UNIVERSITY OF SOUTHAMPTON

FACULTY OF PHYSICAL AND APPLIED SCIENCES

Electronics and Computer Science

Fast, Fully-Automated, Model-Based Fault Localisation and Repair with Test Suites as Specification

by

Geoff Michael Birch

Thesis for the degree of Doctor of Philosophy

December 2016

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF PHYSICAL AND APPLIED SCIENCES
Electronics and Computer Science

Doctor of Philosophy

FAST, FULLY-AUTOMATED, MODEL-BASED FAULT LOCALISATION AND REPAIR WITH TEST SUITES AS SPECIFICATION

by Geoff Michael Birch

Fault localisation, i.e. the identification of program locations that cause errors, takes significant effort and cost. We describe a fast model-based fault localisation algorithm which, given a test suite, uses symbolic execution methods to fully automatically identify a small subset of program locations where genuine program repairs exist. Our algorithm iterates over failing test cases and collects locations where an assignment change can repair exhibited faulty behaviour. Our main contribution is an improved search through the test suite, reducing the effort for the symbolic execution of the models and leading to speed-ups of more than two orders of magnitude over previously published work.

We implemented our algorithm for C programs, using the KLEE symbolic execution engine, and demonstrate its effectiveness on the Siemens TCAS variants. Its performance is in line with recent alternative model-based fault localisation techniques, but narrows the location set further without rejecting any genuine repair locations. We explore extending the low-quality repairs constructed by the localisation process to synthesise a high-quality repair.

We also show how our tool can be used in an educational context to improve self-guided learning and accelerate assessment. We apply our algorithm to a large selection of actual student coursework submissions, providing precise localisation within a sub-second response time. We show this using small test suites, already provided in the coursework management system, and on expanded test suites, demonstrating scaling. We also show that compliance with test suites does not predictably score a class of "almost correct" submissions, which our tool highlights. Finally, we show an extension to our tool which enables a selection of student submissions to be localised beyond programs that conform to a single-fault assumption.

Contents

List of Tables

List of Figures and Listings

Declaration of Authorship

I, Geoff Michael Birch , declare that the thesis entitled *Fast, Fully-Automated, Model-Based Fault Localisation and Repair with Test Suites as Specification* and the work presented in the thesis are both my own, and have been generated by me as the result of my own original research. I confirm that:

- this work was done wholly or mainly while in candidature for a research degree at this University;
- where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
- where I have consulted the published work of others, this is always clearly attributed;
- where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
- I have acknowledged all main sources of help;
- where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
- parts of this work have been published as:

[9] G. Birch, B. Fischer, M. Poppleton: Fast Model-Based Fault Localisation with Test Suites. In *Proc. 9th International Conference on Tests and Proofs*, TAP '15, LNCS **9154**, pp. 38–57. Springer, 2015.

[10] G. Birch, B. Fischer, M. Poppleton: Using Fast Model-Based Fault Localisation to Aid Students in Self-Guided Program Repair and to Improve Assessment. In *Proc. 21st Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '16, pp. 168–173. ACM, 2016.

Signed:.....

Date:.....

Acknowledgements

This academic research was funded by the Engineering and Physical Sciences Research Council, UK. Funding number 1239954.

With thanks to:

Michael Poppleton and Bernd Fischer for their guidance, co-authorship of our papers, and support throughout this PhD;

Ewan Tempero and Paul Denny at the University of Auckland for access to an anonymised copy of their student exercises and submissions to those exercises, from which we generated our student data set;

Jeremy Morse for the education and assistance getting up to speed with ESBMC;

Eli Bendersky, author of the PyCParser;

the Automated Software Testing Group at Beihang University for Hawk-Eye;

all contributors to the various downstream components and tools used in this work including KLEE, ESBMC, LaTeX/TeXworks, and Mint-Linux;

my friends and family for supporting me during the last four years;

and the Southampton University Feminist Society.

In loving memory,

to Professor Charles Percival Whittingham.

Terminology

Throughout the literature, some terms relating to debugging are not used consistently. In this glossary, we outline the definitions of potentially ambiguous terminology used in this thesis to ensure the correct meaning can be read upon first encountering a term.

A **fault** [82, 87] (also called a defect [68, 86, 99], error [63], coding error, or bug [2, 16]) in source code is a minimal fragment of code whose execution can generate an incorrect behaviour. This is for some input (which includes the environment of execution), against whatever specification exists to declare what is and is not correct behaviour for the program. Note that a fault is defined for a specific source code against a given specification. A fault is still a fault even if it is not exercised or does not cascade into error/failure during a test case.

Faults can be **repair**ed [34, 82] by substituting in a replacement block of code into the block reported as defective which returns the program to executing in a way that does not violate the specifications.

An **error** [84, 86] (sometimes also called fault [19] or infection [19]) in program execution or modelled/simulated execution is the consequence of executing a defective block of code and the resulting creation of an erroneous state for the program. This effect may not be visible and not all errors will be exposed by surfacing as a failure. An error is the result of a fault being exercised by an execution which is susceptible to that fault. Other authors use 'error' in place of 'fault' [63], 'failure' [41], or meaning the mental mistake before the fault is penned [87].

A **failure** [19, 82] in program execution or modelled/simulated execution is the surfacing of an error state by the observation of behaviour in violation of the program specification. It is therefore correct to say that a failure was experienced for a given test case due to a chain of erroneous states that originated with the execution of a fault that caused the error.

For example, *"The program failed by returned the wrong result due to an error caused by the fault at line 5 where integer Var was assigned the value of 6 and not 10."*

To illuminate the scale of the terminology issue in this area, we provide this quote from the 1044-2009 revision to the IEEE Standard Classification for Software Anomalies:

> The 1993 version of IEEE 1044 characterized the term "anomaly" as a synonym for error, fault, failure, incident, flaw, problem, gripe, glitch, defect, or bug, essentially deemphasizing any distinction among those words. [54, p. iv]

A **trace** is a sequence of state transitions which describe the execution of a program or the simulation of a program execution. Traces can come in various levels of granularity, from providing precise navigation of the fully enumerated states of the program up to only providing the general flow, logging at the function or module transition level the path taken.

A **path** through a **control flow graph** [38] (or **program flow**) is a possible traversal of a program. These are often graphed to show the connectivity of a program and how state transitions can flow during execution. We will use **path** to refer to the collection or set of traces which share the same control flow path through the program.

Debugging [2, 16] is the process through which faults are removed from a program. This process is composed of the discovery, localisation, and repair phases. Discovery is the identification of errors in the program flow or failures as they report themselves. Localisation is the tying of these errors to the causal faults. Repair is the correction of the underlying faults to provide a new version of the program with correct execution within the specifications.

A **test case** is the partial specification of a program or program fragment for a given input and with a postcondition (output). This required output can simply be the lack of raising of an error flag, or an assertion violation, but is usually specified in more detail with a desired output or a known incorrect output. When applied to a program fragment this is commonly referred to as a **unit test**. Unit testing is often applied to individual modules or even functions/methods before integration testing. Test cases can **pass** or **fail** [34, 36] to comply with the program specification, giving respectively the correct or incorrect result. The incorrect result may be abnormal termination or other failure (thus making the "result" incorrect while possibly still providing the expected output).

The **unit under test** (UUT), also known as the device under test (DUT) in the original context of electronic engineering, is the program or program fragment being tested for compliance with a specification.

A program **(formal) specification** [34] is a description of the correct execution range of a program (i.e. the set of all valid traces). This is often given by providing error states that become failure conditions, therefore defining the set by what is not in it. This specification must be written in such a way as to be violable. A violation of the specification is always a failure. The formal descriptor of the specification means that the description is mathematically precise; it can be interpreted by formal methods to provide a clear true/false result without human interpretation.

We use a **weak-strong scale** of specification to indicate the extent to which the program flow is constrained by this specification. Many formal specifications create (internal) pre- and post-conditions on blocks of program code (often at the function/method scale) which operate to quickly convert from an error to a failure with early detection of the specification failure. An example of this would be using a coded assertion to surface an incorrect result at a block transition, which promotes it to a failure, earlier than it would otherwise be seen without a tested post-condition. We refer to this as a **strong specification**.

We refer to programs that are only specified by the underlying language constraints and a set of test cases as **weakly specified**. This specification only restricts the program flow by providing a subset of all inputs and required limitation on the output of the program; either the weaker 'must not' which does not require an oracle (correct) output to compare or the stronger 'must be' which does provide a known correct output.

Most programming languages supply added internal specification, to a greater or lesser extent depending on the language design, where certain states can lead to program flow that is illegal under the language specification. An example of this is any undefined behaviour in the language specifications, such as dereferencing a null pointer in C.

Chapter 1

Introduction

Debugging is the process of generating (from surfacing of an error) or collecting a report of a failure, finding the fault location that caused the failure, and correcting that fault. Fault localisation, the identification of program locations that can cause erroneous state transitions which eventually lead to observed program failures, is a critical component of this debugging cycle. However, this process puts a significant time [82, 87] and specific expertise burden [2, 112, 115] on programmers or dedicated debuggers for software projects.

To compound this issue, debugging is not typically the focus of software engineering or computer science education, despite commonly falling under the purview of the professional programmers developing projects [2]. Automated tools to assist in the debugging cycle have the potential to significantly reduce this time cost and lower the expertise bar required to iterate on source code to remove faults. While automated fault localisation is not the panacea for all debugging and testing problems, the total scale of feasible savings from all forms of improvement are significant and even a small percentage of that being attained would be highly valuable.

> [T]he [2002 U.S.] national cost estimate of an inadequate infrastructure for software testing is $59.5 billion. The potential cost reduction from feasible infrastructure improvements is $22.2 billion. [109, §8, p. 8-1]

A variety of methods are used for automated debugging. Failure detection is automated by tools which exercise or analyse the unit under test to generate and collect new failure reports, for example the use of automated test case generators to find an input leading to an error trace. Fault localisation and repair broadly fall into the categories of static code analysis and dynamic analysis, the latter using program execution to probe a compiled program for actual execution output. Static code analysis models a program and reasons on the representation of the program and potential control flow based on

the specification as given by the programming language. This can therefore reason over a range of potential values, leaving an input unconstrained/non-deterministic and so symbolic rather than concrete.

A hybrid process of dynamic analysis with static reasoning has been developed and named concolic testing [15, 101], which combines concrete execution with symbolic evaluation. This accelerates reasoning on a symbolic model by allowing the execution of any concrete statements and expressions, thus narrowing the state space. This allows the testing of a subset of all potential flows based on the program evaluation in the current environment of execution. This can be thought of as a maximally optimising model checker or symbolic analyser with the added limitation of importing the specific environment on which the concolic testing is done, rather than working within the full breadth of possible environments possible under pure modelling of the full language specifications.

Localisation by examining counter-example traces, test cases, or other output from static and dynamic analysis tools is an active area of research [16, 19, 39, 41, 61, 93, 96, 98].

Spectrum-based fault localisation techniques, compared in [79, 121, 123], are dynamic analysis methods that operate by examining the traces of passing and failing test cases. They assume that faults are more likely to be exercised by failing test cases and less likely to be exercised by passing test cases. The statements in a program can then be ranked based on different weighting techniques. The best-known example of this technique is the Tarantula tool [60, 98]. This can be used with the GCov code coverage tool in GCC to decorate the unit under test and track which statements are executed for each input from a test suite. These tools are popular due to their fast, predictable processing times. A GCov-decorated program will execute in approximately the same time as the undecorated program when compiled with GCC and executed on the test platform. State of the art spectrum-based fault localisation methods have recently been compared using different theoretical frameworks [79, 123]. Several methods have been identified as optimal under these frameworks but their real-world narrowing performance on small benchmark localisation test programs lags significantly behind state of the art model-based localisation.

Delta debugging [126] is a family of approaches that involve splitting up a large set of changes to find the minimal set that "flip" the program behaviour from correctly functioning to exhibiting a failure. This has variously been used to minimise inputs and traces but was later extended to source code exploration. The principle applied here [19] is to look at passing and failing traces and minimise the differences between them to isolate the failing components. This is reminiscent of a binary search, looking for interesting subset behaviour to narrow down variables that correlate with failure. However, this does require the existence of at least one passing trace and the localisation

performance of delta debugging on small benchmark localisation test programs [19] is worse than Tarantula's [60] results.

Model-based fault localisation [92] (sometimes also called model-based debugging [20]) is the application of model-based diagnosis methods [25] to programs. It involves three main steps: (*i*) the construction of a logical model from the original program; (*ii*) the symbolic analysis of this model; and (*iii*) mapping any faults found in the model back to program locations. One popular approach to model-based fault localisation is to transform the program so that a symbolic program verification tool can be reused for all three steps.

For example, Griesmayer et al. describe a method [39] in which the model (in the form of a logical satisfiability problem) is derived by running the CBMC model checker over a transformed program, and analysed by means of the model checker's integrated SAT solver. The transformation "inverts" the program's specification (cf. section 3.1), producing failures where the original program would complete successfully and blocking paths where the original program would fail, and replaces each original assignment by a conditional assignment with either the original value or an unconstrained symbolic value, depending on the value of a toggle variable. The actual localisation can then be reduced to extracting the possible values of the toggle variable from the satisfying assignments that the SAT solver returns.

However, the technique initially described by Griesmayer et al. requires detailed specifications to achieve acceptable precision—the weaker the specification, the more program locations are flagged as potential faults. Such detailed specifications rarely exist in practice. What *does* commonly exist are extensive unit test suites, in particular in the context of modern test-driven design approaches. Griesmayer et al. have shown that this technique can be extended to work with (failing) test cases, but the published results [39] are prohibitively slow (typically executing in hundreds to thousands of seconds per each small benchmark program). Griesmayer et al. have improved the original implementation [40] but still only achieve times typically in the hundreds of seconds.

Könighofer and Bloem have developed the FoREnSiC system [63, 64] (a Formal Repair Environment for Simple C) which includes a similar method of localisation. Their published results only cover a few variants of the benchmark localisation test programs used but where they have publishable results, localisation times are still measured at over a hundred seconds. Könighofer et al. report improved times [65] (around 37 seconds per program) but only after manually annotating all functions with contracts, so they are no longer working purely from the benckmark's test suites.

Jose and Majumdar convert an input C program to a maximum Boolean satisfiability problem which is analysed with the MAX-SAT solver [61]. However, because this returns the complement of the maximal subset of clauses that can be true for each single test case, their approach can omit genuine repair locations. This technique therefore

relies on summing the results of the different test cases, providing a ranking of most to least commonly flagged locations. This is the same trade-off used by all heuristic-based fault localisation techniques, like the spectrum-based techniques, and so inherits their strengths *and* weaknesses. Approximating model-based fault localisation techniques for test suites can run faster but can also miss a true fault location when evaluating a test case.

1.1 Research Questions

Our research seeks to answer how model-based fault localisation can be used without detailed program specifications. Can precise, test-driven, model-based fault localisation be executed with the low run-time costs associated with spectrum-based techniques? We aim to develop tools that will localise efficiently on real-world faulty programs as well as common small localisation benchmark programs. Specifically, we look at education as an area with a specific need for high quality automated localisation to aid novice debuggers who are learning to program. In this thesis, we therefore investigate the following research questions:

RQ1 Does model-based fault localisation improve over the more common spectrum-based methods when using large test suites?

RQ2 Does model-based fault localisation accurately pin-point fault locations (in absolute and relative terms) on real programs?

RQ3 Do model-based fault localisation techniques expand to synthesise high-quality repairs?

RQ4 Does model-based fault localisation provide results beyond a single-fault assumption?

RQ5 Does model-based fault localisation offer benefits to the pedagogical domain?

We start with the technique presented by Griesmayer et al. as an initial framework for model-based localisation which is potentially expandable to model-based, high-quality repair synthesis.

1.2 Main Contributions

The run-time performance in the area of thousands of seconds for small benchmark programs which have been published by Griesmayer et al. is due to a naïve search algorithm, which simply iterates over all test cases and runs an unoptimized “full width”

search over all possible locations for each test case (cf. section 3.1 for more details). However, the more locations the solver needs to explore, the longer the analysis of each test case takes. Moreover, the algorithm contains no optimizations to deal with test cases that generate intractable problems for the solver. We reimplemented this technique using modern hardware and a recent model checker on top of a state of the art SMT solver in order to evaluate the current performance issues of this design, as described, with results and analysis provided in chapter 4.

We have developed, implemented, and evaluated a novel approach that addresses these shortcomings, which leads to typical speed-ups of more than two orders of magnitude over Griesmayer's results, and yields a performance in line with current approximation techniques such as Jose and Majumdar [61]. Our approach still iterates over the failing test cases and runs a Griesmayer-style localisation task for each individual test case, but it maintains a *whitelist* of still viable fault locations which is narrowed down as the localisation tasks return. Our algorithm manages individual localisation tasks via a task pool to take advantage of the underlying multi-core hardware of modern systems, and dispatches the tasks in batches to percolate improvements in whitelist narrowing generationally. Tasks that fail to complete in a dynamically adjusted time are terminated and, if the whitelist is smaller, resubmitted at the tail of the iteration, where they may have become tractable due to the reduced search space leading to a smaller SMT problem for the solver. This early termination/resubmission increases the speed with which we can process larger test suites, without harming the localisation performance, as they typically have more redundant (for localisation purposes) test cases. It also prevents a loss of completeness in the results that model-based approaches can provide, within the limits of the symbolic analyser's accuracy.

Our approach is compatible with the modern test-driven design approach of specification by unit test suites. It inherits the typical strength of dynamic analysis, in that no prior knowledge of the program under test is required beyond test cases being flagged as passing or failing. We implemented our algorithm in a tool with the use of ESBMC [23] or KLEE [15] symbolic analysers and with PyCParser [7] handling the program transformations to encode the specification and other model constraints. The algorithm inherits the underlying behaviour, in respect of library and method calls, unrolling loops, and so on, of the symbolic analyser used. We initially demonstrate [9] this algorithm on the defective TCAS program variants from the Siemens [53] repository, a set of common localisation benchmark programs. This is presented in section 4.3. Our work on the extension of this technique to automated repair of the located assignment(s) provided negative results. We provide an analysis of why, for this technique, concrete execution is a more efficient approach to this search problem of narrowing to a high-quality repair form, with no level of further optimisations possible to provide tractability for non-trivial scale programs, in section 4.5.

We have then applied this tool to a corpus of student programs [10] and presented our analysis in chapter 5. Our experiments with real-world student code from an existing submissions database are encouraging. Our tool can, for some programs, pin-point fault locations and so should assist students in surmounting the final hurdle to completing writing of source code that fully complies with a test suite without requiring advanced debugging skills. The short list of locations where an assignment repair is possible, provided by our tool, can aid students in pinpointing improvement sites before they resubmit their code. The run-times of our tool are competitive with common, fast spectrum-based localisation techniques while providing more accurate fault localisation. This is of value to novice programmers, who stop debugging when provided with large lists of potential repair sites. Sub-second processing times and use of test suites for specification allows our tool to work with existing coursework submission systems and workflows, both for self-training and grading, in principle even at the scale required by Massive Open Online Courses (MOOCs).

Our tool can also provide instructors with automated assistance detecting some submissions which would provide perfect compliance after a single assignment edit (e.g. only fail due to a small mistake such as failing to increment a counter before returning it as the final value) but potentially fail most or all of a test suite. This will allow test-suite-based automated graders to identify submissions whose quality is being underestimated by this functional assessment method. We show that real-world student submissions which are a single assignment edit away from full compliance with a test suite are not provided with a fair mark by grading against that test suite. Repair location information provided by our tool can also help instructors who are manually grading submissions to more quickly find and understand the mistakes made by students.

Our contribution includes:

- Development of the automated application of test case specifications to C programs for modelling. [RQ5]
- Evaluation of Griesmayer's design on modern hardware with a current SMT solver. [RQ1]
- Development of a fast search through test suites to accelerate test-driven model-based fault localisation. [RQ1, RQ2]
- Demonstration of several orders of magnitude speed improvements from this tool. [RQ1]
- Investigation of this tool applied to a database of actual student programs using existing test suites. [RQ2, RQ5]
- Demonstration on a database of student programs using expanded test suites. [RQ1, RQ2, RQ5]

- Investigation of the database of actual student programs, expanded to dual-fault programs. [RQ4, RQ5]
- Demonstration that functional correctness over test suites does not predict near correct (single-fault) student programs. [RQ5]
- Investigation into an expansion of the model into repair synthesis, showing negative results vs the concrete execution implementation of the same search space, even with fast search optimisations. [RQ3]

1.3 Thesis Structure

In chapter 2 we expand on the overview of automated debugging outlined earlier in this chapter and provide an analysis of similar techniques in a review of related work, including an overview of the educational domain in which many of our results lie. We detail the full design of our tools in chapter 3, including the algorithms developed in the design and implementation of a search space acceleration process for use with an inverted model fault localisation technique.

In chapter 4 we present the results of our tool against our reimplementation of the naïve search algorithm to provide a performance improvement on a set of common localisation benchmark programs. We extend this to include a comparison against a state of the art approximate model-based localisation technique and provide a tentative comparison to several spectrum-based localisation techniques [RQ1]. This chapter ends with section 4.5, where the negative results from the repair extension of this technique are presented and analysed [RQ3].

In chapter 5 we provide the results of our tool directly compared to a spectrum-based localisation tool on a corpus of actual student programs [RQ1, RQ2, RQ5]. Here, we also present an extension of the search design which enables the capture of localisation data from defective programs which contain more than one fault without paying the cost of widening the search to an n-fault assumption at the symbolic analysis level [RQ4]. This demonstrates preliminary practical results beyond the strict assumptions that enables our search space optimisation.

We discuss the totality of these results in chapter 6, including extended discussion towards our conclusions. We conclude with discussion of the limitations of our approach and any threats to the validity of our conclusions. Finally, we summarise our conclusions in chapter 7 and provide potential avenues for future research.

Chapter 2

Background and Related Work

In this chapter we outline the historical context and main concepts developed upon in this thesis. We review the state of the art literature around related research questions in section 2.5 to section 2.8.

2.1 Program Specification

Program annotation, the insertion of comments into the flow of sequential programs, is as old as the writing of sequential programs. Initially these annotated comments included a class of properties which could be manually tested for truth at the point of notation in the program flow. By the late 1960s these properties, called specifications, were explored with techniques proposed for proving the consistency of the annotations [35, 49]. These axiomatic proposals were refined and, by the mid 1970s, specific techniques were being developed to formally express these properties [71], facilitating the proof of these (now formal) specifications.

> The most important property of a program is whether it accomplishes the intentions of its user. [... T]hese intentions can be described rigorously by making assertions about the values of variables at the end (or at intermediate points) of the execution of the program[.] [49, p. 579]

A program's formal specifications are expressions in a formal language which define a collection of properties a system should satisfy. These can be written at various levels of abstraction. They are mathematical descriptions of a(n informal) concept which establish the correctness of a class of programs which provably are equivalent to those specifications. This must allow automatic analysis of the specifications; requiring human interpretation ("the steps in the reasoning place too much dependence on human ingenuity and intuition" [71, p. 72]) defines an informal specification.

Violation of a specification is classified as a program failure. A failure will be experienced for a given specification due to a chain of erroneous states that originated with the execution of a fault that caused the error. Without a formal specification, no automatic reasoning on the correctness of programs is possible. However, once a program has been shown to fail an informal specification, when a human (oracle) decides that the program has failed to execute correctly at least once, this failure defines the start of a formal specification. The input used and incorrect execution generated can be formally specified.

We use a weak-strong scale of specification to indicate how detailed the program flow is constrained by a formal specification. Many formal specifications create (internal) pre- and post-conditions on blocks of program executions (often at the function/method scale) which operate to restrict the state space by quickly converting from an error to a failure. An example of this would be by surfacing an error at a block transition, which promotes it to a failure, earlier than it would otherwise be seen without a tested post-condition. We refer to this as a strong specification. We refer to programs that are only specified by the underlying language constraints and a set of test cases as weakly specified. This specification only restricts the program flow by providing a subset of all inputs and required limitation on the output of the program, either the weaker 'must not' which does not require an oracle (correct) output to compare or the stronger 'must be' which does provide an oracle (known correct) output.

The environment under which the program executes also contributes to the total specification. Calls that pass to the operating system or a shared library can have platform-dependent behaviours which are part of the correct execution of the program and must be accounted for. Some programs are specified only for execution on specific environments/platforms. Additionally, programming languages define the translation from source code to potential execution. However, the range of programs that are compilable often extends beyond this strict translation range. Once the C programming language was standardised in 1989/90 by ANSI, it became a well specified family (C89, C99, C11) of languages with a standard library. The various mainstream C compilers (MS, GNU, LLVM) generate code for a significantly wider set of input programs than required by the specification. This includes supporting legacy behaviour for pre-ANSI specification code and compiler-specific extensions and decoration to the language, defining platform and compiler dependent behaviour. Typical coding can also lead to the creation of programs that, under some conditions, fall outside the language specifications or specifically violate those specifications. An example of this is any undefined behaviour in the language specifications, such as dereferencing a null pointer in C. This provides weakly specified programs with another layer of specification, even if they have not been written to include any code that detects failures of the program.

2.2 Debugging

When a program violates one or more properties which make up its formal specifications, it must be brought back into conformance with those specifications. Debugging is the process of generating or collecting a report of a failure, finding the fault location that caused the failure, and correcting that fault. Reports of failures typically originate from software testing, discussed in section 2.4. Fault localisation, the identification of program locations that can cause erroneous state transitions which eventually lead to observed program failures, is a critical component of this debugging cycle. However, this process puts a significant time [82, 87] and specific expertise burden [2, 112, 115] on programmers or dedicated debuggers for software projects. This is explored further in section 2.5 and section 3.1. Localisation must occur before the final major cost [6, 43, 68, 109] of debugging, creating the repair to the fault that a failure exposes. We discuss automated approaches to this in section 2.7.

To compound this issue, debugging is not typically the focus of software engineering or computer science education, despite commonly falling under the purview of the professional programmers developing projects [2]. Automated tools to assist in the debugging cycle have the potential to significantly reduce this time cost and lower the expertise bar required to iterate on source code to remove faults. While automated fault localisation is not the panacea for all debugging and testing problems, the total scale of feasible savings from all forms of improvement are significant and even a small percentage of that being attained would be highly valuable.

> [T]he [2002 U.S.] national cost estimate of an inadequate infrastructure for software testing is $59.5 billion. The potential cost reduction from feasible infrastructure improvements is $22.2 billion. [109, §8, p. 8-1]

A variety of methods are used for automated debugging. Failure detection is automated by tools which exercise or analyse the unit under test to generate and collect new failure reports, for example the use of automated test case generators (cf. subsection 2.4.2) to find an input leading to an error trace. Fault localisation and repair broadly fall into the categories of static code analysis (cf. section 2.3) and dynamic analysis, the latter using program execution to probe a compiled program for actual execution output. Static code analysis models a program and reasons on the representation of the program and potential control flow based on the specification as given by the programming language. This can therefore reason over a range of potential values, leaving an input unconstrained/non-deterministic and so symbolic rather than concrete.

A hybrid process of dynamic analysis with static reasoning has been developed and named concolic testing [15, 101], which combines concrete execution with symbolic evaluation (cf. subsection 2.3.3). This accelerates reasoning on a symbolic model by allowing

the execution of any concrete statements and narrowing of the state space, testing a subset of potential control flows based on the executed program evaluation. This can be thought of as a maximally optimising model checker/symbolic analyser with the added limitation of importing the specific environment on which the concolic testing is done, rather than working within the full breadth of possible environments possible under pure modelling.

2.3 Model-Based Diagnosis

Model Based Diagnosis compares the specified output of a system to the actual execution to detect failures. The difference is used to assist in diagnosing potential faults and other analysis tasks. The process was originally developed with the aim of providing reasoning for the debugging of hardware faults in components of an electronic engineering system [25, 92].

This technique uses a compositional construction of a system from defined components and connection protocols to model a complete system.[1] This can provide an analysis of the measured outputs (possibly from tap points to measure intermediate results between component boundaries) of an actual system. From this the tool can compare results to determine components failing to comply with their model, and so identify candidates for repair or replacement [26]. When this family of techniques was brought over to the software testing field tap points became program instrumentation, which provides the equivalent functionality.

More advanced implementations of this technique can also indicate future tap points if the series of input/output pairs are insufficient to accurately diagnose a defective component [26]. This can be seen as the equivalent of an analysis tool that uses the program control flow to develop variables to be watched to explain failures.

2.3.1 Model Checking

Model checking assesses a specified model of a system to ensure the model under simulation does not violate the given specification. Any violation leads to the generation of a counter-example showing the path taken to the specification failure. These techniques can also be applied to software. Model checking for software systems is a form of static code analysis that creates a finite state interpretation of the software system as temporal logic to verify its correctness. A more extensive overview of the topic is provided [77] with later performance comparisons [76]. This provides a high level overview of the

[1] A more extensive overview of the topic can be found at `http://www.dekleer.org/Publications/mbd-figure.pdf` [26].

competing model checker types and their differing abilities to reason over the models extracted from source code translation.

For bounded model checking, this is done by translating an unfolded version of a computer program to a propositional satisfiability problem which encodes the constraints on program flow. This checks for behaviour by demanding that an assertion on the state (an inverted error condition) can be reached and violated. A Boolean satisfiability (SAT) solver is called on this formula which either returns the variables used to satisfy it or the verdict that is it unsatisfiable. This symbolic analysis may be accelerated with the use of built-in theories to compactly reason over the logic expression [81]. The core operation of calling the solver inside the model checker is NP-complete [21]. Moreover, this type of SAT problem has been shown [17] to not provide predictable tractability. Hence, the time it takes to process this task is not predictable with any degree of certainty.

A bounded model checker puts a limit on the unfolding of some program blocks to allow the exploration of a limited subset of a program's paths to try and find a counter-example without generating all the possible paths through a program that contains structures like an unwound loop or inlined recursive call. These structures may not have a discoverable limit to the unwinds, which would cause an unconstrained number of paths through the program with at least one for each unwinding length (iterations before the loop broke). When the loop is unrolled X times and tailed with a failing assertion then the rewritten program is now specified as all paths that iterate that loop X or less times. If the assertion is triggered then the bound can be increased to check paths above the originally chosen bound.

Due to bound limits and any approximation used to optimise the search or preprocess the formula to make it more likely to be tractable, this approach is not perfect. Translation from source code language is often not a perfect representation of the complete language and library specifications. So failure to find a counter-example is not proof that no counter-example exists.

As well as encoding a specification as an assertion that is tested and whose violation leads to the generation of a counter-example, model checking also uses assumptions which encode that any paths through that statement must ensure the truth of the statement assumed. This provides a constraint that does not raise a violation and so can be used to narrow the potential traces possible.

A Boolean satisfiability (SAT) solver requires the problem passed to it to be encoded entirely in Boolean logic. Reasoning over a software program which has been entirely converted to Boolean form can quickly provide intractable results due to the size of the problem and the non-linear processing time explosion for solving of large arbitrary Boolean expressions. The application of satisfiability modulo theories[2] (SMT) to this

[2]See de Moura and Bjørner [28] for a more detailed overview.

problem [5, 22] allows the use of more complex theories to reason over the logic expression, avoiding the need to deconstruct every element into a pure Boolean expression. This maps well onto type systems and symbol constraints with mathematical operations that can be performed by a programming language, using theories of number systems and data structures to reason on the logic expression. SMT interoperability is assisted by the common model checker problem language of LibSMT which has been under development as a common input and output standard since 2003.

2.3.2 Counter-Example Analysis

The first stage in applying the results of model checking to static code analysis is reasoning based on the counter-examples returned from a specification violation. A full trace of the program execution up to the point of failure will provide more information than many analysis tasks require and will overwhelm human debuggers.

Jin et al. provides a method [59] for paring down counter-examples to relevant elements which contribute to finding the failing expression that generated the counter-example. This level of refinement to the provided counter-example is now common for integration into model checker output systems. The level of value add provided by dedicated counter-example analysis tools is now focussed on providing visualisations to human debuggers [18], with automated tools moving to further analysis of the counter-examples beyond simplification.

Related to the delta debugging localisation discussed in subsection 2.5.2, Groce and Visser demonstrate [42] an application of that technique using several counter-examples. These are compared to instrumented passing runs to narrow the program flow differences between passing and failing traces. This is used to infer the areas of the program where a fault exists exposed by the key difference in states. An expanded version of this idea by Jalbert and Sen uses several traces with multi-threaded programs to find context switches which may be involved in errors [57]. This provides meaning to a set of counter-examples that would not necessarily identify a correlation between problematic context switches and failures.

2.3.3 Concolic Testing

A hybrid process of dynamic analysis with static reasoning has been developed and named concolic testing in CUTE [101]. The EXE tool, later renamed to KLEE, was being developed at approximately the same time as CUTE and was first reported in the ACM Computer and Communications Security Conference (CCS) in 2006 [14] with an expanded paper in 2008 [15]. This combines concrete execution with symbolic evaluation. At all points in the evaluation of the program, all expressions and sub-expressions that

can be executed concretely are evaluated and their results injected before continuing. This can be thought of as aiming for a maximally optimised pre-processing of a symbolic program to reduce all concrete components to their simplest form under the specification of the current execution environment.

This differs from an optimising symbolic evaluation by taking the environment of the program as known and so executable and the returned value to be the correct result. The simplest example of this difference is that a symbolic evaluation of a program that waits for standard input would assign an unconstrained symbolic value to the input, a concolic execution takes the concrete value provided as the actual input and so constrains the program flow to that actual input passed to the program under test. This would also clearly provide less program exploration than the limits of the language specifications for C because concrete execution occurs on, for example, fixed width integers set by the platform rather than all potential integer widths that comply with the language specifications. This introduces an area of divergence between execution platforms and favours program comprehension where the debugging platform is the same as the execution platform.

This also accelerates reasoning on a program modelled by inputs as symbols by allowing the concrete execution of a chosen path to drive the program flow down which the symbolic values are tested for specification compliance, thus testing a subset of potential control flows based on the limits of the inputs. Each time the symbolic inputs enable more than one path to be taken then the model checker problem is split to allow each path to be evaluated independently. Program elements carry their constraints from the earlier statements in the control flow with them based on the range of values an input can take.

This can also be thought of as a symbolic computation driven by a concrete trace. The concrete execution of the program defines a flow from which the symbolic evaluation can generate a solver problem. By negating branch conditions, the symbolic analysis can target symbolic inputs which exercise other paths and so efficiently generate concrete inputs which have a high branch coverage. These concolic tools were designed for test case generation by exploring the different paths reachable in the program and generating inputs based on the generated constraints matching each path. When compared with random testing, using random inputs to search for a good coverage of test cases, these methods provide a directed approach.

2.4 Testing

2.4.1 Test-Oriented Development

Current development paradigms broadly referred to as agile software development are a family of different management and software construction methods that provide rapid cycle development processes with continuous integration of new code, shipping products early and often, with feedback from the customer. Such methods typically include specification verification via compliance with various unit tests for modules, with the tests being developed ahead of the module construction. These test cases can be augmented after development with more expanded white box tests whose inputs are automatically generated via code path exploration to give good coverage of the module. In order to properly test the modules, the creation of a test harness is sometimes required to simulate the rest of the program and provide accurate input and collect outputs at interface points.

The rise of such agile development practices [119] and collaboration tools has provided an incentive to generate more test-based specifications testable at all stages of the program's development. Rather than having a program constructed over a long cycle before going through testing, the software is constantly at a state of being flagged as operational and ready for testing. These methods are, to a greater or lesser extent, replacing the old processes in many development studios.

> [In] Dr. Dobbs Global Developer Technographics Survey, Q3 2009, 35% of respondents stated that Agile most closely reflects their development process, with the number increasing to 45% if you expand what you include in Agile's definition. Both waterfall and iterative approaches are giving ground to much lighter, delivery-focused methods based on the principles the Agile Manifesto describes. [119, §1, p. 3]

The use of static code analysis tools via integration into common IDEs and as augmentations to common C compilers to improve warning and error feedback has pushed forward analysis-friendly coding with mantras that support treating compiler warnings as errors and coding to pedantic compiler warning levels to allow better code comprehension. While this has not pushed developers towards strong program specification, it has provided much needed hooks for automated localisation and repair tools which can provide added value and so reinforce this loop of valuing strict, analyser-friendly coding standards.

2.4.2 Test Case Generation

When a code base lacks unit tests, and so is possibly too weakly specified for reasoning, then an accelerated way of generating a test suite is to request a minimal set of test cases with good coverage criteria of the explored program paths. With some effort for an oracle (to calculate the required output) for the generated inputs, this can cheaply create a weak specification to work with.

Modern approaches [15, 74, 101] use concolic methods (cf. subsection 2.3.3) that interleave a concrete exploration for any deterministic elements and external method calls with symbolic checking. This symbolic evaluation provides a trace back to the inputs that are likely to provide good coverage by taking a different path at a conditional statement. Tools such as KLEE provide path exploration and statement coverage strategies to provide various coverage criteria from which to generate new counter-examples [15].

Fraser et al. have provided an overview [37] of using model checkers for the generation of test suites. Riener et al. use a bounded model checker to generate test cases that are capable of exercising the seeded faults added by mutation testing [94]. The original program and a mutant are both encoded with assumptions that the inputs are the same and an assertions that the output is not. This model generates text cases that demonstrate that the mutant exhibits differing behaviour from the program it is modified from. This also test demonstrates the capability of fault detection using only automatically generated test cases and indicates how these topics are highly linked in the literature with knowledge and exploitation of one technique used to validate the other.

Majumdar and Sen combine mixed random testing and concolic execution of C programs [74] to generate test cases with both deep and wide state space exploration.

Hildebrandt and Zeller applied the delta debugging technique discussed in subsection 2.5.2 to the input to a program (treating it as a simple string) to find an input that exercises a fault and generates a failure [48]. This is a form a test case refinement, which is sometimes required to augment a generator that does not generate a minimal set of test cases during the generation process.

Another technique developed to more efficiently deal with very large test suites with high pass rates is test case prioritisation. This was originally designed to accelerate regression testing [33], which is an incremental testing process that aims to find failures in a new version of a program. It was later applied to spectrum-based fault localisation [125], discussed in subsection 2.5.1, to provide an order to the test suite which maximises the fault localisation results when only exploring a partial set of test cases.

Use of these methods ensures that programs can be efficiently provided with a weak specification via test suite which is compact and offers a high coverage of the program statements.

2.4.3 Mutation Testing

A code mutation is a modification that changes a small piece of a program, taking its name from genetic mutations which are small changes in DNA which result in different genetic expression in the organism. The intent of these mutations is to simulate the result of a programmer mistake leading to a small fault in a program. As these mutations are inserted into otherwise correct programs, they allow the exploration of programs that are known defective with a known fault location without requiring fault localisation to determine that location.

Mutation testing, first proposed in 1978 [27], provides a series of "almost correct" variants of an original program whose behaviour should diverge from that of the original program. These are used to test the performance of various fault localisation and repair tools as well as test case generators. Injecting modifications into programs provides a known fault with a prescribed characteristic without the need for debugging existing programs to isolate a fault.

The mutants (where each mutation is considered to be a fault) are used to show that changes to the parent program are detected as failures from the test cases/suites that examine the program traces. One of the problems with automated generation of mutants (for testing the efficiency of detecting faults) is ensuring that the inserted mutation is an actual fault. Without analysis it could merely be a code change which does not alter the flow of the program as it relates to the specifications.

In mutation testing, new mutants of a program are generated; these variants exhibit different behaviour from their parent program due to small changes injected into their code. While injecting source code modifications into existing programs is a simple transformation, some care must be taken to ensure the variant exhibits some visible behaviour changes from the parent program. A mutant which behaves identically to the parent is not a useful tool for mutation testing.

Riener et al. demonstrate mutation testing [94], the generation of new mutants of a program which exhibit different behaviour from their parent program. This is best done via parsing (front end of a compiler) the parent program to allow the abstract syntax tree (AST) to provide meaning to the tokens and then executing various translations to the AST before printing back to the original source representation. The seeding of these mutations provides performance metrics for catching new errors by generating program variants that differ from the original, ie potentially defective variants.

Schuler et al. provide an example of solving this problem without a formal specification by using dynamic pre- and post-condition discovery to verify that the mutants being generated are different from the parent in testable ways [99]. This avoids the issue of false positives where the mutant is not functionally different from the parent and so the mutation could not be discovered by an automated repair or even localisation process.

Mutation testing can also be used to test the quality of debugging tools by providing a set of programs with known faults. The Siemens Test Suite [53], used for our performance metrics and discussed in section 4.1, is a set of mutation seeded variants. The TCAS program in this suite is a common testing metric for automated localisation and repair tools for C since being introduced in 1994 [53]. Of the 41 variants, one of them (v38) provides no failing test cases. For this variant, all 1608 tests provide the same output as the oracle program. This indicates a primitive seeding with mutations to generate these variants without testing for visible faults under the provided test suite.

2.5 Fault Localisation

Fault localisation, the identification of program locations that can cause erroneous state transitions which eventually lead to observed program failures, is a critical component of debugging. However, this process puts a significant time [82, 87] and specific expertise burden [2, 112, 115] on programmers or dedicated debuggers for software projects. Localisation by examining counter-example traces, test cases, or other output from static and dynamic analysis tools is an active area of research [16, 19, 39, 41, 42, 61, 93, 96, 98] that can be divided into several broad approaches, several of which are detailed in the subsections below.

The static code analysis of C programs has been an ongoing area of research and commercial development since the application of model based diagnosis to software systems. This provides mature tools for model checking, test case generation, and fault localisation with ongoing research in these areas and developing work in automated repair. The current state of the art in C program fault localisation shows limitations in performance, automation, and ability to deal with programs that are weakly specified using large test suites.

2.5.1 Spectrum-Based Fault Localisation

Spectrum-based fault localisation techniques, compared in [79, 121, 123], are dynamic analysis methods that operate by examining traces of passing and failing test cases. They assume that faults are more likely to be exercised by failing test cases and less likely to be exercised by passing test cases. All the statements in a program can then be ranked based on different weighting techniques. The analysis of the performance of these approaches is typically based on several scoring formulas that roughly correspond to how much of a program must be explored, given an ordered list of locations as tool output, before the genuine fault is found.

The best-known example of this technique is the Tarantula tool [60, 98]. This can be used with the GCov code coverage tool in GCC to decorate the unit under test and

track which statements are executed for each test case. These tools are popular due to their fast, predictable processing times. A GCov-decorated program will execute in approximately the same time as the undecorated program when compiled with GCC and executed on the test platform. Tarantula provides over 50% of the various variants of the Siemens small programs (including TCAS) with a localisation performance that ranks the injected fault location in the top 10% of lines in order of suspiciousness. But this performance is inconsistent and 7% of these variants are ranked such that the injected fault location is not in the top 80% of ranked locations, requiring debuggers to explore virtually all of the program before encountering the seeded fault.

State of the art spectrum-based fault localisation methods have recently been compared using different theoretical frameworks [79, 123]. Several methods have been identified as optimal under these frameworks. Xie et al. expanded upon previously discovered formulas to generate several new optimal and distinct weightings using genetic programming [124]. The frameworks in these papers include the proof that these competing formulas cannot be placed into a hierarchy where one is strictly best for all inputs.

There is also empirical data over various of the Siemens small localisation benchmark programs. While Tarantula is not optimal [123], it is also not far behind the state of the art on the tested small benchmark programs. In empirical results [79, Table XI, p. 11:23], the only method identified as optimal under that paper's framework ranked the injected fault location on average at the 17th returned location (9.9%) over all TCAS variants. Tarantula returned the injected fault location at an average location between the 18th and 19th ranked location (10.8%).

2.5.2 Slice-Based and State-Based Fault Localisation

Delta debugging [126] is a family of approaches that involve splitting up a large set of changes to find the minimal set that flip the program behaviour from correctly functioning to exhibiting a failure. This has variously been used to minimise inputs and traces but was later extended to source code exploration.

The analysis of traces of a program, which are generated by tools similar to the front-end of a model checker, can be used [93] to look at programs, ideally with few failing runs and many passing inputs, to find the minimal difference between traces on a failing run to the nearest passing. Mapping back to the cause of these flow changes can create a set of possible fault locations.

Zeller proposes a method for dividing up changes to localise the one that introduced a fault called delta debugging [126]. This process involves splitting up a large set of changes to find the minimal set that flip the program behaviour from correctly functioning to exhibiting the fault. This was originally show in 1999 working on hundreds of thousands of changes to GDB (GNU debugger) to isolate a single fault. The approach was expanded

upon [127] to consider grouping changes by their scope (proximity based on analysis of text files and their containing folders) and shrinking the size of the changes tested in making new mutations as the refinement continued. Combining these techniques halved the number of tests required to find a minimal set of changes. This splitting and recombining for delta debugging to pry out the smaller elements of change that are important is seen applied in several areas, as already covered.

Advancing delta debugging to an automated localisation tool for executables, the test case reduction technique of Hildebrandt and Zeller is applied to manage the exploration of compiled program's execution flow [128]. This is used to find the root difference in execution chains from a passing test case that is minimally different to the provided failing one. This was implemented into a robust package [129] that provides results on executables with one passing and one failing input, outputting a cause-effect chain of events that cause the difference in execution flow. This is the smallest gap in inputs between the given failing input and a passing input.

The chain of advances and applications of delta debugging is expanded by Cleve and Zeller to source code exploration [19] and moves from checking program states to also look at the temporal chain of events and ties into object scopes so that a failure in one variable can be tied back to that other variables in scope when it was created and work back to a root cause. This allows for the root fault, in source, of a failure to be correctly identified by this technique with performance that outclasses the competing methods, such as Renieres and Reiss [93]. The principle applied here [19] is to look at passing and failing traces and minimise the differences between them to isolate the failing components. This is reminiscent of a binary search, looking for interesting subset behaviour to narrow down variables that correlate with failure. However, this does require the existence of at least one passing trace and the localisation performance of Delta Debugging on small benchmark localisation test programs [19] is worse than Tarantula's [60] results.

The state of the art in slicing and heuristic led localisation is currently exploring [96] the generation of program invariants from correctly executing inputs that are close to failing inputs and analysis the cause chains that force these invariants to fail. This approach shows where the localisation work is progressing towards processing larger and more complex programs in a reasonable time after reaching a peak for accuracy and minimum false positives.

Using model checker counter-examples, Groce et al. use distance metrics with a semi-automated approach for fault localisation [41]. Once a failing run is found and a counter-example produced, a solver is used to generate a similar (in terms of execution) passing run, and the differences between these runs are minimised by slicing and then returned as the localisation result. This work indicates the difficulty of automated localising on even a program as simple as the TCAS variants.

Around the same time Wang et al. were also looking at removing the manual process of going from counter-examples to continuing the debugging process with localisation. A single counter-example trace is used to find potential fault locations. The technique [114] uses a counter-example to build a concrete execution trace and then generate weakest pre-conditions for the eventual failure working back along the execution. These can be chained back to the root fault. This approach contains similarity to Cleve and Zeller's work but using abstractions of the execution, similar to those explored by symbolic model checking.

2.5.3 Model-Based Fault Localisation

Model-based fault localisation [92] (sometimes also called model-based debugging [20]) is the application of model-based diagnosis methods [25] to programs. It involves three main steps: (*i*) the construction of a logical model from the original program; (*ii*) the symbolic analysis of this model; and (*iii*) mapping any faults found in the model back to program locations. One popular approach to model-based fault localisation is to transform the program so that a symbolic program verification tool can be reused for all three steps.

For example, Griesmayer et al. describe a method [39] in which the model (in the form of a logical satisfiability problem) is derived by running the CBMC model checker over a transformed program, and analysed by means of the model checker's integrated SAT solver. The transformation "inverts" the program's specification (cf. section 3.1, producing failures where the original program would complete and blocking paths where the original program would fail), and replaces each original assignment by a conditional assignment with either the original value or an unconstrained symbolic value, depending on the value of a toggle variable. The actual localisation can then be reduced to extracting the possible values of the toggle variable from the satisfying assignments that the SAT solver returns.

However, the technique initially described by Griesmayer et al. requires detailed specifications to achieve acceptable precision—the weaker the specification, the more program locations are flagged as potential faults. Such detailed specifications rarely exist in practice. What *does* commonly exist are extensive unit test suites, in particular in the context of modern test-driven design approaches. Griesmayer et al. have shown that his technique can be extended to work with (failing) test cases, but the published results [39] are prohibitively slow (typically executing in hundreds to thousands of seconds per each small benchmark program). Griesmayer et al. have improved the original implementation [40] but only achieve times typically in the hundreds of seconds.

Griesmayer et al.'s approach has also been applied to hardware designs in SystemC [67], often combined with different solver technologies such as QBF [106] or unsatisfiable

cores [108]. Our results indicate that "plain old SAT/SMT" is still sufficient, but these technologies could be considered as alternative approaches to surmount any future limitations with our underlying solver tool chain.

Könighofer and Bloem have developed the FoREnSiC system [63, 64] (a Formal Repair Environment for Simple C) which includes a similar method of localisation. Their published results only cover a few variants of the benchmark localisation test programs used but where they have publishable results, localisation times are still measured at over a hundred seconds. Könighofer et al. report improved times [65] (around 37 seconds per program) but only after manually annotating all functions with contracts, so no longer working from the benckmark's test suites alone.

A hybrid approach to localisation using concolic execution [16] is shown on Java programs by Chandra et al.. Their process creates a family of different programs that each take a different potential fault point. Each of these program is explored with concrete values up to their potential fault point, after which they use symbolic execution to explore potential program flows that can lead to failure. A potential fault point is flagged as a localisation if there is a change to the expression (value substitution) that allows the program to continue to avoid the failure and if this change also allows passing runs to still pass if they take a different value at this repair point. This attempt to remove false positives by constraining on passing test cases also blocks localisation to points where a passing test case requires an exact value to continue (no range of values all lead to successful termination).

Jose and Majumdar convert an input C program to a maximum Boolean satisfiability problem which is analysed with the MAX-SAT solver [61]. However, because this returns the complement of the maximal subset of clauses that can be true for each single test case, their approach can omit genuine repair locations. This technique therefore relies on summing the results of the different test cases, providing a ranking of most to least commonly flagged locations. This is the approach, and so inherits the strengths *and* weaknesses, of heuristic-based fault localisation techniques like the spectrum-based techniques. Approximating model-based fault localisation approaches for test suites can run faster but can also miss a true fault location when evaluating a test case.

2.6 Griesmayer's Model-Based Fault Localisation in Detail

Griesmayer et al. described one implementation of a fault localisation technique based on model checking [39], a field outlined in subsection 2.5.3. This technique is the foundation on which our research has developed. We have provided fully automated tools to implement the described process and expanded the technique in several areas. Critically, we have developed a smart search process to manage the unpredictable return time for

each element of the search for a location under the specification of a specific test case, including removing redundant search nodes from the original proposed process.

```
1 /* ... */
2 int v = a*b;
3 /* ... */
4 assert(o==d); //added spec
5 exit(0);
```

Figure 2.1: Program transform process: Code with test case specification

The original technique described by Griesmayer et al. uses an already-specified input C program (such as that seen in figure 2.1 - where the output value of the program `o` is asserted to equal a variable assigned to with the known desired output of the program `d`) which is reconfigured to relax the assignment constraints at any single locations and to use an inverted version of the specification to generate the desired information in the symbolic analyser's provided counter-examples. An example of this transformation is seen in the modifications to the code snippet in figure 2.1. This transformation is applied to the C source code before the model is generated by the CBMC model checker, which then analyses the model using a SAT solver.

```
1 int t = symbolic();
2 /* ... */
3 int v = (t==5) ? symbolic() : a*b;
4 /* ... */
5 assert(o==d);
6 exit(0);
```

Figure 2.2: Program transform process: Model of specified code

To enable the search for potential fault locations, a toggle variable, `t`, is added that allows any one location to run alternative code. In the program body this toggle is made symbolic and constrained to the range of locations being explored (cf. line 1, figure 2.2, constraining possible values not shown). Assignments in the source program are modified to become conditional assignments that flip to an unconstrained symbolic value when toggled (cf. line 3, figure 2.1 to figure 2.2). The use of a single toggle value makes a single-fault assumption.

```
1 int t = symbolic();
2 /* ... */
3 int v = (t==5) ? symbolic() : a*b;
4 /* ... */
5 assume(o==d);
6 assert(0); exit();
```

Figure 2.3: Program transform process: Inverted model of specified code

To invert the program specification, each potentially failing `assert`-statement is replaced by a blocking `assume`-statement with the same parameter (line 5, figure 2.2 to figure 2.3). This requires that all traces through these lines conform to the assertion's test without

generating a counter-example if the expression can be violated. Failing `assert(0)`-statements are inserted before the program's terminal nodes to force the generation of new counter-examples (line 6, figure 2.3). The purpose of this inversion is to ensure traces that originally returned counter-examples due to specification failure do no longer, and traces that satisfy the specification generate counter-examples. The exploration becomes one searching for a trace to an exit point that satisfies the initial specifications. Thus, in a simple (no symbolic assignment insertion) inverted program an exiting trace will not be found for a failing test case.

The generated counter-examples from this new model provide the toggle values that identify candidate assignment locations as sites of possible repair. The technique described by Griesmayer et al. requires that the input programs have detailed specifications. In practice, these typically do not exist. Griesmayer et al. demonstrated [39] an extension to this technique using a test suite as specification. A failing test case is first encoded into the input program, providing a concrete value of `d` in figure 2.1. The results for each toggle value therefore identify candidate repairs for that failing test case. Any assignment that is identified by all failing test cases becomes a location flagged as being the site of a potential repair.

The localisation process effectively generates a look-up table at each flagged location that will repair all failing test cases. For each failing test case, the value of the alternative assignments can be different and will be reported for each location flagged by this process. The flagging process means the chosen value leads to the end of the program without failure of the original specification. All passing test cases define their own correct values for the assignment. So this look-up table is a genuine repair for the full test suite.

This approach, where each test case is treated as an independent specification and all results are collected independently, results in prohibitively slow execution time. Using test suites demonstrates a strength of dynamic analysis: no prior knowledge of the program beyond flagged test cases as correct or incorrect is required. But the published results indicated that the run-time cost is too high using the common localisation performance measure of the Siemens TCAS [53] variants.

2.7 Automatic Repair

At the limit of localisation techniques is the use of automated software repair, where the location that caused the fault is diagnosed to the point where a viable repair expression is provided. This is not usually explored in isolation so automated repair tools are responsible for localising before or as part of generating repair expressions.

Griesmayer et al. moved from localisation work on deterministic C programs to automated repair of the potentially non-deterministic program over Boolean data types that

represents a simple C program. This tool [38] explores the different paths of execution in the Boolean program and, assuming a known localisation, changes that element of the execution to an arbitrary execution.This allows the program to jump to another point or change a value within scope as the alternative execution at this point. Model checking is used to narrow the possible substituted expression to ones that do not violate the program specifications. When localisation data is not provided then the tool iterates over all expression locations and repeats the repair process. Any repair must be mapped back to the C program from which the Boolean program was constructed. It is noted by Griesmayer et al. that the generated repairs from this process require human interaction to choose the correct one that solves the debugging problem, if it is generated, and so is not a suitable choice for automated repair.

A recent approach by Wei et al. provides well specified Eiffel programs with automatic repairs via schema generated mutations [117] which are validated against failing and passing test cases with the pre- and post-conditions and assertions on objects. This exploits the strong, granular specifications of the programs to apply a library of mutation substitutions to search for valid repairs over the full specification, driven by the test case set. This tool, AutoFix-E, is later refined [84] by Pei et al. to reduce the requirements on the specifications, removing the need for public interfaces showing object specification, although this still leaves the need for a strong specification/contract to reason about repairs.

The automated repair work using delta debugging concludes with the release of GenProg [69]. This work applies the search technique to accelerate and later direct the use of genetic/evolutionary mutation of a defective program towards a potential repair variant [34]. The mutations pick code from inside the defective program to apply at a repair site, which has been previously shown to be a surprisingly effective technique. The original implementation [118] used the AST of a C program with path as weighting, limiting mutations to only operate on the execution of the faulty program trace. The first version used delta debugging to minimise the size of this mutated program that repaired the failure without breaking passing runs. This was later reprinted with further examples [36].

The approach by Fast et al. uses test cases to encode the requirements of the program and trigger the failure and reduces the issues with mutation creep outside of well specified programs by delta debugging to try and minimise the mutation from the original buggy program. It was extended to assembly [100] rather than a C AST by Schulte et al. and optimised for faster more accurate results [70] by Le Goues et al., using better mutations and breeding algorithms for the evolutionary stage. Results for TCAS are provided by Nguyen et al. [82]; they indicate scaling issues for even medium sizes of test case sets, even with programs as simple as TCAS. We consider their performance further in section 4.5.

The delta debugging method, with modifications, has been shown to scale to very large programs by moving from AST and weighted paths to committed diffs (patches) for an ongoing project in source control [68]. It is offered as a cost effective repair solution when running at cloud computing scale/prices. Scaling out the execution hardware to cloud deployment helps contain the costs of evolutionary mutation and testing each mutant for success at repairing the found failure without generating new failures in the specification, provided days or massively parallel processing are available for turnaround.

Konighofer and Bloem provide results on TCAS [63, Table 1, p. 98] using a model checker driven mutation search of expressions based on A * x + B * y + ... + Z for integer literals as capitals and every variable in scope at the point of the assignment as lower case in that formula. However, the published results indicate the full range of this mutation space is not being fully exploited as the only repairs produced reject all variables in scope, repairing the TCAS variants which can be fixed with a single integer literal, the Z in the above formula. We consider their performance further in section 4.5.

The tool DSDSR by Hussain and Csallner automatically repairs Java data structures using dynamic symbolic execution [52]. This is a limited scope repair tool looking for a very specific fault type. It builds on top of DSD-Crasher [24], which collects invariants to generate a specification via dynamic analysis, which restricts the input domain of the program. DSD-Crasher then uses static analysis to search this smaller input space for emerging failures, and then verifies the discovered failures with a final dynamic pass on generated test cases to confirm the static analysis approximation applies to actual Java execution.

ClearView [86] is a live program repair system which monitors an executing x86 Windows binary to generate a specification from the observed invariants and input ranges. By inserting monitors to detect errors this specification is refined to classify any invariants that possibly lead to errors or failures. The tool then works to create a patch to the binary to block (control flow change) or repair (state change) the path to failure. This is shown in a security context on x86 binaries on a limited number of test cases and with manually defined monitors checking the binary execution.

Nguyen et al. provide SemFix [82] as a KLEE (symbolic virtual machine) based tool with expression generation using a small subset of test cases to generate a valid repair under the specification of that subset. This provides good results for TCAS compared to GenProg's method [82, Fig. 4, p. 779], assuming that the repairs under that more limited specification are useful for repairing the expression under the full specification. It is not reported if these repairs generated on the limited test case sets are genuine repairs that apply to the full specification as given by the complete universe but extrapolating the times out to running the process on the full set of test cases would not provide competitive performance if this is not so. The later data they provide on found repairs suggests the reported repairs are not necessarily valid over the full test case set specification but

this is not conclusive. No per variant results are provided, only averaged performance. We consider their performance further in section 4.5.

Recent work by Jose and Majumdar [61] converts an input C program to a maximum Boolean satisfiability problem to be explored with MAX-SAT. This operates on a single defective test case as specification and uses the trace this generates to construct the formula. Using this formula the process looks for a maximal subset of clauses that can be true and reasons that the complement of this set are tied to the location of potential repairs. This localisation approach is expanded to repair synthesis by selecting this set of potentially faulty lines and modifying any computation there to account for the fault class of 'off by one' errors. Results are shown for localisation on TCAS as well as a range of small Siemens Test Suite programs. To provide tractability for MAX-SAT, the traces of programs other than TCAS were minimised with a combination of program slicing, concolic execution, and isolating failure-inducing input using delta debugging. The fault class repair process is shown working on some toybox examples. This is only looking for repairs and localisation results that are specified by a single test case, not all 1608 test cases for TCAS.

Other approaches provide a manual rather than programmatic approach to program repair, for example von Essen and Jobstmann [113] writing a strong specification in LTL for the correct program execution to provide a directed repair process.

2.8 Assessment and Self-Training

To understand how to automatically assist in the debugging of programs requires the understanding of how programmers make mistakes and work to correct them manually. Novices offer both an increased need for high-quality tools and the opportunity to understand the psychology of programming through the analysis of how skills develop. Education is an area with a specific need for high quality automated localisation to aid novice debuggers and, as we discuss in subsection 2.8.3, provides a significant volume of suitable real-world faulty programs on which to test localisation tools.

2.8.1 Programmer Types and Programming Psychology

In the "Psychology of Programming" [50] Hoc et al. note that much of the mental model for the activity of programming can be influenced by external factors such as the non-programming education of the coder, the programming language in use, and even the native language of the programmer. There is also a wide range of domain-specific expertise and programmers cannot be assumed to have reached a high level in any area [50, §3.2.1, p. 225] before being able to gain the benefits of automated debugging tool assistance.

Novice programmers can be divided into three classes [85, 120]: *stoppers*, who give up; *tinkerers*, who appear to modify their code at random; and *movers*, who are already able to engage with feedback to make progress. Even though the movers already demonstrate debugging skills beyond the average of their cohort, all three classes still need high-quality, novice-friendly debugging feedback to allow a self-guided refinement of their solutions towards a correct answer.

Giving tinkerers viable repair locations massively reduces the mutation space they are exploring. Giving movers viable repair locations directs their effort, allowing faster debugging times [83]. This should increase the chance that both will realise the repair and convert a program failure into a learning event. For programmers to retain their status of movers, they must be provided with feedback in terms of suitable scaffolding to overcome progress stalls and achieve an otherwise unattainable goal [122]. Tinkerers who are unable to be scaffolded into movers will eventually become stoppers due to lack of progress [120]. This generates the need for high quality, novice-friendly debugging feedback.

A significant part of the debugging process is finding the location where the file needs to be changed to repair the fault [115]. Tools like AskIgor [129] provide cause chains to assist programmers in localising faults from test failures. While this lowers the required expertise threshold, it still requires too much debugging expertise for novices. Only experts debugging small programs see significant benefit [83] from localisation tools with the precision of Tarantula due to the requirement for debugging skills to be capable of working through significant ranked lists of potential repairs before even reaching the real fault location. Since the absolute number of locations shown before the actual repair site is critical to allowing progress before programmer interest drop-off generates stoppers [83], the provided list of locations to investigate must be short. Unfortunately, widespread spectrum-based fault localisation methods are unable to provide sufficiently short candidate lists. Pham et al. tested Tarantula and Ochiai tools and on three variants of an algorithm [88], with an average length of 13 statements, they found over 9 statements shared the same top suspiciousness ranking. Localisation performance thus becomes a function of luck.

Analysis of student code submissions [72, 78, 91, 103] shows a wide range of submitted program forms and performance characteristics. Students working towards functionally correct submissions do not necessarily work towards solutions that are similar to an instructor-authored model answer. Features such as code length and memory footprint are independent of each other and of compliance with a question's functional requirements [95]. This makes it inadvisable to only provide feedback directing students towards a model answer as the shortest path to a high quality submission.

Enhanced (syntax) error messages appear to be ineffectual in assisting novice debuggers apply feedback [29]. However, when students are given hints about failures over a

test suite, their effort increases compared to students not given hints [13]. On-demand programming feedback provides students the motivation to iterate on submissions [75]. Moreover, a correlation has been found between sessions where students were provided with feedback from a test suite and sessions that improved the student's score [104], indicating this assisted self-training.

Complete program synthesis repair tools such as Auto Grader [102] remove all debugging or repair self-training from the process, trivialising the contribution and so learning of the student. Auto Grader uses an error model and constraint solver to transform student submissions to try and formulate/synthesise repairs, this requires the writing of a manually provided, specialized error models for each problem to allow the synthesis tool to provide repairs.

There is a need for debugging feedback that acts as scaffolding, the tutoring required to facilitate a novice to achieve a goal they would otherwise find unattainable [122]. This must bridge the gap between standard error reports, which many students lack the expertise to use, and fully automated repair suites that trivialise the contribution and so learning of the student. A viable automated localisation tool must provide high quality localisation feedback for student self-training which lowers the prerequisite debugging skills, adding necessary scaffolding to the programming experience when approaching a correct solution, without restricting students to converging towards a model answer.

2.8.2 Programmer Feedback and Guidance

Coursework and charettes, traditionally manually assessed by educators and instructors, have seen various attempts at automation since the 1960s (cf. subsection 2.8.3). Modern, generic assessment frameworks provide a series of built-in grading tools or just provide the framework into which external tools can be plugged in, which feed back their results into the system. Automated submission and assessment systems deliver immediate feedback to students for self-training, which is enjoyed by students and feeds into improvements throughout courses that use it [66]. They include tools to help accelerate grading of final submissions, providing both self-training and grading benefits [107]. When manual feedback from submissions is provided, even when timely and electronically delivered, it generates a drop-off in engagement [73] over the run of a course.

But scaffolding, the tutoring required to facilitate a novice to achieve a goal they would otherwise find unattainable [122], is necessary to bridge this gap between standard error reports and fully automated repair suites that trivialise the contribution and so learning of the student. When students are given hints about failures in an automated testing suite, their effort increases compared to students not given hints [13]. Moreover, a correlation has been found between sessions where students were provided with feedback from a test suite and sessions that improved the student's score [104], indicating this

assisted self-training. Using the Marmoset submission system [105], students were provided with feedback from a small test suite. This was tracked over a number of sessions. A correlation was found between when the student collected feedback and sessions that improved the student's score [104], indicate this assisted self-training. In this example, to avoid grade-like feedback being abused to brute-force answers, the use of tokens restricted how often feedback was available to the students. However, despite its benefits, students are somewhat resistant to a pure test-driven development (TDD) approach [12]. For example, the Web-CAT [31, 32] submission system, which provides students with a workflow that mandates the writing of test suites as well as source code. This is at least partly because TDD also requires expertise in developing test suites as a prerequisite to demonstrating programming ability.

2.8.3 Coursework Submission and Assessment Systems

The modern pedagogical style of continuous assessment delivered to ever-increasing class sizes demands increased automation of assessment feedback, including early-stage feedback during coursework attempts. This assessment model includes a self-improvement model which uses the coursework format as an educational tool rather than one of formal assessment. The loop of work and feedback improves the performed knowledge of the student without a formal mark that contributes to a final grade.

The timely, accurate, and detailed feedback from the automated tools is essential to this model of directed continuous student improvement. Without the assessment component, the student does not have a directed experience that affirms their progress.

There is a long history of coursework, normally assessed by teachers manually, seeing some automation. Reports go back to punchcard-based programs in 1959 [51] and executing student programs to compare their output with a stored value in 1960s [46] for automated grading. Source code can be assessed with general evaluation criteria focussed on execution performance, which can bring in some automation, or degree of closeness (to a model answer) assessment, for example a control flow graph comparison starting out in the 1980s [1], which facilitates discrimination between non-functional submissions. A textual analysis is clearly insufficient [78, 91, 103] as there is not only a single form of a solution that correctly solves the programming question. Analysis of student code submissions show [95] a very wide range of program forms and performance characteristics.

Code assessment frameworks and tools are becoming ubiquitous but also require specialist skills [3] to fairly design coursework questions and test suites. Many educational establishments now include real-time automated feedback for both non-graded coursework and, as an initial feedback stage, for graded work, when the coursework is being prepared for submission. Automated submission and assessment systems [8, 47, 56, 62, 90, 111]

use functional and/or unit testing to confirm a student program complies with the assessment test suite. In Computer Science this ensures graded submissions can pass the initial technical hurdles (such as submitted source code compiling) to be submitted to a large, fully automated, rigorous testing environment which provides guidance for the final assigned grade. This real-time feedback can include running through a limited test suite to allow students (who submit early) to improve any easy-to-spot faults in their submission without the assessment becoming an implicit test of their own testing skills and development environment. For non-graded work assigned for self-improvement, the full testing environment can be provided to the students as enhanced feedback. Providing this as a service removes the configuration of a test environment from the courseworks prerequisite Intended Learning Outcomes (ILOs).

Advances on these tools can provide some of the test suite feedback to students to facilitate their debugging. Use of token systems to prevent abuse of this to brute-force the assessment test suite has been examined in Marmoset [105]. The correlation between getting this feedback and that work session improving the student's score [104] indicate this is effective at assisting debugging. Some systems have, in the last decade, e.g. Web-CAT [31], moved to mandate student written tests as well as source code. This self-contained TDD driver also generates significant volumes of test suites for student submissions.

Such assessment tools take many forms. Ceilidh [8] uses regular expressions to specify the test output of compiled student code, providing more freedom to define the expected answers than traditional test suites. Immediate feedback options include compile confirmation messages for source code and functional assessment via test suites. Due to the pluggable nature of these systems, increasingly advanced processing can be provided to students as well as for grading. Style and complexity analysis tools [56] provide metrics beyond test suite correctness (evaluating non-functional aspects such as code quality and efficiency). ASSYST [56] contains style and complexity analysis tools that provide metrics beyond test suite correctness, and also evaluates non-functional aspects such as code quality and efficiency. AutoGrader [45] requires students to program to a public interface, allowing whitebox testing at the cost of restricting the form of the code submissions. Pex4Fun [110] uses automated test case generation to guide student submissions and to scale to MOOC capacities. This approach requires students to demonstrate advanced debugging skills.

ASys [55] attempts to overcome the limitation of functional testing using test suites with a customisable semi-automated grading system. Marking templates describe extensions to the test suite for a question, generating new tests specific to the student submission that verify each assessment property defined in the template. Failure of the system to complete the instructions or if the submission does not conform to the tests will create a report and triggers manual grading using this feedback. In their use on campus, 48% of the grading work was automatically handled by the tools. Analysis of a discrepancy in

marks between the group using the new system and the group going through traditional grading revealed most of the difference was accountable to errors made in the manual marking process.

With FrenchPress [11] the focus is put entirely on assessment of style and code design, ignoring errors and incorrect output. Tools which explore the style and code design of submissions are complementary to an educational approach that provides learning events tied to the test case compliance of student submissions or code compliance result to accelerate marking of submissions. As noted in this paper [11], "large-scale professional projects [generate] bug reports [that] assume a sophisticated understanding that college students are unlikely to attain in their first year of exposure to Java. Professional tools do not look for, and consequently do not find, the program flaws FrenchPress catches, precisely because the errors of an experienced software engineer are not those of a second-semester student."

The graph similarity approach proposed in 1980 [1] exists in modern systems [116, 80]. Scheme-Robo [97] compares the program source to a model answer, assuming that Scheme's functional design limits the variability of meaningfully different programs that correctly answer the question. Although this similarly only provides analysis of how closely the student submission matches the form of the model answer rather than how it functions to comply with the specifications of the question. Vujošević-Janičić et al. [58] use a static verifier as well as test suites when student solutions do not match the program flow of the model answer. A weighting system then mixes these feedback components to guide students to repair code errors and transform the structure towards that of the model answer or to grade the submission.

2.8.4 Massive Open Online Courses

Considering the scaling of these techniques, instructors spending several minutes per student per submission for feedback [73] cannot scale to Massive Open Online Course (MOOC) environments with thousands of students per course. Compounding this issue of the viability of scaling manual feedback, even timely and electronically delivered notes generate a drop-off in engagement with this asynchronous feedback [73], providing students with feedback detached from the mental state when they created the submission. Offering live feedback is even less capable of scaling to larger classes without relying on fully-automated feedback tools.

To test the scaling of an automated hint system to MOOC environments, AutoTeach [4] provides students with access to a hint system which can uncover partial model answers to assist with learning. Such systems, while scaling very well, strictly constrain students to working towards a single model answer which is slowly revealed by the hint system. This will not assist students who are not writing a clone of this model answer.

With Pex4Fun [110] automated test case generation (powered by Pex) is used to guide student submissions and to scale to MOOC capacities without losing the focus on the actual student submitted code. But this loops back to the earlier points regarding TDD and student resistance or lack of skills in this area. Only the movers group, who are already competent debuggers, can use a failing test case to guide their code repair. While dynamic test case generation based on the (hidden) oracle solution provides good personalised feedback for each different student submission, it only focusses on one type of student with a set of existing competencies in testing and debugging, which may not be the focus of assessment in a programming exercise.

However, the deployment of test suite grading for MOOCs is gaining traction [89] with the drawbacks of formatting issues (3.2% of submissions were awarded zero marks due to formatting issues, not functional failures) and failure to detect almost-correct submissions weighed against the benefits of providing some form of code assessment, as manual grading is not viable. Such issues can be reduced via importing more advanced tools into MOOC grading and feedback systems, assuming their run-time cost can be kept low enough for MOOC execution.

Chapter 3

Fast, Test-Driven, Model-Based Fault Localisation and Repair

Our first major contribution area is the development of a smart search process which significantly reduces the processing time of a test-driven, model-based localisation search such as the one by Griesmayer et al. that we have outlined in section 3.1. This is required to manage the unpredictable return time for each node of the search, including removing redundant search nodes from the original proposed technique. A radical reduction in execution time when dealing with large test suites would allow this technique to compete with spectrum-based methods [RQ1] without sacrificing accuracy [RQ2]. A single search node in this technique, under the granularity of our symbolic analyser, is a single potential repair location under the specification of a single failing test case. We developed this algorithm and implemented it into a suite of fully automated tools that implement the process Griesmayer et al. described, expanding the technique in several areas [RQ3, RQ4]. We directly compare this with a pure reimplementation of the original technique, using modern hardware and symbolic analysis tools, as shown in chapter 4. The fully automated nature of this tool allows its execution on large sets of actual C programs to test the viability of this method of test-driven, model-based localisation [RQ2], as exploited in chapter 5 [RQ5].

3.1 Inverted Model-Based Fault Localisation

A common process for model-based fault localisation uses static analysis on a model that is generated from a transformed input program using the language specifications. This transformation is provided by a checker tool that converts the program code into one or more logic expressions, and so analyses the symbolic execution of the system. A solver is invoked that can evaluate the logic expressions. This symbolic analysis may be

accelerated with the use of built-in theories to compactly reason over the logic expressions [81]. This returns constraint satisfiability results for the expressions. The symbolic analyser uses these results to generate traces of specification violating execution paths (counter-examples), if any exist. Some methods operate directly on the satisfiability result for the logic expression, for example exploring common omissions when using a maximum satisfiability solver [61]. Programs can be modified, e.g. augmented with additional predicates, to generate more information from the data in the failing traces.

Griesmayer et al. described a technique [39] where a specified input C program is reconfigured to use an "inverted" version of the specification. This inversion is applied to the C source code before the model is generated by, in their implementation, the CBMC model checker, which then analyses the model using a SAT solver. The reconfiguration generates a model based on a single-fault assumption.

```
1  int payMe(uint nH, uint otH, uint age) {
2    uint BR = 15;
3    uint bonus = BR * (otH * 2);
4    uint normalPay = BR * nH;
5    uint ageHigher = age % 20;
6    uint SelBR = BR + ageHigher;
7    uint SelBonus = otH * (3 * SelBR);
8    uint SpecialNormalPay = nH * SelBR;
9    uint specialPay = BR + 20;
10   if ((age > 20) && (age < 40)) {
11     return SpecialNormalPay + SelBonus;
12   } else if (age >= 40) {
13     return (nH * specialPay) + ((specialPay * 2) * otH);
14   } else {
15     return normalPay + bonus;
16   }
17 }
18
19 int main(int argc, char ** argv) {
20   int result = payMe(atoi(argv[0]), atoi(argv[1]), atoi(argv[2]));
21   assert(result == atoi(argv[3]));
22 }
```

Figure 3.1: Worked transformation: Code with test case specification

Our worked example is based on the form of a student coursework submission used in the data set described in section 5.1. The listing in figure 3.1 is taken after the program specification has already been encoded into the coursework submission, as detailed in subsection 3.2.1 as this is not part of the automated technique described by Griesmayer et al. This program calculates pay for various employees based on the formulation given in a written question. The failing program, at the sixth assignment (line 7), declares an unsigned integer `SelBonus` equal to `otH*3*SelBR`. This expression contains a fault as the question text requires the bonus rate be double pay, not triple pay. This fault can surface, for some inputs, in the final output as a failure when returned on line 11 to `result` on line 20.

> The pay that an employee earns each week depends on their age, the number of hours worked during normal business hours, and the number of overtime

> hours worked. The base pay rate for all workers is $15 per hour. On top of the base rate, each worker over the age of 20 earns an extra $1 per hour for every year their age exceeds 20. However this additional, age-based bonus is only valid up until the age of 40. Finally, any overtime hours are paid at twice the base rate. Write a function which calculates the amount paid to an employee in one week, based on the number of normal and overtime hours worked, as well as their age. You should work in whole numbers (integers) only.[1]

Each test case is taken as a search branch to be explored for potential assignment locations. To enable the search for potential fault locations, an `unsigned int` global toggle variable, `__t`, is added that allows any one location to run alternative code (cf. figure 3.2). In the program body this toggle is made symbolic and constrained to the range of locations being explored (lines 2 and 3). Assignments in the source program are modified to become conditional assignments that flip to an unconstrained symbolic value generated by a call to `symbolic()`, when toggled. The single global toggle variable creates a model with a single-fault assumption.

```
1  int payMe(uint nH, uint otH, uint age) {
2    uint __t = symbolic();
3    assert(__t < 11);
4    uint BR = (__t==0) ? symbolic() : 15;
5    uint bonus = (__t==1) ? symbolic() : BR * (otH * 2);
6    uint normalPay = (__t==2) ? symbolic() : BR * nH;
7    uint ageHigher = (__t==3) ? symbolic() : age % 20;
8    uint SelBR = (__t==4) ? symbolic() : BR + ageHigher;
9    uint SelBonus = (__t==5) ? symbolic() : otH * (3 * SelBR);
10   uint SpecialNormalPay = (__t==6) ? symbolic() : nH * SelBR;
11   uint specialPay = (__t==7) ? symbolic() : BR + 20;
12   if ((age > 20) && (age < 40)) {
13     return (__t==8) ? symbolic() : SpecialNormalPay + SelBonus;
14   } else if (age >= 40) {
15     return (__t==9) ? symbolic() : (nH*specialPay)+((specialPay*2)*otH);
16   } else {
17     return (__t==10) ? symbolic() : normalPay + bonus;
18   }
19 }
20
21 int main(int argc, char ** argv) {
22   int result = payMe(atoi(argv[0]), atoi(argv[1]), atoi(argv[2]));
23   assert(result == atoi(argv[3]));
24 }
```

Figure 3.2: Worked transformation: Model of specified code

Finally, this model is inverted which forces the verifier to suppress any assertion failures in the original model but to generate new counter-examples if the program can terminate normally without violating these assertions. The generated counter-examples from this new inverted model provide the toggle values that identify candidate assignment locations as sites of possible repair.

[1] Taken from `http://www.dreamincode.net/forums/topic/288718-possible-math-error/`.

Each potentially failing `assert`-statement is replaced by blocking `assume`-statement with the same argument. Failing `assert(0)`-statements are inserted before the program's terminal nodes to force the generation of new counter-examples. The purpose of this inversion, shown in figure 3.3, is to provide the inverted output. Hence, traces that originally returned counter-examples due to specification failure do no longer, and traces that satisfy the specification now generate counter-examples. The exploration becomes one searching for a trace to the exit point that satisfies the initial specifications. If the inversion was applied without the toggled symbolic assignments added in figure 3.2, an exiting trace would not be found for a failing test case as it defines a trace that violates the specification `assume`-statements.

```
1  int payMe(uint nH, uint otH, uint age) {
2    uint __t = symbolic();
3    assume(__t < 11);
4    uint BR = (__t==0) ? symbolic() : 15;
5    uint bonus = (__t==1) ? symbolic() : BR * (otH * 2);
6    uint normalPay = (__t==2) ? symbolic() : BR * nH;
7    uint ageHigher = (__t==3) ? symbolic() : age % 20;
8    uint SelBR = (__t==4) ? symbolic() : BR + ageHigher;
9    uint SelBonus = (__t==5) ? symbolic() : otH * (3 * SelBR);
10   uint SpecialNormalPay = (__t==6) ? symbolic() : nH * SelBR;
11   uint specialPay = (__t==7) ? symbolic() : BR + 20;
12   if ((age > 20) && (age < 40)) {
13     return (__t==8) ? symbolic() : SpecialNormalPay + SelBonus;
14   } else if (age >= 40) {
15     return (__t==9) ? symbolic() : (nH*specialPay)+((specialPay*2)*otH);
16   } else {
17     return (__t==10) ? symbolic() : normalPay + bonus;
18   }
19 }
20
21 int main(int argc, char ** argv) {
22   int result = payMe(atoi(argv[0]), atoi(argv[1]), atoi(argv[2]));
23   assume(result == atoi(argv[3]));
24   assert(0);
25 }
```

Figure 3.3: Worked transformation: Inverted model of specified code

Any looping behaviour of the program under test is resolved by the symbolic analyser. Bounded model checking will unfold the program with unwinding of the loops to a bound of k and inlining function calls. Recursive calls will also be inlined, up to the bound. This explores a subset of all traces of the program up to the bound of the unfolding. When a fault exists within a loop then this configuration of symbolic expression insertion will replace every instance of the assignment with a new call to generate a symbolic value. These will either all be triggered by the toggle value or not as each call will be guarded by the same condition. This adds to the solver complexity of any fault in a loop (or function called multiple times) but allows the search to isolate such faults to the same standard as other locations searched.

This complete transformation effectively searches the model to find assignments which are possible repair locations. That is, every such assignment can be edited to a symbolic value which corrects the flow of *one* failing test case and results in the desired output

being generated, i.e. not violating the `assume(result==atoi(argv[3]))` statement in the worked example's inverted model. Assignments where a symbolic value is found for *every* failing test case are returned as repair locations.

The localisation process effectively generates a look-up table at each returned location that will repair all failing test cases. For each failing test case, the alternative value of the assignments will be reported (via a counter-example) for each location flagged by this process. The flagging process means the chosen assignment values lead to the end of the program without failure of the original specification. All passing test cases define their own correct values for the assignments (they already execute within the test suite specification). So this look-up table is a genuine repair for the full test suite, albeit repairing only from the test cases provided as specification.

Unlike most other localisation processes that only aim to isolate locations as suspicious, this process generates artifacts which define a low-quality repair for the specification, as given by the full test suite. The repair is not one which a programmer would accept as complete but it is a repair that, at minimum, provides significant additional debugging information in the form of what values the high-quality repair can output to fix the faulty assignment under the specification of the test suite.

This approach, as originally described by Griesmayer et al., where each test case is treated as an independent specification and all results are collected independently, results in prohibitively slow execution time. Using test suites demonstrates a strength of dynamic analysis: no prior knowledge of the program beyond flagged test cases as correct or incorrect is required. But the published results indicated that the run-time cost was too high using the common localisation performance measure of the Siemens TCAS [53] variants. We modify this approach to make the search of failing test cases dependent on the discovered negative search results, maintaining a whitelist of still viable fault locations which is narrowed down as the localisation tasks return.

To implement this extension of the above process, we use a multiprocess design that takes advantage of modern consumer processor architectures to radically reduce execution time [9]. Each test case is dispatched as a task to a worker pool with any pruning percolated to future tasks. We minimise the symbolic execution load with two features. Firstly, intelligent pruning of the search space, i.e. not searching known-unrepairable assignments in subsequent test cases after the search has started. Secondly, we minimise the disruption of an intractable (a risk of model-checking programs) or slow search branch by monitoring and evicting tasks, which are retried later, once the search space has been pruned.

3.2 The Localisation Algorithm

We propose a fast algorithm which optimises the search of the test suite and viable repair locations to bring this process into line with current performance expectations and without compromising the completeness of the results this technique allows. Each test case in the suite comprises an input string, which becomes the argument vector, and a desired output string from an oracle version of the program variant. We discuss this algorithm design with respect to the C programming language but it can be applied to any language with suitable support for symbolic analysers.

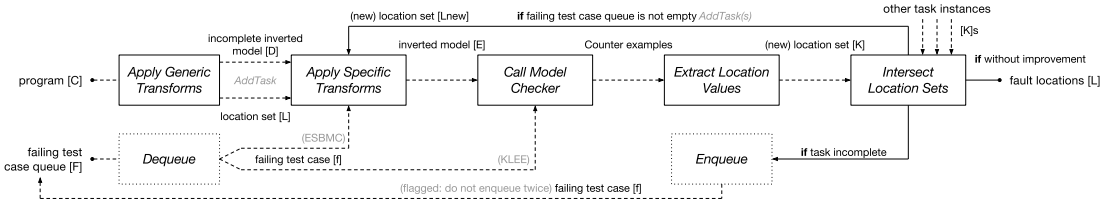

Figure 3.4: Diagram of the data flow through the algorithm architecture

We transform an input C program using an extended Griesmayer inversion process detailed in subsection 3.2.1. This is handed to a worker task (see subsection 3.2.2) with a whitelist of locations to search and a single failing test case. The returned set from this worker is a narrowed whitelist of locations that the test case flagged as being a potential repair. We manage this process using the main tool loop (see subsection 3.2.3). In the algorithms below we denote C programs as C and (after transformation) D and E. Individual failing test cases are f from the non-empty queue F. The refining whitelist of locations is L, which is narrowed during figure 3.6 to generate the smaller whitelist K as output. In the Process Manager algorithm, P is the worker pool manager.

The diagram in figure 3.4 shows the data flow through the tool architecture. The initial inputs, C and F, and the final output of a set of fault locations are noted entering and exiting the tool via circular dots at the extreme left and right of the diagram. Arrows denote the flow of data between functions, which are explained in detail in the algorithm descriptions through section 3.2. The data objects are listed above or below the arrows with the algorithm symbols noted in square brackets. To highlight the transition from the Pool Manager to the Test Case Search task, calls to *AddTask* have been noted on the flow. As described in subsection 3.2.3, the management of the queue, F, is noted via functions in dotted boxes and the decision made after intersecting the returned sets, Ks, is highlighted by solid flow lines on the different paths with the conditions noted after the `if` labels. The difference in failing test case use between KLEE and ESBMC, as explained in subsection 3.2.1 and subsection 3.2.2, is noted with bracketed labels for a split flow line out of *Dequeue*.

We provide two main contributions with this algorithm design. First, the reduction in symbolic execution work via the use of a narrowing process of whitelisting locations

searched. Second, the management of cases where the time to return a narrowed whitelist is significantly beyond the mean time for a failing test case. This later case can be due to either an intractable model representation or poor narrowing compared to other unexplored failing test cases. This management is designed to provide a more consistent completion time over a range of different localisation searches. This aims to avoid slowdown from the highest cost branches of the search rather than focussing on providing an optimal search when all branches are cheap to search.

3.2.1 Extended Inverted Model Transformation

The program under analysis is transformed in two stages. First, an initial generic transformation *ApplyGenericTransforms* is applied once, as described in subsection 3.2.3. This includes the Griesmayer inversion, as outlined in section 3.1, and some accommodation of test case input and output data as inline specification. In the second stage *ApplySpecificTransforms* is applied, as described in subsection 3.2.2, for each subsequent test case processed. This implements any whitelist narrowing possible at this iteration via the toggle; for ESBMC it also includes some test-case-specific input data encoding into the program text.

When we introduced our worked example in section 3.1, figure 3.1 was already specified. A more faithful form of the student coursework submission it is based upon is given in figure 3.5. This is the source that our tool must be able to work with to provide a fully automated tool-chain in chapter 5.

```
1  #DEFINE BR 15
2
3  //normalHours, overtimeHours, age
4  int payMe(uint nH, uint otH, uint age) {
5    uint bonus = BR * (otH * 2);
6    uint normalPay = BR * nH;
7    uint ageHigher = age % 20;
8    uint SelBR = BR + ageHigher;
9    uint SelBonus = otH * (3 * SelBR);
10   uint SpecialNormalPay = nH * SelBR;
11   uint specialPay = BR + 20;
12   if ((age > 20) && (age < 40)) {
13     return SpecialNormalPay + SelBonus;
14   } else if (age >= 40) {
15     return (nH * specialPay) + ((specialPay * 2) * otH);
16   } else {
17     return normalPay + bonus;
18   }
19 }
20
21 int main(int argc, char ** argv) {
22   int result = payMe(atoi(argv[0]), atoi(argv[1]), atoi(argv[2]));
23 }
```

Figure 3.5: Worked transformation: Original faulty code

The previously discussed Griesmayer inversion is always applied during the generic stage, *ApplyGenericTransforms*. Also at this stage, we automatically encode the test case

specification into the input program, avoiding the need to manually hard-code the inputs into the program. The input and output of C programs are defined by the passed program arguments, `argv`, and the standard inputs and standard outputs. For the fully-automated execution of our tool, the input and desired output are encoded into the program via an expanded standard input; we widen `argv` to also accept the desired output as an extra string value. This can be compared in the transformed program against modified stdout commands, replacing calls to e.g. `printf` with comparisons that increment a pointer to the desired output when it matches the previous output of the `printf`. Assertions inserted before the program's terminal nodes, which confirm the pointer to the desired output has reached the end of the string, will complete this encoding of the specification. This is used in the generic implementation of automated specification insertion that our tool defaults to. It can be overridden with additional information about how to directly assert the desired output matches the final variable holding the result of the program.

An optimisation of this string replacement is used for simple programs, such as TCAS and our worked example, that only require a single integer value to be checked for output specification compliance. Rather than encode the output as a string that is walked, it can be handled in the same way most input integers are. A call to `printf("%d", val);` can be replaced with `assert(val == atoi(argv[X]));` for X being the last value of the now-widened input vector. A program that returns a value to the calling program or operating environment will expose `return` calls in `main` that can be prepended with a similar assertion. In our worked example from the coursework database, the test cases define the name of the variable that must store the output value. On line 22 of figure 3.5 the value of `result` is tested by the inserted `assert(result == atoi(argv[3]));` in figure 3.1.

Program transformations can extend the number of assignments visible for repair. For example, a return statement that calculates a result before returning it can be divided into two steps with a temporary variable. Thus, `return a+b*c` becomes `temp=a+b*c; return temp`, exposing the expression to an assignment-based repair. If returns are always considered to be implicit assignments then localisation is expanded from assignment locations to also include all return statements. On lines 13, 15, and 17 of figure 3.2 this can be seen applied to expose the implicit assignments of the expressions returned without needing to make the temporary assignment explicit.

On line 1 of figure 3.5 there is a preprocessor statement `DEFINE BR 15`. This is automatically integrated into the C program from the C Pre-Processor, where it could not be reasoned on by the symbolic analyser as a single global value. This substitution into line 2 of figure 3.1 checks the scope of the global to ensure a single assignment covers all uses of the token. This extension of the transformation process allows the symbolic analysis of this value to test it as a site of potential repair and can be mapped back to the original `DEFINE` by the reporting tool.

During the specific stage, i.e. *ApplySpecificTransforms*, the whitelist of locations must be applied to narrow the range of toggles being searched. We do this by adding assumptions of the form `assume(__t != 7);` immediately after `__t` is assigned a symbolic value, blocking searching the assignment to, in this example, `SpecialNormalPay` on line 10 of figure 3.3.

For ESBMC there is no way to populate `argv` in the program simulation, so the `argv` values need to be hard-coded into the transformed program before handing to the symbolic analyser. As this is linked to the specific test case, this can only be done at the specific stage. In our worked example, line 10 of figure 3.3 would be modified during this transform pass to swap out the `atoi(argv[X])` calls with the values from the test case being tested. KLEE provides a POSIX implementation for program I/O during simulation. This allows the test case input to be fed into the standard `argv` parameters and removes the requirement for this specific stage transform, figure 3.3 would be used as written.

3.2.2 The Test Case Search Algorithm

The algorithm in figure 3.6 outlines a single task, which defaults to being deployed to a single core via the pool manager discussed in subsection 3.2.3. When *AddTask* is called on lines 5 and 24 of figure 3.7 by the manager then an instance of this single unit of work is queued into the worker pool. These tasks run independently until they return their narrowed list of locations, K, on line 9 or are ejected by the pool manager. Each task takes an input program which has already been transformed by the generic stage, a single failing test case, and a whitelist of locations that are indicated by their associated toggle values.

Input: Program D; Failing Test Case f; Location Set L
Output: Fault Location Set K
1: E = ApplySpecificTransforms(D, L, f);
2: $CounterExamples$ = CallModelCheckerOnInput(E, f)
3: K = [];
4: **for** c in $CounterExamples$ **do**
5: **if** c.UnconditionalAssertionFailure() **then**
6: K.Add(ExtractLocationValue(c));
7: **end if**
8: **end for**
9: **return** K;

Figure 3.6: Test case search worker algorithm

ApplySpecificTransforms is discussed in subsection 3.2.1. The final transformed program is generated, ready for submission to the symbolic analyser. The toggle values are restricted to the whitelist and, in the case of ESBMC, the test case is hard-coded into the source code.

CallModelCheckerOnInput passes the transformed program to the model checker. For ESBMC the failing test case has been encoded into the source code (in *ApplySpecific-Transforms*) but KLEE still requires this information. KLEE, using the POSIX runtime environment model, is passed the argument vector (extended to contain the desired output) during the simulation of the program execution.

The call to *CallModelCheckerOnInput* returns the counter-examples for this failing test case, which are a set of traces that result in the raising of specification failure when simulating the execution of the transformed program. The traces are parsed into a format that holds the failure type and the associated assignment to `__t` and `symbolic()`. In the case of ESMBC, this process is iterative as only one counter-example is generated by each instantiation of the symbolic analyser. For ESBMC, a loop that modifies the input program to further narrow the toggle values explored allows the symbolic analyser to run repeatedly until no new toggle value is generated. This, executed inside the call to *CallModelCheckerOnInput* in figure 3.6, generates a full list of toggle values associated with repairs for this failing test case. KLEE does this loop automatically within a single call.

On lines 3 – 8 the old whitelist is replaced with the new list of toggle values returned by the counter-examples, generated by the symbolic analyser execution. This narrowing checks the counter-example type to ensure the assertion raised is the `assert(0);` added before the program's terminal nodes.

The time it takes to process this task is not predictable with any degree of certainty. The core operation of calling the solver inside the symbolic analyser is a logical satisfiability problem, which is NP-complete [21]. This type of SAT problem has been shown [17] to not provide predictable tractability. This unpredictability of the time it takes to process each logic expression in the symbolic analyser is the core issue that our algorithm must cope with. Each counter-example generated has a corresponding solver stage and there are an unknown number of counter-examples multiplying this unknown per-instantiation processing time.

Each counter-example generated increases processing time and so a significantly narrowed whitelist, which blocks off many counter-examples, is highly beneficial. We manage uncertainty via the pool manager and ensure that narrowing results are percolated to new tasks as soon as possible.

3.2.3 The Pool Manager Algorithm

The main tool loop, which manages the process pool and task scheduling, generates an output set of viable repair locations (lines 29 and 19 of figure 3.7). The input is an untransformed C program and a non-empty queue of failing test cases, each of which comprises an input string and a correct output string. To provide our evaluation of

this tool with generalization validity, the queue's order is randomised. This is because the optimised search varies in performance based on the test case order, as discussed in section 4.2.

Input: Program C; Failing Test Case Queue $F \neq []$
Output: Fault Location Set L

```
1:  (D, L) = ApplyGenericTransforms(C);
2:  VisibleCores = Min(Len(F), OS.VisibleCores);
3:  P = EstablishWorkerPool(VisibleCores);
4:  for 1 .. VisibleCores do
5:      P.AddTask(D, Dequeue(F), L);
6:  end for
7:  WithoutImprovement = 0;
8:  while P.HasOpenWorkers() do
9:      Sleep(TickTimerMS);
10:     if P.HasCompletedTasks() then
11:         Sleep(0.25 * P.GetFastestCompletedTaskTime());
12:         for w in P.GetCompletedTasks() do
13:             Lnew = w.Locations ∩ L;
14:             if Lnew == L then WithoutImprovement++;
15:             else WithoutImprovement = 0;
16:             end if
17:             L = Lnew;
18:         end for
19:         if WithoutImprovement > 15 then return L;
20:         end if
21:         F.Enqueue(P.ReturnTCsForIncompleteTasks());
22:         for w in P.GetAllTasks(); do
23:             P.RemoveTask(w);
24:             if Len(F) > 0 then P.AddTask(D, Dequeue(F), L);
25:             end if
26:         end for
27:     end if
28: end while
29: return L;
```

Figure 3.7: Pool manager algorithm

ApplyGenericTransforms on line 1 applies the generic transform stage discussed in subsection 3.2.1. This parsing of the source file, as it walks all the fault locations being searched to apply the assignment transform, is also used to generate the initial whitelist of all toggle values that correlate to a localisation.

A worker pool is established on line 3, which provides the interaction point for all calls involving workers and the tasks they are executing or schedule for future execution. The tool queries the operating system to establish the multiprocessing pool is as wide as the exposed CPU core count, unless there are fewer failing test cases than available cores (line 2). The use of the worker pool is to avoid a single intractable task from stalling the entire search. This is most efficiently achieved on modern multi-core consumer hardware

by dedicating one core to each worker. A similar pool could be managed on a single-core processor using the OS scheduler to manage the tasks, with the added overhead of regularly swapping the current process.

Each worker will process an independent task (i.e. figure 3.6) which takes the transformed program from line 1, a failing test case, and the current whitelist. This will eventually deplete the test case queue. Lines 4 – 6 push an initial batch of tasks to the pool system, which will use the un-narrowed whitelist created on line 1. Batching tasks with a multiprocessing implementation increases throughput, even with an algorithm dependent on the pruning of the search space for efficiency. Critically, this prevents one slow task, whose individual contribution is not required for the search, from completely stalling the full search. A typical four-core CPU executing highly variable return-time tasks with probability p intractable outliers in the task queue will only stall the entire process when all cores are filled with intractable outliers at once. This probability of p^4 is a significant improvement, especially for this process where stalled tasks may become tractable later due to whitelist narrowing of the search space.

The main tool loop starts on line 8 of figure 3.7. The loop exits when the pool manager is not holding any completed tasks (waiting for their return value to be processed), no tasks are in flight, and no tasks are queued waiting to be started.

The pool manager thread sleeps to allow the pool's workers to monopolise the CPU, periodically waking to check if any tasks have completed with *HasCompletedTasks*. When at least one completed task is ready for retiring from the pool, the main thread waits for other tasks to complete. This waits a maximum of 25% of the time the fastest task completed with (line 11). This allows slower tasks with valuable narrowing results to complete and be added to the narrowing before the next generation of tasks is dispatched. The completed tasks after this time-out are iterated on lines 12 – 18. To prepare for the next generation of tasks to be dispatched, a new whitelist is created that includes all narrowing returned from the completed tasks during this generation.

A counter, *WithoutImprovement*, is incremented if the returned narrowing does not prune the existing set. This will eventually trigger an early termination clause (line 19) when the narrowing process has stalled for many failing test cases in a row. This provides enhanced time performance with larger failing test case sets, without harming the narrowing performance, as the larger sets have more redundant (for narrowing) test cases.

Any task which has been flagged as failing to complete in a time consistent with the others, missing the 125% dynamically assessed expected time, is queried on line 21. The failing test case it failed to complete is added back to the tail of the queue. If the whitelist is smaller when this task comes back up then it can be rescheduled, with the reduced search space increasing the likelihood of a fast completion time. Test cases are flagged as repeats so they are not enqueued a third time if they failed to complete a

second time. This protects against intractable tasks which will not provide narrowing data. All of the tasks from a generation will have now been processed, so they are ejected from the pool (line 23). The next batch of tasks is despatched to the pool (line 24) with the newly narrowed whitelist, if there are still failing test cases in the queue.

This generational search process, with percolated combined narrowing, limits the explored search space to relevant branches where a find is still viable and reduces the symbolic execution work for the solver. Some tasks may still be intractable so, to prevent them slipping through any cracks in this process, a global timer that must be configured to denote what is an "unreasonable" processing time is established that ejects tasks that fail to complete in that time. This will only be triggered if all active workers are stalled with intractable problems but does require configuration of what is an unacceptable wait.

Scheduling this task pool over a standard consumer multi-core CPU with these guards against search stalls and early rejection of superfluous searches provides significant performance improvements, as indicated by our preliminary results in chapter 4.

3.3 The Repair Algorithm

The fault localisation inverted model search refined above provides assignments where a repair is possible but does not provide a high-quality repair, only the indication that such a repair exists. Reversing the assumption made in mutation testing (cf. subsection 2.4.3), we researched the possibility that a mutant of a faulty assignment could repair the program, ensuring all test cases passed. Using the same optimised inverted search as above, our repair approach replaced toggle values selecting between potential repair locations with toggle values for various mutants of the expression previously localised as potentially being a repair site. The abstract syntax tree (AST) of the expression where a repair was being searched for was modified to index into tables of function and variable pointers that replaced the (unary or binary) operation and variables or constants in the expression.

Although ultimately a negative result, as discussed with results in section 4.5, we also explored an extension of the inverted model-based localisation to generate a search of potential repairs from a range of synthesised expressions. We aimed to provide a repair synthesis tool which would use the same general inverted search form, with our accelerated search improvements to improve tractability, generating high-quality repairs that could be validated by the programmer before integration into the repaired program without the need for human repair synthesis. This would be provided while still only using the test suite as specification and being end-to-end fully-automated. This would have been a significant contribution to the field discussed in section 2.7, going well beyond

the low-quality repairs in the form of the look-up tables our tool already generates as a byproduct of this specific model-based localisation technique.

We developed a tool that uses a mutation of the existing, defective expression in the assignment as the foundation for creating mutated expressions which may repair the program for all passing and failing test cases. This searches the space of expressions similar to the programmer's assignment attempt and will cover several classes of mistake. For example, several typos, including those made syntactically valid via autocomplete, or mixing up variable names or operators when constructing complex expressions to deadline.

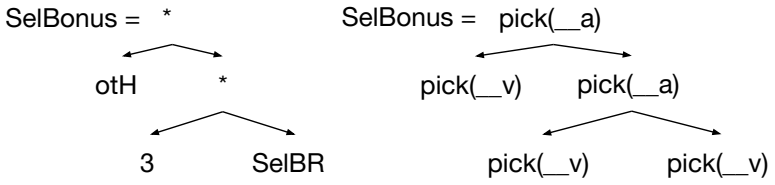

Figure 3.8: AST of transformed expression

Repair transformations are applied on a per-location basis, rather than the localisation transformation which is applied at all viable locations at that stage of the search (the whitelist locations). This process is called at each assignment location where a viable repair was reported to exist by our localisation tool, generating a search of the potential mutants that can repair all test cases. The process uses all test cases, not just the failing ones, as passing test cases will also define the different mutated expressions that can keep the passing test case working.

In figure 3.9, the specified code in our worked example has been transformed to search for a high-quality repair at the assignment on line 7 of figure 3.1. The repair strategy is to retain the form of the original faulty assignment expression but mutate each binary operation within a family of similar operations. Each variable is similarly mutated in-place with an open choice of any variable or literal in scope at that point in the program. In figure 3.8, this expression is shown as the AST representation and how a toggle (pick) is applied at each node referencing into either a table of binary operation function pointers, `__a`, or table of variable pointers, `__v`.

The repair mutation space is defined by a set of toggles, given symbolic values, that index into arrays from which the mutation is selected. This provides the symbolic analyser with the full mutation space. The binary operations are chosen from lists of functions generated by the transformation to match the required prototype and return the expression. In figure 3.9, lines 1–7 define the binary operations available for this mutation with `__a1` and `__a2` being the toggles made symbolic and restricted on lines 10 and 11. The variables are chosen from those which are in scope at that point in

```
1  typedef uint(*funcform)(uint, uint);
2  static inline uint fnmult(uint x, uint y) { return x*y; }
3  static inline uint fndiv(uint x, uint y) { return x/y; }
4  static inline uint fnadd(uint x, uint y) { return x+y; }
5  static inline uint fnminus(uint x, uint y) { return x-y; }
6  static inline uint fndiscardy(uint x, uint y) { return x; }
7  static funcform __a[5] = {fnmult, fndiv, fnadd, fnminus, fndiscardy};
8
9  int payMe(uint nH, uint otH, uint age) {
10   uint __a1 = symbolic(); assume(__a1 < 5);
11   uint __a2 = symbolic(); assume(__a2 < 5);
12   uint __v1 = symbolic(); assume(__v1 < 11);
13   uint __v2 = symbolic(); assume(__v2 < 11);
14   uint __v3 = symbolic(); assume(__v3 < 11);
15   uint BR = 15;
16   uint bonus = BR * (otH * 2);
17   uint normalPay = BR * nH;
18   uint ageHigher = age % 20;
19   uint SelBR = BR + ageHigher;
20   uint __l1 = 2, __l2 = 20, __l3 = 3;
21   uint *__v[11] = {&nH, &otH, &age, &BR, &bonus, &normalPay, &ageHigher,
22                                                &SelBR, &__l1, &__l2, &__l3};
23   //uint SelBonus = otH * (3 * SelBR);
24   uint SelBonus = __a[__a2](*__v[__v3], __a[__a1](*__v[__v1], *__v[__v2]));
25   uint SpecialNormalPay = nH * SelBR;
26   uint specialPay = BR + 20;
27   if ((age > 20) && (age < 40)) {
28     return SpecialNormalPay + SelBonus;
29   } else if (age >= 40) {
30     return (nH * specialPay) + ((specialPay * 2) * otH);
31   } else {
32     return normalPay + bonus;
33   }
34 }
35
36 int main(int argc, char ** argv) {
37   int result = payMe(atoi(argv[0]), atoi(argv[1]), atoi(argv[2]));
38   assume(result == atoi(argv[3]));
39   assert(0);
40 }
```

Figure 3.9: Worked transformation: Repair model of specified code

the program. Lines 12–14 define the toggles for these and line 21 defines the table of pointers to be searched.

When an integer literal is found in the expression to be mutated, or other expressions in that code block, they will have no memory address to dereference so a temporary variable is created to provide access to these substitutions. Line 20 demonstrates this, with the addresses added in line 21. The literal '15' (from line 15) is not added to the table as this would be duplication of `BR` and a simple check ensures these duplicates are avoided to reduce the mutation space size when it does not increase the actually different repairs expressed. Pointers are used rather than an array of copied value to best preserve the behaviour and access order for the data, as the table of variables cannot be generated at the same point as they are used and C expressions can have side effects.

On line 24 of figure 3.9 the assignment being repaired is rewritten as references to the table of appropriate functions being substituted and values in scope to be swapped in. Here `3 * SelBR` has first been transformed into a mutation space defined by `__a1`, `__v1`

and __v2, then this is one of the parameters passed to the call using toggle values of __a2 and __v3 to encode otH * ([...]). This allows the repair space to cover the form of the original expression but with any variable/constant and binary operation swapped out from the tables of options. Different expressions which use unary operators or different binary operations, for example boolean logic operators, can reference into different tables of function forms not shown here.

As with localisation, the specification has been inverted with an identical process. As this process is built on the same algorithms as our localisation refinement process, this is broadly the same as that discussed in chapter 3. We will not go over the parallelism of the process as this is an identical algorithm and the choices made for that implementation are guided by the performance characteristics seen during mutation space refinement with the risk of degenerate cases and a high variability in return times based on which test case is under test.

However, maintaining a whitelist for a combinatorial mutation space can quickly grow with transformations of expressions using many parameters or with programs which expose many variables to the entire scope of the program. Writing an assume, such as assume(__a1 != 2 && __v1 != 1 && __v2 != 4), for each found repair for a single test case quickly introduced too much complexity to the generated solver problem when using ESBMC. We needed to test different forms of assume blocking to most efficiently define the shrunk search space of mutations not already found to repair the test case being searched.

```
1  switch(__a1) {
2    case 0:
3      switch(__v2) {
4        case 0: assume(__v1 != 2 && __v1 != 3 && __v1 != 20); break;
5        case 7: assume(__v1 != 7 && __v1 != 15); break;
6        case 8: assume(__v1 != 1 && __v1 != 4 && __v1 != 18); break;
7        case 10: assume(__v1 != 3 && __v1 != 4); break;
8      }; break;
9    case 2:
10     switch(__v2) {
11       case 0: assume(__v1 != 5 && __v1 != 11); break;
12       case 1: assume(__v1 != 5 && __v1 != 10); break;
13       case 3: assume(__v1 != 7); break;
14     }; break;
15   case 3:
16     switch(__v2) {
17       case 0: assume(__v1 != 1 && __v1 != 4 && __v1 != 8); break;
18       case 1: assume(__v1 != 0 && __v1 != 4 && __v1 != 18); break;
19       case 3: assume(__v1 != 2 && __v1 != 5 && __v1 != 11); break;
20     }; break;
21 }
```

Figure 3.10: Tree of blocking assumes to narrow found counter-examples

Branching, as seen in figure 3.10, is required to rapidly iterate over the potential mutation space without creating thousands of asserts or assumes that are unconditionally called, without which the processing time for an ESBMC conversion to solver problem can explode. The use of assumes to block as soon as all toggle variable are defined, rather

than selective asserts at the tail of the program traces (only exposing an unconditional fail when a whitelist toggle combination is met), may terminate the path search for KLEE more rapidly and so reduces the time before those paths are rejected. Both of these choices are critical to avoiding exploding processing time when moving from localisation with tens of locations to mutations with hundreds or thousands of different options to potentially block.

Even with these optimisations to ensure no new tractability issues are introduced, when tested against anything larger than a toybox scale repair problem, this technique fails to provide results in times comparable with a concrete search of the mutation space. The iterative process for gathering repair mutations leads to a symbolic approach and per result time costs with effectively concrete approach individual results except in cases where only a very small percentage of the mutation space is viable to repair any single test case. This lack of narrowing on a much large search space than the localisation problem is common to typical real-world code, where each test case often has a wide range of repairs possible. We discuss this further and draw conclusion in section 4.5, where we also present results from our tool.

3.4 Conclusions

We have developed a smart search process to significantly reduce the processing time of a test-driven, model-based localisation process. This provides two main contributions. First, the reduction in symbolic execution work via the use of a narrowing process of whitelisting locations searched. Second, the management of cases where the time to return a narrowed whitelist is significantly beyond the mean time for a failing test case. This management is designed to provide a much faster and more consistent completion time over a range of different localisation searches, allowing this technique to compete with spectrum-based methods [RQ1] without sacrificing accuracy [RQ2].

This generational search process, with percolated combined narrowing, limits the explored search space to relevant branches where a find is still viable and reduces the symbolic execution work for the solver. Scheduling this task pool over a standard consumer multi-core CPU with these guards against search stalls and early rejection of superfluous searches aims to provide significant performance improvements, which we later confirm in chapter 4. We go on to test this design against a large set of actual C programs [RQ2, RQ5] in chapter 5, including expanding this narrowing search to enable localisation outside of a single-fault assumption [RQ4].

We also propose an expansion to this inverted model which searches for mutants of localised assignment expressions which may form a high-quality repair for the program under test [RQ3]. This is later tested in section 4.5.

Chapter 4

Fault Localisation and Repair on the Siemens Suite

We initially demonstrate the time and localisation performance of our tool on the Siemens test suite's TCAS program and test universe taken from the Software-artifact Infrastructure Repository (SIR) [30]. For the single-fault variants, we maintain time performance within the same order of magnitude as the current model-based fault localisation state of the art, Jose and Majumdar. We guarantee returning the location of the injected fault in every failing case, which Jose and Majumdar cannot. For 31 of the 33 single-fault variants we improve on the localisation performance of Jose and Majumdar's results.

4.1 Data Set

TCAS is a 173 line C program from which 41 variants have been generated by seeding (injecting) faults. Of these, 33 variants have been seeded with a single fault and exhibit at least one failing output with the test suite of 1608 test cases. These provide meaningful interpretation when comparing the performance of localisers with a single-fault assumption.

Inside the Siemens suite of programs, which share the test universes and fault-seeded variant format, there are six other programs on a similar scale of hundreds (400-750) of LoC. These are also used in the literature to establish localisation performance. Of there programs, *tot_info* has calls to the external C library libm and so is not being considered at this point for testing but should be tractable if these external calls were redirected to local implementations of the required functions (a limitation of KLEE's use of LLVM IR for input). *Schedule*, as seen in the 11 hour completion time in a test

typically only taking seconds others have published [61], provides issues with tractability when modelled and has not provided any successful results for our tool.

The other four programs are *print_tokens*, *print_tokens_2*, *replace*, and *schedule_2*. These programs all use ASCII input/output as part of their specification and include loop or recursive calls (including while loops on strings). They all reference tens of global variables, except for *print_tokens* which contains large global arrays and so exposes over 800 primitive type variables at all points in the program. They each provide between 7 and 32 seeded variants and come with a universe of 1000 to 5500 test cases per program. All make use of C's type system (beyond *TCAS*'s use of only integer variables) but *print_tokens* and *schedule_2* have a complex type system (including structs). The arithmetic binary operations that *TCAS* lacks are well covered in these other programs. The C standard library is exercised far more thoroughly via these programs, especially leaning on the ASCII functionality that *TCAS* does not require (outside of `atoi`).

4.2 Experimental Setup

In table 4.1 and table 4.2 the headings refer to the following data sources and test platforms: Griesmayer's original data [39, §4, Table 1, p. 104] (**G**) uses CBMC on a 2.8GHz Pentium 4 and our naïve reimplementation of Griesmayer's algorithm (**N**) using ESBMC v1.17 on a 3GHz Core2Duo E8400. This reimplementation on a more modern Intel CPU and a recent SMT-based solver is designed to explore the potential performance delta caused by an outdated version of CBMC and the Pentium 4. Our new algorithm using ESBMC (**E**) and KLEE (**K**) as back-end both ran on a 2011-era 3.1GHz Core i5-2400, with the ESBMC [23] v1.21 and KLEE [15] (for LLVM 3.4) symbolic analysers. Jose and Majumdar's results (**J**) are reconstructed from data provided [61, §6, Table 1, p. 443] using MSUnCORE on a 3.16GHz Core2Duo. Boldface entries in the table represent the best performance, underlined entries indicate failure to return the injected fault location for all failing test cases. In table 4.1 the Jose and Majumdar time data has been calculated by taking the number of executions per test case and multiplying by the reported average time to complete a single execution of a failing test case.

We shuffle the test suite to randomise the order of the failing test cases when invoking our tool. This prevents the performance reported by our current tool only reflecting the time performance when provided with the default test case order. The time performance reported is the average of ten runs.

4.3 Model-Based Fault Localisation Results

4.3.1 Run-time Performance on TCAS

Griesmayer et al. provided results on TCAS using state of the art (for the time) model checking tools (CBMC) but indicated the design had not been optimised, saying "we do not concentrate on performance" [39, §4.1, p. 105]. We reimplemented this naïve process as described in the Griesmayer et al. paper. This is running on more modern hardware and updated to use the current, CBMC-derived, SMT-based symbolic analyser ESBMC. We implemented automatic specification encoding into the tool to hard-code test cases. This tool iterates over all failing test cases, waiting for the symbolic analyser to return all flagged locations. The tool then returns the common locations flagged by all the failing test cases.

	G	**N**	**E**	**K**	**J**		**G**	**N**	**E**	**K**	**J**		**G**	**N**	**E**	**K**	**J**
v1	2953	1442	9.0	4.5	**2.1**	v14	594	101	3.2	**1.4**	**1.4**	v26	311	114	3.4	2.1	**1.2**
v2	836	678	3.7	**3.2**	4.7	v16	1263	746	8.8	**3.9**	7.3	v27	153	107	3.3	2.3	**1.1**
v3	423	240	6.7	3.6	**2.2**	v17	1300	365	5.6	3.6	**3.4**	v28	642	711	**2.0**	2.7	6.1
v4	576	307	7.5	2.9	**2.7**	v18	499	188	3.7	**3.0**	3.6	v29	224	112	2.9	3.4	**1.7**
v5	159	106	3.2	2.3	**1.2**	v19	691	193	5.4	3.4	**2.1**	v30	939	508	3.9	**2.8**	3.7
v6	253	134	4.9	2.8	**1.3**	v20	748	196	7.4	4.3	**2.2**	v34	1906	790	4.9	**3.0**	7.7
v7	743	359	5.9	3.9	**2.6**	v21	585	197	6.7	3.7	**1.7**	v35	1069	711	**2.3**	2.8	4.6
v8	26	10	1.7	1.4	**0.1**	v22	223	42	2.8	2.6	**0.6**	v36	877	219	**2.1**	2.4	3.0
v9	114	72	2.0	2.1	**0.8**	v23	885	189	4.2	**2.8**	4.2	v37	822	729	**3.7**	4.1	**3.7**
v12	1664	727	5.0	**3.4**	11.5	v24	254	71	3.3	2.1	**0.6**	v39	66	8	1.0	1.5	**0.3**
v13	149	43	1.9	1.1	**0.3**	v25	68	8	1.0	1.5	**0.2**	v41	956	309	8.0	4.2	**2.4**

Table 4.1: Seconds to return location set for test suite. Griesmayer's original data [39, Table 1, p. 104] (**G**). Naïve reimplementation of Griesmayer's algorithm (**N**). New algorithm using ESBMC (**E**) and KLEE (**K**) as back-end. Jose and Majumdar's results [61, Table 1, p. 443] (**J**)

The average halving, at most six-fold, decrease in completion time from Griesmayer's results (696 seconds average) to our naïve reimplementation (325 seconds average) in table 4.1 shows some performance increase is derived from using a modern symbolic analyser on modern hardware. But, for example, variant 1 moving from over 49 minutes to over 24 minutes to return a location set is not viable compared to the 2.1 seconds of Jose and Majumdar or comparable with our optimised algorithm when also using ESBMC, at 9 seconds.

We have implemented our algorithm as tools interfacing with ESBMC or KLEE. As discussed in section 3.2, this is designed to maximise consistency and avoid worst case processing time, as well as reducing the symbolic execution burden to improve times. We provide run-time numbers for our algorithm using an ESBMC and KLEE back-end in table 4.1. This indicates that using KLEE is often somewhat faster, compared to the

ESBMC back-end, but the use of KLEE as the symbolic analyser is not a major factor in the orders of magnitude time performance gap between the naïve reimplementation of Griesmayer et al. and our algorithm with a KLEE back-end.

We maintain time performance within the same order of magnitude as the current model-based fault localisation state of the art, as presented by Jose and Majumdar, throughout the singe fault TCAS variants, marginally beating their times in ten of the 33 variants. Our tool, using the KLEE back-end, averages a completion time of 2.87 seconds per TCAS variant, compared to an average of 2.80 seconds in Jose and Majumdar's results. The ability of KLEE to scale to larger input programs offsets the few instances where it does not lead our tool's results compared to the ESBMC back-end.

These preliminary results support our claim that a Griesmayer-derived model-based localisation technique can be modified to be fast, comparable to the current alternatives when localising on some small integer programs such as the TCAS variants. Using intelligent pruning of the search space to minimise the symbolic execution load while minimising the disruption of a slow or intractable search node is facilitated by a multi-process design that takes advantage of modern consumer processor architectures.

4.3.2 Localisation Performance on TCAS

The scope of the localisation of a tool quantifies which locations are being searched by the process and flagged as a potential fault. Different localisation scopes for each technique's implementation means their localisation performance is not precisely comparable. The results published by Griesmayer et al. only explore the 34 explicit assignments in the TCAS variants, which increases the localisation performance we would expect to see in table 4.2 as there cannot be more than 20% of the total lines of code returned. Our naïve reimplementation has an expanded scope that finds implicit assignments within the source code, expanding the potential locations returned to 43 assignments, or 25% of the source lines. This accounts for the weak, for Griesmayer-derived, localisation performance. The localisation results for our algorithm using the ESBMC back-end are omitted, but were noted to fall in line with the original Griesmayer et al. results and our current numbers with KLEE. Our current tool, using a KLEE back-end, does not apply all implicit assignment transforms implemented in our naïve reimplementation, only implementing the transforms described in subsection 3.2.1. This reduces the assignments tracked to 39, or 23% of the source lines.

We can conceptualise the Griesmayer-derived searches as building a look-up table for each assignment location returned that, if complete, repairs all failing test cases. Passing test cases already have known correct values for their assignments. Any location flagged by a failing test cases will have, in the `symbolic()` value extractable from the counter-example, the assignment which repairs that trace and so test case. It is thus possible,

	G	N	K	J		G	N	K	J		G	N	K	J
v1	8.7	10.4	**7.5**	8.6	v14	**2.3**	2.9	2.9	8.1	v26	4.6	6.9	**4.0**	9.2
v2	**2.9**	**2.9**	**2.9**	4.6	v16	8.7	10.4	**7.5**	9.2	v27	4.0	8.1	**3.5**	10.9
v3	**4.0**	8.7	5.2	9.8	v17	2.3	**1.7**	2.3	9.2	v28	**1.2**	**1.2**	**1.2**	5.7
v4	8.7	11.0	**8.1**	9.2	v18	**2.3**	**2.3**	**2.3**	6.9	v29	**1.7**	2.3	2.3	5.7
v5	4.0	8.1	**3.5**	8.6	v19	2.3	**1.7**	2.3	9.2	v30	**2.3**	2.9	2.9	5.7
v6	7.5	11.0	**6.9**	8.6	v20	8.7	12.1	**8.1**	9.2	v34	4.0	5.8	**3.5**	8.6
v7	2.3	**1.7**	2.3	9.2	v21	8.7	12.7	**8.1**	8.6	v35	**1.2**	**1.2**	**1.2**	5.7
v8	11.0	16.8	10.4	**8.6**	v22	4.6	**4.0**	4.6	5.7	v36	**1.2**	**1.2**	1.7	2.9
v9	5.2	**4.6**	**4.6**	5.2	v23	5.2	**4.6**	**4.6**	6.3	v37	2.9	**1.7**	2.3	8.6
v12	**4.0**	4.6	**4.0**	9.2	v24	8.7	11.0	**7.5**	8.6	v39	4.6	**3.5**	4.0	6.9
v13	**5.2**	9.2	**5.2**	9.2	v25	4.6	**3.5**	4.0	6.9	v41	**8.7**	12.7	9.2	**8.6**

Table 4.2: Percentage of lines of code returned by localisation; see table 4.1 for legend

for each location flagged by all failing test cases, to construct a complete look-up table that ensures every test case now has a specification-complying trace, i.e. a genuine repair exists at that location. In our results, the injected fault location is always included in the locations returned. But, with this conceptualisation of the process, the other locations are not false positives but additional locations where a genuine repair exists that will allow the test suite to pass, to the limits of the symbolic analyser's accuracy.

All our results confirm roughly comparable localisation performance between the various Griesmayer-derived methods, after accounting for the differences in localisation spaces. Any performance regression in localisation performance, when comparing the original Griesmayer et al. results with our Griesmayer-derived localisation results, is most likely the result of searching a wider assignment space. Localisation performance improvements are likely to have resulted from more modern symbolic analysers providing a more accurate exploration of the input C program, exploring new potential traces. Exact localisation performance, while a common metric for comparison on TCAS and in general, is slightly defocussed as a primary metric here. Each location returned by a Griesmayer-derived tool is based upon a possible program repair that remedies the injected fault. Evaluating the difference between Griesmayer-derived techniques, as they all operate to generate this family of locations with look-up table justification, is to penalise a tool for returning justified fault candidates. Each such location has a genuine repair, despite not being the fault injection location.

Jose and Majumdar, with an approach based on mapping MAX-SAT clauses back to source code, cannot be directly compared in terms of potential C code coverage. The mapping of the MAX-SAT output, from logic clauses in the maximum satisfiable result to source locations, can flag locations other than assignments. However, the granularity of this mapping is not clear. Some of the lack of competitive localisation performance in some variants shown in table 4.2 for Jose and Majumdar, when compared to Griesmayer

et al. or our algorithm can be explained by this different scope of potentially returned locations, where additional potential repair locations are being suggested outside of assignment modifications.

When comparing the localisation performances, even without being apples to apples, this is ultimately comparing sets of proposed fault sites where human developers must search for a genuine repair, possibly the injected one. Our current tool is typically ahead in this metric, sometimes by a significant percentage. In the two variants where our tool performs worse, *v41* returns a set of locations only one larger than those returned by Jose and Majumdar, and *v8* returns a set three locations larger.

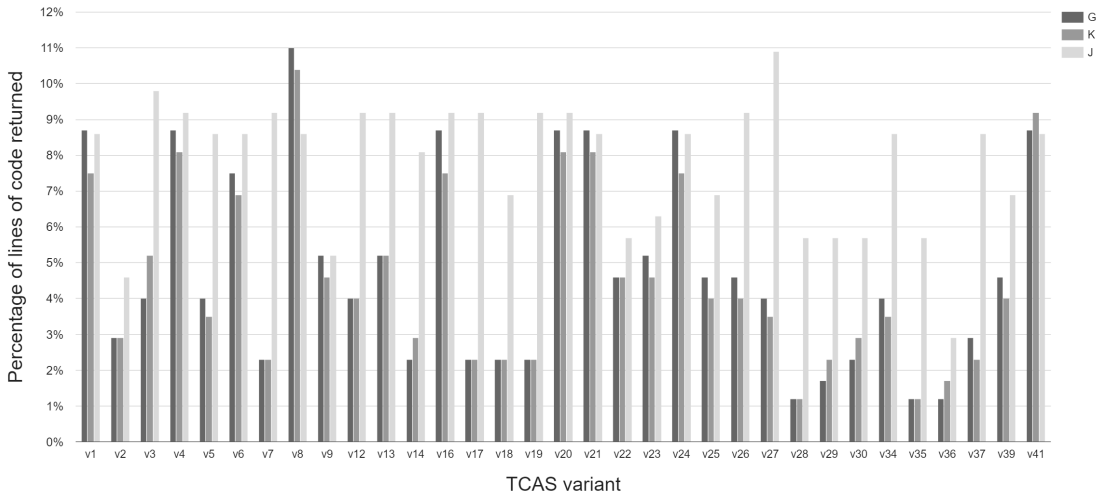

Figure 4.1: Chart of data in table 4.2; see table 4.1 for legend

When comparing the localisation performance of these tools, we must consider that there is an injected fault location for each of the single-fault variants of TCAS. For all the Griesmayer-derived techniques then the injected fault location (the location where a variant was seeded with a fault) is always included in the returned list of locations. Due to the technique's design (where a location is only returned if it is common between all individual failing test cases), this means that this injected location will also be returned when only given a single failing test case from any of the test suites and on any of the single-fault variants. Any subset of the test suite that contains at least one failing test case will, for all Griesmayer-derived tools, flag the injected fault location.

The results from Jose and Majumdar cannot make a similar claim. To account for some failing test cases not indicating the injected fault location, their final set is based on the most commonly indicated locations, not locations that are always flagged by every failing test case. Their results for each full test suite do flag the injected fault location for TCAS as most common. But they indicate that there exist subsets of the failing test cases for which they would not flag the injected fault in the case of six single-fault variants (underlined in the tables).

4.4 Spectrum-Based Fault Localisation Comparison

Comparing our results directly with those published previously which use spectrum-based localisation techniques, discussed in subsection 2.5.1, is even more imprecise than the above noted issues of narrowing comparability in subsection 4.3.2. The granularity of most spectrum-based localisation techniques is down to the line of code as this is how code coverage tools like GCov report their statement coverage output. All locations are ranked by suspiciousness using the various weighting formulas. The localisation performance is typically measured as the average percentage of the program that *doesn't* needs to be traversed by the human debugger before encountering the inserted fault. So a score of 100% means that the inserted fault in the variant is marked as the most suspicious, so a human debugger doesn't need to view any of the non-faulty code lines before encountering the injected fault. The language used also refers to each variant localised as a run, as reproduced in figure 4.2 and imported into our own results in figure 4.3 to retain consistency of labelling. The choice of label name, axis order, and score method is replicated here to retain consistency with earlier papers that establish this convention [19, 60].

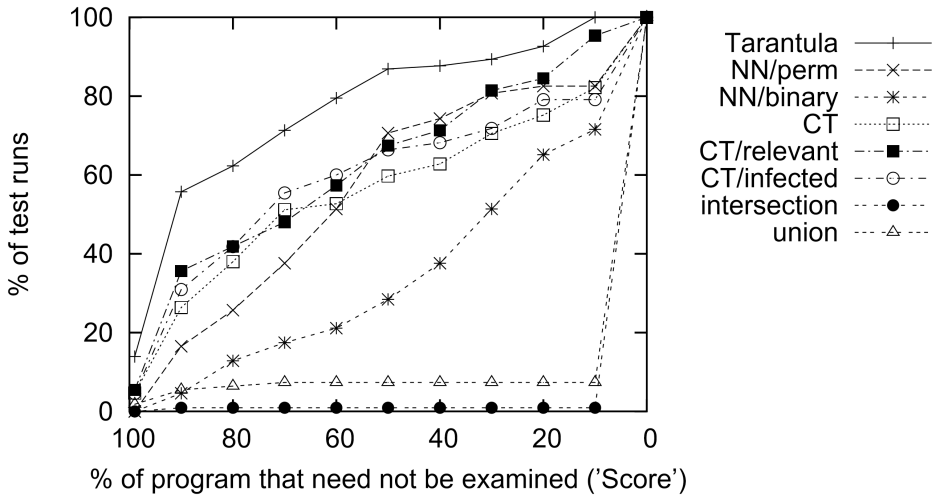

Figure 4.2: Comparison of effectiveness of spectrum based techniques reproduced from Jones and Harrold [60, Figure 2, §3.3]

Jones and Harrold provide results based on the full Siemens suite of programs which is visualised as a chart of the percentage of the program that need not be examined, i.e. the percentage of the program that is ranked as less suspicious than the injected fault location, against the variants that meet this performance criteria. There are a total of 132 different program variants over the full Siemens suite being summed here so the performance of only the 41 TCAS variants are a minority of the results reported. For this reason we have not overlaid our results onto the reproduced chart, as the 'test runs' (program variants) are not equivalent. It should also be noted that each of the lines on the reproduced chart are also referring to slightly different subsets of the total

132 program variants in the Siemens suite as those which cannot generate a result are rejected. Tarantula reports on 122 variants [60, §3.2.1], Cause-Transitions (CT) on 129 variants [19, §7.5], and Nearest Neighbour (NN) on 109 variants [60, §3.2.1].

The chart is constructed into decile buckets except for the initial 100% bucket as not traversing any of the program under test will guarantee that the fault has not yet been traversed. This is replaced by a 99% bucket for variants which have been localised with the injected fault in the top 1% of all locations and a bucket below it for those over 1% but under 10% (98–90% in the scale used). The comparison shows how Tarantula outperforms some contemporary (at the time) techniques and so this solid line is the one we wish to compare our results again. We have reproduced this in figure 4.2. It is clear that the narrowing, while excellent for a few and good for many when using Tarantula on the Siemens variants, does not reach a point of uniform guarantee before the majority of code must be searched. Some variants are localised with a result which ranks the true fault location as less suspicious than most of the rest of the locations in the program, a result worse than chance.

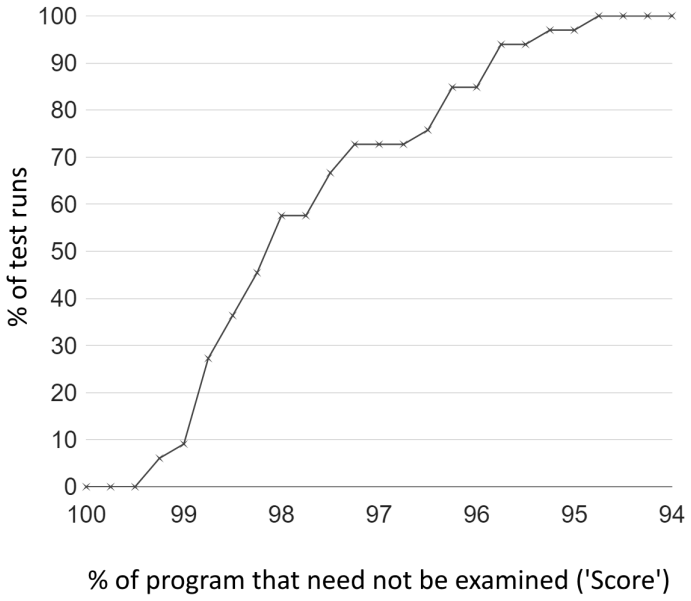

Figure 4.3: Effectiveness of our localisation techniques (partial x-axis)

Using the same method for creating a ranked list as later relied upon in section 5.2, we have converted our results using KLEE from table 4.2. We use a middle-line strategy, ranking equally suspicious statements with the mid-point rank. For example, statements with suspiciousness (0.1, 0.4, 0.8, 0.4, 0.4) are ranked (5th, 3rd, 1st, 3rd, 3rd). Results from our model-based tool have been converted from an unranked list using this strategy, where all locations returned are given a suspiciousness of 1.0 and all others a suspiciousness of 0.0. This effectively provides our results with a ranked narrowing of half the percentage of lines of code returned. This makes sense as the average performance of

our tool as an unranked subset of locations randomly shuffled to provide an arbitrary ranking will, if the fault is in that subset as it is with out localisation lists, rank at an average of half way down the ranking.

However, charting our results using the same decile buckets provides an uninteresting comparison as by the 90% plot, our tool has ranked every variant's injected fault inside of the list. An average traversal of a ranked list generated with our unranked conversion will have passed the injected fault before 10% of the lines have been looked at for every variant. Even looking at the absolute worst case performance of our tool on the single-fault TCAS variants, assuming our random ranking of the list of returned locations always put the injected fault location as least suspicious, we would still have reached 97% of the variants covered at the 90% need not be examined point on the chart.

We have therefore provided in figure 4.3 a zoomed in plot of only the leftmost 6% of the full chart range and have bucketed this data at every quarter percentage. This indicates how our model-based localisation results would be represented on the very leftmost sliver of figure 4.2 before running a straight line at 100% of the variants for the majority of the chart. To visualise the worst case performance of our tool on the single-fault TCAS variants, the x-axis can be doubled in length (so 100–88%) for the same curve. This should not be read as a definitive comparison and is provided with significant caveats. This data is not directly comparable due to variable code coverage criteria and because the chart from Jones and Harrold combines data from various large subsets of all the Siemens variants, not just the 33 single-fault TCAS variants we are charting. In chapter 5 we will present more valuable comparisons between spectrum-based and our model-based tool using a fixed data set.

4.5 Model-Based Fault Repair Results

Our proposed high-quality repair tool, which implements the technique described in section 3.3, executes our localisation tool to generate a list of locations in which an assignment expression has been flagged as potentially a repair site. It iterates over this list and all test cases to isolate any mutations to the expressions which repairs all failing test cases while continuing to allow all passing test cases to pass. When provided with a faulty program and a test suite then our repair tool is fully automated and uses very similar transformations to the localisation tool used above. This research was concluded before the completion of all the optimisations and refinements to our threaded search and represents a snapshot of a branch of inquiry that was ultimately not followed. As there are no situations where a concrete execution of the same search is anything but significantly faster than this approach, all further development was halted. This is provided for completeness and to publish our negative results.

	v1	v4	v9	v16	v17	v20	v25	v36	v39	v40	v41
Best	123s	67s	50s	191s	41s	225s	90s	37s	90s	73s	394s
Worst	438s	455s	328s	888s	42s	752s	570s	37s	552s	511s	764s
Median	245s	206s	110s	300s	42s	510s	240s	37s	172s	217s	554s
Mean	*227s*	*223s*	*133s*	*373s*	*42s*	*452s*	*368s*	*37s*	*210s*	*285s*	*546s*

Table 4.3: End-to-end repair results

Our tool, using ESBMC, synthesises repairs from a mutation space search and verifies that the proposed high-quality repair works under the specification of the full test case universe of the TCAS variants. Our search process relies on using an unordered selection of passing and failing test cases and so the different narrowing performances for each test case essentially randomise the final completion time. Unlike our results in subsection 4.3.1, the scale of the search space and variability of the narrowing here, potentially requiring thousands of mutations to be tracked at each repair location, creates a significantly larger range of run-time results. Any mutation space that was beyond thousands of potential mutants was rejected as intractable by an early termination clause to prevent the tool from failing to return any results. Variants for which we do not publish a result failed to generate the repair for the injected fault within a reasonable execution time and were terminated. In table 4.3 we provide, for a single-threaded end-to-end repair process, a typical response time with a median and mean as well as providing the fastest and the slowest complete run response time from a random selection of 10 runs. This should provide more informative than other reporting methods such as taking each run component and computing the fastest possible run via each minimum observed component completion time.

A prototype of the implementation of threaded optimisations which manages the variable tractability and run-time cost of each search outlined in section 3.2 was applied to our repair tool. In table 4.4 we provide our last results on end-to-end repair of TCAS using the full threaded performance of the tool, with the same statistics as previously used. Of note in the comparison to the results in table 4.3 are the generally large reductions in the worst case processing times to generate the same repair results. The algorithmic choices are tuned to provide better worst case performance at the price of less impressive averages and comparisons between the best case show that the general intractability of this repair process generally

	v1	v4	v9	v16	v17	v20	v25	v36	v39	v40	v41
Best	135s	56s	39s	148s	16s	151s	88s	13s	84s	44s	347s
Worst	252s	129s	110s	167s	118s	273s	270s	13s	177s	171s	519s
Median	152s	76s	44s	157s	18s	172s	99s	13s	100s	102s	406s
Mean	*166s*	*82s*	*53s*	*156s*	*28s*	*176s*	*123s*	*13s*	*105s*	*101s*	*404s*

Table 4.4: End-to-end threaded repair results

Variant 17 indicates that this is only lowering the chances of a worst case run, not eliminating it. If the randomised list of test cases fed into the tool input stacks non-reducing test cases heavily at the start then long processing times which do not refine the mutation space will provide significant performance issues; even when rolling four times as many dice with a four-core, threaded process they can all end up showing sixes. This indicates a limit of our threaded design when dealing with a search space where the majority of branches are very expensive to search and do not pay off with significant narrowing, if not outright intractable.

The competing solutions for end-to-end repair of the TCAS variants have all published incomplete data sets, indicating rejection of performance levels on other variants or inability to recover a genuine repair for all variants or even the subset of variants which have single fault repairs (variants with a single inserted mutation). This indicates that finding an acceptably performing high-quality repair process for the range of TCAS variants using the test suite as specification is still awaiting a first complete success, unlike the localisation process where the techniques are compared by speed and quality of result.

	v2	**v7**	**v8**	**v16**	**v17**	**v18**	**v19**	**v36**
Time	341s	165s	159s	166s	166s	162s	163s	failed

Table 4.5: Konighofer and Bloem repair results

Konighofer and Bloem provide results [63, Table 1, p. 98] using their mutation search of expressions based on A*x+B*y+...+Z for integer literals as capitals and every variable in scope (at the point of the assignment) as lower case elements. Their results, summarised with time taken in table 4.5, show that this only produced repairs that reject all variables in scope and repair the TCAS variants which can be fixed with a single integer literal, i.e. the Z in the above formula.

Test Cases	**10**	**20**	**30**	**40**	**50**
SemFix	12s	23s	30s	46s	52s
GenProg	27s	56s	85s	135s	164s

Table 4.6: Nguyen et al. average repair results

Nguyen et al. [82] provide results for their own method of SemFix as well as running the technique of Le Goues et al. (GenProg) to provide comparison results for TCAS unavailable in the original Le Goues et al. paper announcing their methodology [69].

These results are published for 90 of the 132 variants that make up the complete Siemens suite. Of these variants, with the caveat that the repairs are only validated against the subset of test cases explored, Nguyen et al. report SemFix finding 48 variant repairs using 50 test cases while GenProg only finds 16. These repairs increase to 56 for SemFix and 35 for GenProg when only using 10 test cases.

The data they provide on found repairs suggests the reported repairs are not necessarily valid over the full test suite specification but this is not conclusive stated. No per variant results are provided, but the averaged performance for the TCAS variants indicates SemFix proposes repairs for 34 of the 41 variants while GenProg proposes 11 when using 50 test cases.

Their publication provides a graphical representation of the time performance results for TCAS [82, Fig. 4, p. 779] using a limited number of test cases, up to a maximum of 50 of the 1608 provided by the universe file. We have reproduced these results in table 4.6. It is not made explicit if these repairs generated on the limited test case sets are genuine repairs that apply to the full specification given by the complete universe. Extrapolating the times out to running the process on the full set of test cases would not provide competitive performance if this is not so.

> "The success rate decreases with more [sic] number of tests. Intuitively, it is more difficult to generate a repair to pass more tests. It also shows that the repairs generated with small number of tests may not be valid for some other tests that are not in the test suite." [82, §6.A, p. 778]

In section 3.3 we describe a mutation space that is searched that does not include integer literals outside of those captured by exploring the expressions within the same block as the repair location. This is a symptom of the core weakness of this approach to high-quality repair synthesis: that a symbolic analyser is being used to generate a set of all concrete repairs. Our approach requires the full iteration over all viable concrete repairs on a per-test-case basis and then narrowing of this whitelist, as we do for localisation, but without low limits on the total number of search results (i.e. the toggled locations for localisation). When including integer literal values in the mutation space, this immediately becomes intractable as an iterated solution. For example, a repair where a test case is fixed if the assignment is any positive value would generate repair mutants for every positive integer value if it searched the integer literals as part of the mutant space. Even with small integers this is 32,767 repairs, with most C implementations expecting over two billion valid repairs be synthesised for this constraint.

This is similar to the process of symbolic search used during our localisation phase, only iterating over every potential symbolic value for each test case rather than blocking the toggle value (location) as soon as one working symbolic value is found. For this reason it would not be possible to use this technique to do a search for a repair that includes an unconstrained integer literal both due to tractability concerns and, when using early termination to avoid searching the full space, simply replicating the low-quality repair already generated by our existing localisation tool.

A potential solution to lack of narrowing of the mutation space is to run several test cases sequentially as a single solver problem so that the constraints on the repair are

combined in order to search for the unconditional assertion failure at the end of all programs explored. But this quickly leads to intractable solver problems as the scale of the program being tested can be visualised as every potential path through the first test case with a second test case attached to each end point. Multiply again with a third trace on the end of every one of those branches to simultaneously constrain for 3 test cases. We failed to generate a single result outside of time-outs for prototypes exploring such a solution to a lack of mutation space narrowing from searching a single test case at a time for every mutant.

Using a solver as an integrated part of the tool rather than a black box, as shown by Konighofer and Bloem, is another potential solution. However these results [63, Table 1, p. 98] to synthesise a high-quality repair of the form A*x+B*y+...+Z does not return a high-quality repair.

ESBMC is unable to process the programs other than TCAS in the Siemens Test Suite due to language features (loops and ASCII standard C library features) which generate intractable problems. Expanding to automated repair using KLEE requires competitive performance in this arena. To assess this requirement we tested the CPU time to compile (with Clang) and execute these small programs to concretely test all individual mutants (M#). We did not encounter a significant overhead when batching the compiled mutant executables using Python 3 with an attached stdin/stdout pipe.

	print tokens		print tokens 2		replace		schedule		schedule 2		totinfo	
M#	CPU	KLEE	CPU	KLEE	CPU	KLEE	CPU	KLEE	CPU	KLEE	CPU	KLEE
8	1.3s	3-19s	1.4s	2.5-11s	2.0s	2.5-2.9s	0.9s	t.o.	1.0s	2.5-3s	0.5s	N/A
40	6.7s	69-82s	7.0s	13-26s	9.9s	14s	4.3s	t.o.	4.8s	14-20s	2.3s	N/A
100	16.6s	4-6m	17.5s	41-49s	24.8s	40s	10.9s	t.o.	12.0s	41-56s	5.9s	N/A

Table 4.7: Concrete execution comparison results

In table 4.7 we present the six other Siemens programs against the processing time for KLEE to do a single test case exploration with a repair mutation space that covers M# different mutants. This is not a full end-to-end process or even a full repair stage but simply the time for a single thread to process a single test case with a mutation set to the required width somewhere in the `main` of that program, providing a mutation which will not be traversed more than once by a program loop. This mutant position and format is likely to be a best case situation for KLEE, defining a lower bound for the time to process a single test case. Using various test cases from the program's universe we have provided a range of times in which the mutations are searched.

	totinfo	**print tokens**	**schedule**	**schedule**	**totinfo**	**schedule 2**
Time	0.19s	25s	28s	11h	225s	20s

Table 4.8: Jose and Majumdar localisation results

The published results replicated in table 4.8, taken from Jose and Majumdar [61, Table 3, p. 445], indicate competing localisation results on these programs include an 11 hour execution for the *schedule* program, pointing to an issue with symbolic analysis and this program or even a minimised trace of it. Our results in table 4.7 are that KLEE had stopped working and hung on processing this input program, hence a time-out (t.o.).

The concrete times noted as CPU not only accounts for compilation of all mutants but also executing them for every single test case in the provided universe (450-5500 depending on the program), not just the one the KLEE numbers represent. These CPU numbers will scale almost perfectly linearly with mutation space and provide significantly better value at extracting the repair mutation from the search space with some concrete execution. At this point we consider a concrete execution iteration the best approach to searching a repair mutation space for a viable repair after localisation.

4.6 Conclusions

In this chapter we have presented our results demonstrating a several orders of magnitude reduction in run-time comparing the naïve reimplementation of the localisation algorithm with our optimised search, using the TCAS benchmark variants. We have extended this comparison to include a state of the art approximate model-based localisation technique and provide tentative comparisons to several spectrum-based localisation techniques [RQ1]. We have explained our negative results from the repair extension of this technique with analysis of why this particular approach cannot be accelerated to compete with a purely concrete search, which is an already-established repair search technique [RQ3]. Our positive results are limited to a single loop-free integer benchmark program (TCAS) and the single-fault variants provided in the suite (limitations and threats to validity are covered in section 6.4). However, this qualified success has updated the literature [9] to report that the time performance of this technique is competitive when using our optimised search.

Chapter 5

Fault Localisation for Student Programs

Introductory programming courses typically require instructors to assess a large number of small programs by novice programmers. Many of these programs contain errors, ranging from small mistakes to complete design and implementation failures, reflecting the students' misunderstanding of the task or their solution attempt. With limited resources the best learning outcomes are achieved if students are (in the former case) automatically directed towards the locations of their mistakes to allow self-guided repairs, so that the instructors can focus (in the latter case) on addressing fundamental misunderstandings. However, existing basic assessment tools (such as Ceilidh [8], ASSYST [56], and BOSS [62]) based on simple compilation tests, test suites, or model solutions do not provide the detailed feedback necessary for self-guided repairs and do not support instructors in quickly separating solutions with small mistakes from complete failures. Compilation tests do not give any feedback for syntactically correct programs, test suites can give misleading results if programs contain simple errors that affect a large number of test cases, and model solutions cannot account for the variability of student programs. In this section we apply our fault localisation approach to a corpus of student programs and show that its performance, both in terms of runtime and localisation precision, would allow its use in practice. Specifically, our approach can provide feedback to enable self-guided repair and can locate programs which "almost" conform to question requirements but are scored badly by functional grading using test suites.

5.1 Data Set

We use a data set of around 30k passing and 150k failing Java and Python programs collected by the automated assessment system of the University of Auckland [91]. These

programs were written by students to answer 1693 different Computer Science coursework questions. We translated the programs into a C-representation using a simple converter that was designed to retain the style and functionality of programs without translating any detailed differences between the languages, such as language-specific handling of integer overflows. This translation would allow the mapping back of locations to the original Java or Python source code, although our model-based downstream components currently only reason within the specifications of the C programming language they have been translated into. We consider the translated programs in C to be the corpus on which we are testing the efficacy of our tool. Our methods are more directly applicable for native reasoning on Java or Python programs by using suitable downstream components designed for those languages.

We rejected programs that produced translation failures (typically due to unsupported standard library calls), that use floating-point arithmetic (which is not yet supported for symbolic exploration by our downstream components), that comprise less than three assignments (to avoid trivial localisation tasks), or that KLEE or GCov (which we use as downstream components) could not process (such as programs containing infinite loops). We then analysed the data set to identify pairs where a student had submitted a failing program which, after a single assignment edit, was later found in the database passing the full test suite. This yielded 304 pairs which were answers to a range of different coursework questions. These programs contain an average of 5 assignments in 11 statements (as counted by GCov).

We ran a script to analyse each provided test suite (averaging 7.8 tests per pair) and generated new test suites that randomly picked inputs within an order of magnitude of the existing values. The passing program from the code pair was used as oracle to establish desired output values. These large test suites average 154 tests per pair, with 80 of those tests failing.

5.2 Experimental Setup

We generated all results on a 3.1GHz Core i5-2400 using the KLEE 1.1 symbolic analyser. Spectrum-based fault localisation results using Ochiai and Tarantula formulas [123] are provided by Hawk-Eye [44]. As only the suspiciousness values, not the final rank of the fault location, varied between formulas with this data set, both are referred to simply as the Hawk-Eye result.

Localisation tools typically produce a ranked list of locations, based on the generated suspiciousness value of each location. The rank is calculated using Hawk-Eye's middle-line strategy, ranking equally suspicious statements with the mid-point rank. For example, statements with suspiciousness (0.1, 0.4, 0.8, 0.4, 0.4) are ranked (5th, 3rd, 1st, 3rd, 3rd). Results from our model-based tool have been converted from an unranked list

using the same middle-line strategy (where all locations returned are given a suspiciousness of 1.0 and all others a suspiciousness of 0.0). Absolute localisation performance is reported as the average rank of the fault location in the ranked list.

Percentage localisation performance is reported in the average percentage of other locations that will be searched before the fault is found when iterating over the locations in rank order. This is in contrast to the earlier results reported in subsection 4.3.2, where the *percentage of lines of code returned by localisation* is the total percentage of unranked lines (not searched locations) returned as potential repair sites. So, in the above example, if the second statement is the fault location then the ranked localisation scores 50% in percentage localisation performance. If the first statement (which is ranked last) was the fault then this would score 100%, the worst possible score, indicating every other returned statement would be searched before the fault was reached. A score near 0%, indicating very few locations ranked above the location used by the student to repair the submission in the database, means fewer instances of the debugging process stalling before repair synthesis can begin. Spectrum-based methods have been shown to provide rankings insufficient for novices [83] or even be unable to discriminate between most locations [88].

5.3 Results for Original Test Suites

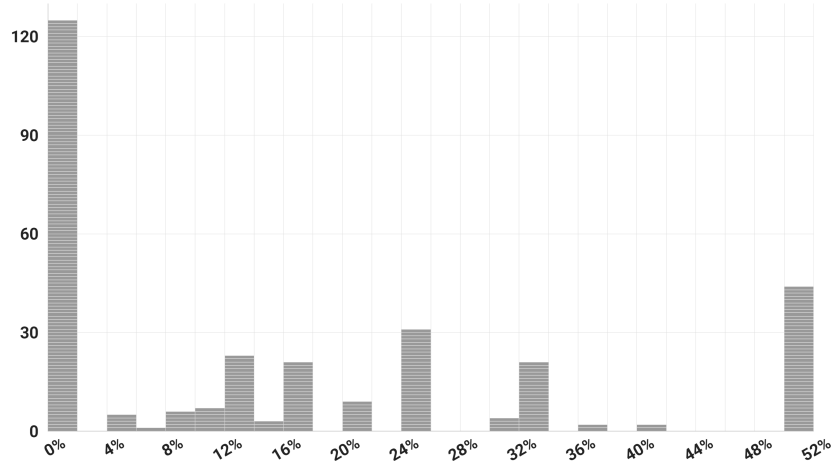

Figure 5.1: Histogram grouping programs by their percentage localisation performance (relative rank of injected fault, x-axis) using the original test suite

On the 304 selected (i.e. failing) student submissions and using the instructor-authored test suites, our tool returns the faulty assignment after, on average, only 16% of other assignments when providing a ranked list of locations. In figure 5.1 these rankings are divided into two percent buckets to show the distribution. A lower percentage score means less of the program must be manually explored by a programmer before the faulty assignment is encountered. Our tool regularly pin-points the assignment later used by

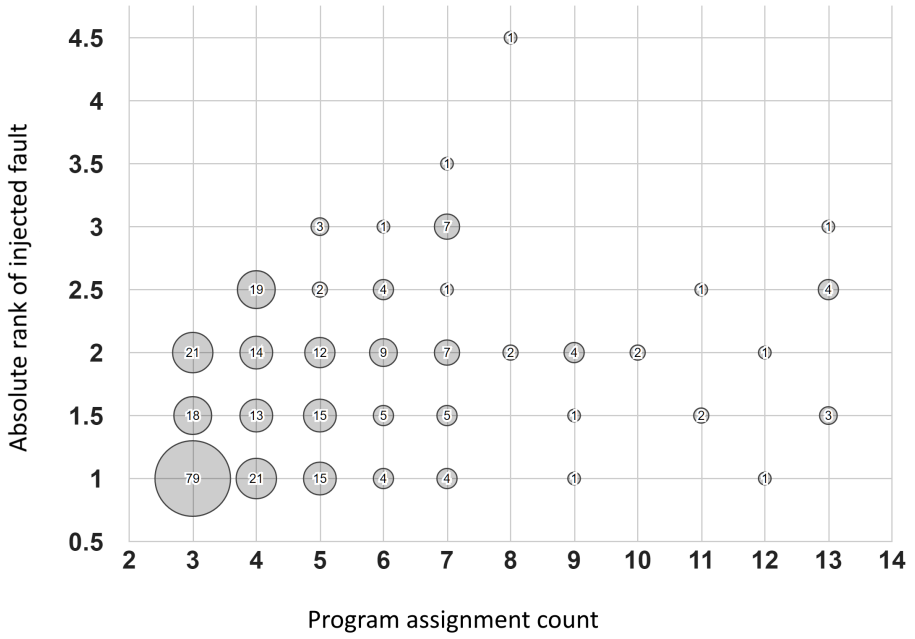

Figure 5.2: Bubble plot counting programs by absolute localisation using the original test suite (x-axis) vs assignment count

the student to bring the submission into compliance with the test suite. 125 of the 304 student submissions were returned from our tool with only that assignment flagged (left-most bucket in the histogram with a 0% percentage localisation), the ideal result [83]. In these cases, all other assignments were eliminated as viable repair candidates for the test suite specification.

The chart in figure 5.2 collects the program's assignment count (how many locations could be ranked by our tool) against the absolute localisation rank (the position in the ranked list of the fault later repaired by the student, using a mid-line strategy when several locations are ranked equally). This demonstrates that small programs (assignment counts of 3 to 5) make up the majority of the student submissions tested but that they don't skew the performance of our tool towards overestimating the absolute narrowing ability of our technique. In fact, looking at the larger assignment counts (8 and above) indicates that percentage localisation performance is weakest for low assignment count programs. These smaller programs is where consistently strong absolute localisation performance is limited by the small list of potential locations, leading to comparatively weak percentage localisation performance. Since the absolute number of locations shown before the actual repair site is critical to facilitating debugging progress [83], strong absolute localisation performance throughout the data set is highly desirable. This topic is discussed further in section 2.8.

In figure 5.3 the p.p. difference to the spectrum-based results (i.e. percentage points closer to a perfect 0% localisation) is plotted for every student submission, ordered by relative advantage. Positive differences indicate our tool's advantage. When comparing

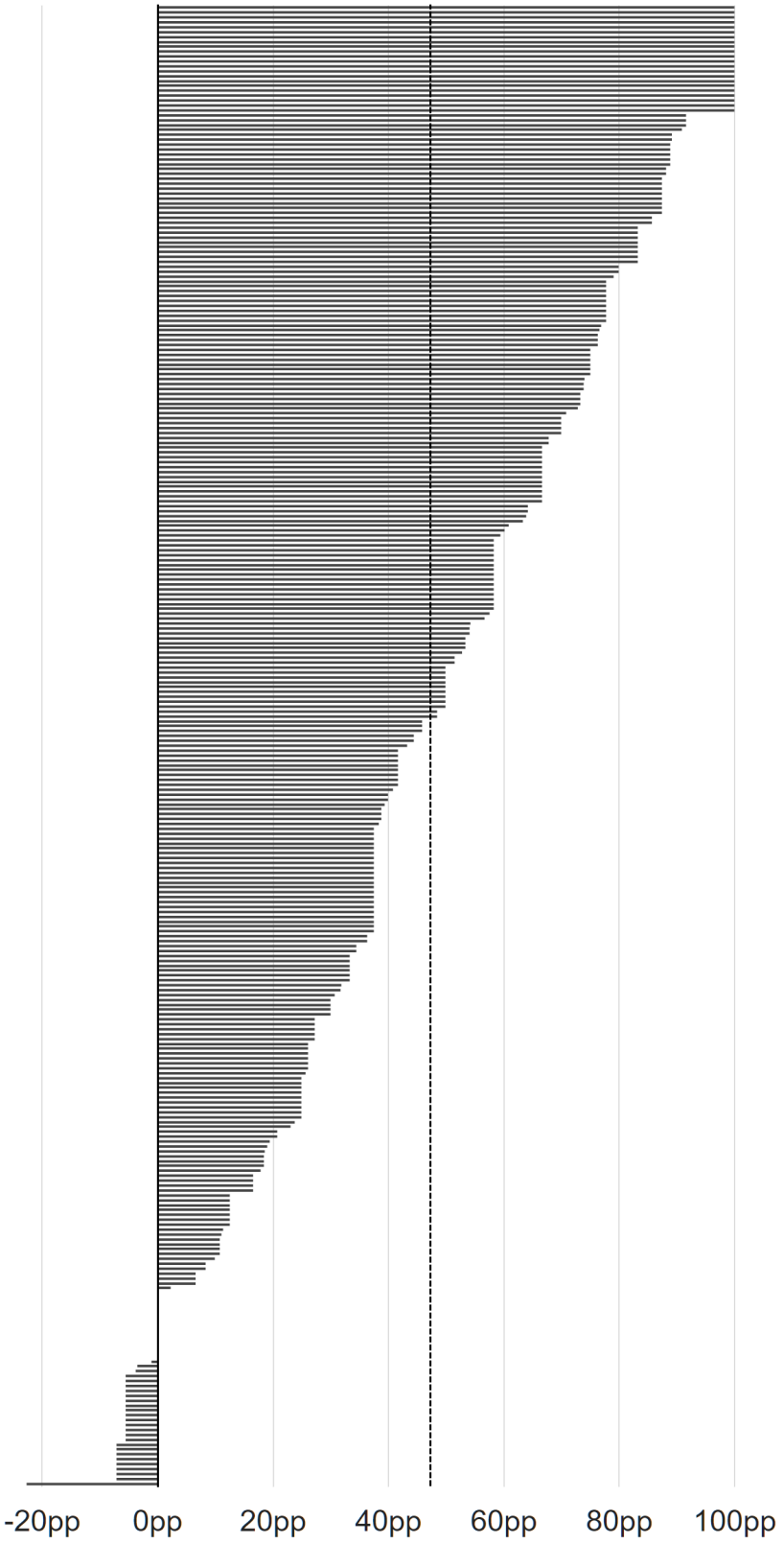

Figure 5.3: Comparison of percentage localisation performance using the original test suite: our tool (+ve) vs Hawk-Eye

the data provided by Hawk-Eye, on this data set, it ranks the statement the student later used to repair the program after an average of 63% of other statements, 47 percentage points adrift of our tool. No correlation was found between the program size and the comparative performance of our tool compared with Hawk-Eye.

On average, Hawk-Eye is provided eight test cases (passing and failing) for each submission which are used to rank eleven statements. Since our tool only requires failing test cases, it is provided with an average of four failing test cases and localises over the average of five assignments in each submission. These localisations are produced in an average of 0.3 seconds per submission by both our tool and Hawk-Eye. 254 times our tool was a fraction of a second ahead, 48 times Hawk-Eye was a fraction of a second ahead, and twice our tool hit too many pathological cases to adapt and only provided results at the tool's ten second time-out.

When looking at the 125 submissions for which our tool returned an ideal result, Hawk-Eye ranked the repair statement after an average of 65% of other statements. Only 15 of those submissions provided the repair in the first third of the ranked list. The best ranking from Hawk-Eye for the statement the student used to repair the program was after 25% of other statements.

The limited number of failing test cases did not hinder our tool on this sample of short, real-world student programs. This strongly supports the use of our tool for assisting students in repairing single-assignment faults in small program submissions, even when only specified by a very small test suite. When a student has made a mistake on constructing an assignment in an otherwise solid submission, an expensive debugging process may be averted by use of this feedback with consistently high absolute localisation performance over the range of student program submissions.

On this selection of actual student submissions which were later brought into full compliance with the test suite with a single assignment edit, our tool provides strong localisation performance, either looking at absolute rank or percentage rank results. We also provide equivalent run-time performance on these localisation tasks when compared to a spectrum-based tool, while ranking the location later used to repair the program significantly higher.

5.4 Results for Extended Test Suites

The provided test suites for these student courseworks were hand-written to provide high coverage with very few tests. The original weakness our tool is designed to overcome [RQ1] is only exposed when localising using a large number of test cases, where

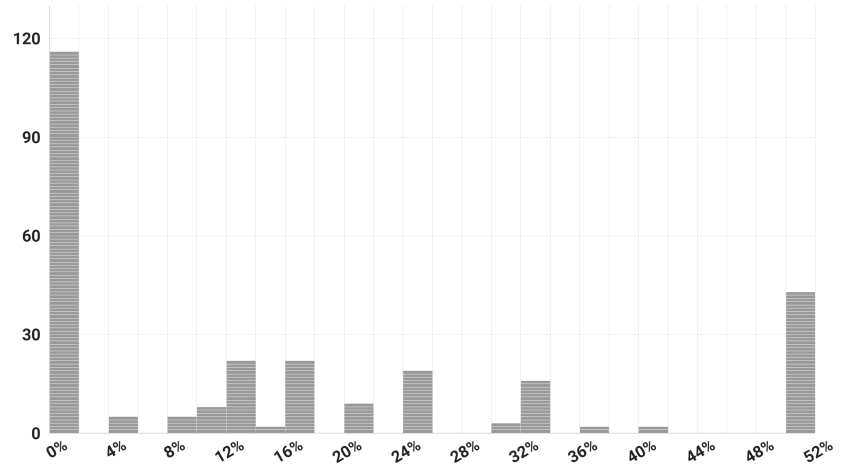

Figure 5.4: Histogram grouping programs by their percentage localisation performance (relative rank of injected fault, x-axis) using the extended test suite

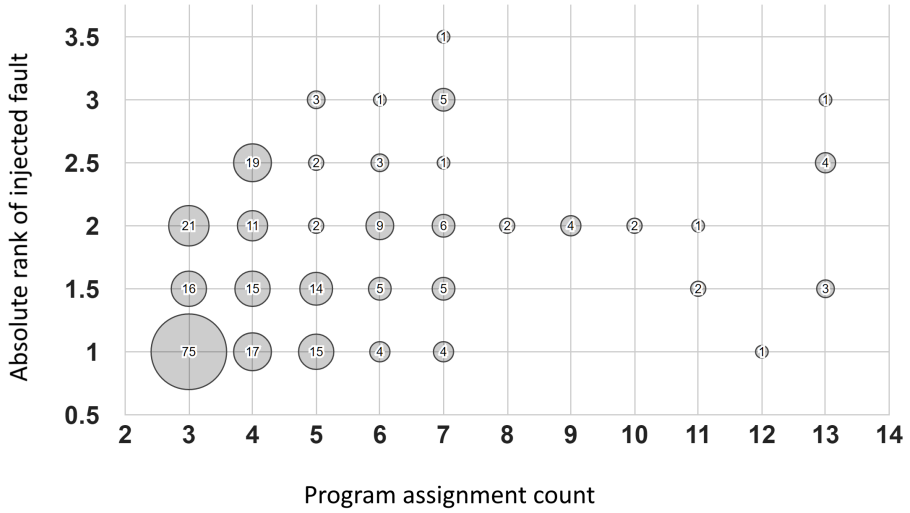

Figure 5.5: Bubble plot counting programs by absolute localisation using the extended test suite (x-axis) vs assignment count

traditionally spectrum-based techniques have significantly outperformed a Griesmayer-derived localisation. We therefore significantly expanded the test suite, as described in section 5.1.

Our tool, provided with an average of 80 failing test cases, did not vary from the 16% localisation score achieved previously. Both the distribution of percentage localisations plotted in figure 5.4 and the plot of the assignment count (x-axis) against the absolute localisation rank (y-axis) in figure 5.5 shows that the localisation performance does not significantly shift when using the extended test suites. Running the average of 154 test cases through each submission, Hawk-Eye showed a modest improvement; although figure 5.6 shows it continued to lag our tool significantly, at a 42 percentage point deficit (dashed line). Hawk-Eye never ranked the statement that was used to repair the

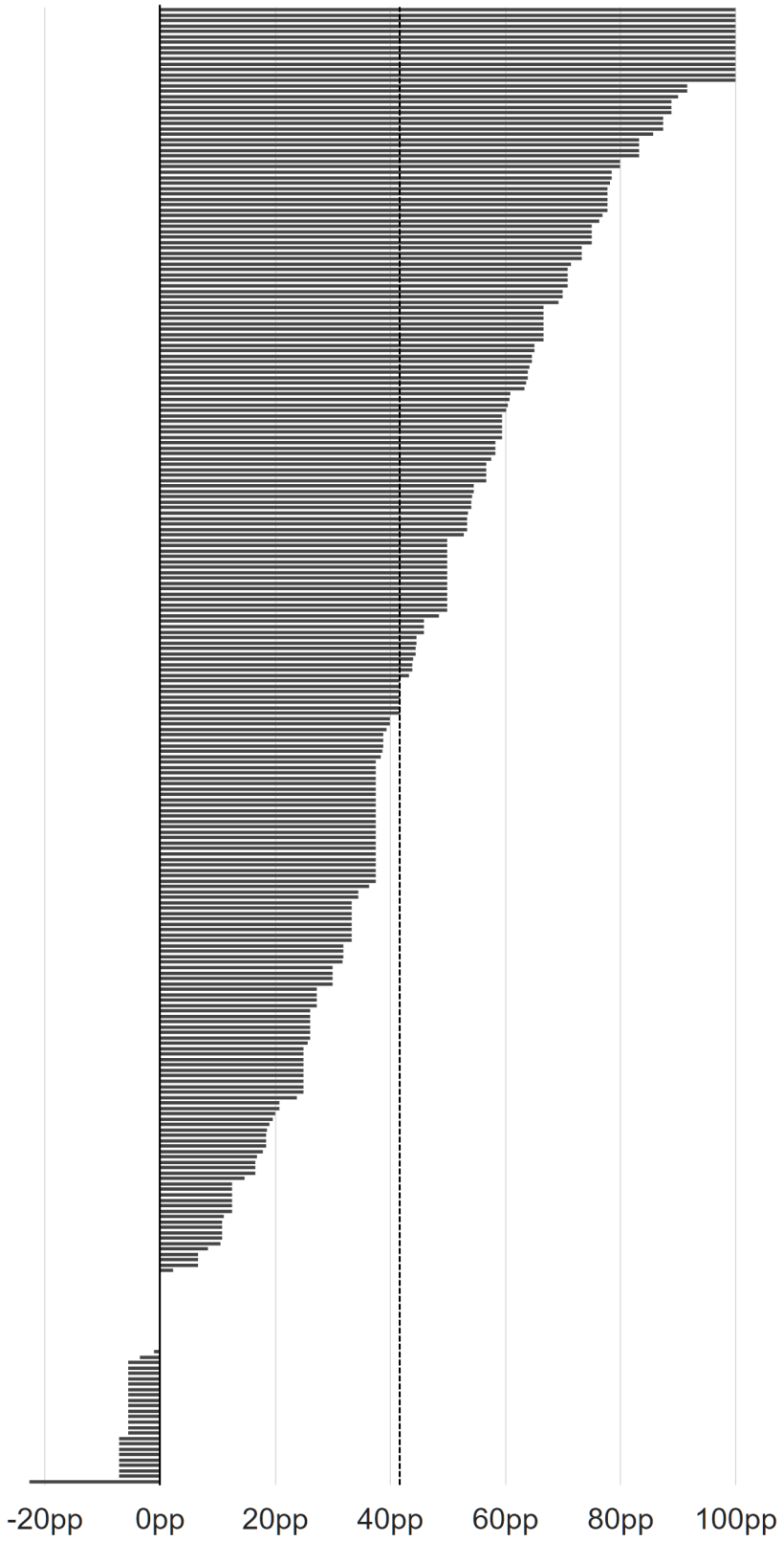

Figure 5.6: Comparison of percentage localisation performance using the extended test suite: our tool (+ve) vs Hawk-Eye

program as the most suspicious when localising on the data set, with either the original or extended test suite.

The consistent localisation performance of our tool is due to a combination of different factors. Over a third of these submissions, with the smaller test suite, already provided perfect results using our tool; some programs will contain several assignments where a genuine repair is possible so there is no more compact list of repair locations; and some programs are resistant to analysis by symbolic analysis, which does not change with test suite size. This may have provided very little room for improvement by our tool.

However, the run-time on these much larger test suites confirm the scalability of our tool, an area where model-based fault localisation has traditionally suffered [39]. Our tool averaged 0.9 seconds per submission while Hawk-Eye lagged behind, averaging 1.1 seconds. Included in that average, three times our tool hit too many pathological cases to adapt and only completed at the time-out.

When significantly extending the test suite size from the instructor-authored suites, our tool retains the same strong localisation performance, looking at absolute rank or percentage rank results. While the spectrum-based technique does improve with the addition of many extra test cases, those results still rank the location later used to repair the program significantly below the ranking provided by our tool. We also provide faster run-time performance on these localisation tasks when compared to a spectrum-based tool, taking the lead due to our efficient scaling thanks to our optimised search process.

5.5 Dual-Fault Extension

The currently discussed algorithm uses a single toggle value which activates a single location, performing the alternative assignment (of a symbolic value). Our whitelist prunes locations based on this single-fault assumption, where a single test case not being repairable causes the location to cease to be searched. If multiple toggles were used with `(__t1 == 3 || __t2 == 3)` conditions activating the modified locations, then the search would be able to produce localisations searching for multiple faults (up to the number of toggles inserted and chained in the or-conditions). This multiple toggle method was proposed as an extension by Griesmayer et al. [39] to provide results for n-fault programs. This increases the solver cost due to additional non-determinism and would also increase the total combination of counter-examples returned, which may severely limit the applicability of this technique to very small input programs in order to retain tractability. We explored an extension to our narrowing whitelist which would allow the collection of more results without using multiple toggle values to explore an n-fault assumption [RQ4].

To explore student submissions pruned from our original slice due to containing more than one fault, as explained in section 5.1, we reprocessed the entire database with slightly tweaked parameters. In this new slice, we rejected submissions using the same criteria except identifying pairs where a student had submitted a failing program which, after exactly *two assignment edits*, was later found in the database passing the full test suite. This yielded 125 pairs. These programs contain an average of 4 assignments in 12 statements (as counted by GCov), making them of the same scale as our initial slice of single-fault programs.

We analysed these pairs by performing a manual code review. The majority of these pairs, 77, contain two faults which cannot both be executed in the same trace. We characterise this class of dual-fault program as *twin-fault* programs, which have a test suite that mixes the two fault-exercising subsets of test cases. The remaining 48 pairs are defective programs where traces from failing test cases will either always or sometimes execute both fault locations during a single trace. We call this class *double-fault* programs. In the example in figure 5.7, if lines 3 and 5 are incorrect then this would be a double-fault program where some test cases would exercise both lines and others would only exercise faulty line 3 (and then exiting via line 7). However, faults on lines 5 and 7 would make it a twin-fault program as all test cases would either only exercise line 5 or only line 7, no trace can exercise both faults.

```
1 int main(void) {
2   /* ... */
3   int z=14*a*c;
4   if((bb-z)<0) {
5     return pack(co1, co2);
6   } else {
7     return pack(re1, re2);
8   }
9 }
```

Figure 5.7: Potential dual-fault program example

For this new data set, the original test suites contain an average of 8.8 test cases per program, 5.7 of them failing. This can be insufficient to exercise each pair of assignments in a dual-fault program and classify the pair as twin-fault or double-fault. In figure 5.7, if the omitted code includes a branch that provides three paths, then simply covering each different path through this small program fragment requires six failing test cases, assuming no duplication of trace paths. Although program flow analysis can detect when a pair of toggled locations being searched will result in a twin-fault program, this analysis can be expensive. In fact, the sorting of test cases alone is as expensive as running Hawk-Eye, due to the similarity in work done to execute and assess each test case trace. Hawk-Eye averages a third of a second per program and provides an average localisation of 67% on this data set using the original test suites. We generated extended test suites in the same manner as with the original data set, discussed in section 5.1. These contain an average of 270 test cases per program, 94 of them failing.

To adapt our tool to search for localisation results in twin-fault and double-fault programs, the whitelisting discussed in section 3.2 had to be replaced. A failing test case which exercises line 5 of figure 5.7 cannot flag line 7 as a potential repair location (as it is not exercised at all) so the lack of a localisation does not remove it from the search space for future test cases. Dual-fault programs require the consideration of previously completed localisations in order to reason about the possible narrowing of the whitelist of locations. If previous failing test case have always either flagged line 5 or 7 but not other lines then, under a twin-fault assumption, we can eliminate other lines from the search space.

However, this whitelist adaptation can, for some programs, lead to no narrowing of the whitelist for the entire duration of the localisation run as no location can be ruled impossible to be one of the two repair sites. This assumption also cannot localise with a failing test case that exercises both locations on double-fault programs; this results in no locations being flagged as a possible repair site. If lines 3 and 5 must always both be changed to fix any failing test case in figure 5.7 then no single toggle value can exercise that widened symbolic behaviour to find that repair. Those negative localisation results must be ignored (dropped) rather than integrated into the narrowing process.

To accelerate the process for larger test suites, we rank the frequency in which the locations are capable of repairing previous failing test cases and use early narrowing to reject locations which have been flagged significantly below the bulk of other locations. This optimisation allows early search space reduction with a very low chance of erroneous rejection in much the same way the early termination in section 3.2 does. This ranking is used to provide a ranked list of all locations with associated suspiciousness, allowing our tool to provide ranked results (cf. section 5.2 where our original algorithm only provides one and zero suspiciousness levels to any assignment). These changes produce a search which narrows less often and less early.

5.5.1 Results for Dual-Fault Extension

	2P	**2nP**	**1P**	**1nP**	**F**	**Loc**	**Rank**	**Time**
Twin (orig)	47	24	4	2	0	24%	2.0	0.3s
Twin (ext)	37	32	4	4	0	24%	2.0	2.5s
Twin (ext, opt)	40	31	4	2	0	24%	2.0	1.0s
Double (orig)	7	34	3	4	0	34%	2.3	0.8s
Double (ext)	6	35	3	4	0	36%	2.3	4.2s
Double (ext, opt)	6	35	3	4	0	37%	2.3	2.0s

Table 5.1: Dual-fault localisation performance. **2P**erfect double localisation; non-perfect double (**2nP**) localisation; **1P**erfect single, localisation; non-perfect single (**1nP**) localisation; **F**ailed localisation; mean **Loc**alisation rank; mean absolute localisation **Rank**; completion **Time**

In table 5.1, after submitting the dual-fault data set to our adapted tool, we analysed (via manual code inspection) and split the results into twin and double-fault subsets based on our knowledge of the repair locations used by students to later bring the programs into full compliance with the test suite. We present the results using the original test suites, where the early narrowing optimisation does not have time to trigger, and for extended test suites both with and without the (early narrowing) ranking optimisation discussed above. We have calculated averages of: the mean absolute localisation rank (**Rank**) of the two faults later used to repair the program; the completion time (**Time**) to run localisation on a program with all failing test cases; and the mean localisation rank (**Loc**) of the two faults as a percentage of the of the full range of rankings possible.

We also divide the 77 twin-fault programs and 48 double-fault programs by the final whitelist results they generated. This classifies our results based on our hybrid approach, described above, where locations are both given suspiciousness ranks and narrowing reduces the list of locations returned. Perfect (**2P**) results flag exactly two locations, both of the faults used to later repair the program to bring it into full conformance with the test suite. Non-perfect (**2nP**) results flag both locations but include other locations as possible repair sites, which is often due to other genuine repairs existing. Some of these non-perfect results rank the two fault locations most highly (i.e. *Loc* of 0%, *Rank* of 1.5) despite also marking other locations as viable repair sites.

Suppose figure 5.7, with a twin-fault on lines 5 and 7, was localised with a test suite that only exercised line 7. A result where the suspiciousness of all locations other than line 7 was zero while line 7 was 100% is plausible. In this case, the dual-fault search indicates a single-fault repair solution has been found as the only repair location. There also exists dependent assignment fault chains, for example `a = x*y; b = a/z-p;`, which generate single-fault repairs (at `b`) to a double-fault program. We use perfect (**1P**) to classify a singleton localisation, non-perfect (**1nP**) includes other repair sites with the one detected fault. Finally, failures (**F**) are where the localisation fails to return either fault location, which we did not experience with this data set.

The dual-fault programs, which are the same scale of source code as the previous data set with similar test suite sizes, are localised within the same time window. Double-faults localise with run-times in the same order of magnitude while twin-faults match the run-times of our earlier single-fault programs, when using our optimisations. The absolute and percentage ranked localisation performance also shows a similar characteristic to the single-fault data set where additional failing test cases do not significantly change the performance of the tool. However, the added complexity of this search does not provide identical narrowing performance. The whitelist localisation results exhibit some variability. This may be caused by the randomised test case order influencing the search space exploration. Twin-faults that find additional locations for potential repairs when exercised by larger test suites (2P to 2nP shift) may also indicate the limitations of smaller test suites to reason over which locations can be dropped as potential repairs.

However, the unvarying ranked performance indicates the locations used by students to repair the submissions are not penalised by this limited data leading to a tighter localisation with the original test suites.

This tool extension offers a path to localising programs beyond our original slice in section 5.1 without the likelihood of exploding symbolic analyser time costs caused by implementing a model-based dual-fault search using multiple toggle variables. We find this extension allows the processing of many additional student programs that occur in our database, extending the use of this tool beyond the single-fault programs previously localised. Programs with twin faults are often perfectly localised without requiring test suites be divided before processing or knowing in advance if the locations being searched are in a twin-fault relationship.

5.6 Grading Support

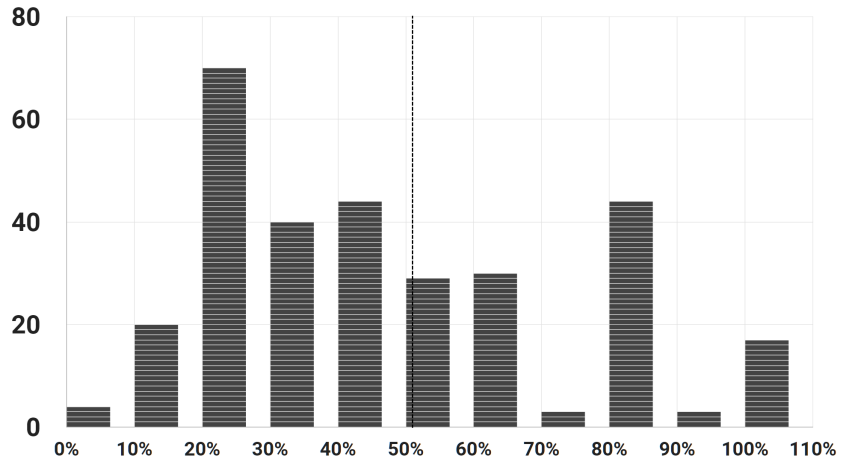

Figure 5.8: Histogram of test suite compliance score for "almost correct" programs using the original test suite

To confirm the value of this localisation data for detecting "almost correct" student submissions that test suites do not highlight, we extracted the test suite results for the data set. As each of these student submissions was selected because there exists a single edit to an assignment which brings it into compliance with the complete instructor-authored test suite, they should be graded within a generally narrow distribution, reasonably close to a fully compliant solution. However, as figure 5.8 shows in a histogram of the test suite compliance score of these programs, this is not the case. The average submission fails for 51% of the test cases (dashed line) and the distribution of scores is very uneven, not clustering around a single value. As there are so few test cases per submission, the histogram buckets have been set to best show the distribution curve.

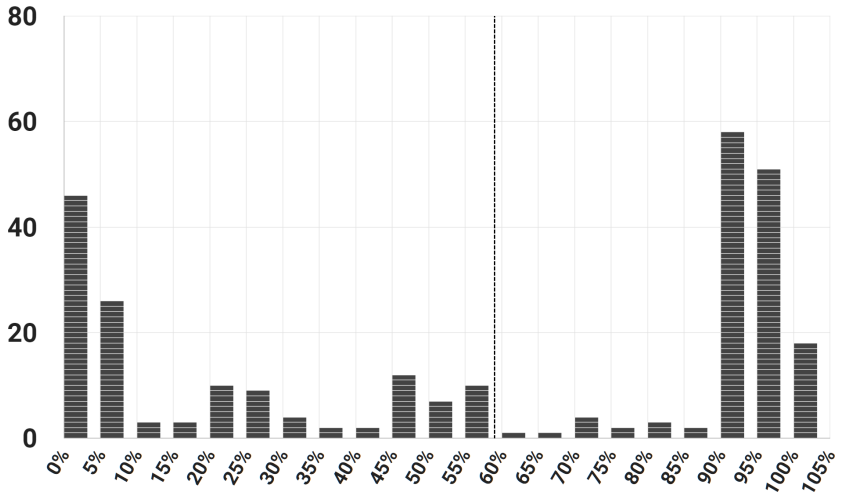

Figure 5.9: Histogram of test suite compliance score for "almost correct" programs using the extended test suite

When expanding out to the much larger test suite in figure 5.9, the average submission fails for 59% of the test cases (dashed line). Here the distribution of grades is less flat, with clumping at both poles. Half of the "almost correct" submissions are scored with 80% or more of the test suite failing. Our tool will accelerate automation-assisted marking by flagging nearly-good code with the likely location of a repair that will radically increase a test suite-based grading.

5.7 Effects of Programming Language

Comparing data for submissions originally written in Java to those in Python showed no significant skew to the data points explored. Translating a simple subset of each language into C syntax generated comparable sets. At least on this simple translated subset, the language originally used does not appear to strongly bias the characteristics of single-faults introduced by students.

5.8 Conclusions

We have applied our tool and comparing those results to spectrum-based methods on real programs [RQ2] with large test suites [RQ1]. This published research [10] into the efficacy of our tool in the pedagogical domain [RQ5] also provides an extension to our narrowing algorithm with some preliminary results on a class of dual-fault programs found in our data set [RQ4]. This allows a wider search for dual-fault repairs if our original single-fault search fails to isolate a single potential repair location. We go on to cover the limitations and threats to validity in section 6.4.

Chapter 6

Discussion

We have described a fast, model-based fault localisation algorithm which, given a test suite, uses symbolic execution methods to fully automatically identify a small subset of program locations within which (under a single-fault assumption) a genuine program repair exists. Our main contribution is an improved search through the test suite that drastically reduces the effort for the symbolic execution of the models.

In chapter 4 we have evaluated this improved search against the originally published results for this method without those optimisations. We also reimplemented Griesmayer's design to allow us to compare it when running on modern hardware with a current SMT solver. This is where, in subsection 4.3.1, we have found several orders of magnitude of speed improvements from using our search algorithms. We also compare our performance on the benchmark program variants used against a competing state of the art model-based fault localisation method. In subsection 4.3.2 we explain how our tools doesn't narrow too quickly or approximate too early and so can always isolate the injected fault for every variant with every failing test case. We also discuss how this benchmark has performed with spectrum-based localisation methods in section 4.4. We have also investigated the generation of high-quality repairs to replace the low-quality repair that our model-based localisation process creates as a side effect of localising using a test suite. However, the negative results shown indicate that repair synthesis by concrete execution of the same search space will outperform an optimised version derived from this search method.

In chapter 5 we apply our tool to real-world student code submissions and compare the results against an implementation of a spectrum-based method. Our speed improvements allow us to retain parity with this fast, inaccurate localisation for small test suites and even slightly improve upon the speed when extending the test suite scale thanks to our better than linear scaling. This tests our tool on many new programs which are not just variants of a single benchmark from 1994 and are formatted in a style consistent with current novice programmers. We also, in section 5.5, have provided positive preliminary

results for an extension to some n-fault programs by demonstrating a relaxed search optimisation on dual-fault programs from the same data source. Finally, we have shown in section 5.6 that functional correctness over test suites does not predictably grade the class of nearly correct student programs which our tool isolates during localisation. Although expanding the test suite does generate grades that cluster around the poles (0% and near 100% compliance), this is only a slight improvement over no clustering as many of the programs are being pushed towards no marks. Our tool will accelerate automation-assisted marking by flagging nearly-good code with the likely location of a repair that will radically increase a test suite-based grading.

6.1 Model-Based Localisation Acceleration

We initially demonstrated the time and localisation performance of our tool on the Siemens test suite's TCAS program variants and test universe, a common localisation benchmark from 1994. We compare the results of our optimised search algorithm against our naïve reimplementation of Griesmayer's algorithm in chapter 4. We demonstrate multiple orders of magnitude completion time improvements over these non-optimised searches. The average halving, at most six-fold, decrease in completion time from Griesmayer's results (696 seconds average) to our naïve reimplementation (325 seconds average) shows some performance increase is derived from using a modern symbolic analyser on modern hardware. But, for example, variant 1 moving from over 49 minutes to over 24 minutes to return a location set is not viable compared to our optimised algorithm when also using ESBMC, at 9 seconds. This quite clearly demonstrates the speed improvements are derived primarily from our search optimisations, with hardware platform and symbolic analyser used making only minor differences to the recorded times for the benchmark variants.

When comparing the localisation performance of these tools, we must consider that there is an injected fault location for each of the single-fault variants of TCAS. For all the Griesmayer-derived techniques then the injected fault location (the location where a variant was seeded with a fault) is always included in the returned list of locations.

Jose and Majumdar convert an input C program to a maximum Boolean satisfiability problem which is analysed with MAX-SAT solver [61]. However, because it returns the complement of the maximal subset of clauses that can be true for each single test case, their approach can omit genuine repair locations. For six of the 33 single-fault variants they indicate that there exist subsets of the failing test cases for which they would not flag the injected fault. It therefore relies on summing the results of the different test cases, providing a ranking of most to least commonly flagged locations. For 31 of the 33 single-fault localisation benchmark variants we improve on the localisation performance of their results, sometimes by a significant percentage. In the two of those 33 variants

where our tool performs worse, *v41* returns a set of locations only one larger than those returned by Jose and Majumdar, and *v8* returns a set three locations larger.

For the single-fault localisation benchmark variants, we always maintain time performance within the same order of magnitude as the current model-based fault localisation state of the art, Jose and Majumdar. Our tool, using the KLEE back-end, averages a completion time of 2.87 seconds per TCAS variant, compared to an average of 2.80 seconds in Jose and Majumdar's results. We guarantee returning the location of the injected fault in every failing case, which Jose and Majumdar cannot. Their results for each full test suite do flag the injected fault location for TCAS as most common. But they indicate that there exist subsets of the failing test cases for which they would not flag the injected fault in the case of six single-fault variants.

These results support our claim that a Griesmayer-derived model-based localisation technique can be modified to be fast, comparable to the current alternatives for small programs, even with large test suites. This is shown with an intelligent pruning of the search space to minimise the symbolic execution load while minimising the disruption of a slow or intractable search node, facilitated by a multiprocess design that takes advantage of modern consumer processor architectures.

6.2 Comparisons to Spectrum-Based Localisation

Spectrum-based fault localisation techniques, compared in [79, 121, 123], operate by examining passing and failing test cases separately. They assume that faults are more likely to be exercised by failing test cases and less likely to be exercised by passing test cases. The statements in a program can then be ranked based on the different weighting techniques. The analysis of the performance of these approaches is typically based on several scoring formulas that roughly correspond to how much of a program must be explored, given an ordered list of locations as tool output, before the genuine fault is found.

We return to the worked example from chapter 3, here in figure 6.1, where line 9 contains an error where the integer literal should be `2`. When a spectrum-based localisation technique is applied to a test suite using this example, the first 12 lines will always be executed. Their suspiciousness will be defined by the ratio of passing and failing test cases in the suite.

Some passing test cases, where the employee is between 21 and 39 years old but doesn't work overtime that week, will exercise line 13; this line will also be exercised by all failing test cases. Surfacing the fault on line 9 as a visible error in output (so creating a failing test case) requires non-zero overtime and an age between 21 and 39 to cause the `SelBonus` value to be incorrect and then used on line 13. All other passing test cases,

```
1  #DEFINE BR 15
2
3  //normalHours, overtimeHours, age
4  int payMe(uint nH, uint otH, uint age) {
5    uint bonus = BR * (otH * 2);
6    uint normalPay = BR * nH;
7    uint ageHigher = age % 20;
8    uint SelBR = BR + ageHigher;
9    uint SelBonus = otH * (3 * SelBR);
10   uint SpecialNormalPay = nH * SelBR;
11   uint specialPay = BR + 20;
12   if ((age > 20) && (age < 40)) {
13     return SpecialNormalPay + SelBonus;
14   } else if (age >= 40) {
15     return (nH * specialPay) + ((specialPay * 2) * otH);
16   } else {
17     return normalPay + bonus;
18   }
19 }
20
21 int main(int argc, char ** argv) {
22   int result = payMe(atoi(argv[0]), atoi(argv[1]), atoi(argv[2]));
23 }
```

Figure 6.1: Worked example: Original faulty (at line 9) code

with ages too low or high, will flag lines 14 to 17 as less suspicious. The cumulative effect of this, for any test suite composition, will be to rank the suspiciousness of line 9 as lower than line 13 and identical to the majority of lines. This gives a good example of how the spectrum-base localisation process can fail to elevate the faulty line of code. If there was a significant volume of code rather than the single statements in the block at line 13 then all those lines would be flagged as the most suspicious. If the if clauses were tested in the opposite order (so the statement on line 13 was on line 17), lines 14 and 16 would also be painted by all test cases. That would leave only two lines as painted as less suspicious than the actual fault location. The localisation would be approximately equal to a random chance ordering.

The best-known example of spectrum-based fault localisation techniques is the Tarantula tool [98]. For over 50% of the variants of the Siemens small programs (including TCAS) Tarantula ranks the injected fault location in the top 10% of lines by suspiciousness. But this performance is inconsistent and for 7% of these variants, it fails to rank the injected fault location in the top 80% of locations. We look at this directly in section 4.4, showing our tool provides 100% of the single-fault TCAS variants with a localisation performance that puts the injected fault location in the top 10% of lines returned. Even if we assume the injected fault was always ranked at the bottom of our list of returned locations, rather than averaging somewhere in the middle, that would only drop to 97% of variants.

State of the art spectrum-based fault localisation methods have recently been compared using different theoretical frameworks [79, 123, 124]. Several methods have been identified as optimal under these frameworks; there is also empirical data over various of the Siemens small programs. While Tarantula is not optimal [123], it is also not far

behind the state of the art for TCAS. In empirical results [79, Table XI, p. 11:23], the only method identified as optimal under that paper's framework ranked the injected fault location on average at the 17th returned location (9.9%) over all TCAS variants. Tarantula returned the injected fault location at an average location between the 18th and 19th ranked location (10.8%).

This is significantly below the worst performance of our tool. To compare these results, we must convert the unranked sets in table 4.2 to ranked lists from which to derive averages. Randomly ranking all the returned lines above a list that randomly ranks all the lines not returned provides this conversion. The injected fault location, when it is returned as part of the unranked set, will, on average, be in the middle of the ranked returned lines. Using this conversion, the KLEE average result over the TCAS single-fault variants is equivalent to returning the injected fault location at the 4th ranked location (2.3%). Jose and Majumdar provide results equivalent to a 3.9% average on the same set of variants.

When we directly compare our tool to a spectrum-based localisation tool in chapter 5, we almost always outperform the spectrum-based ranking. For small test suites we provide run-time parity (0.3s) and even take the lead (0.9s vs 1.1s) when extending the test suite scale, thanks to our better than linear scaling with test suite size. Spectrum-based localisation, as it simply iterates all test cases using concrete execution, scales linearly with test suite size once the overhead has been factored out.

Our tool regularly pin-points the assignment later used by the student to bring the submission into compliance with the test suite, returning the faulty assignment after, on average, only 16% of other assignments. 125 of the 304 student submissions were returned from our tool with only that assignment flagged. All other assignments were eliminated as viable repair candidates for the test suite specification. The spectrum-based tool ranked the statement the student later used to repair the program after an average of 63% of other statements when using the small test suite and only improved to 58% with a large test suite. Our unchanging performance with the larger test suite is still a far higher ranking for the statement the student later used to repair the program.

Our results in chapter 5 strongly support the use of our tool for assisting students in repairing single-assignment faults in small program submissions, even when only specified by a very small test suite. When a programmer has made a mistake on constructing an assignment in an otherwise solid submission, an expensive debugging process may be averted by use of this feedback with consistently high absolute localisation performance over the range of student program submissions.

6.3 Extensions beyond Single-Fault Assumption

Most of our results have been achieved under a single-fault assumption. This is a common assumption [16, 39, 61] but also does not reflect the majority of real-world situations in which automated localisation will be deployed. Our data set in section 5.1 does show that there are real-world single-fault programs on which to apply our tool but expanding this beyond this limitation is highly desirable. However, using n toggles to expand the symbolic analysis to a genuine search for any n-fault repair rapidly increased the solver cost due to additional non-determinism and would also increase the total combination of counter-examples returned. This may severely limit the applicability of this tool to very small input programs in order to retain tractability.

In section 5.5 we have shown a method of relaxing and reshaping the optimised search process we have developed which allows the isolation of some dual-fault programs. The dual-fault programs, which are the same scale of source code as the previous data set with similar test suite sizes, are localised within the same time window. Double-faults, where both faults can be exercised in a single trace, localise within the same order of magnitude while twin-faults equal our earlier single-fault programs, when using our optimisations. The absolute and percentage ranked localisation performance also shows a similar characteristic to the single-fault data set where additional failing test cases do not significantly change the performance of the tool.

Our preliminary results show a valuable extension beyond our single-fault assumption, adding an additional 125 dual-fault pairs to the 304 single-fault pairs in our data set on which we can process to provide localisation results. The 77 twin-fault pairs show the most promising results, which are in-line with our single-fault pairs. The remaining 48 double-fault pairs also provide some positive results and maintain a run-time completion time within the same order of magnitude as our other results.

This tool extension offers a path to localising programs beyond our original slice in section 5.1 without the likelihood of exploding symbolic analyser time costs caused by implementing a model-based dual-fault search using multiple toggle variables. We find this extension allows the processing of many additional student programs that occur in our database, extending the use of this tool beyond the single-fault programs previously localised. Programs with twin faults are often perfectly localised without requiring test suites to be divided before processing or knowing in advance if the locations being searched are in a twin-fault relationship.

6.4 Limitations and Threats to Validity

In subsection 4.3.2 we have already discussed issues with making direct localisation performance comparisons. These are not direct “apples to apples” comparisons as each

localisation method brings with it various limitations and assumptions. The granularity and fault classes being searched for varies between the different model-based and spectrum-based localisation methods compared in this thesis. Different localisation scopes for each technique's implementation means their localisation performance is not precisely comparable which is a caveat noted whenever we have presented a comparison.

We have provided results on both commonly used benchmark program variants and against programs where the repair site is defined where a human debugger later repaired the program after the faulty version was submitted to the system. We also argue that any location returned by our tool is a genuine repair opportunity to allow the entire test suite to pass, due to the method of localisation generating a low-quality repair at the returned locations. So when many locations are returned, this is a function of the weak specification being exposed, not an error in ranking the seeded/used repair location lower than it should be. Fault localisation does, ultimately, result in comparing the size and composition (or rank order) of sets of proposed fault sites where developers must then search for a genuine repair. It is by this metric and run-time performance that we have attempted to offer insight into how the various approaches compare, even if we are not comparing apples to apples.

The fault-seeded variants of the small Siemens program we are testing on are not a representative sample of C programs and the faults they contain. The TCAS variants explored all contain injected faults inserted at return expressions or assignments. Our results may not generalise to other C programs. Our focus on a subset of programs, and use of real-world code which is atypical in the heavy use of global variables, may obscure comparative analysis of performance against other tools with different program features. Performance on relatively small, loop-free programs like TCAS does not provide guidance into how this process scales to large programs with more complex control flow. However, this issue is common to all tools which demonstrate their localisation effectiveness on the TCAS variants.

We have expanded to other library-call-free, integer programs using our database of student code submissions, including some programs with loops. These are generally of a much smaller scale than the Siemens benchmarking programs outside of TCAS and don't use other variable types (which our downstream tools cannot handle due to no floating point symbolic analysis). However, this does provide a real-world niche in which our tool is generating localisation results which would assist students in completing the debugging of their coursework submissions.

The slicing of the student programs via a simple language translation stage, without any translation of library calls, and rejection of unparseable code restricts the form of programs explored. This slice assumes a repair possible via assignment modification. The choice of symbolic analyser (with an integer solver) also restricts the type of programs processed. Some programs are not suitable for automated localisation, such as those

that never terminate on some inputs, and these would not make it through the database slice. These restrictions could add a bias to the student programs explored which could unfairly advantage one tool or call the generalisability of these results into question.

Our tool and the slice used on the database of student programs to generate the pairs for analysis makes a single-fault assumption for the majority of our results. This is a common assumption in the fault localisation field [16, 39, 61] but does not reflect real-world debugging, although the existence of many code pairs in this data set does confirm real-world applicability. We have partially relaxed this constraint to a limited dual-fault assumption in section 5.5.

We can use any (C99 comprehending, supporting `assume` functionality) symbolic analyser to process our generated C code but our results are linked to either KLEE or ESBMC. Any issues related to those tools may affect our results, if not our methods/process. No high performance symbolic analyser of C can perfectly transform an input program into an exact representation according to the full C specifications. The lack of exact specification compliance by the various widely used (optimising) C compilers also makes such an impracticable achievement undesirable. Real compiled code does not perfectly map to a strict adherence to a single, deterministic interpretation of the C specifications.

The open-source script used to execute the spectrum-based localisation was written in Java. Executing Java scripts is known to come with a high initialisation time cost to start the JVM. Due to the short run-times involved, this may have inflated the run-time costs of this technique beyond some competing implementations based on the same GCov underlying tool.

To minimise the risk of over-tuning our design to the test data, our choice of time-out, sleep delay, and early termination values have not been tuned or selected for optimising with respect to the data set as this would compromise the generalisability of our results.

Chapter 7

Conclusions

Our main contribution is an improved search algorithm through the test suite, reducing the effort for the symbolic execution of the model. Our results show Griesmayer's technique works in comparable time to the state of the art when driven with our optimised algorithm for the small C programs tested. This algorithm outperforms the naive reimplementation of the technique and the technique's originally published implementation by more than two orders of magnitude.

We generate genuine lists of repair locations as specified by test cases for any repair that could be expressed as a look-up table for the right-hand side of an assignment, within the limits of symbolic analyser accuracy. These low-quality repair tables provide assignment values to correct all failing test cases in the suite. Our time performance is in line with recent alternative model-based fault localisation techniques, but narrows the location set further without rejecting any genuine repair locations where faults can be fixed by changing a single assignment. This is more consistent than the localisation performance of other techniques and does so without compromising the narrowing extent, which might be done to avoid the false negatives shown in the competition.

Applying our tool in an educational context must meet the demands of novice programmers who are unlikely to be capable of advanced debugging techniques. Coursework and charettes provide an opportunity for fast, accurate fault localisation to assist in educational institutions, which have already built up databases and workflows around test suite specified exercises. We have demonstrated a fast, model-based fault localisation tool on a collection of single-fault, real-world student submissions. The high quality localisation information reduces the search space for novice debuggers working down a list of potential repair locations when compared to the results from spectrum-based techniques. In over a third of the sampled failing student programs, our tool used the test suite to provide a localisation result that uniquely identified the location later used by the student to repair the program and rejected all other locations. This reinforces the qualitative difference between model-based fault localisation that reasons

over a model of the student program to derive a list of feasible repair locations compared to a spectrum-based ranking process that infers suspiciousness of each program statement. These short, often exact singleton, lists of potential assignment repair locations can direct students to the site of improvement, assisting in the construction of a final submission that fully complies with the test suite. The run-time cost of our high quality localisation matches that of fast, inaccurate spectrum-based fault localisation, even with large test suites, ensuring that our approach is viable even at the scale required by MOOCs.

We have demonstrated that submissions which are a single assignment edit away from full compliance with a test suite would not be graded predictably if scoring were based on compliance with the test suite only. This confirms the need for tools which can isolate such "almost correct" student submissions.

We have also proposed an extension to our algorithm to account for collections including n-fault programs with different test case coverage of those faults. We have explored the frequency of different classes of dual-fault programs in our data set of student submissions and demonstrated the preliminary application of our tool using an extended algorithm covering these programs without the solver cost of using a classic extension to n-faults with multiple toggle variables.

In this thesis we have answered the research questions set out in section 1.1:

RQ1 We have found that, for the small programs analysed here, model-based fault localisation provides better narrowing results in the same time constraints as spectrum-based methods when using large test suites.

RQ2 Our model-based fault localisation approach has accurately pin-pointed fault locations (in absolute and relative terms) on the real programs which we have tested it against.

RQ3 We have not found an expansion of this model-based fault localisation techniques to synthesise high-quality repairs from the low-quality repair byproducts of the localisation process in competitive run-time.

RQ4 Our model-based fault localisation approach has provided initial results beyond a single-fault assumption without significantly expanding the symbolic load of analysis of the programs localised.

RQ5 Our approach has shown potential benefits for the pedagogical domain in the areas of automated student feedback, grading feedback, and isolating a class of programs which some automated grading fails to correctly classify as "almost correct". This has been shown applied to real-world student coursework code submissions.

7.1 Future Work

Future studies to continue this research can survey the real-world performance of our tool on new data sets and expanding our tools to new program types (including new languages), fault classes, and symbolic analysis tools. The underlying methods and optimised algorithms developed and implemented into our current tools are programming language and symbolic analyser agnostic. This will allow future tests using other programming languages and beyond the restrictions (such as no floating point symbolic analysis) of the currently selected downstream components. Such an extension would provide results for the limits of scalability of this approach when attempting a wider range of larger programs than the Siemens suite. An extended transformation system that extracted more implicit assignments or modelled non-assignment points of repair would demonstrate localisation on a larger set of program locations.

Our current research points towards the study of tool-assisted learning in classroom environments and the effect on student progress when one group is provided with this high quality localisation information in typical student programming tasks. Such studies can validate this feedback as valuable for novices, improving total debugging time and reducing the number of students who stop before completing a source code submission fully conforming to the provided test suite. The integration of this localisation feedback into student integrated development environments must be managed to maximise and quantify student comfort and views on the ease-of-use of this additional information. Future studies could be done with direct interaction tracking, surveys, or analysis of achievement changes when this tool is introduced to an existing course.

References

[1] A. Adam, J.-P. Laurent: LAURA, a System to Debug Student Programs. In *Artificial Intelligence* **15**(1–2), pp. 75–122. Elsevier, 1980.

[2] M. Ahmadzadeh, D. Elliman, C. Higgins: An Analysis of Patterns of Debugging Among Novice Computer Science Students. In *Proc. 10th Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '05, pp. 84–88. ACM, 2005.

[3] K. M. Ala-Mutka: A Survey of Automated Assessment Approaches for Programming Assignments. In *Computer Science Education* **15**(2), pp. 83–102. Taylor & Francis, 2005.

[4] P. Antonucci, C. Estler, D. Nikolić, M. Piccioni, B. Meyer: An Incremental Hint System For Automated Programming Assignments. In *Proc. 20th Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '15, pp. 320–325. ACM, 2015.

[5] A. Armando, J. Mantovani, L. Platania: Bounded Model Checking of Software Using SMT Solvers Instead of SAT Solvers. In *Proc. 13th International Conference on Model Checking Software*, SPIN '06, LNCS **3925**, pp. 146–162. Springer, 2006.

[6] B. Beizer: Software Testing Techniques. Van Nostrand Reinhold, 1990.

[7] E. Bendersky: Pycparser - C Parser and AST Generator Written in Python. `https://github.com/eliben/pycparser`, 2012.

[8] S. D. Benford, E. K. Burke, E. Foxley, C. A. Higgins: The Ceilidh System for the Automatic Grading of Students on Programming Courses. In *Proc. 33rd Annual Southeast Regional Conference*, ACM-SE 33, pp. 176–182. ACM, 1995.

[9] G. Birch, B. Fischer, M. Poppleton: Fast Model-Based Fault Localisation with Test Suites. In *Proc. 9th International Conference on Tests and Proofs*, TAP '15, LNCS **9154**, pp. 38–57. Springer, 2015.

[10] G. Birch, B. Fischer, M. Poppleton: Using Fast Model-Based Fault Localisation to Aid Students in Self-Guided Program Repair and to Improve Assessment. In

Proc. 21st Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE '16, pp. 168–173. ACM, 2016.

[11] H. Blau, J. E. B. Moss: FrenchPress Gives Students Automated Feedback on Java Program Flaws. In *Proc. 20th Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '15, pp. 15–20. ACM, 2015.

[12] K. Buffardi, S. H. Edwards: Exploring Influences on Student Adherence to Test-Driven Development. In *Proc. 17th Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '12, pp. 105–110. ACM, 2012.

[13] K. Buffardi, S. H. Edwards: Responses to Adaptive Feedback for Software Testing. In *Proc. 19th Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '14, pp. 165–170. ACM, 2014.

[14] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, D. R. Engler: EXE: Automatically Generating Inputs of Death. In *Proc. 13th Conference on Computer and Communications Security*, CCS '06, pp. 322-335. ACM, 2006.

[15] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, D. R. Engler: EXE: Automatically Generating Inputs of Death. In *Trans. on Information and System Security* **12**(2), 10A. ACM, 2008.

[16] S. Chandra, E. Torlak, S. Barman, R. Bodik: Angelic Debugging. In *Proc. 33rd International Conference on Software Engineering*, ICSE '11, pp. 121–130. ACM, 2011.

[17] P. Cheeseman, B. Kanefsky, W. M. Taylor: Where the Really Hard Problems Are. In *Proc. 12th International Joint Conference on Artificial Intelligence* **1**, IJCAI'91, pp. 331–337. Morgan Kaufmann, 1991.

[18] E. Clarke, D. Kroening, F. Lerda: A Tool for Checking ANSI-C Programs. In *Proc. 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS '04, LNCS **2988**, pp. 168–176. Springer, 2004.

[19] H. Cleve, A. Zeller: Locating Causes of Program Failures. In *Proc. 27th International Conference on Software Engineering*, ICSE '05, pp. 342–351. ACM, 2005.

[20] L. Console, G. Friedrich, D. T. Dupré: Model-Based Diagnosis Meets Error Diagnosis in Logic Programs. In *Proc. 1st International Workshop on Automated and Algorithmic Debugging*, AADEBUG '93, pp. 85–87. Springer, 1993.

[21] S. A. Cook: The Complexity of Theorem-Proving Procedures. In *Proc. 3rd Annual Symposium on Theory of Computing*, STOC '71, pp. 151–158. ACM, 1971.

[22] L. Cordeiro: SMT-Based Bounded Model Checking for Multi-Threaded Software in Embedded Systems. In *Proc. 32nd International Conference on Software Engineering*, ICSE '10, pp. 373–376. ACM, 2010.

[23] L. Cordeiro, B. Fischer, J. Marques-Silva: SMT-Based Bounded Model Checking for Embedded ANSI-C Software. In *Trans. on Software Engineering* **38**(4), pp. 957–974. IEEE, 2012.

[24] C. Csallner, Y. Smaragdakis: DSD-Crasher: a Hybrid Analysis Tool for Bug Finding. In *Proc. International Symposium on Software Testing and Analysis*, ISSTA '06, pp. 245–254. ACM, 2006.

[25] J. de Kleer, B. Williams: Diagnosing Multiple Faults. In *Artificial Intelligence* **32**(1), pp. 97–130. Elsevier, 1987.

[26] J. de Kleer, J. Kurien: Fundamentals of Model-Based Diagnosis. In *Proc. 5th IFAC Symposium on Fault Detection, Supervision and Safety of Technical Processes*, pp. 25–36. Elsevier, 2003.

[27] R. A. DeMillo, R. J. Lipton, F. G. Sayward: Hints on Test Data Selection: Help for the Practicing Programmer. In *Computer* **11**(4), pp. 34–41. IEEE, 1978.

[28] L. de Moura, N. Bjørner: Satisfiability Modulo Theories: Introduction and Applications. In *Communications of the ACM* **54**(9), pp. 69–77. ACM, 2011.

[29] P. Denny, A. L. Reilly, D. Carpenter: Enhancing Syntax Error Messages Appears Ineffectual. In *Proc. 19th Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '14, pp. 273–278. ACM, 2014.

[30] H. Do, S. Elbaum, G. Rothermel: Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. In *Empirical Software Engineering* **10**(4), pp. 405–435. Springer, 2005.

[31] S. H. Edwards: Using Software Testing to Move Students from Trial-and-Error to Reflection-in-Action. In *Proc. 35th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '04, pp. 26–30. ACM, 2004.

[32] S. H. Edwards, M. A. Perez-Quinones: Web-CAT: Automatically Grading Programming Assignments. In *Proc. 13th Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '08, pp. 328–328. ACM, 2008.

[33] S. Elbaum, A. G. Malishevsky, G. Rothermel: Prioritizing Test Cases for Regression Testing. In *Proc. International Symposium on Software Testing and Analysis*, ISSTA '00, pp. 102–112. ACM, 2000.

[34] E. Fast, C. Le Goues, S. Forrest, W. Weimer: Designing Better Fitness Functions for Automated Program Repair. In *Proc. 12th Annual Conference on Genetic and Evolutionary Computation*, GECCO '10, pp. 965–972. ACM, 2010.

[35] R. W. Floyd: Assigning Meanings to Programs. In *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics* **19**. AMS, 1967.

[36] S. Forrest, T. Nguyen, W. Weimer, C. Le Goues: A Genetic Programming Approach to Automated Software Repair. In *Proc. 11th Annual Conference on Genetic and Evolutionary Computation*, GECCO '09, pp. 947–954. ACM, 2009.

[37] G. Fraser, F. Wotawa, P. E. Ammann: Testing with Model Checkers: a Survey. In *Journal of Software Testing, Verification and Reliability* **19**(3), pp. 215–261. Wiley, 2009.

[38] A. Griesmayer, R. Bloem, B. Cook: Repair of Boolean Programs with an Application to C. In *Proc. 18th International Conference on Computer Aided Verification*, CAV '06, LNCS **4144**, pp. 358–371. Springer, 2006.

[39] A. Griesmayer, S. Staber, R. Bloem: Automated Fault Localization for C Programs. In *Proc. Workshop on Verification and Debugging*, V&D '06, ENTCS **174**(4), pp. 95–111. Elsevier, 2007.

[40] A. Griesmayer, S. Staber, R. Bloem: Fault Localization Using a Model Checker. In *Software Testing Verification and Reliability* **20**(2), pp. 149–173. Wiley, 2010.

[41] A. Groce, S. Chaki, D. Kroening, O. Strichman: Error Explanation with Distance Metrics. In *International Journal on Software Tools for Technology Transfer* **8**(3), pp. 229–247. Springer, 2006.

[42] A. Groce, W. Visser: What Went Wrong: Explaining Counterexamples. In *Proc. 10th International Conference on Model Checking Software*, SPIN '03, pp. 121–136. Springer, 2003.

[43] B. Hailpern, P. Santhanam: Software Debugging, Testing, and Verification. In *IBM Systems Journal* **41**(1), pp. 4–12. IBM, 2002.

[44] Hawk-Eye. `http://code.google.com/p/hawk-eye/`, 2010.

[45] M. T. Helmick: Interface-based Programming Assignments and Automatic Grading of Java Programs. In *Proc. 12th Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '07, pp. 63–67. ACM, 2007.

[46] J. B. Hext, J. W. Winings: An Automatic Grading Scheme for Simple Programming Exercises. In *Communications of the ACM* **12**(5), pp. 272–275. ACM, 1969.

[47] C. Higgins, T. Hegazy, P. Symeonidis, A. Tsintsifas: The CourseMarker CBA System: Improvements over Ceilidh. In *Education and Information Technologies* **8**(3), pp. 287–304. Kluwer, 2003.

[48] R. Hildebrandt, A. Zeller: Simplifying Failure-Inducing Input. In *Proc. SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '00, pp. 135–145. ACM, 2000.

[49] C. A. R. Hoare: An Axiomatic Basis for Computer Programming. In *Communications of the ACM* **12**(10), pp. 576–583. ACM, 1969.

[50] J.-M. Hoc, T. R. G. Green, R. Samurçay, D. J. Gilmore: Psychology of Programming. Academic Press, 1990.

[51] J. Hollingsworth: Automatic Graders for Programming Classes. In *Communications of the ACM* **3**(10), pp. 528–529. ACM, 1960.

[52] I. Hussain, C. Csallner: DSDSR: a Tool that Uses Dynamic Symbolic Execution for Data Structure Repair. In *Proc. 8th International Workshop on Dynamic Analysis*, WODA '10, pp. 20–25. ACM, 2010.

[53] M. Hutchins, H. Foster, T. Goradia, T. Ostrand: Experiments on the Effectiveness of Dataflow- and Control-Flow-Based Test Adequacy Criteria. In *Proc. 16th International Conference on Software Engineering*, ICSE-16, pp. 191–200. IEEE, 1994.

[54] IEEE Standard Classification for Software Anomalies. Technical Report 1044-2009. IEEE, 2010.

[55] D. Insa, J. Silva: Semi-Automatic Assessment of Unrestrained Java Code: A Library, a DSL, and a Workbench to Assess Exams and Exercises. In *Proc. 20th Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '15, pp. 39–44. ACM, 2015.

[56] D. Jackson, M. Usher: Grading Student Programs Using ASSYST. In *Proc. 28th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '97, pp. 335–339. ACM, 1997.

[57] N. Jalbert K. Sen: A Trace Simplification Technique for Effective Debugging of Concurrent Programs. In *Proc. 18th SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pp. 57–66. ACM, 2010.

[58] M. V. Janičić, M. Nikolić, D. Tošić, V. Kuncak: Software Verification and Graph Similarity for Automated Evaluation of Students' Assignments. In *Information and Software Technology* **55**(6), pp. 1004–1016. Elsevier, 2013.

[59] H. Jin, K. Ravi, F. Somenzi: Fate and Free Will in Error Traces. In *Proc. 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS '02, LNCS **2280**, pp. 445–459. Springer, 2002.

[60] J. A. Jones, M. J. Harrold: Empirical Evaluation of the Tarantula Automatic Fault-localization Technique. In *Proc. 20th International Conference on Automated Software Engineering*, ASE '05, pp. 273–282. ACM, 2005.

[61] M. Jose, R. Majumdar: Cause Clue Clauses: Error Localization Using Maximum Satisfiability. In *Proc. 32nd Conference on Programming Language Design and Implementation*, PLDI '11, pp. 437–446. ACM, 2011.

[62] M. Joy, N. Griffiths, R. Boyatt: The Boss Online Submission and Assessment System. In *Journal on Educational Resources in Computing* **5**(3), 2A. ACM, 2005.

[63] R. Könighofer, R. Bloem: Automated Error Localization and Correction for Imperative Programs. In *Proc. International Conference on Formal Methods in Computer-Aided Design*, FMCAD '11, pp. 91–100. IEEE, 2011.

[64] R. Könighofer, R. Bloem: Repair with On-The-Fly Program Analysis. In *Proc. 8th International Haifa Verification Conference*, HVC '12, LNCS **7857**, pp. 56–71. Springer, 2012.

[65] R. Könighofer, R. Toegl, R. Bloem: Automatic Error Localization for Software Using Deductive Verification. In *Proc. 10th International Haifa Verification Conference*, HVC '14, LNCS **8855**, pp. 92–98. Springer, 2014.

[66] M-J. Laakso, T. Salakoski, A. Korhonen, L. Malmi: Automatic Assessment of Exercises for Algorithms and Data Structures - a Case Study with TRAKLA2. In *Proc. 4th Finnish/Baltic Sea Conference on Computer Science Education*, pp. 28–36. University of Joensuu, 2004.

[67] H. M. Le, D. Grosse, R. Drechsler: Automatic TLM Fault Localization for SystemC. In *Trans. on Computer-Aided Design of Integrated Circuits and Systems* **31**(8), pp. 1249–1262. IEEE, 2012.

[68] C. Le Goues, M. Dewey-Vogt, S. Forrest, W. Weimer: A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 Each. In *Proc. 34th International Conference on Software Engineering*, ICSE '12, pp. 3–13. IEEE, 2012.

[69] C. Le Goues, T. Nguyen, S. Forrest, W. Weimer: GenProg: A Generic Method for Automatic Software Repair. In *Trans. on Software Engineering* **38**(1), pp. 54–72. IEEE, 2012.

[70] C. Le Goues, W. Weimer, S. Forrest: Representations and Operators for Improving Evolutionary Software Repair. In *Proc. 14th Annual Conference on Genetic and Evolutionary Computation*, GECCO '12, pp. 959–966. ACM, 2012.

[71] B. Liskov, S. Zilles: Specification Techniques for Data Abstractions. In *Proc. International Conference on Reliable Software* **10**(6), pp. 72–87. ACM, 1975.

[72] R. Lister, T. Clear, Simon, D. J. Bouvier, P. Carter, A. Eckerdal, J. Jacková, M. Lopez, R. McCartney, P. Robbins, O. Seppälä, E. Thompson: Naturally Occurring

Data As Research Instrument: Analyzing Examination Responses to Study the Novice Programmer. In *SIGCSE Bulletin* **41**(4), pp. 156–173. ACM, 2010.

[73] T. MacWilliam, D. J. Malan: Streamlining Grading Toward Better Feedback. In *Proc. 18th Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '13, pp. 147–152. ACM, 2013.

[74] R. Majumdar, K. Sen: Hybrid Concolic Testing. In *Proc. 29th International Conference on Software Engineering*, ICSE '07, pp. 416–426. IEEE, 2007.

[75] L. Malmi, A. Korhonen, R. Saikkonen: Experiences in Automatic Assessment on Mass Courses and Issues for Designing Virtual Courses. In *Proc. 7th Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '02, pp. 55–59. ACM, 2002.

[76] W. Mayer, M. Stumptner: Evaluating Models for Model-Based Debugging. In *Proc. 23rd International Conference on Automated Software Engineering*, ASE '08, pp. 128–137. IEEE, 2008.

[77] W. Mayer, M. Stumptner: Model-Based Debugging - State of the Art And Future Challenges. In *Proc. Workshop on Verification and Debugging*, V&D '06, ENTCS **174**(4), pp. 61–82. Elsevier, 2007.

[78] M. McCracken, V. Almstrum, D. Diaz, M. Guzdial, D. Hagan, Y. B. D. Kolikant, C. Laxer, L. Thomas, I. Utting, T. Wilusz: A Multi-National, Multi-Institutional Study of Assessment of Programming Skills of First-Year CS Students. In *Working Group Reports from 6th Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE-WGR '01, pp. 125–180. ACM, 2001.

[79] L. Naish, H. J. Lee, K. Ramamohanarao: A Model for Spectra-Based Software Diagnosis. In *Trans. on Software Engineering and Methodology* **20**(3), 11A. ACM, 2011.

[80] K. A. Naudé, J. H. Greyling, D. Vogts: Marking Student Programs Using Graph Similarity. In *Journal of Computers and Education* **54**(2), pp. 545–561. Elsevier, 2010.

[81] G. Nelson, D. C. Oppen: Simplification by Cooperating Decision Procedures. In *Trans. on Programming Languages and Systems* **1**(2), pp. 245–257. ACM, 1979.

[82] H. D. T. Nguyen, D. Qi, A. Roychoudhury, S. Chandra: SemFix: Program Repair via Semantic Analysis. In *Proc. 35th International Conference on Software Engineering*, ICSE '13, pp. 772–781. ACM, 2013.

[83] C. Parnin, A. Orso: Are Automated Debugging Techniques Actually Helping Programmers? In *Proc. International Symposium on Software Testing and Analysis*, ISSTA '11, pp. 199–209. ACM, 2011.

[84] Y. Pei, Y. Wei, C. A. Furia, M. Nordio, B. Meyer: Code-Based Automated Program Fixing. In *Proc. 26th International Conference on Automated Software Engineering*, ASE '11, pp. 392–395. IEEE, 2011.

[85] D. N. Perkins, C. Hancock, R. Hobbs, F. Martin, R. Simmons: Conditions of Learning in Novice Programmers. In *Journal of Educational Computing Research* **2**(1), pp. 37–55. Sage, 1986.

[86] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W. F. Wong, Y. Zibin, M. D. Ernst, M. Rinard: Automatically Patching Errors in Deployed Software. In *Proc. SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pp. 87–102. ACM, 2009.

[87] H. Pham: Software Reliability. Springer, 2000.

[88] L. H. Pham, G. V. Trinh, M. H. Dinh, N. P. Mai, T. T. Quan, H. Q. Ngo: Assisting Students in Finding Bugs and their Locations in Programming Solutions. In *International Journal of Quality Assurance in Engineering and Technology Education* **3**(2), pp. 12–27. IGI, 2014.

[89] V. Pieterse: Automated Assessment of Programming Assignments. In *Proc. 3rd Computer Science Education Research Conference on Computer Science Education Research*, CSERC '13, pp. 45–56. ACM, 2013.

[90] K.A. Reek: The TRY System -or- How to Avoid Testing Student Programs. In *Proc. 20th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '89, pp. 112–116. ACM, 1989.

[91] A. L. Reilly, P. Denny, D. Kirk, E. Tempero, S. Y. Yu: On the Differences Between Correct Student Solutions. In *Proc. 18th Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '13, pp. 177–182. ACM, 2013.

[92] R. Reiter: A Theory of Diagnosis from First Principles. In *Artificial Intelligence* **32**(1), pp. 57–95. Elsevier, 1987.

[93] M. Renieres, S. P. Reiss: Fault Localization with Nearest Neighbor Queries. In *Proc. 18th International Conference on Automated Software Engineering*, ASE '03, pp. 30–39. IEEE, 2003.

[94] H. Riener, R. Bloem, G. Fey: Test Case Generation from Mutants Using Model Checking Techniques. In *Proc. 4th International Conference on Software Testing, Verification and Validation Workshops*, ICSTW '11, pp. 388–397. IEEE, 2011.

[95] T. Rosenthal, P. Suppes, N. Ben-Zvi: Automated Evaluation Methods with Attention to Individual Differences: a Study of a Computer-Based Course in C. In

Proc. 32nd Annual Frontiers in Education, FIE '02, pp. T1B-7–T1B-12. IEEE, 2002.

[96] S. K. Sahoo, J. Criswell, C. Geigle, V. Adve: Using Likely Invariants for Automated Software Fault Localization. In *Proc. 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pp. 139–152. ACM, 2013.

[97] R. Saikkonen, L. Malmi, A. Korhonen: Fully Automatic Assessment of Programming Exercises. In *Proc. 6th Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '01, pp. 133–136. ACM, 2001.

[98] R. Santelices, J. A. Jones, Y. Yu, M. J. Harrold: Lightweight Fault-Localization Using Multiple Coverage Types. In *Proc. 31st International Conference on Software Engineering*, ICSE '09, pp. 56–66. IEEE, 2009.

[99] D. Schuler, V. Dallmeier, A. Zeller: Efficient Mutation Testing by Checking Invariant Violations. In *Proc. 18th International Symposium on Software Testing and Analysis*, ISSTA '09, pp. 69–80. ACM, 2009.

[100] E. Schulte, S. Forrest, W. Weimer: Automated Program Repair Through the Evolution of Assembly Code. In *Proc. 25th International Conference on Automated Software Engineering*, ASE '10, pp. 313–316. ACM, 2010.

[101] K. Sen, D. Marinov, G. Agha: CUTE: a Concolic Unit Testing Engine for C. In *Proc. 10th European Software Engineering Conference*, ESEC/FSE-13, pp. 263–272 ACM, 2005.

[102] R. Singh, S. Gulwani, A. S. Lezama: Automated Feedback Generation for Introductory Programming Assignments. In *Proc. 34th Conference on Programming Language Design and Implementation*, PLDI '13, pp. 15–26. ACM, 2013.

[103] E. Soloway, J. C. Spohrer: Studying the Novice Programmer. Erlbaum, 1988.

[104] J. Spacco, D. Fossati, J. Stamper, K. Rivers: Towards Improving Programming Habits to Create Better Computer Science Course Outcomes. In *Proc. 18th Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '13, pp. 243–248. ACM, 2013.

[105] J. Spacco, D. Hovemeyer, W. Pugh, F. Emad, J. K. Hollingsworth, N. P. Perez: Experiences with Marmoset: Designing and Using an Advanced Submission and Testing System for Programming Courses. In *Proc. 11th Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '06, pp. 13–17. ACM, 2006.

[106] S. Staber, R. Bloem: Fault Localization and Correction with QBF. In *Proc. 10th International Conference on Theory and Applications of Satisfiability Testing*, SAT '07, LNCS **4501**, pp. 355–368. Springer, 2007.

[107] M. Striewe, M. Balz, M. Goedicke: A Flexible and Modular Software Architecture for Computer Aided Assessments and Automated Marking. In *Proc. 1st International Conference on Computer Supported Education*, CSEDU '09, pp. 54–61. INSTICC, 2009.

[108] A. Sülflow, G. Fey, R. Bloem, R. Drechsler: Debugging Design Errors by Using Unsatisfiable Cores. In *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, MBMV '08, pp. 159–168. ITG/GI/GMM Workshop, 2008.

[109] G. Tassey: The Economic Impacts of Inadequate Infrastructure for Software Testing. RTI Technical Report 7007-011. NIST, 2002.

[110] N. Tillmann, J. de Halleux, T. Xie, S. Gulwani, J. Bishop: Teaching and Learning Programming and Software Engineering via Interactive Gaming. In *Proc. 35th International Conference on Software Engineering*, ICSE '13, pp. 1117–1126. IEEE, 2013.

[111] G. Tremblay, F. Guérin, A. Pons, A. Salah: Oto, a Generic and Extensible Tool for Marking Programming Assignments. In *Journal of Software Practice and Experience* **38**(3), pp. 307–333. Wiley, 2008.

[112] I. Vessey: Expertise in Debugging Computer Programs: A Process Analysis. In *International Journal of Man-Machine Studies* **23**(5), pp. 459–494. Academic Press, 1985.

[113] C. von Essen, B. Jobstmann: Program Repair Without Regret. In *Proc. 25th International Conference on Computer Aided Verification*, CAV '13, LNCS **8044**, pp. 896–911. Springer, 2013.

[114] C. Wang, Z. Yang, F. Ivančić, A. Gupta: Whodunit? Causal Analysis for Counterexamples. In*Proc. 4th International Conference on Automated Technology for Verification and Analysis*, ATVA'06, LNCS **4218**, pp. 82–95. Springer, 2006.

[115] Q. Wang, C. Parnin, A. Orso: Evaluating the Usefulness of IR-based Fault Localization Techniques. In *Proc. International Symposium on Software Testing and Analysis*, ISSTA '15, pp. 1–11. ACM, 2015.

[116] T. Wang, X. Su, Y. Wang, P. Ma: Semantic Similarity-Based Grading of Student Programs. In *Journal of Information and Software Technology* **49**(2), pp. 99–107. Elsevier, 2007.

[117] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, A. Zeller: Automated Fixing of Programs with Contracts. In *Proc. 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pp. 61–72. ACM, 2010.

[118] W. Weimer, T. Nguyen, C. Le Goues, S. Forrest: Automatically Finding Patches Using Genetic Programming. In *Proc. 31st International Conference on Software Engineering*, ICSE '09, pp. 364–374. IEEE, 2009.

[119] D. West, T. Grant, M. Gerush, D. D'Silva: Agile Development: Mainstream Adoption has Changed Agility. Forrester Research, 2010.

[120] J. Whalley, N. Kasto: A Qualitative Think-Aloud Study of Novice Programmers' Code Writing Strategies. In *Proc. 19th Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '14, pp. 279–284. ACM, 2014.

[121] W. E. Wong, R. Gao, Y. Li, R. Abreu, F. Wotawa: A Survey on Software Fault Localization. In *Trans. on Software Engineering* **42**(8), pp. 707–740. IEEE, 2016.

[122] D. Wood, J. S. Bruner, G. Ross: The Role of Tutoring in Problem Solving. In *Journal of Child Psychology and Psychiatry* **17**(2), pp. 89–100. Pergamon, 1976.

[123] X. Xie, T. Y. Chen, F. C. Kuo, B. Xu: A Theoretical Analysis of the Risk Evaluation Formulas for Spectrum-Based Fault Localization. In *Trans. on Software Engineering and Methodology* **22**(4), 31A. ACM, 2013.

[124] X. Xie, F. C. Kuo, T. Y. Chen, S. Yoo, M. Harman: Provably Optimal and Human-Competitive Results in SBSE for Spectrum Based Fault Localisation. In *Proc. 5th International Symposium on Search Based Software Engineering*, SSBSE '13, LNCS **8084**, pp. 224–238. Springer, 2013.

[125] S. Yoo, M. Harman, D. Clark: Fault Localization Prioritization: Comparing Information Theoretic and Coverage Based Approaches. In *Trans. on Software Engineering and Methodology* **22**(3), 19A. ACM, 2013.

[126] A. Zeller: Yesterday, my Program Worked. Today, it Does Not. Why? In *Proc. 7th European Software Engineering Conference*, ESEC/FSE-7, LNCS **1687**, pp. 253–267. Springer, 1999.

[127] A. Zeller: Automated Debugging: Are We Close. In *Computer* **34**(11), pp. 26–31. IEEE, 2001.

[128] A. Zeller: Isolating cause-effect chains from computer programs. In *Proc. 10th SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '02/FSE-10, pp. 1–10. ACM, 2002.

[129] A. Zeller: Isolating Cause-Effect Chains with AskIgor. In *Proc. 11th IEEE International Workshop on Program Comprehension*, IWPC '03, pp. 296–297. IEEE, 2003.