unchanged were blocked. We have shown that by doing so we do not damage the completeness of the program repair algorithm, since all programs that are blocked are definitely buggy. Finally, our experimental results have confirmed that the program repair algorithm of this paper in fact produces all repairs that were produced before, and, in some cases, significantly faster.

Chapter 6

Conclusion and Discussion

In the process of software production and maintenance, much effort and many resources are invested in order to ensure that the product is as bug free as possible. Manual bug repair is time-consuming and requires close acquaintance with the checked program. Therefore, there is a great need for research on automated program repair, as done in this thesis.

Prior research on program repair has been mostly empirical, and has focused on heuristic, search-based, techniques, and on test-based repair. In contrast, the emphasis of this work is on examining the formal aspect of program repair, both in terms of the methods used and the specification provided. Thus, this work has contributed to closing this research gap.

The body of our work was presented in three papers whose connecting thread is the enhancement of the basic generate-and-validate scheme for program repair. Improvements were made to both the generate stage, which samples patched programs from the search space, and to the validate stage, which checks the correctness of the sampled program. The generate stage was improved by introducing efficient pruning of the search space: In chapter 3 this was done by removing non-minimal repaired programs upon finding a correct program, and in chapter 5 it was done also by using fault localization upon finding an incorrect program. The validate stage was improved by using incrementality: In chapter 3 we employed incremental SAT and SMT solving, and in chapter 4 we developed algorithms for incremental verification.

The main contributions of the research done as part of this thesis are as follows:

- Development of new algorithms and proof of their correctness
  - **Program repair algorithms**: AllRepair (Algorithm 3.4) and FLAllRepair (Algorithm 5.4)
  - **Trace-abstraction-based incremental verification algorithms**: Eager (Figure 4.5) and Lazy (Figure 4.6)
  - **Fault localization algorithm**: FOSFL (Algorithm 5.2)

- Experimental evaluation of the new algorithms
  - In section 3.6, our AllRepair algorithm was compared with two state-of-the-art techniques by Könighofer and Bloem [KB11, KB13] on the tcas benchmark, which contains 41 faulty versions of a program implementing a traffic collision avoidance system for aircrafts. Results have shown a clear trade-off between repairability and efficiency: When using a short list of mutations our method repairs less faulty versions than [KB11, KB13], but is significantly faster. On the other hand, when adding more mutations, the number of faulty versions we fix increases, and becomes better than [KB11, KB13], but the average time to repair also increases significantly, making the method slower than [KB11, KB13].
  - In section 4.5, our incremental verification algorithms were compared with stand-alone verification and with another state-of-the-art technique by Beyer et al. [BLN$^+$13] on a benchmark set based on different revisions of device drivers from the Linux kernel. Results have proved that incrementality pays-off: our method, both when used with the Eager algorithm and with the Lazy one, manages to save the user a considerable amount of time, for the vast majority of these benchmarks. Both algorithms obtain a nontrivial speedup of around $\times 4.7$ in analysis time (compared to stand-alone verification), which is $\times 1.5$ larger than the mean analysis speedup of [BLN$^+$13].
  - In section 5.7, the FLAllRepair algorithm was compared to the AllRepair algorithm, using programs from two benchmarks: tcas and Codeflaws. Our conclusion from the results was that fault localization is indeed very helpful in pruning the search space of program repair: in many cases FLAllRepair found repairs significantly faster than AllRepair. The savings were especially significant in cases where it took a long time to produce the repair using AllRepair, and these are the cases where time savings are most needed.
- Open source tools implementing our algorithms
  - **AllRepair** a program repair tool for C programs with assertions. Available for download at https://github.com/batchenRothenberg/AllRepair. This tool implements both the AllRepair and the FLAllRepair algorithms, and was developed from scratch as part of this research.
  - **Ultimate Automizer** an automata-based software model checker developed by the software engineering group of Andreas Podelski at the University of Freiburg. Available for download at https://monteverdi.informatik.uni-freiburg.de/tomcat/Website/?ui=tool&tool=automizer. The contribution of this research to the tool was the implementation of the Eager and the Lazy incremental verification algorithms. A new configuration of the tool was

introduced, allowing the user to run it in incremental mode and to choose the desired algorithm from the two.

6.1 Future Work

The work of this thesis can be expanded in several interesting directions. Next, we portray our ideas for what we believe is the most promising future work.

Using Incremental Verification for Repair The program repair algorithms described in this thesis (AllRepair, FLAllRepair) instantiate the generate and validate scheme using a bounded verification technique for the validate stage. Other repair algorithms could result from using the same method for the generate stage, but replacing the verification technique with any other off-the-shelf technique. But, if the verification techniques are used as a black box, solving the verification problem of each patched program we wish to validate in isolation, the efficiency of these algorithms will be limited.

Instead, incremental verification should be used. Incremental verification aims at facilitating the re-verification of successive revisions of a program; changes made from the previous revision are taken into account in an attempt to limit the analysis to only the parts of the program that need to be reanalyzed, and partial verification results are reused. The syntactically closer the two revisions are, the more likely incremental verification is to be useful. Therefore, it can be especially useful for generate and validate-based repair, since patched programs are usually very similar to each other syntactically.

There are many incremental verification techniques that could be used for this purpose. The first one is our work, presented in chapter 4, which is based on the use of trace abstraction for verification. This is a particularly promising approach, since trace-abstraction-based verification has been proved to be very successful, winning the SV-comp verification competition several times. Other techniques for reuse of verification results include reuse of function summaries for bounded model checking [CGS12], contextual assumptions for assume-guarantee reasoning [HMW16], parts of a proof or counter-example obtained through ic3 [CIM+11] and inductive invariants [FGS14].

While using incremental verification techniques for the validation stage can be effective, it can be even more effective if we do not use these methods as a black box. Instead, they can be provided with information on the code changes allowed for repair, and hence the expected differences between successive patched programs being verified. We believe that this information can be used to refine existing algorithms, or even to design variants of algorithms that are specific to a particular type of code change.

Fault Localization The fault localization problem, i.e., finding a set of locations responsible for a certain bug, is a long standing research problem. However, traditional fault localization methods are intended for use by humans trying to manually repair the code. As research on automated program repair progresses, a new use for these methods has emerged: many repair algorithms use them as a first step, in order to figure out which lines of code should be repaired, and then try to make changes to those lines. The difference between the use of these methods in algorithms and their use by human beings is significant, since human beings have freedom of action as to the nature of the repair, while the changes made by repair algorithms are limited and known in advance. Therefore, it is necessary to rethink fault localization from the point of view of automated repair.

Our work on fault localization, presented in Chapter 5, is a successful example of such rethinking. This work shows that an accurate definition of the properties of fault localization, can enable more efficient repair, without sacrificing its desired properties. For example, the definition of the must-fault-localization property (definition 2) allows programs to be removed from the repair search space, while ensuring that no good repair is lost.

We believe that there are other, similar, properties of fault localization that can be thought of, which, like the must definition, depend on the repair scheme, and that will be useful for repair. Given a bug in a program, and a particular repair scheme, the set of all ways to repair the bug using the repair scheme can be seen as set S of sets of locations, where changes from the scheme can be applied to make the program correct. The must definition, as defined in definition 2, actually requires that the set returned by fault localization always be a hitting set of S. Similarly, we could require that one of the sets in S will be returned exactly, or that all the locations that are in any set from S will be returned (i.e., that the returned set will contain the union of all sets from S). Each of these definitions leads to a different property of fault localization, or, in fact, to a different definition of what it means to "localize the fault".

New definitions such as these can, as mentioned, be used in repair algorithms and lead to their improvement. However, their contribution will not be just that; they can also lead to a better understanding and evaluation of existing fault localization methods. Although these methods have been the focus of research for many years, comparison and analysis of them has been and remains largely empirical. Defining additional formal properties will also allow for theoretical comparison, by sorting the methods into those in which a property holds and those in which it does not.

A theoretical comparison such as this is important in order to complete the picture obtained from an empirical comparison, which is sometimes partial and misleading. For example, in an empirical comparison some formal method may look bad compared to other methods, since its localization result for a particular set of programs is large and takes a long time to obtain; In a theoretical comparison, however, it may turn out that the formal method in fact guarantees certain properties that will hold in all cases, and

not just for the given programs, which is an important advantage.

Program Repair that Learns from Mistakes In chapter 5 we have presented algorithm FLAllRepair, which incorporates fault localization with a traditional generate and validate loop: whenever a program was found incorrect during the validation stage, fault localization was applied to it in order to obtain a set of suspicious lines, which were then used to derive a set of patched programs that could be safely removed from the search space. But, in fact, applying fault localization is only one example of how you can gain information from a failed repair attempt.

In general, we propose to replace the traditional generate and validate working scheme with an enhanced, three-staged, working scheme, called *generate-validate-analyze*. In the generate-validate-analyze scheme, whenever a program is found to be incorrect, a witness of the bug is further analyzed to obtain a search hint, which improves the sampling of patched programs in the future. We call it "repair that learns from mistakes".

In FLAllRepair, the analyze stage was realized by a fault localization component, the search hint was a must-location-set, and it was used to prune patched programs that are certainly erroneous because the content of all locations in the set was not changed. A possible extension of this, is to provide a search hint that not only indicates problematic locations, but also problematic changes. For example, in the case of mutation-based repair, the analyze stage can be given the list of mutations, and return a set of tuples of locations and mutations of the form (l, m), indicating that for every tuple in the set the mutation m should not be applied to the location l.

Another possible extension is to provide a search hint that results in pruning of programs that are only likely to be erroneous, but are not guaranteed to be so. In this case, the analyze stage can be realized using any off-the-shelf fault localization technique, and not necessarily one that is a must-fault-localization, as defined in definition 2. Also, one can use several heuristics to prioritize certain patched programs over others, based on information observed from the error trace.

Instead, one can think of an analyze stage that also gets information about the changes made when creating patched programs (e.g., the list of mutations in the case of AllRepair), and results in a search hint that is not a set of locations, but a set of tuples of locations and change actions. A tuple of the form (l, c), where l is a location and c is a change action, can be seen as either a *do* or a *don't-do* search hint: it can indicate that the code in location l should be changed using change action c, or that it shouldn't.

Integrating Test-Based and Formal Repair Test-based repair is the problem of repairing a program with respect to a test-suite; a program is considered correct iff it passes all tests in the test-suite. Overall, test-based techniques have shown to scale well and be able to automatically repair bugs and vulnerabilities from real life applications [WCW+18, HZWK18a, XMD+16, GPKS17, MYR16, LR16].

In recent years, however, the quality of repairs produced by test-based tools has been

studied more closely, and has been found to be problematic. It turns out that many tools produce repairs that are overfitting to the test suite. That is, despite passing all tests, these repairs will not be accepted by a human developer, since they break other desired functionalities, unspecified by the tests. In some cases, overfitting repairs might even introduce new bugs or cause the program to crash. This problem is now referred to as the overfitting problem and is believed to be a major obstacle for test-based repair [SBLGB15, LTLLG18, YMDM20].

On the other hand, formal program repair, as done in this work, produces repairs with an inherent guarantee of correctness. However, it is far less scalable, showing results only for small programs.

Therefore, the combined use of tests and formal methods seems to be a promising approach, with the potential of getting the best of both worlds. One example of a successful combination in the context of verification is concolic execution [SMA05], where concrete inputs guide the symbolic exploration of the program. We believe a similar idea can also work for repair: tests can guide the generation of programs, but formal methods will be used to verify that the result is correct.

Appendix A

Computing Important Variables

Definitions Let φ be a first-order SMT formula over a quantifier-free theory T. A *(partial) valuation* of the formula is a (partial) function from $vars(\varphi)$ to D, the domain of variables in the theory T. For a valuation μ, we denote by $\mu[v]$ the value of the variable v in μ. For a partial valuation ρ, $\rho[v]$ is either the value of v in ρ, if ρ is defined for v, or a don't-care symbol, $\perp$, otherwise ($\perp \notin D$). A partial valuation ρ is a *reduction* of a valuation μ, if ρ and μ agree on their common variables. That is, if for every $v \in vars(\varphi)$, either $\rho[v] = \perp$ or $\rho[v] = \mu[v]$. Symmetrically, in such a case μ is said to be an *extension* of ρ. The *V-reduction* of a valuation μ, denoted $\mu\lfloor_V$, is the (unique) reduction of μ that is defined for the exact set of variables V. A partial valuation ρ is a *partial model* of φ, denoted $\rho \models \varphi$, if every extension μ of it is a model of φ.

ImportantVars In this section we present a procedure, ImportantVars, which receives a quantifier-free SMT formula φ and a model μ of it, and finds a set of variables V such that the partial valuation $\mu\lfloor_V$ is a partial model of φ. Although in the literature there exist several algorithms for finding a partial model of an SMT formula, these algorithms aim at finding minimal or even minimum [DDMA12] models. Therefore, they have an exponential cost that would incur a severe overhead for our repair algorithm. Thus, we use a naive algorithm with linear cost, that does not guarantee that the produced model is minimal but manages to produce small partial models in practice.

The naive algorithm is presented in Algorithm A.1. It is given a quantifier-free SMT formula, and a model μ of it, and produces a partial model for the formula by creating a reduction of μ. As a first step, the formula is converted to negation normal form (NNF), by pushing negation inwards (this is done by the function ToNNF). Then, there is a recursive search for all sub-formulas with an or ($\vee$) operator, and for each of them, only the first disjunct satisfied by μ is kept. When a sub-formula with an and ($\wedge$) operator is reached, all of its conjuncts are kept. Finally, whenever the search reaches an atomic formula it computes the set of variables that appears in it and adds it to the *result* set. This set is then returned at the end, after the recursive traversal of the formula is completed.

Algorithm A.1 Find Important Variables

Input: quantifier-free SMT formula φ and a model μ of φ.
Output: set V of variables from φ.

Procedure $ImportantVars(\varphi, \mu)$

1: φ_{NNF} := ToNNF(φ)
2: $result := \emptyset$
3: $RecursiveImportantVars(\varphi_{NNF}, \mu, result)$
4: **return** $result$

Procedure $RecursiveImportantVars(\varphi, \mu, result)$

1: **if** φ is an $\wedge$ formula **then**
2: **for every** c, a conjunct of φ **do**
3: $RecursiveImportantVars(c, \mu, result)$
4: **else if** φ is an $\vee$ formula **then**
5: **for every** d, a disjunct of φ **do**
6: **if** $\mu \models d$ **then**
7: $RecursiveImportantVars(d, \mu, result)$
8: **break**
9: **else** ▷ φ is an atomic formula
10: $result := result \cup vars(\varphi)$

Proposition 2. Let V be the result of Algorithm A.1 on the formula φ and its model μ. Then, $\mu|_V \models \varphi$.

Appendix B

Proofs

B.1 Proof of Theorem 2

Instead of proving the theorem as stated, we will prove the following, stronger, lemma:

Lemma B.1.1. *For every φ, μ, v, and $v' \in InflVars_{\varphi,\mu}(v)$, the following holds:*

$$\left[\bigwedge_{c \in Slice_{\varphi,\mu}(v)} c \wedge \bigwedge_{v_i \in InputVars} (v_i = \mu[v_i])\right] \implies (v' = \mu[v']) \tag{B.1}$$

Since $v \in InflVars_{\varphi,\mu}(v)$, Theorem 2 will be implied.

Lemma B.1.1. Let φ, μ and v be a program formula, a model of it, and a variable, respectively. Let μ' s.t.

$$\mu' \models \left[\bigwedge_{c \in Slice_{\varphi,\mu}(v)} c \wedge \bigwedge_{v_i \in InputVars} (v_i = \mu[v_i])\right] \tag{*}$$

For every $v' \in InflVars_{\varphi,\mu}(v)$, if $v' \in InputVars$ then the constraint $(v' = \mu[v'])$ is a conjunct of *, and therefore clearly $\mu' \models (v' = \mu[v'])$.

Next, we show that also for every computed variable $v' \in InflVars_{\varphi,\mu}(v)$, $\mu' \models (v' = \mu[v'])$. Let us denote $Assign(v')$ by $(v' = e)$ (recall that this constraint encodes either an assignment or a Φ-assignment). First, since $v' \in InflVars_{\varphi,\mu}(v)$, $Assign(v') \in Slice_{\varphi,\mu}(v)$, which means that $\mu' \models (v' = e)$ (as a conjunct of *). Also, since $\mu \models \varphi$, and $Assign(v')$ is a conjunct of φ, $\mu \models Assign(v')$. That is, $\mu[v'] = \mu[e]$. Putting these two facts together, we get that it is sufficient to show that $\mu' \models (e = \mu[e])$, in order to get that $\mu' \models (v' = \mu[v'])$, and finish the proof.

In fact, it is thus sufficient to show that for every $v_e \in vars(e)$, $\mu' \models (v_e = \mu[v_e])$, since the value of $\mu'[e]$ is uniquely determined by the value μ' gives to the variables of e.

With that in mind, we use induction to show that for every computed variable $v' \in InflVars_{\varphi,\mu}(v)$, $\mu' \models (v' = \mu[v'])$. The induction is on the set of computed variables, which is in fact an induction on the location number in the SSA program

where the computed variable v' was assigned. Note that, in an SSA program, for every assignment or Φ-assignment of the form `x:=e` in a location l, all the variables in $vars(e)$ are assigned *before* l (if assigned at all).

base case Let $v' \in InflVars_{\varphi,\mu}(v)$ be a computed variable assigned on $l = 1$, and denote by $v' = e$ the constraint $Assign(v')$. Since this is the first location of the program, $vars(e) \subseteq InputVars$. Therefore, for every $v_e \in vars(e)$, $\mu' \models (v_e = \mu[v_e])$, because it is a conjunct of *.

inductive step Let $v' \in InflVars_{\varphi,\mu}(v)$ be a computed variable assigned on l, and denote by $v' = e$ the constraint $Assign(v')$. Let us explore the different options for $Assign(v')$:

1. If $Assign(v')$ is an assignment constraint: Let $v_e \in vars(e)$. If $v_e \in InputVars$, then $\mu' \models (v_e = \mu[v_e])$ as a conjunct of *. Otherwise, $v_e \in ComputedVars$. Also $v_e \in InflVars_{\varphi,\mu}(v)$, since $vDD^*_{\varphi,\mu}v'$ ($v' \in InflVars_{\varphi,\mu}(v)$) and also $v'DD_{\varphi,\mu}v_e$ (by definition of $DD_{\varphi,\mu}$). Therefore, from induction hypothesis, $\mu' \models (v_e = \mu[v_e])$.

2. If $Assign(v')$ is a Φ-assignment constraint, where $e = ite(b, v_1, v_2)$: $b \in InflVars_{\varphi,\mu}(v)$, since $vDD^*_{\varphi,\mu}v'$ and also $v'DD_{\varphi,\mu}b$. Therefore, $\mu' \models (b = \mu[b])$ (this is either from induction hypothesis, if $b \in ComputedVars$, or from *, if $b \in InputVars$). This in particular means that $\mu'[b] = \mu[b]$. Let us explore the different options for $\mu[b]$:

   (Ł) If $\mu[b] = true$: Since $\mu'[b] = \mu[b] = true$, the value of $\mu'[e]$ is determined by the value of $\mu'[v_1]$, and the value of $\mu[e]$ is determined by the value of $\mu[v_1]$. Therefore, if we can show that $\mu' \models (v_1 = \mu[v_1])$, then $\mu' \models (e = \mu[e])$, as needed.

   Since $\mu[b] = true$, by the definition of $DD_{\varphi,\mu}$, $v'DD_{\varphi,\mu}v_1$. In combination with $vDD^*_{\varphi,\mu}v'$, $v_1 \in InflVars_{\varphi,\mu}(v)$. Thus, indeed, $\mu' \models (v_1 = \mu[v_1])$ (this is either from induction hypothesis, if $v_1 \in ComputedVars$, or from *, if $v_1 \in InputVars$).

   (Ł) If $\mu[b] = false$: This case is symmetric to the one where $\mu[b] = true$. ■

B.2 Proof of Theorem 3

To prove this theorem we will prove the following two lemmas, from which the theorem is applied:

Lemma B.2.1. *Algorithm FOSFL is an $\mathcal{S}_{arb}$-must fault localization algorithm.*

Lemma B.2.2. *Every $\mathcal{S}_{arb}$-must fault localization algorithm is also an $\mathcal{S}_{mut}$-must fault localization algorithm.*

lemma B.2.1. To simplify the presentation of the proof, we begin with some notations. First, given a program formula φ, we denote by φ^{dem} the formula constructed from the conjunction of all constraints in S_{demand}. This is, in fact, the `demandFormula` computed on line 2 of the FOSFL algorithm, and it encodes only the requirements from assumptions and negation of assertions.

Also, we denote by φ^{comp} the formula constructed from the conjunction of all constraints in S_{assign} and S_{phi}. This formula encodes the computation of the program, without any requirements from assumptions or assertions, and therefore is always satisfiable. Furthermore, for every input I, there exists a model μ_I of φ^{comp} s.t. for every input variable $v \in InputVars$, $\mu_I[v]$ is the value of v in I. We say that such a model represents the execution of P on I. The correspondence between models of φ^{comp} and inputs also works in the other direction: for every model μ^{comp} of φ^{comp}, μ^{comp} represents the execution of P on I, where I is the input in which the value of v, for every $v \in InputVars$, is $\mu^{comp}[v]$.

Now that we are done presenting notations, we begin the actual proof: Assume, by contradiction, that FOSFL is not an $\mathcal{S}_{arb}$-must fault localization algorithm. Let P be a program with a program formula φ_P, and μ be a model of φ_P, s.t. the set F returned by FOSFL for φ_P and μ is not an $\mathcal{S}_{arb}$-must location set. That is, there exists a minimal $\mathcal{S}_{arb}$-repairable set R, s.t. $R \cap F = \emptyset$. Let P_R be an $\mathcal{S}_{arb}$-repair defined over R, and let φ_R be the program formula of P_R.

First, we claim that all constraints of the united slice S computed for φ_P and μ on lines 4-6 of the FOSFL algorithm, remain unchanged in φ_R (or, in fact, in φ_R^{comp}). Formally, let $VIP(\varphi_P^{dem}, \mu)$ be the set of important variables computed on line 3 of the algorithm for φ_P and μ. Then, for every constraint c in

$$S = \bigcup_{v_{ip} \in VIP(\varphi_P^{dem}, \mu)} Slice_{\varphi_P, \mu}(v_{ip})$$

c is a conjunct of φ_R^{comp}. This is because, the only statements changed in P_R are those on locations from R, and therefore the only constraints of φ_R that are different from φ_P are those encoding assignments in locations from R. This, in combination with the fact that $R \cap F = \emptyset$, implies what we want, since F is the set of all locations whose assignments are encoded by constraints in S (lines 7-9 of the algorithm).

Next, consider the formula

$$\varphi_R^{comp} \wedge \bigwedge_{v_i \in InputVars} (v_i = \mu[v_i]) \qquad (**)$$

This formula is satisfiable. Specifically, let I be the input that μ represents. The model μ_I^R of φ_R^{comp} that represents the execution of P_R on I, satisfies **. Note that, this input I is the buggy input we are trying to repair (P_R is an $\mathcal{S}_{arb}$-patched program that is correct for I).

Let $v_{ip} \in VIP(\varphi_P^{dem}, \mu)$. Since we have shown that every constraint c in S is a conjunct of φ_R^{comp}, and $Slice_{\varphi,\mu}(v_{ip}) \subseteq S$,

$$\mu_I^R \models \bigwedge_{c \in Slice_{\varphi,\mu}(v_{ip})} c$$

Taking into account that also

$$\mu_I^R \models \bigwedge_{v_i \in InputVars} (v_i = \mu[v_i])$$

we get from Theorem 2 that $\mu_I^R \models (v_{ip} = \mu[v_{ip}])$. That is, $\mu_I^R[v_{ip}] = \mu[v_{ip}]$.

Thus, we have shown that $\mu_I^R\restriction_{VIP(\varphi_P^{dem},\mu)} = \mu\restriction_{VIP(\varphi_P^{dem},\mu)}$. From proposition 2, $\mu\restriction_{VIP(\varphi_P^{dem},\mu)} \models \varphi_P^{dem}$. Therefore, $\mu_I^R\restriction_{VIP(\varphi_P^{dem},\mu)} \models \varphi_P^{dem}$. Since no changes are made in P_R to assumptions or assertions, $\varphi_P^{dem} = \varphi_R^{dem}$. So, $\mu_I^R\restriction_{VIP(\varphi_P^{dem},\mu)} \models \varphi_R^{dem}$, which means that $\mu_I^R \models \varphi_R^{dem}$. Put together with the fact that μ_I^R satisfies the formula **, this means that $\mu_I^R \models \varphi_R^{comp} \wedge \varphi_R^{dem}$. Thus, $\mu_I^R \models \varphi_R$.

Since μ_I^R represents the execution of P_R on I, the fact that $\mu_I^R \models \varphi_R$ means that P_R has a bug on I (recall that φ_R^{dem} encodes a disjunction of the negation of assertions, so the satisfiability of φ_R implies that an assertion is violated). Thus, this is a contradiction to P_R being an $\mathcal{S}_{arb}$-repair (i.e., an $\mathcal{S}_{arb}$-patched program that is correct for I).

lemma B.2.2. It is sufficient to show that every $\mathcal{S}_{arb}$-must location set is also an $\mathcal{S}_{mut}$-must location set. Let H_{arb} be an $\mathcal{S}_{arb}$-must location set. That is, H_{arb} contains at least one location from each minimal $\mathcal{S}_{arb}$-repairable set.

Let F_{mut} be a minimal $\mathcal{S}_{mut}$-repairable set. Since every $\mathcal{S}_{mut}$-patch is also an $\mathcal{S}_{arb}$-patch, F_{mut} is also an $\mathcal{S}_{arb}$-repairable set. Though F_{mut} itself is not necessarily a *minimal* $\mathcal{S}_{arb}$-repairable set, it must contain a non-empty subset F'_{arb} that is. Therefore, H_{arb} contains at least one location from F'_{arb}. Since $F'_{arb} \subseteq F_{mut}$, H_{arb} in fact contains at least one location from F_{mut}.

Thus, we have shown that for every minimal $\mathcal{S}_{mut}$-repairable set F_{mut}, H_{arb} contains at least one location from F_{mut}. This makes H_{arb} also an $\mathcal{S}_{mut}$-must location set. ■

Bibliography

[AB18] Fatmah Y Assiri and James M Bieman. Mut-apr: Mutation-based automated program repair research tool. In *Future of Information and Communication Conference*, pages 256–270. Springer, 2018.

[ABS17] Paul C Attie, Kinan Dak Al Bab, and Mouhammad Sakr. Model and program repair via sat solving. *ACM Transactions on Embedded Computing Systems (TECS)*, 17(2):1–25, 2017.

[ABV16] John Adler, Ryan Berryhill, and Andreas Veneris. Revision debug with non-linear version history in regression verification. In *IEEE International Verification and Security Workshop (IVSW)*, pages 1–6. IEEE, 2016.

[AH90] Hiralal Agrawal and Joseph R Horgan. Dynamic program slicing. *ACM SIGPlan Notices*, 25(6):246–256, 1990.

[AZVG06] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. An evaluation of similarity coefficients for software fault localization. In *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*, pages 39–46. IEEE, 2006.

[BDF+12] Roderick Bloem, Rolf Drechsler, Görschwin Fey, Alexander Finder, Georg Hofferek, Robert Könighofer, Jaan Raik, Urmas Repinski, and André Sülflow. Forensic–an automatic debugging environment for c programs. In *Haifa Verification Conference*, pages 260–265. Springer, 2012.

[BLN+13] Dirk Beyer, Stefan Löwe, Evgeny Novikov, Andreas Stahlbauer, and Philipp Wendler. Precision reuse for efficient regression verification. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 389–399, 2013.

[BMD00] B Brandin, Robi Malik, and Petra Dietrich. Incremental system verification and synthesis of minimally restrictive behaviours. In *Proceedings of the 2000 American Control Conference. ACC (IEEE Cat. No. 00CH36334)*, volume 6, pages 4056–4061. IEEE, 2000.

[BOR13] Marcel Böhme, Bruno CDS Oliveira, and Abhik Roychoudhury. Partition-based regression verification. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 302–311. IEEE, 2013.

[BPRT13] John Backes, Suzette Person, Neha Rungta, and Oksana Tkachuk. Regression verification using impact summaries. In *International SPIN Workshop on Model Checking of Software*, pages 99–116. Springer, 2013.

[BUVHW15] Bernhard Beckert, Mattias Ulbrich, Birgit Vogel-Heuser, and Alexander Weigl. Regression verification for programmable logic controller software. In *International Conference on Formal Engineering Methods*, pages 234–251. Springer, 2015.

[BV16] Mohit Bangalore Venkatesh. A case study in non-functional regression verification. Master's thesis, Uppsala University, Department of Information Technology, 2016.

[CESW13] Jürgen Christ, Evren Ermis, Martin Schäf, and Thomas Wies. Flow-sensitive fault localization. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 189–208. Springer, 2013.

[CGS12] Sagar Chaki, Arie Gurfinkel, and Ofer Strichman. Regression verification for multi-threaded programs. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 119–135. Springer, 2012.

[CHM+19] Maria Christakis, Matthias Heizmann, Muhammad Numair Mansur, Christian Schilling, and Valentin Wüstholz. Semantic fault localization and suspiciousness ranking. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 226–243. Springer, 2019.

[CIM+11] Hana Chockler, Alexander Ivrii, Arie Matsliah, Shiri Moran, and Ziv Nevo. Incremental formal verification of hardware. In *2011 Formal Methods in Computer-Aided Design (FMCAD)*, pages 135–143. IEEE, 2011.

[CKL04] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ansi-c programs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176. Springer, 2004.

[CKMS15] Hana Chockler, Daniel Kroening, Leonardo Mariani, and Natasha Sharygina. *Validation of evolving software*. Springer, 2015.

[CKY03] Edmund Clarke, Daniel Kroening, and Karen Yorav. Behavioral consistency of c and verilog programs using bounded model checking. In *Proceedings 2003. Design Automation Conference (IEEE Cat. No. 03CH37451)*, pages 368–371. IEEE, 2003.

[CPMB07] Kai-hui Chang, David A Papa, Igor L Markov, and Valeria Bertacco. Invers: an incremental verification system with circuit similarity metrics and error visualization. In *8th International Symposium on Quality Electronic Design (ISQED'07)*, pages 487–494. IEEE, 2007.

[DB09] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: An appetizer. In *Formal Methods: Foundations and Applications*, pages 23–36. Springer, 2009.

[DDMA12] Isil Dillig, Thomas Dillig, Kenneth L McMillan, and Alex Aiken. Minimum satisfying assignments for smt. In *International Conference on Computer Aided Verification*, pages 394–409. Springer, 2012.

[DER05] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005.

[DHM+17] Daniel Dietsch, Matthias Heizmann, Betim Musa, Alexander Nutz, and Andreas Podelski. Craig vs. newton in software model checking. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 487–497, 2017.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[DSS16] Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. Qlose: Program repair with quantitative objectives. In *International Conference on Computer Aided Verification*, pages 383–401. Springer, 2016.

[DW10] Vidroha Debroy and W Eric Wong. Using mutation to automatically suggest fixes for faulty programs. In *2010 Third International Conference on Software Testing, Verification and Validation*, pages 65–74. IEEE, 2010.

[DW14] Vidroha Debroy and W Eric Wong. Combining mutation and fault localization for automated program debugging. *Journal of Systems and Software*, 90:45–60, 2014.

[DXLM14] Favio DeMarco, Jifeng Xuan, Daniel Le Berre, and Martin Monperrus. Automatic repair of buggy if conditions and missing preconditions with smt. In *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis*, pages 30–39. ACM, 2014.

[EDC10] W. Eric Wong, Vidroha Debroy, and Byoungju Choi. A family of code coverage-based heuristics for effective fault localization. *Journal of Systems and Software*, 83(2):188–208, 2010.

[ES04] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Theory and applications of satisfiability testing*, pages 502–518. Springer, 2004.

[ESW12] Evren Ermis, Martin Schaf, and Thomas Wies. Error invariants. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7436 LNCS:187–201, 2012.

[FGK+14] Dennis Felsing, Sarah Grebing, Vladimir Klebanov, Philipp Rümmer, and Mattias Ulbrich. Automating regression verification. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 349–360, 2014.

[FGS14] Grigory Fedyukovich, Arie Gurfinkel, and Natasha Sharygina. Incremental verification of compiler optimizations. In *NASA Formal Methods Symposium*, pages 300–306. Springer, 2014.

[FKP+12] Vojtěch Forejt, Marta Kwiatkowska, David Parker, Hongyang Qu, and Mateusz Ujma. Incremental runtime verification of probabilistic systems. In *International Conference on Runtime Verification*, pages 314–319. Springer, 2012.

[GPKS17] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. Deepfix: Fixing common c language errors by deep learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, pages 1345–1351, 2017.

[GZL15] Pei Gong, Ruilian Zhao, and Zheng Li. Faster mutation-based fault localization with a novel mutation execution strategy. In *International Conference on Software Testing, Verification and Validation Workshops, ICSTW*, pages 1–10. IEEE, 2015.

[HHP09] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Refinement of trace abstraction. In *International Static Analysis Symposium*, pages 69–85. Springer, 2009.

[HHP13] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Software model checking for people who love automata. In *International Conference on Computer Aided Verification*, pages 36–52. Springer, 2013.

[HLK+15] Shin Hong, Byeongcheol Lee, Taehoon Kwak, Yiru Jeon, Bongsuk Ko, Yunho Kim, and Moonzoo Kim. Mutation-based fault localization for real-world multilingual programs (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 464–475. IEEE, 2015.

[HMW16] Fei He, Shu Mao, and Bow-Yaw Yaw Wang. Learning-based assume-guarantee regression verification. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9779:310–328, 2016.

[HSNB+16] Andreas Holzer, Daniel Schwartz-Narbonne, Mitra Tabaei Befrouei, Georg Weissenbacher, and Thomas Wies. Error invariants for concurrent traces. In *International Symposium on Formal Methods*, pages 370–387. Springer, 2016.

[HW12] Birgit Hofer and Franz Wotawa. Spectrum enhanced dynamic slicing for better fault localization. In *ECAI*, volume 12, pages 420–425, 2012.

[HYFD+18] Matthias Heizmann, Chen Yu-Fang, Daniel Dietsch, Marius Greitschus, Jochen Hoenicke, Yong Li, Alexander Nutz, Betim Musa, Christian Schilling, Tanja Schindler, and Andreas Podelski. Automizer and the Search for Perfect Interpolants. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2018. To Appear.

[HZWK18a] Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. Sketchfix: A tool for automated program repair approach using lazy candidate generation. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 888–891, 2018.

[HZWK18b] Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. Towards practical program repair with on-demand candidate genera-

tion. In *Proceedings of the 40th international conference on software engineering*, pages 12–23, 2018.

[JCK13] Kenneth Johnson, Radu Calinescu, and Shinji Kikuchi. An incremental verification framework for component-based software systems. In *Proceedings of the 16th International ACM Sigsoft symposium on Component-based software engineering*, pages 33–42. ACM, 2013.

[JGB05] Barbara Jobstmann, Andreas Griesmayer, and Roderick Bloem. Program repair as a game. In *Computer Aided Verification*, pages 226–238. Springer, 2005.

[JHS01] James A Jones, Mary Jean Harrold, and John T Stasko. Visualization for fault localization. In *in Proceedings of ICSE 2001 Workshop on Software Visualization*. Citeseer, 2001.

[JM11] Manu Jose and Rupak Majumdar. Cause clue clauses: error localization using maximum satisfiability. *ACM SIGPLAN Notices*, 46(6):437–446, 2011.

[JXZ+18] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. Shaping program repair space with existing patches and similar code. In *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*, pages 298–309, 2018.

[KB11] Robert Könighofer and Roderick Bloem. Automated error localization and correction for imperative programs. In *Formal Methods in Computer-Aided Design (FMCAD), 2011*, pages 91–100. IEEE, 2011.

[KB13] Robert Könighofer and Roderick Bloem. Repair with on-the-fly program analysis. In *Hardware and Software: Verification and Testing*, pages 56–71. Springer, 2013.

[KKK15] Etienne Kneuss, Manos Koukoutos, and Viktor Kuncak. Deductive program repair. In *Computer Aided Verification*, pages 217–233. Springer, 2015.

[KL88] Bogdan Korel and Janusz Laski. Dynamic Program Slicing. *Information Processing Letters*, 29:155–163, 1988.

[KNSK13] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 802–811. IEEE Press, 2013.

[LBBO01] Yassine Lakhnech, Saddek Bensalem, Sergey Berezin, and Sam Owre. Incremental verification by abstraction. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 98–112. Springer, 2001.

[LDVFW12] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *34th International Conference on Software Engineering (ICSE)*, pages 3–13. IEEE, 2012.

[LGL+20] Yiling Lou, Ali Ghanbari, Xia Li, Lingming Zhang, Haotian Zhang, Dan Hao, and Lu Zhang. Can automated program repair refine fault localization? a unified debugging approach. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 75–87, 2020.

[LKB+19] Kui Liu, Anil Koyuncu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, and Yves Le Traon. You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems. In *2019 12th IEEE conference on software testing, validation and verification (ICST)*, pages 102–113. IEEE, 2019.

[LM12] Mark H Liffiton and Jordyn C Maglalang. A cardinality solver: more expressive constraints for free. In *Theory and Applications of Satisfiability Testing–SAT 2012*, pages 485–486. Springer, 2012.

[LN16] Si-Mohamed Lamraoui and Shin Nakajima. A Formula-based Approach for Automatic Fault Localization of Multi-fault Programs. *Journal of Information Processing*, 24:88–98, 2016.

[LNFW12] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. GenProg: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, 2012.

[LNH15] Si-Mohamed Lamraoui, Shin Nakajima, and Hiroshi Hosobe. Hardened flow-sensitive trace formula for fault localization. In *2015 20th International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 50–59. IEEE, 2015.

[LPMMS16] Mark H Liffiton, Alessandro Previti, Ammar Malik, and Joao Marques-Silva. Fast, flexible mus enumeration. *Constraints*, 21(2):223–250, 2016.

[LR15] Fan Long and Martin Rinard. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 166–178, 2015.

[LR16] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 298–312, 2016.

[LS08] Mark H Liffiton and Karem A Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *Journal of Automated Reasoning*, 40(1):1–33, 2008.

[LTLLG18] Xuan Bach D Le, Ferdian Thung, David Lo, and Claire Le Goues. Overfitting in semantics-based automated program repair. *Empirical Software Engineering*, 23(5):3007–3033, 2018.

[Mes92] Pedro Meseguer. Incremental verification of rule-based expert systems. In *Proceedings of the 10th European conference on Artificial intelligence*, pages 840–844. John Wiley & Sons, Inc., 1992.

[MKKY14] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. Ask the mutants: Mutating faulty programs for fault localization. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 153–162. IEEE, 2014.

[MM15] Matias Martinez and Martin Monperrus. Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering*, 20(1):176–205, 2015.

[MNN+18] Sergey Mechtaev, Manh-Dung Nguyen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. Semantic program repair using a reference implementation. In *Proceedings of the 40th International Conference on Software Engineering*, pages 129–139, 2018.

[MVP15] Djordje Maksimovic, Andreas Veneris, and Zissis Poulos. Clustering-based revision debug in regression verification. *Proceedings of the 33rd IEEE International Conference on Computer Design, ICCD 2015*, pages 32–37, 2015.

[MYR15] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Directfix: Looking for simple program repairs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 448–458. IEEE, 2015.

[MYR16] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th international conference on software engineering*, pages 691–701, 2016.

[NLR11] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. A model for spectra-based software diagnosis. *ACM Transactions on Software Engineering and Methodology*, 20(3), 2011.

[NQRC13] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. SemFix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 772–781. IEEE Press, 2013.

[NTC19] Thanh-Toan Nguyen, Quang-Trung Ta, and Wei-Ngan Chin. Automatic program repair using formal verification and expression templates. In *Verification, Model Checking, and Abstract Interpretation*, pages 70–91. Springer International Publishing, 2019.

[NWKF17] ThanhVu Nguyen, Westley Weimer, Deepak Kapur, and Stephanie Forrest. Connecting program synthesis and reachability: Automatic program repair using test-input generation. In Axel Legay and Tiziana Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 301–318, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.

[PFN+14] Yu Pei, Carlo A Furia, Martin Nordio, Yi Wei, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. *IEEE Transactions on Software Engineering*, 40(5):427–449, 2014.

[PT15] Mike Papadakis and Yves Le Traon. Metallaxis-FL: mutation-based fault localization. *Software Testing, Verification and Reliability*, Volume 21(Issue 3):195–214, 2015.

[QML13a] Yuhua Qi, Xiaoguang Mao, and Yan Lei. Efficient automated program repair through fault-recorded testing prioritization. In *2013 IEEE International Conference on Software Maintenance*, pages 180–189. IEEE, 2013.

[QML+13b] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. Does genetic programming work well on automated program repair? In *2013 Fifth International Conference on Computational and Information Sciences (ICCIS)*, pages 1875–1878. IEEE, 2013.

[QX08] Ju Qian and Baowen Xu. Scenario oriented program slicing. *Proceedings of the ACM Symposium on Applied Computing*, pages 748–7752, 2008.

[RDH18] Bat-Chen Rothenberg, Daniel Dietsch, and Matthias Heizmann. Incremental verification using trace abstraction. In *International Static Analysis Symposium*, pages 364–382. Springer, 2018.

[RG16] Bat-Chen Rothenberg and Orna Grumberg. Sound and complete mutation-based program repair. In *International Symposium on Formal Methods*, pages 593–611. Springer, 2016.

[RG20] Bat-Chen Rothenberg and Orna Grumberg. Must fault localization for program repair. In *International Conference on Computer Aided Verification*, pages 658–680. Springer, 2020.

[RHJ+12] Urmas Repinski, Hanno Hantson, Maksim Jenihhin, Jaan Raik, Raimund Ubar, Giuseppe Di Guglielmo, Graziano Pravadelli, and Franco Fummi. Combining dynamic slicing and mutation operators for ESL correction. In *2012 17th IEEE European Test Symposium (ETS)*, pages 1–6. IEEE, 2012.

[SBLGB15] Edward K Smith, Earl T Barr, Claire Le Goues, and Yuriy Brun. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 532–543, 2015.

[SDLLR15] Stelios Sidiroglou-Douskos, Eric Lahtinen, Fan Long, and Martin Rinard. Automatic error elimination by horizontal code transfer across multiple applications. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 43–54, 2015.

[SFS12] Ondrej Sery, Grigory Fedyukovich, and Natasha Sharygina. Incremental upgrade checking by means of interpolation-based function summaries. In *2012 Formal Methods in Computer-Aided Design (FMCAD)*, pages 114–121. IEEE, 2012.

[SG08] Ofer Strichman and Benny Godlin. Regression verification-a practical way to verify programs. *Verified Software: Theories, Tools, Experiments*, 4171 LNCS:496–501, 2008.

[SGZL18] Shuyao Sun, Junxia Guo, Ruilian Zhao, and Zheng Li. Search-based efficient automated program repair using mutation and fault localization. In *2018 IEEE 42nd Annual Computer Software and*

Applications Conference (COMPSAC), volume 1, pages 174–183. IEEE, 2018.

[SMA05] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. *ACM SIGSOFT Software Engineering Notes*, 30(5):263–272, 2005.

[SSNW13] Martin Schäf, Daniel Schwartz-Narbonne, and Thomas Wies. Explaining inconsistent code. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 521–531, 2013.

[SVBV16] Ofer Strichman, Maor Veitsman, Ofer Strichman B, and Maor Veitsman. Regression Verification for unbalanced recursive functions. In *21st International Symposium on Formal Methods (FM)*, pages 645–658. Springer, 2016.

[TGK17] Anna Trostanetski, Orna Grumberg, and Daniel Kroening. Modular demand-driven analysis of semantic difference for program versions. In *International Static Analysis Symposium*, pages 405–427. Springer, 2017.

[TYM+17] Shin Hwei Tan, Jooyong Yi, Sergey Mechtaev, Abhik Roychoudhury, et al. Codeflaws: a programming competition benchmark for evaluating automated program repair tools. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 180–182. IEEE, 2017.

[TYPR16] Shin Hwei Tan, Hiroaki Yoshida, Mukul R Prasad, and Abhik Roychoudhury. Anti-patterns in search-based program repair. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 727–738, 2016.

[VGD12] Willem Visser, Jaco Geldenhuys, and Matthew B Dwyer. Green: Reducing, Reusing and Recycling Constraints in Program Analysis. *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 58:1–58:11, 2012.

[VJ15] Christian Von Essen and Barbara Jobstmann. Program repair without regret. *Formal Methods in System Design*, 47(1):26–50, 2015.

[vTG18] Rijnard van Tonder and Claire Le Goues. Static automated program repair for heap properties. In *Proceedings of the 40th International Conference on Software Engineering*, pages 151–162, 2018.

[WCW+18] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. Context-aware patch generation for better automated program repair. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 1–11. IEEE, 2018.

[WDGL14] W. Eric Wong, Vidroha Debroy, Ruizhi Gao, and Yihao Li. The DStar method for effective software fault localization. *IEEE Transactions on Reliability*, 63(1):290–308, 2014.

[WFF13] Westley Weimer, Zachary P Fry, and Stephen Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *2013 IEEE/ACM 28th International Conference on Automated Software Engineering (ASE)*, pages 356–366. IEEE, 2013.

[WGL+16] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016.

[Wot10] Franz Wotawa. Fault localization based on dynamic slicing and hitting-set computation. *Proceedings - International Conference on Quality Software*, pages 161–170, 2010.

[WPF+10] Yi Wei, Yu Pei, Carlo A Furia, Lucas S Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 61–72. ACM, 2010.

[WPP+14] Yan Wang, Harish Patil, Cristiano Pereira, Gregory Lueck, Rajiv Gupta, and Iulian Neamtiu. DrDebug: Deterministic replay based cyclic debugging with dynamic slicing. *Proceedings of the 12th ACM/IEEE International Symposium on Code Generation and Optimization, CGO 2014*, pages 98–108, 2014.

[XMD+16] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering*, 43(1):34–55, 2016.

[XWY+17] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. Precise condition synthesis for program repair. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 416–426. IEEE, 2017.

[YMDM20] He Ye, Matias Martinez, Thomas Durieux, and Martin Monperrus. A comprehensive study of automatic program repair on the quixbugs benchmark. *Journal of Systems and Software*, page 110825, 2020.

[YQM17] Deheng Yang, Yuhua Qi, and Xiaoguang Mao. An empirical study on the usage of fault localization in automated program repair. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 504–508. IEEE, 2017.

[ZGG07a] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. A study of effectiveness of dynamic slicing in locating real faults. *Empirical Software Engineering*, 12(2):143–160, 2007.

[ZGG07b] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. Locating faulty code by multiple points slicing. *Software - Practice and Experience*, 39(7):661–699, 2007.

מעולם האימות הפורמלי וטכניקות לפתרון נוסחאות בוליאניות ונוסחאות בלוגיקה מסדר ראשון.

הפוקוס שלנו במחקר היה על שיטות תיקון באגים מבוססות חיפוש. בשיטות מבוססות חיפוש מחליטים מראש על אוסף שינויים סינטקטיים שמאפשרים לעשות לתוכנית, ואז בונים אוסף תוכניות המורכב מכל התוכניות הנוצרות מהתוכנית המקורית ע"י הפעלת שינויים מתוך האוסף הנ"ל. נהוג לקרוא לאוסף התוכניות שנוצר "מרחב החיפוש של התיקון" (או "מרחב החיפוש" בקצרה) ולכל אחת מהתוכניות בו "תוכנית משופצת".

מרבית השיטות מבוססות החיפוש פועלות בלולאת "צור ובדוק": בכל איטרציה, בשלב היצירה בוחרים תכנית משופצת כלשהי מתוך מרחב החיפוש, ובשלב הבדיקה בודקים האם היא נכונה או לא. המטרה המשותפת של כל האלגוריתמים שפיתחנו במחקר זה היא לאפשר לייעל את הסכמה הפשוטה הנ"ל. שניים מהאלגוריתמים המוצגים עושים זאת ע"י ייעול שלב היצירה, בכך שהם מוציאים מראש ממרחב החיפוש תוכניות משופצות שאינן רלוונטיות. אלגוריתם נוסף עושה זאת ע"י ייעול שלב הבדיקה, בכך שהוא מאפשר להשתמש בתוצאות ביניים של בדיקה של תוכנית משופצת מסוימת בשביל הבדיקה של התוכנית המשופצת הבאה בתור.

התזה בנויה כאסופת מאמרים, ואנו מציגים בה שלושה מאמרים שלנו שפורסמו בכנסים מובילים בתחום. המאמר הראשון מציג אלגוריתם לתיקון באגים באופן פורמלי שמחזיר את כל התיקונים האפשריים והוא גם נכון וגם שלם (כלומר, כל תוכנית מתוקנת היא בהכרח נכונה, וכל תוכנית נכונה בהכרח מוחזרת בשלב כלשהו). אולם, הגדרת הנכונות בו היא ביחס לחישובים באורך חסום ולא באורך כלשהו. המאמר השני עוסק באימות אינקרמנטלי של תוכניות, ומאפשר לייעל את שלב הבדיקה בלולאת "צור ובדוק". המאמר השלישי והאחרון משפר את אלגוריתם התיקון שהוצג במאמר הראשון ע"י שימוש בשיטה חדשה למציאת מיקום השגיאה.

תקציר

עבודת המחקר המוצגת בתזה זו עוסקת בתיקון אוטומטי של באגים (כלומר, שגיאות) בתוכניות מחשב, מתוך נקודת מבט פורמלית.

בזמנים המודרניים, מערכות תוכנה נמצאות בכל מקום: בטלפונים ניידים, טלוויזיות, מכוניות, מטוסים ועוד. מערכות שכאלו הן על פי רוב מסובכות לתכנון ולבדיקה. לכן, על־אף מאמצים רבים למניעת באגים שנעשים מצד המתכננים, הן בהרבה מקרים מגיעות לידי הלקוח כשהן עדיין מכילות באגים נסתרים רבים. כאשר באג נסתר שכזה מתגלה בזמן השימוש במערכת התוצאה היא נזק למשתמש שיכול לנוע בין סתם אי־שביעות רצון ועד לנזק ממשי למידע, רכוש, או במקרים קיצוניים אפילו חיי אדם. לכן, חברות המייצרות מערכות תוכנה משקיעות משאבים רבים לצורך גילוי ותיקון שגיאות.

ואולם, למרות שמאמצי החברות מצליחים להביא לגילוי באגים רבים, לא תמיד ישנם המשאבים המספקים כדי לתקן את כולם. בהרבה מקרים חברות נאלצות לתעדף ולהשאיר באגים מסוימים ללא תיקון. זאת משום שתיקון הבאגים הוא מלאכה קשה שנעשית בצורה ידנית ודורשת הכרה מעמיקה של הקוד ורמת מיומנות גבוהה. לכן, ישנו ביקוש גבוה לשיטות שהופכות את תהליך התיקון לאוטומטי, איפה וככל שניתן.

בשנים האחרונות המחקר בתחום זה ראה התקדמות משמעותית, עם פיתוחן של שיטות שהוכחו כשימושיות עבור תיקון שגיאות אמיתיות בתוכניות בקנה מידה גדול. אולם, רוב העבודות התמקדו בתיקון ביחס לטסטים. טסט הוא ניסוי שבו מריצים את התוכנית עבור קלט ספציפי, שעבורו ידועה התוצאה הרצויה של התוכנית; אם התוכנית אכן נתנה את התוצאה הרצויה הטסט נחשב כעובר, אחרת הוא נכשל. תיקון ביחס לטסטים פירושו תיקון של התוכנית ממצב שבו ישנו לפחות טסט אחד שנכשל למצב שבו כל הטסטים עוברים. הבעיה המרכזית עם תיקון באופן הנ"ל היא שהוא מבטיח התנהגות נכונה של התוכנית רק עבור קבוצת הקלטים שנבחרה עבור הטסטים.

בניגוד לכך, עבודה זו מתמקדת בתיקון תוכניות ביחס למפרט פורמלי. מפרט פורמלי הוא תיאור לוגי, מתמטי, של ההתנהגות הרצויה של התוכנית. תיקון ביחס למפרט פורמלי מבטיח כי התוכנית המתוקנת תקיים את המפרט תמיד, לכל קלט אפשרי. לכן, הוא מאפשר ביטחון גדול יותר בתכנית המתוקנת. מנגד, בעיית התיקון היא קשה יותר והאלגוריתמים צפויים להיות פחות יעילים. שוני נוסף של עבודה זו מעבודות אחרות הוא בשיטות בהן השתמשנו. הדגש בעבודה היה על שימוש בשיטות פורמליות אשר מאפשרות להוכיח נכונות של מפרטים פורמלים. השיטות העיקריות בהן השתמשנו הן טכניקות

המחקר בוצע בהנחייתה של פרופסור ארנה גרימברג, בפקולטה למדעי המחשב.

חלק מן התוצאות בחיבור זה פורסמו כמאמרים מאת המחבר ושותפיו למחקר בכנסים ובכתבי־עת במהלך תקופת מחקר הדוקטורט של המחבר, אשר גרסאותיהם העדכניות ביותר הינן:

Bat-Chen Rothenberg, Daniel Dietsch, and Matthias Heizmann. Incremental verification using trace abstraction. In *International Static Analysis Symposium*, pages 364–382. Springer, 2018.

Bat-Chen Rothenberg and Orna Grumberg. Sound and complete mutation-based program repair. In *International Symposium on Formal Methods*, pages 593–611. Springer, 2016.

Bat-Chen Rothenberg and Orna Grumberg. Poster: Program repair that learns from mistakes. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 218–219. IEEE, 2018.

Bat-Chen Rothenberg and Orna Grumberg. Must fault localization for program repair. In *International Conference on Computer Aided Verification*, pages 658–680. Springer, 2020.

תודות

ברצוני להודות בראש ובראשונה למנחה שלי, ארנה גרימברג, שלא יכולתי לבקש לעצמי מתנה יותר טובה ממנה למסע הזה שנקרא דוקטורט. תודה על שנתת לי יד חופשית לחקור מה שמעניין אותי, על שתמיד היית שם לסייע ולתת רעיונות טובים, ועל שלימדת אותי מקרוב שהדבר הכי חשוב זה להנות מהדרך.

תודה נוספת לכל מי שהיה שותף למחקר שלי. לכותבים השותפים שלי מתיאס ודניאל – תודה על שיתוף פעולה מועיל שלמדתי ממנו רבות. לבוחנים שלי בבחינת המועמדות ובבחינת התזה – תודה על ההערות המועילות, ההכוונה והדיונים המפרים. לשותפי למשרד ולחברי בפקולטה לאורך השנים – תודה על שהייתם לי אוזן קשבת בשעות הקשות, שתמיד עזרתם בכל שאלה או בקשה, ושנתתם לי להרגיש הכי בבית בעולם.

תודה ענקית להורי, אבי ונעמי, שתמיד האמינו בי ואף פעם לא דחקו בי להיות שום דבר מלבד מי שאני. תודה לחמותי איליין, לאחים שלי נדב ושאנה, ולכל המשפחה המורחבת על שאתם תמיד שם בשבילי ללא תנאים. אני אוהבת אתכם.

תודה מיוחדת לילדים שלי, ליאור וגלי, שמזכירים לי תמיד מה חשוב בחיים, ומכניסים אותי לפרופורציות. כל מה שאני עושה הוא בשבילכם.

ותודה אחרונה ומיוחדת במינה לאחד שלי, החבר הכי טוב שלי, והשותף שלי לחיים, בעלי דני. תודה שאתה מלווה אותי לאורך כל הדרך, בטוב וברע. לא הייתי יכולה לעשות את זה בלעדייך.

הכרת תודה מסורה לטכניון, למרכז המחקר לאבטחת סייבר ע"ש הירושי פוג'יוארה ולמערך הסייבר הלאומי על מימון מחקר זה.

תיקון אוטומטי של שגיאות בתוכנה בעזרת שיטות פורמליות

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר
דוקטור לפילוסופיה

בת–חן רוטנברג

הוגש לסנט הטכניון --- מכון טכנולוגי לישראל
חשוון התשפ"א חיפה אוקטובר 2020

תיקון אוטומטי של שגיאות בתוכנה
בעזרת שיטות פורמליות

בת–חן רוטנברג