Concolic Fault Abstraction

Chanseok Oh
New York University
Email: chanseok@cs.nyu.edu

Martin Schäf
SRI International
Email: martin.schaef@sri.com

Daniel Schwartz-Narbonne
New York University
Email: dsn@cs.nyu.edu

Thomas Wies
New York University
Email: wies@cs.nyu.edu

***Abstract*—An integral part of all debugging activities is the task of diagnosing the cause of an error. Most existing fault diagnosis techniques rely on the availability of high quality test suites because they work by comparing failing and passing runs to identify the error cause. This limits their applicability. One alternative are techniques that statically analyze an error trace of the program without relying on additional passing runs to compare against. Particularly promising are novel proof-based approaches that leverage the advances in automated theorem proving to obtain an abstraction of the program that aids fault diagnostics. However, existing proof-based approaches still have practical limitations such as reduced scalability and dependence on complex mathematical models of programs. Such models are notoriously difficult to develop for real-world programs. Inspired by concolic testing, we propose a novel algorithm that integrates concrete execution and symbolic reasoning about the error trace to address these challenges. Specifically, we execute the error trace to obtain intermediate program states that allow us to split the trace into smaller fragments, each of which can be analyzed in isolation using an automated theorem prover. Moreover, we show how this approach can avoid complex logical encodings when reasoning about traces in low-level C programs. We have conducted an experiment where we applied our new algorithm to error traces generated from faulty versions of UNIX utils such as gzip and sed. Our experiment indicates that our concolic fault abstraction scales to real-world error traces and generates useful error diagnoses.**

I. Introduction

Debugging is one of the most time consuming aspects of software development. Several studies have shown that programmers spend at least 50% of their time on debugging (see, e.g., [13], [29]). There is no doubt that any kind of automation with the effect of reducing the manual effort involved in debugging can have a significant impact on software productivity.

An integral part of all debugging activities is to diagnose the cause of the error. For example, a program execution may fail with a segmentation fault. The point of failure could be a dangling pointer that is dereferenced. The actual defect causing this failure could be a premature deallocation of memory that created the dangling pointer. Often the defect and the point of failure are far apart in the program source code. Hence, fault diagnostics becomes more and more challenging as the size of the program and the number of control flow paths increases. Automated fault diagnosis techniques alone have the potential to reduce the overall debugging time by 40% [22]. In addition, they are a key enabling technology for automating other debugging tasks, in particular, program repair [32]. In this paper, we propose a novel fault diagnosis technique that conservatively over-approximates the behavior of a faulty program trace such that only the error-relevant portions of the program are preserved and the defect is precisely identified. This abstraction of the error trace is obtained by using a combination of concrete execution and symbolic reasoning. In reference to the idea of *concolic testing* [11], [28], we refer to our approach as *concolic fault abstraction*.

Research on automated fault diagnostics has mostly focused on dynamic fault localization techniques, which try to identify a single instruction in the program that is responsible for the error or rank the instructions accordingly (e.g., [2], [23], [25], [35], [37]). The results that have been achieved with these techniques are impressive [5], [17]. However, they also have limitations. Dynamic techniques assume the availability of passing runs that are comparable to failing runs. The authors of [12] observe that the usefulness of dynamic fault localization techniques strongly correlates with the quality of the test suites that generate the passing runs. Andreas Zeller, author of the seminal work on Delta Debugging [5], [35], recently commented that a major factor for the lack of adoption of existing automated fault localization techniques is the assumption that test suites with high code coverage are available [36]. According to Zeller, this assumption contradicts the reality of how software is developed in large parts of the software industry. It therefore seems necessary to explore alternative techniques that extend and complement dynamic fault localization.

One such alternative is to analyze the faulty program fragment statically [10], [19], [30], [33]. Particularly promising are novel static approaches that leverage the recent advances in automated theorem proving technology, specifically Boolean satisfiability (SAT) and satisfiability modulo theories (SMT) solvers [10], [19]. These approaches encode error paths into *error trace formulas*. An error trace formula is an unsatisfiable logical formula. A refutation proof of this formula captures the reason why the execution of the error trace fails. By applying an automated theorem prover to obtain such proofs, the relevant portions of the trace can be automatically identified without relying on the availability of test suites. For example, in our previous work [10], we have used Craig interpolation to extract formulas from the obtained refutation proof. These so-called *error invariants* provide an explanation of the error for a given position in the trace and can be used for fault abstraction.

While proof-based fault diagnosis techniques are a promising alternative to dynamic techniques, they have practical limitations that have not been addressed in previous work. To precisely capture the behavior of an erroneous program one has to model such features as dynamic memory allocation, function calls, and inputs, over traces that can extend to large numbers of instructions. This limits to applicability of these

This work was supported in part by the National Science Foundation under grant CCS-1350574.

techniques for two reasons: 1) the length of an error trace may exceed what can be handled by existing theorem provers, and 2) there are currently no robust interpolation procedures for the theory of arrays, making it difficult to automatically infer invariants about heap allocated objects.

To address these limitations of purely static approaches, we propose a novel fault diagnosis technique that combines dynamic and static analysis of the error trace. In particular, we note that given an executable error trace, we can determine the concrete value of every variable, and every memory address, using program instrumentation and dynamic execution. This information can then be used to dramatically simplify the encoding of the error trace for static analysis. Instead of relying on complex mathematical theories to reason about pointer aliasing, we can directly determine which memory location a pointer references using the debugger. If a trace contains too many statements to analyze using a theorem prover, we can split the trace into subtraces, and use the program states observed during the concrete execution to seed the analysis of the individual subtraces.

Contributions. The paper makes the following contributions to the state of the art of static fault diagnostics:

- An algorithm that leverages information from dynamic execution to subdivide the error trace into smaller subtraces, each of which is amenable to static analysis using an automated theorem prover.
- A new memory model that uses dynamic information about the heap to enable the application of interpolation-based fault abstraction to real programs.

We demonstrate the ability of our approach to simplify and explain error traces on two real-world examples taken from faulty versions of the open source UNIX tools sed and gzip. Our experiment shows that concolic fault abstraction can scale to real-world error traces, and that our technique is effective at diagnosing error causes in failing traces without relying on the availability of additional passing traces that are sufficiently similar to the failing trace.

II. Overview

We illustrate our fault diagnosis approach using a bug from the SIR bug benchmark suite [8][1]. In particular, we consider the bug *sed1*, which is a seeded fault in the UNIX stream editor sed. Figure 1 shows the portion of sed's source code that is relevant for the seeded bug. The actual source code spans several thousand lines. The bug suppresses a call to a function that initializes a pointer. The uninitialized pointer is dereferenced later on, leading to a segmentation fault.

Generating the Error Trace. As a first step, we generate an error trace by instrumenting the program using a custom cilly [20] transformation, and then executing the program with an input that triggers the bug. The generated error trace is shorter than the full program, but still consists of 976 statements and 89 function calls.

[1]The full source code and the test input that reveals the bug are taken from the BugRedux [14], [16] distribution, and are available online.

Analyzing the Error Trace. The basis of our approach for analyzing the generated error trace is the static fault diagnosis algorithm presented in [10], [27]. This algorithm computes an *abstract slice* of the error trace that explains the faulty program behavior. The abstract slice is a subsequence of statements of the error trace that are correlated with the bug. Figures 2 and 3 show two possible abstract slices that we expect to be computed for the error trace of the faulty sed program. In fact, this is the output generated by our new algorithm. The line numbers refer to the program source code in Figure 1. Note that some of the statements encode the passing of parameters and return values of function calls.

Both of these abstract slices consist of concrete executable statements which alternate with assertions that hold before and after execution of the statements in the abstract slice. We refer to these assertions as *error invariants*. For example, in Figure 3 the error invariant `I02` holds at the beginning of the error trace, after the initialization of the global variable `last_regex`, and before the execution of the statement at line 27. That is, error invariants summarize the behavior of the statements that have been sliced from the full trace. They only capture information that is preserved by the sliced statements and that is relevant for helping the programmer isolate the cause of the bug.

Both of these possible abstract slices are useful for understanding the cause of the bug. The *address-insensitive* slice in Figure 2 represents an answer to the question "why did a particular memory location have a particular value?" In this case, the answer is that `last_regex` was initialized to 0, therefore `addr->addr_regex` pointed to the value 0, and hence `rxb` was assigned 0 before being dereferenced.

This information would be enough for the programmer to debug the problem. However, in some cases the programmer might also wonder "why did `addr->addr_regex` point to the location that it did?" In this case, we need the *address-sensitive* slice shown in Figure 3, which shows the flow of pointer assignments that were necessary to cause the bug.

Static Fault Abstraction with Error Invariants. The algorithm in [10] works by encoding the error trace into a logical formula whose satisfying assignment exactly corresponds to the states that are observed during the concrete execution of the trace. The encoding uses standard symbolic execution techniques. Namely, we first convert the trace into static single-assignment form, then translate each individual statement into a logical constraint, and finally conjoin all the constraints to a single formula. We call the resulting conjunction the *error trace formula* of the trace.

By conjoining the error trace formula with the property that is violated at the end of the trace, we obtain an unsatisfiable formula. For example, in the case of the failing sed trace, the error trace formula is conjoined with the formula `rxb != 0`. Since this property is violated at the end of the execution of the error trace, we obtain a contradiction. We then use an interpolating SMT solver to generate an error invariant for each position in the trace from a refutation proof of the unsatisfiable formula. The computed error invariants have two important properties: (1) they are satisfied by the concrete state that is observed at the respective position in the trace; and (2) any execution of the remainder of the trace starting from that

```
1  struct vector *the_program = 0;
2  ...
3  struct re_pattern_buffer *last_regex;
4  ...
5  int main (int argc, char **argv)
6  {
7    ...
8    compile_file (...);
9    ...
10   read_file (...);
11 }
12 ...
13 void compile_file (...)
14 {
15   ...
16   the_program = compile_program (...);
17   ...
18 }
19 ...
20 struct vector * compile_program (...)
21 {
22   struct sed_cmd *cur_cmd;
23   ...
24   vector = (struct vector *) malloc (sizeof (struct
         vector));
25   vector->v = (struct sed_cmd *) malloc (40 *
        sizeof (struct sed_cmd));
26   ...
27   vector->v_length = 0;
28   ...
29   cur_cmd = vector->v + vector->v_length;
30   ...
31   compile_address (&(cur_cmd->a1));
32   ...
33   return vector;
34 }
35 ...
36 int compile_address(struct addr *addr)
37 {
38   ...
39   #ifndef FAULTY_F_AG_19
40    compile_regex ();
41   #endif
42   addr->addr_regex = last_regex;
43   ...
44 }
```

```
45 void read_file (...)
46 {
47   ...
48   execute_program(the_program);
49   ...
50 }
51 ...
52 void execute_program (struct vector *vec)
53 {
54   struct sed_cmd *cur_cmd;
55   ...
56   cur_cmd = vec->v;
57   ...
58   match_address (&(cur_cmd->a1));
59   ...
60 }
61 ...
62 int match_address (struct addr *addr)
63 {
64   ...
65   re_search (addr->addr_regex, ...);
66   ...
67 }
68 ...
69 int
70 re_search (struct re_pattern_buffer *rxb, ...)
71 {
72   ...
73   unsigned num_regs = rxb->re_nsub + 1;
74   ...
75 }
```

Fig. 1. Relevant excerpt of sed's source code that shows the seeded bug. The pre-processor directive on line 39 in `compile_addr` suppresses the call to `compile_regex`, which causes `last_regex` to stay uninitialized. This, in turn, leads to a segmentation fault when accessing `rxb` on line 73 of `re_search`. The indicated call sequence causes `rxb` to point to the same address as `last_regex`.

position with a state that satisfies the error invariant will still violate the property at the end of the trace.

The generated error invariants are then propagated along the trace to identify maximal subtraces such that the same error invariant holds at both the beginning and the end of the subtrace. For instance, in the error trace for sed, the theorem prover will produce the error invariant `last_regex==0` for the position that is reached after line 3 has been executed. This error invariant also holds at the position before the statement on line 27. From the two properties of error invariants, it follows that the statements between these two positions are irrelevant for understanding the bug and, hence, can be sliced from the trace. After line 27 we now also need to keep track of the fact that `vector->v_length==0` holds. Hence, we need a new error invariant. That is, the trace may no longer fail if we continue execution after line 27 with a state that does not satisfy this property. Since we cannot further propagate the error invariant `last_regex==0` across line 27, we keep the statement at this line. We proceed like this for the entire error trace to produce the abstract slice.

Challenges to Static Fault Abstraction. Unfortunately, the simple algorithm that we outlined above cannot be applied directly to real-world error traces. There are several reasons for this. First, real-world error traces quickly become too large to be processed by today's theorem provers. This scalability issue is rooted in the NP-hardness of the underlying decision problem of checking satisfiability of the error trace formula. While the length of the sed trace is still manageable, the performance of the provers degrades for longer traces. We therefore need a way of breaking the error trace down into smaller fragments that can be analyzed in isolation.

Second, the symbolic encoding of the error trace needs to be based on a precise memory model that takes into account low-level details such as pointer arithmetic, as in lines 29 and 31 of Figure 1. Typically, in deductive reasoning about programs one uses the theory of mutable maps (commonly referred to as array

```
  //I00: true
3 last_regex = 0;
  //I01: last_regex==0 && addr==0x123
42 addr->addr_regex = last_regex;
  //I02: mem[0x123]==0 && addr==0x123
65 rxb = addr->addr_regex;
  //I03: rxb==0
73 num_regs = rxb->re_nsub + 1;
  //I04: false
```

Fig. 2. Address-insensitive slice produced by our algorithm for the sed error trace.

```
  //I00: true
3 last_regex = 0;
  //I01: last_regex==0
27 vector->v_length = 0;
  //I02: last_regex==0 &&
  //     vector->v_length==0
29 cur_cmd = vector->v + vector->v_length;
  //I03: last_regex==0 &&
  //     cur_cmd==vector->v
31 addr = &cur_cmd->a1;
  //I04: last_regex==0 &&
  //     addr==&vector->v->a1
42 addr->addr_regex = last_regex;
  //I05: vector->v->a1.addr_regex==0
16 the_program = vector;
  //I06: the_program->v->a1.addr_regex==0
48 vec = the_program;
  //I07: vec->v->a1.addr_regex==0
56 cur_cmd = vec->v;
  //I08: cur_cmd->a1.addr_regex==0
58 addr = &cur_cmd->a1;
  //I09: addr->addr_regex==0
65 rxb = addr->addr_regex;
  //I10: rxb==0
73 num_regs = rxb->re_nsub + 1;
  //I11: false
```

Fig. 3. Address-sensitive abstract slice produced by our algorithm for the sed error trace.

theory) to model the memory; see, e.g., [6], [24]. However, there is no robust implementation of an interpolating decision procedure for this theory in any off-the-shelf theorem prover. Also, the use of these theories makes automated reasoning less scalable.

Finally, certain operations that occur on the trace might not be expressible in any decidable logical theory. For example, some statements on the trace may contain nonlinear arithmetic expressions over integer variables or involve calls to library functions whose source code is not available.

Concolic Trace Splitting. Inspired by concolic testing [11], [28], we propose a concolic fault abstraction algorithm that addresses the above challenges by combining concrete execution and symbolic reasoning about the error trace.

First, we describe how we use the concrete execution of the trace to improve the scalability of our static fault abstraction. Observe that an error invariant serves as an interface between the two halves of the trace. For example, the error invariant `I10` splits the error trace into two segments, a prefix trace that executes up to the call site of function `re_search` on line 65, and a suffix trace that executes `re_search` up to line 73. In particular, we can analyze the prefix trace in isolation by viewing it as an error trace that violates the negation of `I10` at the end of its execution.

However, how can we infer an error invariant to split the error trace into smaller chunks without resorting to static analysis of the full trace? To solve this problem, we use the concrete execution of the trace to first seed the analysis of the suffix trace and then use the first error invariant in the computed abstract slice of the suffix to obtain the violated postcondition for the analysis of the prefix trace.

Concretely, in the executable error trace that we obtain from Figure 1, we insert breakpoints at each function call and at each return statement. We end up with 158 breakpoints (for some of the 89 calls we do not create breakpoints at the procedure exit because the procedure exit is not reached on our trace). The breakpoints define 159 subtraces to be analyzed. The last breakpoint is at the entry of the procedure `re_search`. Each of these subtraces is analyzed in isolation starting with the last subtrace. That is, we then run gdb up to the last breakpoint and extract the values of all variables that are active from this breakpoint to the segmentation fault. In this final segment, no global variable is read or written, so the only variables that we have to watch are the formal parameters of the procedure.

Using the information about the values of the relevant variables at the breakpoint we generate a shorter executable trace that leads to the segmentation fault. This shorter trace consists of assignments from the formal parameters of `re_search` to the extracted concrete values, and all the statements from the breakpoint to the end of the trace. On this trace segment we now apply our error-invariant-based fault diagnostics. We then extract the first error invariant from the generated abstract slice, which is `rxb==0`, negate it, and recursively proceed with the analysis of the preceding subtrace defined by the previous breakpoint. Finally, the abstract slice for the full trace is obtained by concatenating the computed abstract slices of all the subtraces. By using this new algorithm, static fault abstraction is only ever applied to small segments of the full error trace. We describe this algorithm in full detail in Section IV.

Concolic Pointer Elimination. To avoid the use of complex memory models in the encoding of trace formulas, we take advantage of the fact that we only deal with a single finite trace. That is, only finitely many memory locations are accessed during the execution of the trace. Hence, we can explicitly represent each of these memory locations by a distinct local program variable. Effectively, we reduce the error trace to a behaviorally equivalent trace that no longer dereferences any pointer variables. The encoding of the resulting trace only requires simple logical theories that are supported by standard interpolating SMT solvers.

The encoding that we outlined above requires us to identify all aliases between the dereferenced pointers in the trace. For this purpose, we instrument the error trace to record the actual address that is stored in each pointer whenever the pointer is dereferenced during execution. Each pointer dereference is then replaced by a distinguished variable associated with the address that is stored in the pointer at the respective position in the trace. We call this pointer elimination address-insensitive because the association between pointers and the memory addresses that they reference is lost in the encoding. Using this form of pointer

elimination, our algorithm produces the abstract slice shown in Fig. 2.

Sometimes it is important to keep the association between pointers and the referenced locations intact to diagnose the faulty program behavior. We therefore propose a more precise address-sensitive pointer elimination. In this alternative encoding, every access to the place-holder variable for the location of a dereferenced pointer is conditioned by a check that the pointer indeed stores the right address. This way, the theorem prover will need to explain why the pointer was holding the specific address if the corresponding location was involved in the error. Using the address-sensitive encoding, our algorithm produces the abstract slice shown in Fig. 3.

Other operations that are not supported by the SMT solver are eliminated in a similar manner. We discuss the details of the elimination of pointers and problematic operations in Sec. V.

III. Preliminaries

Before we explain our concolic fault abstraction algorithm in detail, we provide a brief summary of the formal foundations of error invariants and abstract slices.

An *error trace* is a sequence of simple program statements (such as assignments) together with an input determining the initial state, and an assertion that is violated at the end of the trace. We assume that invoking the original program on the given input executes the sequence of statements in the error trace. Often the assertion violation may be implicit in the program, e.g., the program may crash due to a run-time error. In such cases, the violated assertion can be generated automatically by inspecting the failing program statement and the program state immediately before the execution of that statement. We encode the initial state directly in the error trace by adding initial assignments of input values to program variables. Similarly, we fold the violated assertion F into the error trace by adding the statement $\texttt{assert}(F)$ at the end of the trace. Thus, we represent an error trace as a finite sequence of statements $\sigma = st_1; \ldots; st_n$ that has no normally terminating execution.

Throughout this paper, we use quantifier-free first-order logic formulas to describe error traces. We assume standard syntax and semantics of such formulas and use $\top$ and $\bot$ to denote the Boolean constants for *true* and *false*, respectively. Let X be a set of program variables. A *state* is a valuation of the variables from X. A *state formula* F is a first-order constraint over free variables from X. A state formula F represents the set of all states s that satisfy F and we write $s \models F$ to denote that a state s satisfies F.

For a variable $x \in X$ and $i \in \mathbb{N}$, we denote by $x^{\langle i \rangle}$ the variable which models the value of x in a state that is shifted i time steps into the future. We extend this shift function from variables to sets of variables, as expected, and we denote by X' the set of variables $X^{\langle 1 \rangle}$. For a formula F with free variables from Y, we write $F^{\langle i \rangle}$ for the formula obtained by replacing each occurrence of a variable $y \in Y$ in F with the variable $y^{\langle i \rangle}$.

A *transition formula* T is a first-order logic formula over $X \cup X'$ that models the semantics of a single program statement, respectively, a summary of multiple program statements. A transition formula T represents a binary relation on states and we write $s, s' \models T$ to denote that the pair of states (s, s') is in the relation represented by T. For example, the semantics of an assert statement $\texttt{assert}(F)$ is given by the transition formula $F \wedge X = X'$. Other examples of how statements in the programming language C can be encoded into transition formulas are given in Sec. V. In the following, we write T_{st} for the transition formula that encodes the semantics of a statement st.

Given an error trace $\sigma = st_1; \ldots; st_n$, the *error trace formula* of σ, denoted $\tau(\sigma)$, is the conjunction $T_1, \wedge \ldots \wedge T_n$ where $T_i = T_{st_i}^{\langle i-1 \rangle}$. Since σ has no normally terminating executions, the formula $\tau(\sigma)$ is unsatisfiable, i.e., there is no assignment to the variables in $\tau(\sigma)$ such that the formula evaluates to true.

Error Invariants. From the proof of unsatisfiability of a given error trace formula, $\tau(\sigma) = T_1, \wedge \ldots \wedge T_n$, we can use Craig interpolation to infer a sequence of *error invariants* [10], $I_0, \ldots, I_n$. Each error invariant I_k is a state formula that satisfies the following two properties:

1) $T_1 \wedge \ldots \wedge T_k \implies I_k^{\langle k \rangle}$, and
2) $I_k^{\langle k \rangle} \wedge T_{k+1} \wedge \ldots \wedge T_n \implies \bot$

The first property states that the error invariant I_k provides an over-approximation of the set of all states that are reachable by executing the prefix of the trace up to position k. The second property states that any execution of the suffix trace starting from position k with a state that satisfies I_k will still fail.

Abstract Slices. We call an error invariant I *inductive* for positions $k < j$ in a trace $\sigma = st_1; \ldots; st_n$ if I is an error invariant for both k and j. If we find such an inductive error invariant, we can replace the transition formulas $T_k \wedge \cdots \wedge T_{j-1}$ in $\tau(\sigma)$ by I and the resulting formula remains unsatisfiable. That is, in the original error trace, the statements $st_k; \ldots; st_{j-1}$ do not make progress towards the error and can be sliced from the trace. More generally, we can replace the statements $st_k; \ldots; st_{j-1}$ by an *abstract statement* that nondeterministically updates the program state such that I_k remains true. We refer to an error trace where statements are replaced by such abstract statements as an *abstract slice* of the error trace. The abstract trace fails with the same error, and the same error cause as the original error trace, but unlike the error trace, the abstract slice only executes statements for which no inductive error invariant can be found.

Our fault abstraction approach relies on techniques that have been traditionally applied to loop invariant synthesis in program verification. However, it is important to realize that finding inductive error invariants is much simpler than finding inductive loop invariants. Error invariants only need to generalize over a finite trace segment in a given trace while loop invariants need to generalize over potentially infinitely many traces of the loop.

IV. Concolic Fault Abstraction

The high-level overview of our concolic fault abstraction approach is given in Algorithm 1. The algorithm takes two input parameters: a program *Prog* and an input *In* for *Prog*

that leads to a certain error. The algorithm returns an abstract slice, $abstractSlice$, that explains the observed error.

Algorithm 1: Concolic fault abstraction.

Input: $Prog$: program with known bug
In : input that triggers the known bug
Output: $abstractSlice$: abstract slice
begin
  $abstractSlice \leftarrow \epsilon$
  $\sigma \leftarrow \texttt{CreateTrace}(Prog, In)$
  $b_1, \ldots, b_m \leftarrow \texttt{CreateBreakpoints}(\sigma)$
  $lastInvariant \leftarrow \bot$
  for i ***from*** $m-1$ ***to*** 1 **do**
    $st_{init} \leftarrow \texttt{GetConcreteState}(\sigma, b_i)$
    $st_{post} \leftarrow \texttt{assert}(\neg lastInvariant)$
    $st'_1, \ldots, st'_k \leftarrow$
    $\texttt{ExtractSubtrace}(\sigma, b_i, b_{i+1})$
    $\sigma' \leftarrow st_{init}; st'_1; \ldots; st'_k; st_{post}$
    $I[0] \ldots I[k+1] \leftarrow \texttt{Interpolate}(\tau(\sigma'))$
    $lastInvariant \leftarrow I[1]$
    $abstractSlice \leftarrow$
    $\texttt{Abstract}(\sigma', I) \,@\, abstractSlice$
  end for
end

The algorithm starts by initializing its return value $abstractSlice$ to the empty trace. Then it calls the procedure **CreateTrace** to extract an executable error trace σ for the given input In. We describe this procedure in more detail in the next section. On the executable trace σ, we set a sequence of breakpoints $b_1, \ldots, b_m$ for the debugger using the procedure **CreateBreakpoints**. Here, we assume that the first breakpoint b_1 is at the beginning of the trace and the last breakpoint b_m at the end of the trace.

Setting breakpoints at the locations where control enters or leaves functions in the original code turned out to be practical in our experiments. In general, however, **CreateBreakpoints** could set breakpoints at arbitrary program locations, as long as the code between two breakpoints is small enough to be processable by the theorem prover.

Once we have generated the trace and set the breakpoints, the actual fault abstraction loop starts. The loop iterates backwards through the generated breakpoints starting with the second last breakpoint (because the last breakpoint is where the actual fault happens). Our algorithm executes the full trace σ in a debugger, stops execution at the selected breakpoint and collects the values of all variables and memory locations necessary using **GetConcreteState** (see Sec. V). From these values, we create a sequence of assignment statement st_{init} that assigns all program variables to their corresponding values.

Further, we create a statement, st_{post}, that asserts the negation of the last known error invariant $lastInvariant$. In the first iteration, $lastInvariant$ is false because there is no execution that goes past the `assert` statement at the end of the trace.

Now, we use the procedure **SubTrace(trace,** b_i, b_{i+1}**)** to extract the sequence of statements, $st'_0, \ldots st'_k$, between the current, and the last breakpoint. That is, in the first iteration, we extract the statements from the second last breakpoint until the end of the trace. Together with the initialization statement st_{init} and the assertion st_{post}, we obtain an error trace σ', that has no uninitialized variables (every variable is assigned by st_{init}), and whose execution always terminates abnormally (because the last statement st_{post} is always asserting a negated error invariant).

From this trace, we then create the error trace formula $\tau(\sigma')$. This formula must be unsatisfiable (because σ' has no normally terminating executions). Thus, we can compute a sequence of error invariants $I[0] \ldots I[k+1]$ for the statements in σ' using the procedure **Interpolate**. This procedure calls the underlying interpolating SMT solver. It is important to note that $I[1]$ is the interpolant *after* st_{init}. The interpolant $I[0]$ before st_{init} is always $\top$. We use $I[1]$ to compute the postcondition for the error trace fragment that is analyzed in the next iteration by assigning $I[1]$ to $lastInvariant$. That is, instead of requiring the subtrace in the next iteration to end in the state constructed by st_{init}, we use the weaker condition $I[1]$.

Note that due to the concrete nature of the trace we analyze, this slicing technique does not lose any information, and generates interpolation problems that are mathematically equivalent to the interpolants for the full trace. In particular, since every variable in the trace takes on a single concrete value after st'_k, the conjunction of the assertions representing the statements $st'_0, \ldots st'_k$ is equivalent to the assertion that the variables are equal to their concrete values at the breakpoint. This means that replacing $st'_0, \ldots st'_k$ with a statement assigning all variables to their concrete values after st'_k does not affect the mathematical structure of the calculation for $I[k+1]$.

With σ' and the sequence of error invariants I, we can now apply our error-invariant-based fault abstraction from [10], denoted by the procedure **Abstract**. The procedure **Abstract** checks for each interpolant $I[k]$ whether it is an inductive error invariant on σ'. More precisely, **Abstract** tries to identify the smallest subset of error invariants in I that cover the largest possible number of statements in σ'. Then **Abstract** returns the corresponding abstract slice for σ' where all trace fragments for which an inductive error invariant has been found are replaced by this invariant. This abstract slice is then prepended to $abstractSlice$. That is, $abstractSlice$ holds the abstract slice for the suffix of the full trace σ that has been analyzed in previous iterations.

Once all fragments of the error trace have been processed, our algorithm terminates and returns the accumulated abstract slice.

V. Error Trace Construction

Our technique produces an executable trace which allows for easy reproducibility of the bug. We start by instrumenting our program using a custom cilly [20] pass. When the instrumented program is run on an input, it generates a "C" program which precisely represents the instructions executed by the original program on that input. This new program has the property that is it branch free (all branches have been resolved), loop free (all loops have been unrolled the correct number of times), and effectively function-call free (all local functions have been inlined). Since system calls cannot be inlined, they are

represented in the trace as by the appropriate function call. All that is needed to reproduce the bug is to add the appropriate headers, compile the trace using any "C" compiler, and then execute the program with the bug causing input.

As discussed in Section II, pointers are challenging for interpolating SMT solvers. We therefore take an aggressive approach to constructing executable error traces. Unlike existing work (e.g., [1]), our encoding translates every memory cell that is used on the trace into a separate variable to allow the theorem prover in our static fault abstraction to reason more efficiently about the trace. Note that this is only possible because we consider a single trace and a fixed input to the program.

As outlined in Section II, we propose two different approaches for constructing executable error traces: an *address-sensitive* encoding and an *address-insensitive* encoding. In both cases, we encode pointer accesses to the memory using local variables. For each used memory cell, we generate one unique local variable of appropriate type. For the address-sensitive encoding, we encode updates to pointers by assigning the pointer variable to an integer value representing the respective memory address. Each memory access is guarded by the assumption that the pointer carries the correct address (which is always the case, because we only consider one trace with a fixed input). With an address-sensitive encoding, our fault abstraction is able to generate abstract slices that contain information about how pointer values change over time, as shown in Fig. 3.

For the address-insensitive encoding, we also generate one variable per memory location, but we do not explicitly track the connection between pointer values and memory locations. This way, our error trace only contains assignments to actual memory locations. The information about the pointer variables that are used to access this memory is lost. This encoding allows our fault abstraction to construct much denser abstract slices as shown in Fig. 2, but may lose useful information such as why a dangling pointer in an error trace actually pointed to the specific location. This information is useful if the error was caused by assigning a wrong pointer variable rather than, say, a wrong or missing assignment to the location referenced by the pointer.

A. Address-Sensitive Pointer Elimination

For the address-sensitive encoding, we create one variable per memory location that can be accessed during the execution of the heap. As we only consider one fixed input, the number of variables that need to be generated is always finite. For each pointer variable, we log the memory addresses it carries during execution of the trace. For each memory location (e.g., `0x123`) collected during this step, we create a local variable (e.g., `mem_0x123`). Now, we iterate over the executable trace and replace all pointer dereferences by the corresponding generated variable (i.e., `mem_0x123`), guarded by the condition that the pointer points to this memory cell. For example, the assignment statement from our sed example:

```
vector->v_length = 0;
```

is translated into the conditional assignment:

```
if(vector+8==0x123) {mem_0x123 = 0;}
```

where the magic constant "8" represents

```
offsetof(typeof(vector),v_length)
```

Adding the conditional `(vector+8==0x123)` may seem odd at first sight because we know it must always be `true` on the trace for the given input. However, it serves an important purpose during the static fault abstraction: the conditional turns into an implication when translating into first-order logic ($vector + 8 = 0x123 \implies mem\_0x123 = 0$). This forces the theorem prover to include the variable $vector$ into the proof when showing that the formula is unsatisfiable. Therefore, we obtain a different set of Craig interpolants and our fault abstraction preserves all statements that are necessary to understand why `vector->v_length` points to the memory location `0x123`. This results in a detailed abstract slice as shown in Fig. 3. This trick of encoding conditional dependencies between variables using implications is inspired by [4] where the same idea is used to make static fault localization aware of branching conditions.

B. Address-Insensitive Pointer Elimination

For the address-insensitive encoding we take a similar approach: we create one local variable per used memory location and replace pointer dereferences by the corresponding generated variable. The difference is that this time, we do not add the condition that the pointer points to the right memory location. That is, the assignment:

```
vector->v_length = 0;
```

is now translated into the simple assignment:

```
mem_0x123 = 0;
```

This way, when translating the program into first-order logic, the theorem prover will not reason about the address that `vector->v_length` is pointing to. Hence, our fault abstraction will only contain the statements that actually change the value of this memory location. This results in a more compact abstract slice as shown in Fig. 2.

C. Additional Benefits of Concolic Reasoning

One of the advantages of concolic reasoning is that it provides a simple way of dealing with library calls. Formal verification typically requires full specifications for library calls, such as the stubs provided by tools such as Frama-C [7]. In concolic analysis, on the other hand, we only need to capture the effect of such calls for one concrete input. This is much easier to automate and also leads to simpler formulas (e.g., quantification is not necessary).

Calls with side-effects such as `malloc` and `realloc` can be translated into proper assignments to the generated local variables. Other calls, for example to the math library, can be eliminated by recording their return values.

VI. Real-World Examples

We illustrate the effectiveness of our approach along two examples of bugs in open source programs: sed and gzip. Both bugs were drawn from the SIR benchmark [8] suite. The actual source code and the test input that reveals the bugs are taken from the BugRedux [14], [16] distribution.

		sed	gzip
0	LOC Full Program	11990	7328
1	Execution Time (s)	0.052	0.042
2	Execution Time instrumented (s)	0.305	0.172
3	Slowdown	$5.8\times$	$4.1\times$
4	Log size	36kB	18kB
5	LOC Trace Executable code	985	232
6	LOC Address-Sensitive Slice	11	51
7	LOC Address-Insensitive Slice	4	49
8	Bug-Assist reported error locations[2]	0 valid, 2 spurious	2 valid, 1 spurious
9	F^3 ranking of error location	3/19	3/80

TABLE I. EXPERIMENTAL EVALUATION OF CONCOLIC FAULT ABSTRACTION TO BUGGY VERSIONS OF SED AND GZIP. ROWS 0 – 4 DESCRIBE CONCOLIC TRACE GENERATION; ROWS 5 – 7 DESCRIBE THE EFFECT OF FAULT ABSTRACTION, AND ROWS 8 – 9 COMPARE TO FAULT LOCALIZATION TOOLS BUG-ASSIST AND F^3.

sed is a UNIX utility which parses and transforms text, based on a compact scripting language. We analyzed a bug in "GNU sed version 2.05", an open source version of sed with 11990 lines of code. As described in Section II, the bug is that the global variable `last_regex` is never initialized, leading to a segmentation fault when the pointer is finally dereferenced.

gzip is a UNIX utility which compresses and decompresses files. We analyzed a bug in version 1.2.4, which had 7328 lines of code. The root cause of the bug is that the function `huft_build` returns an incorrect error code after a failed allocation. The program then takes the wrong branch in the error-handling path, and dereferences a null pointer.

A. Implementation Details

Our procedure to generate fault-localized error traces requires a set of steps. We evaluated the practicality of each step on both of our examples.

Trace generation. The first step of our procedure is to generate an executable trace leading to the error. We used the default bug-triggering inputs provided by the benchmark suite — `file000001` and `pattern000001` in the case of the sed bug, and `file000001` in the case of the gzip bug.

As discussed in Section V, we used a custom cilly pass which instrumented the buggy program to generate an executable "C" program which represents the set of instructions that led to the bug. The runtime overhead of this step, compared to executing the unmodified version of the program directly in the shell, was approximately $5\times$, as shown in Table I. We expect that this slowdown should be independent of the length of both the program, and of the error trace. Logfile lengths were also quite manageable, measuring in the tens of kilobytes.

The executable trace for sed had 976 executable statements (plus variable decelerations, typedefs, etc). The final executable trace for gzip consisted of 415 lines, of which 232 were executable statements, and the rest were boilerplate variable decelerations, typedefs, etc.

Formula Generation. As we discuss in Section V, we generated two different traces that served as basis for the formula generation using an address-insensitive, and an address-sensitive encoding.

[2]Applied to error traces as Bug-Assist could not process the full programs.

sed makes active use of pointers and the heap, and hence there was a visible difference between the generated abstract slices for the two encodings: the address-sensitive slice contains 11 statements, while the address insensitive slice contains 4 (the abstract slices themselves can be seen in Figures 2 and 3). In both cases invariant generation completed almost instantaneously on a 2GHz Intel Core i7 with 8GB of RAM.

We similarly encoded our gzip error trace into an SMT formula. Unlike sed, which made significant use of heap allocated storage and pointer arithmetic, there were only four statements in the error trace that depended on the address of heap objects, and hence there was effectively no difference between the address-sensitive and the address-insensitive encodings of the trace. For implementation reasons, our encoding did not precisely follow that of Section V: for example, some small arrays have been modeled using uninterpreted functions, and pointers to the arrays were substituted with plain integer variables as indices.

We further experimented on our new divide-and-conquer algorithm with this particular gzip trace and confirmed the potential and promising applicability of our error-invariant technique on long traces. We set one breakpoint in the middle of the trace, splitting the entire trace into two parts and observed that the overall time to compute the abstract slice is considerably reduced compared to the time to compute from the single entire trace. The Nov. 2010 version of smtinterpol used in [10] took 12.2 and 9.2 seconds to process the two subtraces respectively, but failed with an out-of-memory exception after 141 seconds against the entire trace. A similar result was obtained with the latest version of smtinterpol 2.1: it took 2.49 and 0.94 seconds for the two subtraces respectively, totalling 3.43 seconds, vs 4.53 seconds to process the entire trace.

Address-Insensitive Encoding. Analyzing the quality of a debugging tool is an inherently subjective task. Quantitatively, we note that for sed, our technique reduced a 976 statement error trace to 4 statements address-insensitive statements; for gzip, it reduced 232 statements to 49.

In both cases, a qualitative analysis showed that the new trace effectively highlighted the path that led to the bug. In the case of sed, it is harder to imagine a simpler trace that would show the bug. Similarly, in the case of gzip, the generated trace was effective in concisely explaining why the null pointer was invalid. We were unable to remove any additional lines from the trace that our algorithm automatically produced.

We also noted that the invariants generated by the Craig interpolation procedure were logically identical to the invariants that we developed by hand, using human analysis.

Address-Sensitive Encoding. Quantitatively, the reductions to the address-sensitive memory trace was highly effective, reducing 985 to 11 lines in the case of sed, and 232 to 51 lines in the case of gzip. As before, the discovered invariants were equivalent in quality to those we would have hand-generated, and we were not able to remove any additional lines that were not removed automatically by our tool.

Qualitatively, our experience with attempting to understand the cause of this bug was that the shorter simplified trace was much easier to reason about than the full trace.

B. Comparison with other tools

Static fault localization. We attempted to run the static analyzer Bug-Assist [19] on our two case studies. Bug-Assist takes a different approach to bug localization: whereas we generate an abstract slice which explains the *path* to the bug, it generates a list of possible bug *locations*.

Bug-Assist was unable to parse the full versions of either program. This is not particularly surprising, as the Bug-Assist designers warn that it is a research prototype, and is not necessarily expected to scale to complex programs.

We were able to run Bug-Assist on the executable error traces. Note that these represent a best-case scenario for static fault localization, because we remove all conditional branches and loops, which would otherwise require effort to analyze. We further had to supply the `--slice` option to remove statements unrelated to the violated assertions in order for the tool to scale to our traces. In the gzip case, it reported three candidate bug locations, two of which were related to the assertion violation, although neither of them pinpointed the precise bug location. In the case of sed, Bug-Assist reported two candidate bug locations, neither of which was related to the actual bug.

Dynamic fault localization. As we discuss in the related work section, there has been significant work on dynamic fault localization tools, such as Igor[3]. These tools require a set of both passing and failing executions. In our case, we only have a single failing execution, and hence cannot use these tools.

Hybrid fault diagnostics. F^3 uses a concolic approach to solve this problem. Given a single failing trace, it uses a concolic test generator (BugRedux) to generate a set of both failing and passing tests, which it then uses to do dynamic fault localization. In some ways, this approach is the mirror image of ours: whereas we do a static analysis on a single dynamically generated error trace, they do static analysis to generate many error traces, which they then dynamically analyze.

They also take a different approach to bug diagnosis: F^3 generates a ranked list of possible bug *locations*. The effectiveness of F^3 on our test cases is reported in [16]. In the case of the sed fault, F^3 ranked the actual bug location 13 out of 19 candidate locations; for gzip, it ranked the bug location 3/80.

VII. Related Work

Concolic testing. Our work was inspired by the success of concolic testing tools such as DART [11], CUTE [28], KLEE [3], Java PathFinder [31], and Yogi [21]. These tools combine symbolic analysis (which achieves high coverage) with concrete execution (which allows the static analysis to scale). Traditional concolic testing is orthogonal to our work: it focuses on how to *detect the existence of* errors, whereas our concolic fault location is designed to *explain the cause* of an error.

Dynamic Fault Localization. Research on automated fault localization has mostly focused on dynamic techniques (e.g., [9], [23], [26], [35], [37]). These techniques repeatedly execute the faulty program fragment, comparing failing and passing runs that are sufficiently similar to identify the defect. Individual dynamic fault localization approaches differ in the way the failing and passing executions are obtained, the way they are compared, and in the information they report to the user. A detailed survey about the differences between these approaches can be found in [34]. All of these approaches have in common that they depend on the availability of high-quality test suites to compare passing and failing executions. The main difference to our approach is that we do not need any passing program executions and hence do not depend on high-quality test suites.

Static fault diagnostics. The best explored static approach is program slicing [33]. For a survey of static slicing techniques see [30]. While slicing uses a form of abstraction, it is defined purely in terms of the program syntax because it only tracks data flow and control flow dependencies.

Bug-Assist [18], [19] reduces the problem of error localization to the maximal satisfiability problem (MAX-SAT). It first does bounded model checking to encode the semantics of a bounded unrolling of a failing program into a Boolean trace formula. By repeatedly calling a MAX-SAT solver, it finds a maximal set of statements that does not violate the desired property so as to report the complement set as potential bug locations.

Hybrid fault diagnostics. In addition to purely dynamic and static techniques there also exist hybrid techniques that combine the two approaches. Notable is the technique described in [12], which improves dynamic fault localization by analyzing the unsatisfiable cores of error path formulas. F^3 [16] uses the concolic tool BugRedux [15] to generate the passing test cases that are needed for dynamic fault localization.

VIII. Conclusions

We have presented a novel approach to automated fault diagnostics and demonstrated that it delivers useful results on real-world error traces. At the core of this technique is a static analysis that uses an automated theorem prover to compute an abstract slice of the faulty program, which explains the faulty behavior. In this paper, we have shown that this fault abstraction technique can be scaled to real error traces by combining it with a dynamic analysis. In particular, we use dynamic analysis to split the faulty program trace into smaller subtraces that are amenable to static analysis. Moreover, the use of dynamic analysis allows us to significantly simplify the logical encoding of error traces. With the address-sensitive and address-insensitive encodings presented in this papers, we are able to avoid expensive memory models and the use of complex reasoning theories.

We believe that our two encodings of the memory can provide valuable information for debugging. For example, the address-insensitive abstract slice could be presented directly to the programmer, while the address-sensitive slice could be used to generate breakpoints in the program that partially automate a more detailed analysis of the error trace with the help of a debugger.

We conclude that concolic fault abstraction is a very promising approach. Exploiting information about concrete

[3] http://www.st.cs.uni-saarland.de/askigor/downloads/

memory states eliminates the main bottlenecks and complexities of purely static approaches.

REFERENCES

[1] M. Aung, S. Horwitz, R. Joiner, and T. Reps. Specialization slicing. Technical Report UW-CS-TR-1711, University of Wisconsin - Madison Computer Sciences Department, March 2012.

[2] T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: localizing errors in counterexample traces. *SIGPLAN Not.*, pages 97–105, 2003.

[3] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224. USENIX Association, 2008.

[4] J. Christ, E. Ermis, M. Schäf, and T. Wies. Flow-sensitive fault localization. In *VMCAI*, volume 7737 of *LNCS*, pages 189–208. Springer, 2013.

[5] H. Cleve and A. Zeller. Locating causes of program failures. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 342–351. ACM, 2005.

[6] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *TPHOLs*, pages 23–42, 2009.

[7] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C - A Software Analysis Perspective. In *SEFM*, pages 233–247, 2012.

[8] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Softw. Engg.*, 10(4):405–435, Oct. 2005.

[9] M. Dwyer, R. Purandare, and S. Person. Runtime verification in context: Can optimizing error detection improve fault diagnosis? In *Runtime Verification*, volume 6418 of *LNCS*, pages 36–50. Springer, 2010.

[10] E. Ermis, M. Schäf, and T. Wies. Error invariants. In *FM*, volume 7436 of *LNCS*, pages 338–353. Springer, 2012.

[11] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *PLDI*, pages 213–223. ACM, 2005.

[12] D. Gopinath, R. N. Zaeem, and S. Khurshid. Improving the effectiveness of spectra-based fault localization using specifications. In *ASE*, pages 40–49, 2012.

[13] B. Hailpern and P. Santhanam. Software debugging, testing, and verification. *IBM Systems Journal*, 41(1):4–12, 2002.

[14] W. Jin. Reproducing and debugging field failures in house. In *ICSE*, pages 1441–1443, Piscataway, NJ, USA, 2013. IEEE Press.

[15] W. Jin and A. Orso. Bugredux: Reproducing field failures for in-house debugging. In *ICSE*, pages 474–484. IEEE, 2012.

[16] W. Jin and A. Orso. F3: Fault localization for field failures. In *ISSTA*, pages 213–223. ACM, 2013.

[17] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *ASE*, pages 273–282. ACM, 2005.

[18] M. Jose and R. Majumdar. Bug-Assist: Assisting fault localization in ANSI-C programs. In *CAV*, volume 6806 of *LNCS*, pages 504–509. Springer, 2011.

[19] M. Jose and R. Majumdar. Cause clue clauses: error localization using maximum satisfiability. In *PLDI '11*, pages 437–446. ACM, 2011.

[20] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 213–228, London, UK, UK, 2002. Springer-Verlag.

[21] A. V. Nori, S. K. Rajamani, S. Tetali, and A. V. Thakur. The yogi project: Software property checking via static analysis and testing. In *TACAS*, LNCS, pages 178–181. Springer, 2009.

[22] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *ISSTA*, pages 199–209. ACM, 2011.

[23] D. Qi, A. Roychoudhury, Z. Liang, and K. Vaswani. Darwin: an approach for debugging evolving programs. In *ESEC/SIGSOFT FSE*, pages 33–42, 2009.

[24] Z. Rakamaric and A. J. Hu. A scalable memory model for low-level code. In *VMCAI*, volume 5403 of *LNCS*, pages 290–304. Springer, 2009.

[25] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *ASE*, pages 30–39. IEEE Computer Society, 2003.

[26] S. Roychowdhury and S. Khurshid. A novel framework for locating software faults using latent divergences. In *ECML PKDD*, pages 49–64. Springer, 2011.

[27] M. Schäf, D. Schwartz-Narbonne, and T. Wies. Explaining inconsistent code. In *ESEC/SIGSOFT FSE*, pages 521–531, New York, NY, USA, 2013. ACM.

[28] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In *ESEC/SIGSOFT FSE*, pages 263–272. ACM, 2005.

[29] F. Shull, V. R. Basili, B. W. Boehm, A. W. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero, and M. V. Zelkowitz. What we have learned about fighting defects. In *IEEE METRICS*, pages 249–. IEEE Computer Society, 2002.

[30] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[31] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *ASE*, pages 3–11, 2000.

[32] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. In *ISSTA*, pages 61–72, New York, NY, USA, 2010. ACM.

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

[34] W. E. Wong and V. Debroy. Software fault localization, 2009.

[35] A. Zeller. Isolating cause-effect chains from computer programs. In *SIGSOFT FSE*, pages 1–10, New York, NY, USA, 2002. ACM.

[36] A. Zeller. Comment in an open discussion at the 2nd International Workshop on the Future of Debugging, July 2013.

[37] X. Zhang, N. Gupta, and R. Gupta. Locating faults through automated predicate switching. In *ICSE*, pages 272–281. ACM, 2006.