Fast Model-Based Fault Localisation with Test Suites

Geoff Birch[1(✉)], Bernd Fischer[2], and Michael R. Poppleton[1]

[1] University of Southampton, Southampton SO17 1BJ, UK
{gb2g10,mrp}@ecs.soton.ac.uk
[2] Stellenbosch University, Stellenbosch 7602, Matieland, South Africa
bfischer@cs.sun.ac.za

Abstract. 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 the previously published implementation by Griesmayer et al.

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 where faults can be fixed by changing a single assignment.

Keywords: Automated debugging · Fault localisation · Symbolic execution

1 Introduction

Fault localisation, i.e. the identification of program locations that can cause erroneous state transitions which eventually lead to observed program failures, is a critical component of the debugging cycle. Since it puts a significant time [26, 27] and expertise burden [1,34] on programmers, a variety of different automated fault localisation methods have been proposed [4,6,11,13,14,17,29–31].

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 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.

J.C. Blanchette and N. Kosmatov (Eds.): TAP 2015, LNCS 9154, pp. 38–57, 2015.
DOI: 10.1007/978-3-319-21215-9_3

Model-based fault localisation [28] (sometimes also called model-based debugging [7]) is the application of model-based diagnosis methods [18] 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 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 [11] describes a method in which the model (in form of a logical satisfiability problem) is derived by running the CBMC model checker over the transformed program, and analysed by means of a SAT solver. The transformation "inverts" the program's specification (cf. section 2, producing failures where the original program would complete and blocking paths where the original program would fail), and replaces each assignment by 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, Griesmayer's technique requires detailed specifications to achieve acceptable precision—the weaker the specification, the more program locations are flagged as potential faults. Unfortunately, such detailed specifications rarely exist in practice. What *does* commonly exist, though, are extensive unit test suites, in particular in the context of modern test-driven design approaches. Griesmayer has shown that his technique can be extended to work with (failing) test cases, but the published results [11] are prohibitively slow. The bad performance is caused by Griesmayer's 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 2 for more details). However, the more locations the solver needs to explore the longer each analysis takes. Moreover, the algorithm contains no optimizations to deal with test cases that generate intractable problems for the solver. Approximating model-based fault localisation approaches for test suites, such as Jose and Majumdar [17] (cf. section 6), can run faster but can also miss a true fault location when evaluating a test case.

Our approach addresses these shortcomings, which leads to typical speed-ups of more than two orders of magnitude compared to Griesmayer's results, and yields a performance in line with current approximation techniques such as [17]. It still iterates over the failing test cases and runs a Griesmayer-style localisation task for each individual test case, but maintains a *whitelist* of still viable fault locations which is narrowed down as the localisation tasks return. The algorithm manages individual localisation tasks via a task pool to take advantage of the underlying multi-core hardware, 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 become tractable. 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 suite. 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 [9] or KLEE [3] symbolic analysers, and thus Z3 [23] as underlying solver, and with PyCParser [2] handling the program transformations to encode the specification and other model constraints. The algorithm inherits the underlying behaviour, in respect of method calls, unrolling loops, and so on, of the symbolic analyser used. We demonstrate this algorithm on the defective TCAS program variants from the Siemens [15] repository.

2 Model-Based Fault Localisation

Model-based fault localisation techniques are derived from the extension of model-based diagnosis [18,28], used to reason about digital circuits, into the software domain.

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 which analyses the symbolic execution of the system symbolically. A solver is invoked that can evaluate the logic expressions. This returns constraint satisfiability results for the expressions.

This symbolic analysis may be accelerated with the use of built-in theories to compactly reason over the logic expression [25]. The symbolic analyser uses these results to generate traces of specification violating execution paths (counterexamples), 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 [17]. Programs can be modified, e.g. augmented with additional predicates, to generate more information from the data in the failing traces.

Griesmayer [11] described a technique 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 the CBMC model checker, which then analyses the model using a SAT solver. 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 counterexamples. The purpose of this inversion is to provide the inverted output. Hence, traces that originally returned counterexamples due to specification failure do no longer, and traces that satisfy the specification now generate counterexamples. The exploration becomes one searching for a trace to the exit point that satisfies the initial specifications. Thus, in the simple inverted program an exiting trace will not be found for a failing test case.

To enable the search for potential fault locations on this now inverted program, an unsigned int global toggle variable, `__toggle`, 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 (in our example 39 locations). In a KLEE-style API this would be achieved by the calls:

```
klee_make_symbolic(&__toggle); klee_assume(__toggle < 39);
```

Assignments in the source program are modified to become conditional assignments that flip to an unconstrained symbolic value, `__sym`, when toggled. Hence, if we assume that `var = a + b;` is the third assignment in the program, the translation yields:

```
var = ((__toggle == 2)? __sym : a + b);
```

The generated counterexamples from this new model provide the toggle values that identify candidate assignment locations as sites of possible repair. The technique described by Griesmayer requires that the input programs have detailed specifications. In practice, these typically do not exist. Griesmayer demonstrated [11] an extension to this technique using a test suite as specification. A failing test case is encoded into the input program and then inverted. The results for each toggle value therefore identify candidate repairs for that failing test case. Any assignment that was 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, `__sym`, can be different and will be reported for each location flagged by this process. The flagging process means the chosen `__sym` 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 was too high using the common localisation performance measure of the Siemens TCAS [15] variants.

3 The 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.

We transform an input C program using an extended Griesmayer inversion process (see subsection 3.1). This is handed to a worker task (see subsection 3.2) 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 3.3). 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 renamed to K during narrowing inside Algorithm 1. In the Process Manager algorithm, P is the worker pool manager.

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.

3.1 Program Transformation into Model

The program under analysis is transformed in two stages. First, an initial generic transformation *ApplyGenericTransforms* is applied once, as described in subsection 3.3. This includes the Griesmayer inversion, as outlined in section 2, 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, 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.

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. To encode the desired output 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.

An optimisation of this is used for simple programs, such as TCAS, that only require a single 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 variables 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.

Program transformations can extend the number of assignments visible for repair, for example by treating any return statement that returns more than a single variable as an assignment to a temporary variable that is then returned. If returns are always considered to be implicit assignments then localisation is expanded from assignment locations to also include all return statements, as our tool does. `return (y * z);` becomes `return ((__toggle == 3)? __sym : y * z);` for symbolic value `__sym` at the fourth assignment in the program.

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 `klee_assume(__toggle != 11);` to the program body for any values to be omitted from this search (blocking the twelfth assignment in this example).

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. 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.

3.2 The Test Case Search Algorithm

Algorithm 1 outlines a single task, which defaults to being deployed to a single core via the pool manager discussed in subsection 3.3. When *AddTask* is called on lines 5 and 20 of Algorithm 2 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 7 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: **return** K;

Algorithm 1. Test Case Search Worker Algorithm

ApplySpecificTransforms is discussed in subsection 3.1. 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 *ApplySpecificTransforms*) 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 counterexamples 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 `__toggle` and `__sym`. In the case of ESMBC, this process is iterative as only one counterexample 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 Algorithm 1, 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 – 6 the old whitelist is replaced with the new list of toggle values returned by the counterexamples, generated by the symbolic analyser execution. This narrowing checks the counterexample 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 [8]. This type of SAT problem has been shown [5] 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 counterexample generated has a corresponding solver stage and there are an unknown number of counterexamples multiplying this unknown per-instantiation processing time.

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

3.3 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 21 and 16 of Algorithm 2). 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 must

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: WithoutImprovement = 0;
7: while P.HasOpenWorkers() do
8:    Sleep(TickTimerMS);
9:    if P.HasCompletedTasks() then
10:      Sleep(0.25 * P.GetFastestCompletedTaskTime());
11:      for w in P.GetCompletedTasks() do
12:         Lnew = w.Locations ∩ L;
13:         if Lnew == L then WithoutImprovement++;
14:         else WithoutImprovement = 0;
15:         L = Lnew;
16:      if WithoutImprovement > 15 then return L;
17:      F.Enqueue(P.ReturnTCsForIncompleteTasks());
18:      for w in P.GetAllTasks(); do
19:         P.RemoveTask(w);
20:         if Len(F) > 0 then P.AddTask(D, Dequeue(F), L);
21: return L;
```

Algorithm 2. Pool Manager Algorithm

be an unordered list. This is because the optimised search varies in performance based on the test case order, as discussed in section 4.

The tool applies the generic transform stage discussed in subsection 3.1 during line 1's *ApplyGenericTransforms*. 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. Algorithm 1) 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 and 5 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 7 of Algorithm 2. 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 125% of the time the fastest task completed with (line 10). 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 timeout are iterated on lines 11 – 15. 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 16) 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 17. 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 19). The next batch of tasks is despatched to the pool (line 20) 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.

4 Preliminary Experimental Results

4.1 Experimental Setup

We 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) [10]. 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. 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.

In Table 1 and Table 2 the headings refer to the following data sources and test platforms: Griesmayer's original data [11, §4, Table1, p. 104] (**G**) uses CBMC on a 2.8GHz Pentium 4; our naïve reimplementation of Griesmayer's algorithm (**N**) uses ESBMC v1.17 on a 3GHz Core2Duo E8400; our new algorithm using ESBMC (**E**) and KLEE (**K**) as back-end both ran on a 3.1GHz Core i5-2400, with the ESBMC [9] v1.21 and KLEE [3] (for LLVM 3.4) symbolic analysers; and Jose and Majumdar's results (**J**) are reconstructed from data provided [17, §6, Table1, 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 repair for all failing test cases. In Table 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.

Table 1. Seconds to Return Location Set for Test Suite. Griesmayer's original data [11, 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 [17, Table1, p. 443] (**J**).

	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**

4.2 Run Time Performance

Griesmayer 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" [11, §4.1, p. 105]. We reimplemented this naïve process as described in Griesmayer's paper. This is running on more modern hardware and updated to use the current, CBMC-derived, SMT 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.

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 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, 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 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 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

Table 2. Percentage of Lines of Code Returned by Localisation; see Table 1 for legend

	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**

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. 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 multiprocess design that takes advantage of modern consumer processor architectures.

4.3 Localisation Performance

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 only explore the 34 explicit assignments in the TCAS variants, which increases the localisation performance we would expect to see in Table 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 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 3.1. 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 a known correct value for their assignments. Any location flagged by a failing test cases will have, in the `__sym` value extractable from the counterexample, the assignment which repairs that trace and so test case. It is thusly possible, for locations flagged by all failing test cases, to construct a complete look-up table at that assignment location 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 repair 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 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 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. 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; while not the injected fault location, they are locations with a repair.

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 2 for Jose and Majumdar, when compared to Griesmayer or our algorithm can be explained by this different scope of potentially returned locations, where additional genuine repairs 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.

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 Limitations

In subsection 4.3 we have already discussed issues with making direct localisation performance comparisons. The single-fault assumption that underpins our model prevents any meaningful localisation performance on the seven TCAS variants which contain multiple injected faults, where positive localisation results would be derived from blind chance. The performance of a single-fault localiser will be faster than more extensive searches that include k-fault analysis. However, the single-fault assumption is common in fault localisation techniques.

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 complicated control flow. This issue is common to all tools which demonstrate their localisation effectiveness on the TCAS variants.

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.

Familiarity with the specific problem being solved (the small Siemens program) could have subconsciously influenced our research direction towards a process that is unfairly high performing for this specific problem and does not adequately generalise to C code. To minimise this risk, our choice of time-out,

sleep delay, and early termination values have not been tuned or selected for optimising with respect to the test suite; that would compromise our preliminary results.

5 Extensions

We currently present results where the known correct output is provided with the test case input in the test suite. This is encoded as the assertion that the correct output is generated by the program to comply with the specification for that test case. This can be weakened if a test case only specifies that the output does *not* take a given value, i.e. for input x, the output is *not* y. This weakened specification only requires the assertion that the known incorrect output is not generated. These two types of test cases can be mixed, providing a test suite with both known correct outputs and known wrong outputs for failing test cases. They would require two separate transformed programs that encoded this difference (at the generic stage). This would be a small extension for a tool that can, separately, operate on both types of failing test case. Weaker specifications would be far less restrictive on the locations where a potential repair could exist, as any location where the final output could be nudged to no longer generate the same value would be flagged. This may limit the value of localising with such a weak specification.

As described in subsection 3.1, the current tool's transformation process modifies return statements when the returned expression is more than a single variable. This allows localisation to an implicit assignment hidden by any return statement not creating a temporary variable for the returned value. We can extend this to other locations where an assignment is implicit, with a process similar to Single Assignment C transformations. This can be expanded in many ways to inspect for fault classes outside of assignments. For example, the search for spurious statements, i.e. looking for superfluous lines of code that can be safely removed. That is, we remove a statement that enables the failing test cases to pass and not regress passing test cases. Rather than inserting the toggle into the right hand side of an assignment, the entire statement can be made optional by the toggle test `if(loc != [LOC_VAL])original_statement;`. The location, when toggled, would explore how the program functioned without that statement. This sort of fault class exploration provides narrowing of the whitelist of locations when exploring passing test cases, a feature not seen in the modified assignments currently employed. This is not providing a widening of the program functionality with symbolic values but a mutation of the program when a location is activated by the toggle. These more extensive modifications to location searches outside of assignments can be done in combination with current searches or independently.

The currently discussed algorithm uses a single toggle value to activate a single location to perform the alternative assignment of a symbolic value. If multiple toggles were used with `(__toggle1 == 3 || __toggle2 == 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 extension was proposed as an extension in the Griesmayer paper [11]. This would increase the solver cost due to additional non-determinism and would also increase the total combination of counterexamples returned. It may be severely limited to small input programs to retain tractability but is a theoretical extension of this process to a k-fault assumption.

The scalability limitations of our current approach will be quantified by exploring the tool's performance on larger, more complex input programs and production code samples. The transition from ESBMC to KLEE as the back-end and a concolic execution approach facilitates this expansion of the scope of input C programs.

6 Related Work

Localisation by examining counterexample traces, test cases, or other output from static and dynamic analysis tools is an active area of research [4,6,11,13, 14,17,29–31].

Griesmayer et al. [11] have first applied model-based diagnosis methods to software. Our work follows the same lines; see section 2 for a more detailed discussion. Griesmayer et al. [12] improve the original implementation to achieve times roughly comparable to our own initial re-implementation (see section 4 for details). They also expand the possible fault locations to non-assignments (e.g. expressions in control flow guards), which could easily be applied to our approach as well, although the higher number of locations considered can lead to more complicated solver problems and thus higher run times.

Königshofer and Bloem [19,20] have developed the foREnSiC system which includes a Griesmayer-style localisation. They have applied this to TCAS as well, but published results only for a few variants; here localisation times are more than an order magnitude slower (around 120 seconds) than our results. Königshofer et al. [21] report slightly improved times (around 37 seconds) but had to annotate all functions with contracts, and so do no longer work from test suites alone.

Griesmayer's approach has also been applied to hardware designs in SystemC [22], often combined with different solver technologies such as QBF [32] or unsatisfiable cores [33]. Our results indicate that "plain old SAT/SMT" is still sufficient, but these technologies could be considered as alternatives in our approach as well.

Jose and Majumdar [17] convert an input C program to a maximum Boolean satisfiability problem which is analysed with MAX-SAT solver. 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. It 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/weaknesses, of many heuristic-based fault localisation techniques. As discussed in section 4, our approach provides comparable localisation times but a higher precision.

Spectrum-based fault localisation techniques, compared in [24,35,36], 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. The best-known example of this technique is the Tarantula tool [31] with TCAS results provided earlier [16]. Tarantula provides over 50% of runs on various of the Siemens small programs (including TCAS) with a localisation performance that puts the genuine fault location in the top 10% of lines returned. But this performance is inconsistent and 7% of these runs fail to narrow the localisation list so that the genuine fault location is in the top 80% of locations.

State of the art spectrum-based fault localisation methods have recently been compared using different theoretical frameworks [24,36]. 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 [36], it is also not far behind the state of the art for TCAS. In empirical results [24, 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 the symbolic model checking approaches detailed in Table 2. The spectrum-based reporting metrics provide the average rank, as a percentage, in a ranked list of all lines of code. The symbolic model checking results report the total unranked lines flagged as suspicious, as a percentage of total lines of code. To compare these results, we must convert the unranked sets in Table 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 (4.6%) over the TCAS single defect variants is equivalent to returning the injected fault location at the 4th ranked location (2.3%). As noted in subsection 4.3, the different localisation scopes involved with each technique mean these results are not directly comparable. A spectrum-based approach will not only localise to assignment locations and TCAS is not ideally suited to providing these approaches with easily differentiable statements.

Delta Debugging [37] 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 [6] 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 the small Siemens programs [6] is worse than Tarantula's [16] results.

7 Conclusions

Our main contribution in this paper is an improved search 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. 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. 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.

References

1. Ahmadzadeh, M., Elliman, D., Higgins, C.: An analysis of patterns of debugging among novice computer science students. In: SIGCSE, pp. 84–88 (2005)
2. Bendersky, E.: Pycparser: C parser and AST generator written in Python (2012). https://github.com/eliben/pycparser
3. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: EXE: Automatically Generating Inputs of Death. ACM Trans. Inf. Syst. Secur. **12**(2), 10A (2008)
4. Chandra, S., Torlak, E., Barman, S., Bodik, R.: Angelic debugging. In: ICSE, pp. 121–130 (2011)
5. Cheeseman, P., Kanefsky, B., Taylor, W.M.: Where the really hard problems are. In: IJCAI, pp. 331–337 (1991)
6. Cleve, H., Zeller, A.: Locating causes of program failures. In: ICSE, pp. 342–351 (2005)
7. Console, L., Friedrich, G., Dupre, D.T.: Model-based diagnosis meets error diagnosis in logic programs. In: Fritzson, P.A. (ed.) AADEBUG 1993. LNCS, vol. 749, pp. 85–87. Springer, Heidelberg (1993)
8. Cook, S.A.: The complexity of theorem-proving procedures. In: Proc. ACM Symposium on Theory of Computing, pp. 151–158 (1971)
9. Cordeiro, L., Fischer, B., Marques-Silva, J.: SMT-Based Bounded Model Checking for Embedded ANSI-C Software. IEEE Trans. Softw. Engg. **38**(4), 957–974 (2012)

10. Do, H., Elbaum, S., Rothermel, G.: Supporting Controlled Experimentation with Testing Techniques. Empirical Softw. Engg., 405–435 (2005)
11. Griesmayer, A., Staber, S., Bloem, R.: Automated Fault Localization for C Programs. Electronic Notes in Theoretical Computer Science, 95–111 (2007)
12. Griesmayer, A., Staber, S., Bloem, R.: Fault localization using a model checker. Softw. Test. Verif. Reliab. **20**(2), 149–173 (2010)
13. Groce, A., Chaki, S., Kroening, D., Strichman, O.: Error explanation with distance metrics. Int. J. Softw. Tools Technol. Transf. **8**(3), 229–247 (2006)
14. Groce, A., Visser, W.: What Went Wrong: Explaining Counterexamples. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 121–135. Springer, Heidelberg (2003)
15. Hutchins, M., Foster, H., Goradia, T., Ostrand, T.: Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria. In: ICSE, pp. 191–200 (1994)
16. Jones, J.A., Harrold, M.J.: Empirical Evaluation of the Tarantula Automatic Fault-localization Technique. In: ASE, pp. 273–282 (2005)
17. Jose, M., Majumdar, R.: Cause Clue Clauses: Error Localization Using Maximum Satisfiability. SIGPLAN Not **46**(6), 437–446 (2011)
18. de Kleer, J., Williams, B.: Diagnosing multiple faults. Artificial Intelligence **32**(1), 97–130 (1987)
19. Konighofer, R., Bloem, R.: Automated error localization and correction for imperative programs. In: FMCAD, pp. 91–100 (2011)
20. Könighofer, R., Bloem, R.: Repair with On-The-Fly Program Analysis. In: Biere, A., Nahir, A., Vos, T. (eds.) HVC. LNCS, vol. 7857, pp. 56–71. Springer, Heidelberg (2013)
21. Könighofer, R., Toegl, R., Bloem, R.: Automatic Error Localization for Software Using Deductive Verification. In: Yahav, E. (ed.) HVC 2014. LNCS, vol. 8855, pp. 92–98. Springer, Heidelberg (2014)
22. Le, H.M., Grosse, D., Drechsler, R.: Automatic TLM Fault Localization for SystemC. Trans. Comp.-Aided Des. Integ. Cir. Sys. 31(8), 1249–1262 (2012)
23. de Moura, L., Bjørner, N.S.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
24. Naish, L., Lee, H.J., Ramamohanarao, K.: A Model for Spectra-based Software Diagnosis. ACM Trans. Softw. Eng. Methodol. **20**(3), 11A (2011)
25. Nelson, G., Oppen, D.C.: Simplification by Cooperating Decision Procedures. ACM Trans. Program. Lang. Syst., 245–257 (1979)
26. Nguyen, H.D.T., Qi, D., Roychoudhury, A., Chandra, S.: SemFix: program repair via semantic analysis. In: ICSE, pp. 772–781 (2013)
27. Pham, H.: Software reliability. Springer (2000)
28. Reiter, R.: A theory of diagnosis from first principles. Artificial Intelligence **32**(1), 57–95 (1987)
29. Renieres, M., Reiss, S.P.: Fault localization with nearest neighbor queries. In: ASE, pp. 30–39 (2003)
30. Sahoo, S.K., Criswell, J., Geigle, C., Adve, V.: Using likely invariants for automated software fault localization. In: ASPLOS, pp. 139–152 (2013)
31. Santelices, R., Jones, J.A., Yu, Y., Harrold, M.J.: Lightweight fault-localization using multiple coverage types. In: ICSE, pp. 56–66 (2009)
32. Staber, S., Bloem, R.: Fault Localization and Correction with QBF. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 355–368. Springer, Heidelberg (2007)

33. Sülflow, A., Fey, G., Bloem, R., Drechsler, R.: Debugging design errors by using unsatisfiable cores. In: MBMV, pp. 159–168 (2008)
34. Vessey, I.: Expertise in debugging computer programs: A process analysis. International Journal of Man-Machine Studies **23**(5), 459–494 (1985)
35. Wong, W.E., Debroy, V.: A survey of software fault localization. Tech. Rep. UTDCS-45-09, Uni. of Texas at Dallas (2009)
36. Xie, X., Chen, T.Y., Kuo, F.C., Xu, B.: A Theoretical Analysis of the Risk Evaluation Formulas for Spectrum-based Fault Localization. ACM Trans. Softw. Eng. Methodol. **22**(4), 31A (2013)
37. Zeller, A.: Yesterday, my program worked. Today, it does not. Why? SIGSOFT Softw. Eng. Notes **24**(6), 253–267 (1999)