Static fault localization for simple bugs

ANONYMOUS AUTHOR(S)

We consider the problem of fault localization. In particular, we identify relevant statements in an error trace. An error trace is the output of a typical software verification tool that reports a bug. In practice, an error trace can contain hundreds of statements and can be tedious and time-consuming to analyze. Identifying relevant statements can help the user to understand the root cause of the bug more easily.

Intuitively, a statement is relevant in a trace if a modification of the statement could be involved in a bug fix. However, bugs can be very intricate and in general a bug fix could involve every statement. Therefore, we restrict ourselves to simple bugs. We consider a bug simple if a modification of a single statement can fix the bug. We formalize this intuition in a new definition of relevance.

Furthermore, we generalize our approach from traces to programs by applying it to a set of traces. To make the generalization applicable, we extended an existing software verification algorithm such that it can output several error traces.

We have implemented our approach and evaluate it on programs from the competition on software verification.

ACM Reference format:
Anonymous Author(s). 2017. Static fault localization for simple bugs. 1, 1, Article 1 (July 2017), 26 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Developing good software is still a hard job in 2017. Just as in the past, a developer's basic workflow consists of writing code, evaluating the code (usually employing static and dynamic analysis), and debugging. All these steps have been greatly improved by (semi- or fully) automatic tools and algorithms developed by the research community. And yet, debugging software still takes a long time in this process, a major amount of which is usually dedicated to actually localizing the *fault*, i.e., the cause of the error.

It does not come as a surprise that fault localization has invoked a lot of interests with about 350 publications from many different disciplines of computer science [41]. Each of them comes with their own strengths and application domains.

Our motivation for considering fault localization stems from the urge to improve the feedback of static software verifiers. If such a tool finds a violation of the program specification, it usually outputs an error trace (i.e., a sequence of statements that, when executed, leads to the error). While an error trace is often sufficiently helpful to find the bug, error traces can easily become prohibitively long to be analyzed by a human developer. This can happen

XXXX-XXXX/2017/7-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

for instance if the program is just so large or if the program has a *deep* bug, i.e., a bug that only manifests itself after lots of loop unwindings.

Given a long error trace, usually not all statements are actually related to the bug. One option to support the developer in her task of identifying the fault is to additionally highlight those statements that might be "more related" to the bug than others. Most fault localization techniques can be phrased in this way. In this paper we call such statements *relevant*, and the task that we try to solve is the detection of such statements.

Before we can address this task, we first have to specify our notion of when a statement is relevant. We identified the following three properties of relevance that we think would be useful for a developer who uses a software verifier for debugging purposes.

(1) The relevant statements should be *statically computable from a single error trace* without the need for additional context like, e.g., sampling of executions along that trace.
(2) To understand *why* a statement was declared relevant can help in understanding how a bug could be fixed. Hence it is desirable to have a notion of relevance that is *formally defined in a declarative way* and not given by the outcome of a specific algorithm.
(3) Ideally, the relevance information should be useful not only to find out *where* to fix the bug but also *how*. Accordingly, relevance of a statement should hint at the possibility that a *modification of the statement can fix the bug*.

Many existing fault localization approaches require other (error and/or correct) traces in order to compute their result. Such approaches do not satisfy Property (1). Other approaches compute a program abstraction, usually by omitting error-unrelated parts ("slicing"), and then report all statements occurring in the reduced program to the developer. Such approaches do not satisfy Property (2) and Property (3) because the information for the developer is only which statements she should *not* look at.

This motivated us to come up with a new fault localization technique that satisfies all three properties. Given a single error trace, we call a statement relevant if it is an assignment statement and, when replacing it with a specific constant assignment, the error trace becomes infeasible (i.e., not executable).[1] Observe that this notion of relevance satisfies the given properties; in particular, it does not depend on a concrete implementation. We present this idea in more details in Section 3.

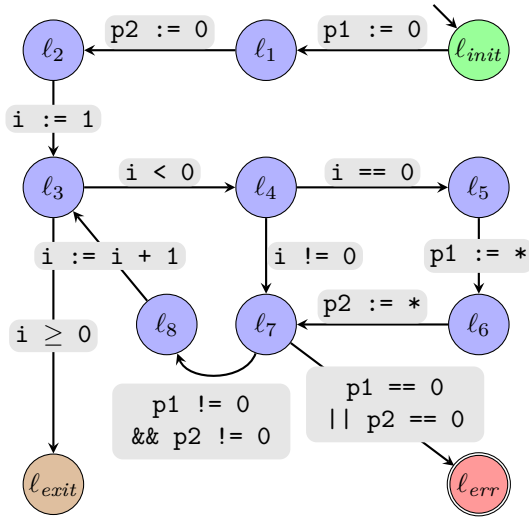
We explain the definition on the program that is given in Figure 1(a) (see also the corresponding control flow graph in Figure 1(b)). The program initializes two pointers to null and enters a loop. The first loop iteration is supposed to allocate memory and assign it to the pointers, but the programmer accidentally initialized the loop counter `i` to 1 instead of 0, and hence this step is skipped. The safety specification is given by the `assert` statement which expresses that the pointers should not point to null at the end of the loop body.

The (unique) error trace is shown in Figure 1(c). First observe that the trace corresponds to a single execution (i.e., there is no nondeterminism involved; every variable is initialized). If we apply the definition, then the assignment to `p1` is relevant only if we can find another assignment to `p1` that makes this trace infeasible. However, this is not the case here because `p2` would still be initialized to 0 and thus the error condition at the end would still be satisfied. The assignment to `p2` is also not relevant with an analogous argument. We can,

[1] At this point we assume that a trace only has one corresponding execution for simplicity. We will later explain the consequences for nondeterminism in the trace.

```
int *p1, *p2, i;
p1 := 0;
p2 := 0;
i := 1; // bug: should be 0
while (i < 10) {
   if (i == 0) {
      p1 := malloc(sizeof(int));
      p2 := malloc(sizeof(int));
   }
   assert p1 != 0 && p2 != 0;
   i := i + 1;
}
```

(a) Program code.



(b) Control flow graph where we replaced `malloc` by `*`.

`p1 := 0` `p2 := 0` `i := 1` `i < 10` `i != 0` `p1 == 0 || p2 == 0`

(c) Error trace where we simplified negated conditions.

Fig. 1. An erroneous program with a single error trace.

however, find an assignment to `i` that makes the trace infeasible. One choice would be the value 0, which corresponds to the actual fix of the bug. Another choice would be the value 10.

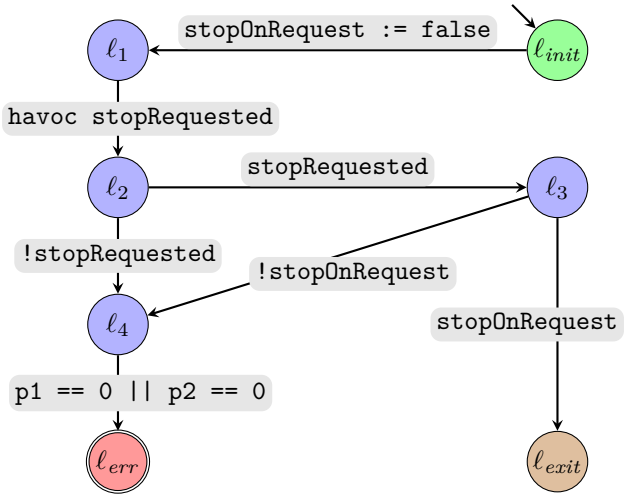
Other bug fixes that one can imagine are to initialize the two pointers outside the loop or replacing the `if` condition with `i == 1`. Thinking it out, we could also try to restructure the control flow of the program. However, we would then open Pandora's box and enter the field of program synthesis, which we strictly want to avoid. We rather follow the principle of Ockham's razor and settle for our *simple bug fix*, namely replacement by another assignment. If we cannot find such a simple bug fix, the only help we can give to the developer is that *no simple bug fix exists* (according to our definition), which itself might be a valuable information, e.g., when estimating debugging time or prioritizing jobs. In such cases one may also fall back to other fault localization techniques (if they are applicable).

An analysis based on a single error trace has inherent limitations. For instance, when the trace takes the `else` branch of a condition, we cannot include the statements of the other branch in our analysis. To address these limitations, we also lift our definition of relevant statements to whole programs in a straightforward way: We say that a statement is relevant in a program if it is relevant in each error trace.

We motivate this approach with the following example. Consider again the program from Figure 1(a). We extend it by the option to terminate the loop upon user request before the assertion is checked. To model user input, we use a `havoc` statement which takes a variable as argument and assigns a nondeterministic value from the variable domain. The request should only be respected if a certain flag is activated. In the example the flag is deactivated. The code snippet that should replace the old `assert` statement is shown in Figure 2(a).

```
bool stopOnRequest;
bool stopRequested;
stopOnRequest := false;
havoc stopRequested;
if (stopRequested) {
  if (stopOnRequest) {
    return;
  }
}
assert p1 != 0 && p2 != 0;
```

(a) Program code.



(b) Control flow graph with fresh locations.

```
stopOnRequest := false  havoc stopRequested
                        stopRequested  !stopOnRequest  p1 == 0 || p2 == 0

stopOnRequest := false  havoc stopRequested
                                       !stopRequested  p1 == 0 || p2 == 0
```

(c) Two error trace suffixes where we simplified negated conditions. Line breaks have no meaning.

Fig. 2. An extension of the program in Figure 1(a) with two error traces.

We still assume the old bug with the loop counter `i` being initialized to 1. Thus there are now two error traces, one that enters the new outer `if` block and one that skips it. The suffixes of the traces are given in Figure 2(c).

In the first error trace the assignment to `stopOnRequest` is relevant because assigning the value `true` makes the trace infeasible. However, in the second error trace the assignment to `stopOnRequest` has no effect. Hence we say that this assignment is not relevant in the program. This also exemplifies our notion of a simple bug fix: We cannot guarantee that a change to the variable will prevent the error. In fact, activating the flag will only fix the bug in case the user requests termination.

We can apply our definition also to the nondeterministic assignment to `stopRequested`. This time, for each error trace we can find an assignment that makes the trace infeasible (namely `false` in the first case and `true` in the second case. Thus we would call the assignment to `stopRequested` relevant. But in fact, there is no assignment to this variable that fixes the bug. This shows that by going from single traces to programs we lose our "simple bug fix" guarantees: If replacing a particular assignment by another one fixes every single error trace ($\forall\exists$), we cannot derive that there exists *one* assignment that fixes *all* error traces ($\exists\forall$).

In the example we have seen that we can also handle nondeterministic assignments. This was an important feature to have for us because software verifiers typically model

undetermined data such as memory pointers, user inputs, or calls to library functions through nondeterminism.

A program may contain infinitely many error traces. In this case we can only approximate the set of relevant statements using a finite subset of all error traces. However, the choice of error traces is rather important. Intuitively we prefer to analyze *different* error traces. To increase the variety of error traces, we propose an extension of a software verification approach. In a nutshell, after an error trace was found, we let the software verifier continue to search for error traces that are not similar to error traces that have been found so far. We describe our extension to programs in more details in Section 4.

To summarize the contributions of this paper:

- We describe a new notion of relevance for usage in formal software analysis tools.
- We give an algorithm that computes all relevant statements for a given feasible error trace that may contain nondeterministic assignments.
- We show how our definition can be lifted from single traces to programs if one has an algorithm that produces multiple feasible error traces.
- We present an extension of a software verification approach that allows us to produce multiple feasible error traces.
- We implemented the approach and evaluate its performance on programs from the competition on software verification (SV-COMP).

2 PRELIMINARIES

We follow the presentation in [20] and identify programs by their control flow graph over a set of variables V where the nodes correspond to program locations and each edge is labeled with a statement. The statements are taken from a finite set Σ called the *alphabet*. Formally, a program is a graph $P = (Loc, \delta, \ell_{init}, \ell_{err})$ where Loc is a finite set of nodes (*locations*), $\delta \subseteq Loc \times \Sigma \times Loc$ is the transition relation, ℓ_{init} is the initial node called the *initial location*, and ℓ_{err} is the accepting node called the *error location*. We sometimes consider another dead-end location called ℓ_{exit} to mark the regular exit of the program.

An example of a control flow graph is given in Figure 1(b). We employ a standard state-based view of programs: A (program) *state* is a valuation of the program variables. We express (sets of) program states by predicates over the program variables, and statements by formulas over primed and unprimed variables. Given a predicate φ over unprimed variables, $\varphi[V'/V]$ (or φ' for short) denotes the predicate where all variables are primed.

A *trace* π is a sequence of statements. Let $\pi = \langle \mathfrak{st}_1, \ldots, \mathfrak{st}_n \rangle$ be a trace and $\xi = s_0, \ldots, s_n$ be a sequence of states. We call ξ an *execution* of π if the formula $s_{i-1} \wedge \mathfrak{st}_i \rightarrow s'_i$ is satisfiable for all $1 \leq i \leq n$. If only a prefix of ξ, say, $s_0, \ldots, s_j$ for $j < n$ is an execution, and $s_j \wedge \mathfrak{st}_{j+1}$ is unsatisfiable, we call ξ a *blocking execution*. A trace is called *feasible* if there exists an execution of it, and *infeasible* otherwise.

An *error trace* is a trace π from ℓ_{init} to ℓ_{err}. Note that an error trace is not necessarily feasible. In the previous section we have omitted the word "feasible" for convenience, but from now on we say *feasible error trace*.

Given a trace $\pi = \langle \mathfrak{st}_1, \ldots, \mathfrak{st}_n \rangle$, a state s is *reachable at position i* if there exists an execution $s_0, \ldots, s_i$ of the prefix trace $\langle \mathfrak{st}_1, \ldots, \mathfrak{st}_i \rangle$ such that $s_i = s$. Similarly, a state s is *coreachable at position i* if there exists an execution $s_i, \ldots, s_n$ of the suffix trace $\langle \mathfrak{st}_{i+1}, \ldots, \mathfrak{st}_n \rangle$ such that $s_i = s$. A state is *bireachable at position i* if it is both reachable and coreachable at that position. A state is *reachable* (*coreachable*, *bireachable*) if it is reachable (coreachable, bireachable) at some position i. Note that every state is reachable at position 0

and coreachable at position n. For a trace π and predicates φ, ψ, we may restrict the executions of π to those starting in φ (the *precondition*) and ending in ψ (the *postcondition*), i.e., ξ is of the form $s_0, \ldots, s_n$ such that $s_0 \models \varphi$ and $s_n \models \psi$.

We sometimes switch between a symbolic and a set interpretation of states. For instance, when we write $s \in \varphi \subseteq \psi$ for some state s and predicates φ, ψ, we mean $s \models \varphi$ and $\varphi \models \psi$ if interpreted in the symbolic view.

To describe program statements in control flow graphs and traces we use guarded commands [11]: the deterministic assignment `x := e` for some expression `e`, the nondeterministic assignment `havoc`(x), and the assume statement `assume(`φ`)` for some predicate φ that blocks the execution if φ does not hold. We usually just write φ for `assume(`φ`)`. For example, the feasible error traces in Figure 2(c) consisted of the three types of guarded commands.

We recall the well-known predicate transformers **wp** (weakest precondition), **pre** (precondition), and **sp** (strongest postcondition). Let φ, ψ be predicates and $\mathfrak{st}$ be a statement.

$$\begin{aligned}
\mathbf{wp}(\psi, \mathfrak{st}) &\equiv \forall V'.\ \mathfrak{st} \implies \psi[V'/V] \\
\mathbf{pre}(\psi, \mathfrak{st}) &\equiv \neg\mathbf{wp}(\neg\psi, \mathfrak{st}) \equiv \exists V'.\ \mathfrak{st} \wedge \psi[V'/V] \\
\mathbf{sp}(\varphi, \mathfrak{st}) &\equiv \exists V''.\ \varphi[V''/V] \wedge \mathfrak{st}[V''/V][V/V']
\end{aligned}$$

The functions can be easily lifted to traces by applying them recursively, where the application to the empty trace $\langle\rangle$ is the identity, e.g., $\mathbf{wp}(\psi, \langle\rangle) := \psi$.

We can rephrase the characterization of (co-/bi-)reachable states as follows. Assume a trace $\pi = \langle \mathfrak{st}_1, \ldots, \mathfrak{st}_i, \ldots, \mathfrak{st}_n \rangle$, a precondition φ and a postcondition ψ. A state s is reachable at position n if $s \in \mathbf{sp}(\varphi, \pi)$. A state s is coreachable at position 0 if $s \in \mathbf{pre}(\psi, \pi)$. A state s is bireachable at position i if $s \in \mathbf{sp}(\varphi, \langle \mathfrak{st}_1, \ldots, \mathfrak{st}_{i-1} \rangle) \cap \mathbf{pre}(\psi, \langle \mathfrak{st}_i, \ldots, \mathfrak{st}_n \rangle)$.

We assume the reader is familiar with the notion of a *Hoare triple* which we write $\{\,\varphi\,\}\,\mathfrak{st}\,\{\,\psi\,\}$ for precondition φ, statement $\mathfrak{st}$, and postcondition ψ. A Hoare triple $\{\,\varphi\,\}\,\mathfrak{st}\,\{\,\psi\,\}$ is valid iff the precondition φ implies the weakest precondition $\mathbf{wp}(\psi, \mathfrak{st})$.

3 RELEVANT STATEMENTS IN A SINGLE TRACE

In this section we try to answer the following question. Given a feasible error trace, what are the statements that allow a simple bug fix?

3.1 Relevance

A simple bug fix tries to replace existing statements by other statements in order to fix the problem. However, we believe that conditional (i.e., `assume`) statements are ill-suited for replacement. Using sufficiently strong conditions one can usually fix any feasible error trace by just blocking the respective executions (in the extreme case consider the statement `assume(false)`). Accordingly, we focus on replacing (possibly nondeterministic) assignment statements by another deterministic assignment statement. Let us use the term *assigning statement* to uniformly talk about deterministic and nondeterministic assignment (i.e., `havoc`) statements. Moreover, we do not change the variable that is assigned, i.e., we only replace the right-hand side of a deterministic assignment and we only replace a statement of the form `havoc(x)` by `x := e` for some `e`. This constraint is not very strict in practice because the developer usually had the right intuition which variable to change. Last, we require that the right-hand side of the new assignment is a constant value.

To rephrase our approach, we try to identify those assigning statements in a feasible error trace that, when replaced by an assignment statement, can prevent reaching the error. We

call such statements *relevant*. Before we formally define relevance of statements, let us first give some explanatory descriptions.

For an assignment statement st of the form `x := e` the intuition is that it is relevant for the feasible error trace if the value that was assigned to the variable `x` plays a role for reaching the end of the feasible error trace. Or, ex negativo: Statement st is not relevant only if the error trace is feasible for *any* nondeterministic assignment to `x` at this point. Observe that this corresponds to replacing st by the statement `havoc(x)` without making the trace infeasible.

We also consider nondeterministic assignment statements st, e.g., of the form `havoc(x)`. Such statements are usually not part of a human-written program, but they typically arise from modeling user input or calls to library functions. Just as for a deterministic assignment, we say that a `havoc` statement is relevant if the (previously feasible) error trace becomes infeasible for *some* deterministic assignment at this point.

We now formally capture the definition of relevance.

Definition 3.1 (Relevance of assigning statements for single traces). Let $\pi = \langle \mathrm{st}_1, \ldots, \mathrm{st}_n \rangle$ be a feasible error trace of length n with st_i being an assigning statement that assigns a new value to some variable `x`.

The statement st_i is *relevant for* π if there is an execution $s_0, \ldots, s_n$ of π and some value v such that every execution of the trace $\langle x := v, \mathrm{st}_{i+1}, \ldots, \mathrm{st}_n \rangle$ starting in s_{i-1} is blocking.

We usually only say that a statement is relevant without explicitly mentioning π if it is clear from the context. Note that the condition for relevance is weaker than just requiring that the value of `x` is never read.

3.2 Computing the relevant statements of a single trace

Now that we have defined what a relevant statement is, we can turn to the question of how we can compute such statements. When we say "compute" we assume an oracle that can decide satisfiability of formulas over the first-order theories that correspond to the statements in the trace.

Let $\pi := \langle \mathrm{st}_1, \ldots, \mathrm{st}_n \rangle$ be the feasible error trace. In a first step we compute the coreachable and bireachable states along the trace. The coreachable states C_i can be iteratively computed as follows:

$$C_i := \begin{cases} \mathit{true} & i = n \\ \mathbf{pre}(C_{i+1}, \mathrm{st}_{i+1}) & i < n \end{cases}$$

Analogously, we can compute the reachable states R_i as follows:

$$R_i := \begin{cases} \mathit{true} & i = 0 \\ \mathbf{sp}(R_{i-1}, \mathrm{st}_i) & i > 0 \end{cases}$$

Finally, the bireachable states are simply obtained from the intersection, $B_i := R_i \cap C_i$.

Given the sequences C_i and B_i, for any assigning statement st_j with assigned variable `x`, we construct the following Hoare triple.

$$\{\, B_{j-1} \,\}\, \texttt{havoc}(x)\, \{\, C_j \,\}$$

The statement will be relevant iff the Hoare triple is not valid. Hoare triple validity can be checked by a theory solver for many theories that are used in practice [24]. We summarize the procedure in Algorithm 1. Note that the definition of relevance is individual for each

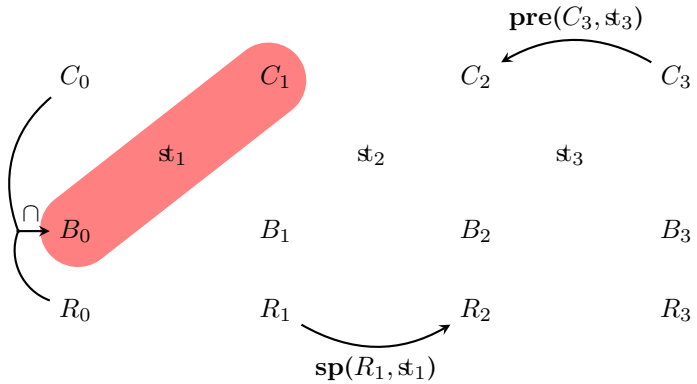


Fig. 3. Trace and state predicates. A Hoare triple is marked in red.

Algorithm 1: Relevant positions in a trace.

Input: $\mathtt{st}_i$ for $1 \leq i \leq n$: sequence of statements
C_i for $0 \leq i \leq n$: sequence of coreachable states
B_i for $0 \leq i \leq n$: sequence of bireachable states
Output: res: list of relevant positions

```
1 res ← [];
2 for j = 1 to n do
3 |   if st_j is not an assigning statement then continue;
4 |   Let x be the assigned variable in st_j;
5 |   if { B_{j-1} } havoc(x) { C_j } is invalid then res.append(j);
6 end
```

statement, i.e., we can make the computations independently from each other. Accordingly, the loop in Algorithm 1 can be parallelized.

3.3 Algorithm correctness

We now show that Algorithm 1 computes the relevant statements according to our definition.

THEOREM 3.2. *Let $\pi = \langle st_1, \ldots, \mathtt{st}_i, \ldots, \mathtt{st}_n \rangle$ be a feasible error trace of length n with $\mathtt{st}_i$ being an assigning statement that assigns a new value to some variable $\mathtt{x}$. Let φ be the set of bireachable states at position i and ψ be the coreachable states at position $i+1$. Then $\mathtt{st}_i$ is relevant iff*

$$\{\, \varphi \,\}\ \mathtt{havoc}(x)\ \{\, \psi \,\} \text{ is invalid.}$$

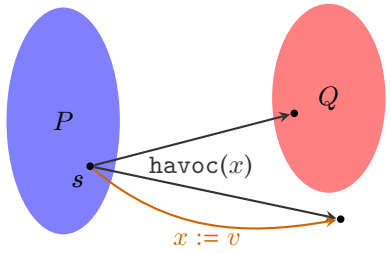


Fig. 4. Additional explanation for the proof of Theorem 3.2.

Proof. Let $\mathcal{D}$ be the domain of variable x.

$$
\begin{aligned}
& \mathfrak{st}_i \text{ is relevant} \\
\iff & \exists s \in \varphi.\ \exists v \in \mathcal{D}.\ \text{every execution of } \langle x := v, \mathfrak{st}_{i+1}, \ldots, \mathfrak{st}_n \rangle \text{ starting in } s \text{ is blocking} \\
\iff & \exists s \in \varphi.\ \exists v \in \mathcal{D}.\ \mathbf{sp}(s, x := v) \notin \psi \\
\overset{(1)}{\iff} & \exists s \in \varphi.\ s \notin \mathbf{wp}(\psi, \texttt{havoc}(x)) \\
\iff & \varphi \not\subseteq \mathbf{wp}(\psi, \texttt{havoc}(x)) \\
\overset{(2)}{\iff} & \{\, \varphi \,\}\, \texttt{havoc}(x)\, \{\, \psi \,\} \text{ is invalid}
\end{aligned}
$$

We only explain the nontrivial steps marked with $(\cdot)$.

(1) We graphically illustrate the situation in Figure 4. First note that, since $s \in \varphi$, there is at least one assignment to x that leads to ψ. Moreover, since from state s there is an assignment to x that leads outside of ψ, this successor can also be reached by a `havoc` of x, i.e., not all successors of s under `havoc(x)` are in ψ. On the other hand, every successor of s under `havoc(x)` can be reached by an assignment to x.

(2) A Hoare triple $\{\, \varphi \,\}\, \mathfrak{st}\, \{\, \psi \,\}$ is valid iff the precondition φ implies the weakest precondition $\mathbf{wp}(\psi, \mathfrak{st})$ (for any statement $\mathfrak{st}$). □

4 COMPUTING THE RELEVANT STATEMENTS OF A PROGRAM

The goal in this section is to identify the relevant statements in whole programs. A straightforward idea is to analyze a collection of *different* feasible error traces. We then say that a statement is relevant only if it is relevant in *all* traces. In certain applications, one may even say that those statements that are relevant in all feasible error traces should be ignored; this can for instance be reasonable when searching for concurrent bugs that should only occur for certain interleavings [23].

The main challenge is how to obtain *different* feasible error traces. One way is to take all failing tests from a test suite, possibly obtained from random testing. Here we propose another, fully automatic way of obtaining different feasible error traces.

4.1 Trace abstraction algorithm

We implemented our verification approach that can produce multiple (feasible) error traces and an extension of the *trace abstraction* software model checking algorithm. We shortly give a broad overview of the classical algorithm and refer the interested reader to the literature [16, 17]. The main idea of the trace abstraction algorithm is to separate two concerns:

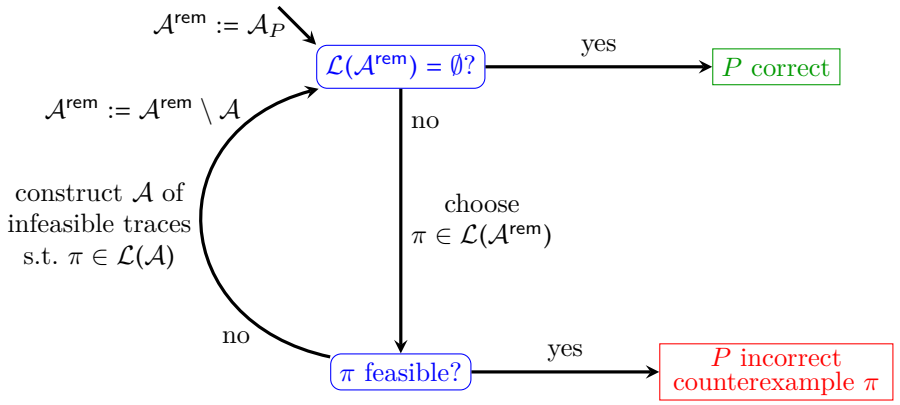


Fig. 5. Trace abstraction scheme.

(1) the control flow of the analyzed program and (2) the semantics of the programming language in which the program was written. The first concern is handled via automata-theoretical techniques, the second concern is handled via logical formulas and SMT solvers. Traces (i.e., sequences of statements) are classified in two dimensions.

(1) A trace is either an error trace or not an error trace. A trace is an error trace if it occurs as a labeling of a path from an initial location to an error location in the control flow automaton.
(2) A trace is either feasible or infeasible. We call a trace feasible if the programming language allows to execute the sequence of statements in a row. Formally, if the **sp** operator applied to *true* and this sequence returns a result different from *false*, the trace is feasible.

The algorithm now checks if for the given program there exists some feasible error trace. In case all error traces are infeasible, the program is correct.

The algorithm follows the iterative counterexample-guided abstraction refinement scheme [8] and is depicted in Figure 5. It maintains an automaton $\mathcal{A}^{\text{rem}}$ that represents all error traces that remain to be analyzed. The automaton $\mathcal{A}^{\text{rem}}$ is initialized with the control flow automaton $\mathcal{A}_P$ (whose error locations are the final states which define when a word is accepted). Afterwards, we do the following iteratively. We choose an error trace π and analyze its feasibility. If π is feasible the analysis is finished and the user gets the feasible error trace as an output. If π is infeasible, we want to subtract π from the remaining error traces and continue the analysis. In order to speed up the analysis (and this might be the key reason for its success) we do not only subtract the infeasible error trace π, but first generalize π to a set of traces that are infeasible. This set is represented as an automaton $\mathcal{A}$ (called Floyd-Hoare automaton [15, 17]) and subtracted from $\mathcal{L}(\mathcal{A}^{\text{rem}})$ via an automata-theoretical difference operation. If after some number of iterations the language of the $\mathcal{A}^{\text{rem}}$ is empty, we have shown that every error trace is infeasible and report to the user that the input program is safe.

4.2 Let trace abstraction compute several feasible error traces

In order to get multiple feasible error traces, we propose an extension of the trace abstraction algorithm. The idea of our extension is to not stop the iterative analysis after a feasible

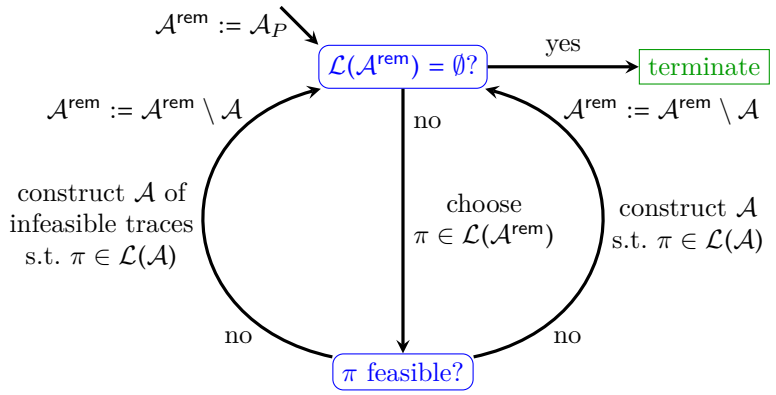


Fig. 6. Trace abstraction scheme with generalization of feasible error traces.

error trace was found but to continue in order to find more feasible error traces. A naive implementation of this idea would be to subtract the feasible error trace π from the set of error traces that still have to be analyzed $\mathcal{A}^{\text{rem}}$. It is easy to see that following this naive implementation we will systematically search for *every* feasible error trace. This number may, however, be too large (even infinite) for our application. Within a given timeout, we might get only feasible error traces that stem from different numbers of loop unwindings of similar paths through the control flow graph.

Hence, we propose to not only subtract π from $\mathcal{A}^{\text{rem}}$ but some automaton $\mathcal{A}$ that accepts π. The extended trace abstraction algorithm is depicted in Figure 6. We note that we do not put any further restrictions on $\mathcal{A}$. Especially, the automaton $\mathcal{A}$ may also accept infeasible traces. However, if $\mathcal{A}$ accepts many feasible error traces, the algorithm might terminate after a few iterations and will not return many feasible error traces.

In the remainder of this section we discuss automata that we consider a reasonable trade-off between both extremes.

4.3 Danger invariants

In the next subsection we introduce *danger automata* which are based on the notion of *danger invariants* [10]. In this subsection we briefly explain danger invariants.

A danger invariant serves as a proof of feasibility. The original definition was formulated for programs of the following form (adapted to our notation).

```
assume I;
while (G) {
  T;
}
assert A;
```

Here $I(s)$ is the initial condition, $G(s)$ is the loop guard, $T(s, s')$ is the transition relation describing the loop body, and $A(s)$ is the error condition.

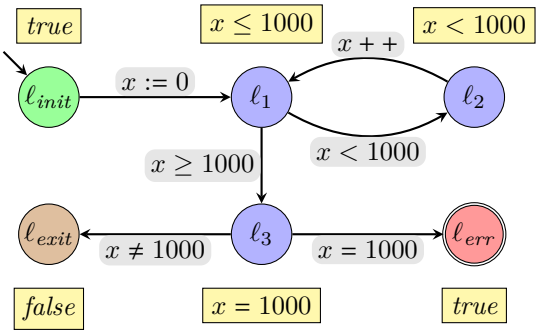


Fig. 7. A danger automaton together with a mapping that assigns to each location/state a predicate.

Definition 4.1 (Danger invariant [10]). A predicate over program states D is a *danger invariant* for the loop given above iff it satisfies the following criteria:

$$\exists s.\ I(s) \wedge D(s) \tag{1}$$

$$\forall s.\ D(s) \wedge G(s) \implies \exists s'.\ T(s, s') \wedge D(s') \tag{2}$$

$$\forall s.\ D(s) \wedge \neg G(s) \implies \neg A(s) \tag{3}$$

A danger invariant is a proof of feasibility in the following sense. Conditions (1) and (2) say that D is inductive, i.e., D contains at least one initial state and is invariant under the loop execution. Condition (3) says that after exiting the loop, the error must be reached. That is, every execution that starts in D must eventually reach the error or loop indefinitely.

4.4 Danger automata

Now we lift the idea of a danger invariant to automata as follows.

Definition 4.2 (Danger automaton). Let $\mathcal{A} = (Q, \delta, Q_{\mathsf{init}}, Q_{\mathsf{fin}})$ be an automaton over the alphabet of program statements Σ such that the set of initial states Q_{init} is not empty and the set of final states Q_{fin} is not empty. We call $\mathcal{A}$ a *danger automaton* if there exists a mapping $f : Q \to \mathsf{Preds}$ such that the following two properties hold.

$$f(q_{\mathsf{init}}) \neq \mathit{false} \quad \text{for some } q_{\mathsf{init}} \in Q_{\mathsf{init}} \tag{4}$$

$$f(q) \subseteq \bigcup_{(q, \mathfrak{st}, q') \in \delta} \mathbf{pre}(f(q'), \mathfrak{st}) \quad \text{for each } q \notin Q_{\mathsf{fin}} \tag{5}$$

The automaton depicted in Figure 7 is a danger automaton. Note that a state q that does not have any successors has to be a final state or $f(q)$ is *false*. Consequently, ℓ_{exit} is labeled with *false*. The language of the automaton can be written as the following regular expression.

`x := 0` `(x < 1000 x++)`* `x ≥ 1000` `x = 1000`

This language contains many infeasible traces such as $\langle$`x := 0` `x ≥ 1000` `x = 1000`$\rangle$. The language has, however, the property that every feasible prefix of a word has a feasible continuation. This is no coincidence and follows from Condition (5). We formalize this observation as follows.

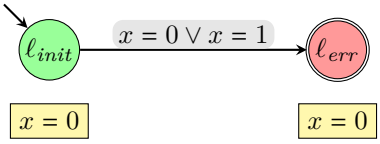


Fig. 8. A danger automaton whose initial and error conditions are not equivalent to *true*.

Proposition 4.3. *Given a danger automaton, for each reachable, non-dead end location $q \in Q$ and for each program state $s \in f(q)$ there exists a statement st, a successor location $q' \in \delta(q, st)$, and a program state $s' \in f(q')$ such that s, s' is an execution of $\langle st \rangle$.*

In the remaining section we also discuss the termination behavior of programs. We lift the definition of traces and executions to infinite traces and infinite executions as usual. We say that an infinite trace π is nonterminating if there exists a corresponding infinite execution. A typical example for an infinite trace that is not nonterminating is (`x ≥ 1` `x-`)$^\omega$. No matter how large the starting value of the variable `x` is, the statement `x ≥ 1` will eventually block the execution.

In the following we also consider words along infinite runs of automata, and we make the following definition. Given an automaton $\mathcal{A}$, we call a word w an *infinite word* of $\mathcal{A}$ if $\mathcal{A}$ has an infinite run over π. E.g., `x := 0` (`x < 1000` `x++`)$^\omega$ is an infinite word of the automaton depicted in Figure 7.

Proposition 4.4. *Given a danger automaton $\mathcal{A}$, $\mathcal{A}$ accepts a feasible trace or $\mathcal{A}$ has an infinite word $\pi = st_0, st_1, \ldots$ that is nonterminating.*

Note that in the previous proposition we do not have to refer to a specific mapping f.

Consider the simple danger automaton in Figure 8. An execution of a feasible trace needs not satisfy the predicates at each location. For example, the sequence of states $x = 1$, $x = 1$ is an execution for the trace $\langle$`x = 0 ∨ x = 1`$\rangle$, but `x = 1` $\not\models$ `x = 0`. This can happen if the predicate at the initial location is not equivalent to *true*.

Is any danger automaton a suitable candidate for our extension of the trace abstraction algorithm that was presented in Subsection 4.2? The answer to this question is 'no' as we will see on the following example. Let us presume that our representation of the remaining traces $\mathcal{A}^{\mathsf{rem}}$ is the automaton whose state graph is similar to the state graph of the automaton depicted in Figure 7 but without location ℓ_2. Let us also presume that the automaton depicted in Figure 7 is the automaton $\mathcal{A}$ that we want to subtract from $\mathcal{A}^{\mathsf{rem}}$ in this iteration. The automaton $\mathcal{A}$ is a danger automaton and hence accepts some feasible trace. However, no feasible trace accepted by $\mathcal{A}$ is a trace accepted by $\mathcal{A}^{\mathsf{rem}}$ and hence we do not make any progress in reducing the number of feasible error traces that are accepted by the automaton $\mathcal{A}^{\mathsf{rem}}$.

Next we make a definition and state a proposition that will help us to characterize automata that are suitable candidates for a refinement.

Given two automata $\mathcal{A}_1$ and $\mathcal{A}_2$, we say that $\mathcal{A}_2$ is *inferior to* $\mathcal{A}_1$ if

- each word accepted by $\mathcal{A}_1$ is accepted by $\mathcal{A}_2$, and
- each infinite word of $\mathcal{A}_1$ is an infinite word of $\mathcal{A}_2$.

Proposition 4.5. *Let $\mathcal{A}$ and $\mathcal{A}_D$ be two automata over the set of statements Σ. If*

- *all infinite traces of $\mathcal{A}$ are terminating,*

- $\mathcal{A}_D$ *is a danger automaton, and*
- $\mathcal{A}_D$ *is superior to* $\mathcal{A}$

then there exists some feasible trace π *that is accepted by the automaton* $\mathcal{A}$ *and by the danger automaton* $\mathcal{A}_D$*.*

We conclude that if we construct danger automata that are superior to the current representation of the remaining traces $\mathcal{A}^{\mathsf{rem}}$ and the analyzed program is terminating, we can guarantee that the algorithm makes progress in the following sense. In each iteration in which a feasible error trace was found, we subtract some feasible error trace from the set of traces that still have to be analyzed (i.e., from the set of traces that is given by the automaton $\mathcal{A}^{\mathsf{rem}}$). That this claim holds can be justified as follows.

In case a program is terminating, then also all infinite traces of the automata $\mathcal{A}^{\mathsf{rem}}$ in the trace abstraction algorithm will be terminating. Hence, also all infinite traces of $\mathcal{A}_D$ will be terminating (first bullet of 'inferior to' definition). Since every danger automaton accepts some feasible trace and the set of $\mathcal{A}$'s traces is a superset of $\mathcal{A}_D$'s traces (second bullet of 'inferior to' definition), the automaton $\mathcal{A}^{\mathsf{rem}}$ in the next iteration (obtained via a set difference operation $\mathcal{A}^{\mathsf{rem}} \backslash \mathcal{A}_D$) will reject some feasible error trace that was accepted by $\mathcal{A}^{\mathsf{rem}}$ in the iteration before.

4.5 The canonical danger automaton

In this section we will present the construction of a danger automaton that is inferior to a given automaton $\mathcal{A}^{\mathsf{rem}}$ that accepts a feasible trace.

The input to this construction will be an automaton $\mathcal{A}^{\mathsf{rem}} = (Q, \delta, Q_{\mathsf{init}}, Q_{\mathsf{fin}})$ (which will in the trace abstraction algorithm represent the error traces whose feasibility status is yet unknown) and a feasible trace $\pi = \mathtt{st}_1, \ldots, \mathtt{st}_n$ that is accepted by $\mathcal{A}^{\mathsf{rem}}$. The output of this construction will be a danger automaton that we call the *canonical danger automaton*.

As a first step, we construct the sequence of predicates $\varphi_0, \ldots, \varphi_n$ where $\varphi_n = \mathit{true}$ and $\varphi_i = \mathbf{pre}(\varphi_{i+1}, \mathtt{st}_{i+1})$. Let us use Preds to refer to the set of the sequence's elements. Next, we construct an annotation $\alpha : Q \to 2^{\mathsf{Preds}}$ that assigns to each state of $\mathcal{A}^{\mathsf{rem}}$ a subset of Preds.

Initially the mapping α assigns to each non-final state the empty set and to each final state the set that contains only the *true* predicate. Then we apply the following inference rule for each automaton state $q \in Q$ and for each predicate $\varphi \in \mathsf{Preds}$ until a fixpoint is reached.

$$\frac{\varphi \subseteq \bigcup_{(q,\mathtt{st},q') \in \delta} \mathbf{pre}(\alpha(q'), \mathtt{st})}{\varphi \in \alpha(q)}$$

Given the mapping α, we the *canonical danger automaton* for the automaton $\mathcal{A}^{\mathsf{rem}}$ and the feasible trace π is the automaton $\mathcal{A}^D = (Q^D, \delta^D, Q^D_{\mathsf{init}}, Q^D_{\mathsf{fin}})$ that is defined as follows.

- $Q^D = \{\langle q, \varphi \rangle \mid \text{ for each } q \in Q, \varphi \in \alpha(q)\}$
- $\delta^D = \{(\langle q, \varphi \rangle, \mathtt{st}, \langle q', \varphi' \rangle) \mid (q, \mathtt{st}, q') \in \delta,\ \varphi \cap \mathbf{pre}(\varphi', \mathtt{st}) \neq \emptyset\}$
- $Q^D_{\mathsf{init}} = \{\langle q, \varphi \rangle \in Q^D \mid q \in Q_{\mathsf{init}}\}$
- $Q^D_{\mathsf{fin}} = \{\langle q, \varphi \rangle \in Q^D \mid q \in Q_{\mathsf{fin}}\}$

Proposition 4.6. *A canonical danger automaton is a danger automaton.*

Proof. (Sketch) Let $\mathcal{A}_D$ be the canonical danger automaton that was constructed for an automaton $\mathcal{A}^{\mathsf{rem}}$ and a feasible trace $\pi = \mathtt{st}_1, \ldots, \mathtt{st}_n$. If we use the mapping defined by

$f(\langle q, \varphi\rangle) = \varphi$ the property (5) from the definition of a danger automaton holds trivially. We show that property (4) also holds as follows.

Since π is accepted by $\mathcal{A}^{\mathsf{rem}}$, there exists a run $q_0, ..., q_n$ of $\mathcal{A}^{\mathsf{rem}}$ over π. We have that for i, the predicate φ_i is in the set $\alpha(q_i)$ because for each i, q_i has the outgoing transition $(q_i, \mathfrak{st}_{i+1}, q_{i+1})$ and $\varphi_i \subseteq \mathbf{pre}(\varphi_{i+1}, \mathfrak{st}_{i+1})$ holds by definition of the predicate sequence. Since π is feasible each predicate in the sequence is different from *false*. This holds especially for φ_0 which is the annotation of the initial state $\langle q_0, \varphi_0\rangle$. □

Proposition 4.7. *The canonical danger automaton $\mathcal{A}_D$ constructed for the automaton $\mathcal{A}^{\mathsf{rem}}$ is inferior to $\mathcal{A}^{\mathsf{rem}}$.*

Proof. (Sketch) For each infinite run $\langle q_0, \psi_0\rangle, \ldots$ of $\mathcal{A}_D$ over an infinite trace π, the sequence $q_0, \ldots$ is an infinite run of $\mathcal{A}^{\mathsf{rem}}$ over π. The same holds for finite runs and if $\langle q_i, \psi_i\rangle$ is accepting, then q_i is accepting. □

The preceding propositions shows that if we use the canonical danger automaton in the trace abstraction algorithm, we get the progress property that we discussed at the end of Subsection 4.4.

5 EVALUATION

We have implemented our fault localization approach in the software analysis framework Ultimate. For the analysis on programs, which for convenience we call *multi-trace* analysis in this section, we modified the trace abstraction implementation Ultimate Automizer [15]. To evaluate our approach, we run Ultimate Automizer on 50 randomly chosen programs from the competition on software verification (SV-COMP) [5] that contain bugs, once in the single-trace setting where we terminate upon finding the first error trace, and once in the multi-trace setting where we continue as long as the abstraction is empty or the analysis runs into a timeout. For comparison we also evaluate the single-trace analysis in the context of Ultimate Automizer.

We performed the experiments on a PC with an Intel i7 3.60 GHz CPU running Linux, where we set the timeout to 120 s and restricted the available memory to 6 GiB. On 9 of the programs Ultimate Automizer crashed or found no error trace within these limitations; we omitted these programs from our evaluation.

We present the results in Table 1 for the single-trace analysis and in Table 2 for the multi-trace analysis, respectively. The program names are abbreviated by their category/folder name due to space constraints. The full file names can be found in Appendix A.

In the first two columns we report on the number of relevant statements compared to the total number of statements; in the multi-trace analysis the total number of statements corresponds to the sum of the statements in all traces where we ignored duplicates. There is no clear trend and the numbers range from roughly 10% to 50%. Only for the program systemc4 there was no relevant statement left in the multi-trace analysis.

In the next two (three) columns we report on the runtime of the fault localization, of the danger automaton construction in case of the multi-trace analysis, and on the total analysis time of Ultimate Automizer (including fault localization and danger automaton construction). Observe that for some programs the multi-trace analysis terminated earlier than the 120 s timeout, which means that the abstraction became empty. We see that the fault localization is comparably cheap and takes less than a second on average. The danger automaton construction time usually dominates the total runtime (sometimes it takes more than 90%).

Table 1. Results from single-trace analysis.

	Number of relevant statements	Number of statements in trace	Fault localization time	Total analysis time	Total number of iterations
locks1	16	96	1	1	1
locks2	17	102	1	1	1
loop-ac5	5	12	0	0	2
loop-ac11	22	47	0	1	8
loop-ac12	15	47	0	1	8
loop-lit1	4	19	0	0	8
loops3	6	13	0	0	2
loops5	16	36	0	1	5
loops10	23	63	0	1	7
loops12	5	16	0	0	2
ntdrivers1	46	227	2	6	36
ntdrivers2	49	295	3	5	36
ntdrivers3	52	310	4	7	40
ntdrivers4	30	203	1	2	16
plines9	26	80	0	1	4
plines10	26	81	0	1	4
plines11	26	86	0	1	5
plines12	27	86	0	1	5
ssh1	32	278	3	4	14
ssh2	34	280	3	4	14
ssh3	39	305	4	5	14
ssh4	35	286	4	5	14
ssh5	15	161	1	1	5
ssh6	46	373	11	22	47
ssh7	46	339	7	10	37
ssh8	37	269	4	6	24
ssh9	24	203	2	3	13
ssh10	22	273	4	5	10
ssh11	22	279	5	6	10
systemc1	130	450	4	7	29
systemc2	114	378	3	76	42
systemc3	22	93	0	0	2
systemc4	385	1678	41	53	34
systemc5	66	230	1	2	13
systemc6	104	349	2	5	23
systemc7	150	494	6	20	32
systemc8	79	354	2	7	40
systemc9	79	352	2	7	38
systemc10	43	156	0	1	6
systemc11	76	261	1	3	18
systemc12	117	392	3	9	25

The next column shows the total number of iterations of Ultimate Automizer (cf. Figure 6). From Table 1 we can read the iteration in which the first error trace was found. Table 2 additionally shows another column for the total number of error traces that were found (and hence also the number of fault localizations and danger automaton constructions that were performed). We can see that for some programs, e.g., those from the category ssh, only a few error traces were found, while for others, e.g., those from the category plines, in

Table 2. Results from multi-trace analysis.

	Number of relevant statements	Number of statements in all traces	Fault localization time	Automaton construction time	Total analysis time	Total number of iterations	Total number of error traces
locks1	16	173	17	23	119	71	29
locks2	17	154	8	13	120	51	9
loop-ac5	5	19	0	0	1	3	2
loop-ac11	7	16	0	2	3	9	1
loop-ac12	5	16	0	0	1	9	1
loop-lit1	3	40	4	110	120	67	51
loops3	6	20	3	117	120	18	17
loops5	16	35	4	115	120	19	15
loops10	7	19	0	0	1	8	1
loops12	5	18	0	0	0	3	2
ntdrivers1	44	243	8	10	28	89	4
ntdrivers2	48	368	41	32	78	69	8
ntdrivers3	51	383	46	41	95	86	8
ntdrivers4	30	203	1	1	4	29	1
plines9	25	97	37	81	120	88	76
plines10	25	94	33	85	120	62	51
plines11	26	99	36	82	120	59	48
plines12	27	138	40	80	120	65	47
ssh1	31	235	19	100	120	23	6
ssh2	33	237	15	104	120	22	5
ssh3	38	255	10	109	120	18	3
ssh4	33	243	12	106	120	20	4
ssh5	15	161	1	1	18	63	1
ssh6	40	258	16	93	120	51	3
ssh7	37	242	18	99	120	42	4
ssh8	31	223	23	94	120	34	6
ssh9	21	201	14	25	72	67	6
ssh10	22	304	30	78	120	59	8
ssh11	22	306	33	77	120	59	8
systemc1	68	235	8	23	36	37	3
systemc2	96	333	11	29	120	50	4
systemc3	22	230	8	19	29	34	5
systemc4	0	500	0	108	120	34	1
systemc5	54	191	30	87	120	49	24
systemc6	90	299	29	86	120	41	12
systemc7	134	433	22	79	120	42	5
systemc8	66	259	21	92	120	56	10
systemc9	66	257	18	95	120	53	9
systemc10	37	137	4	8	13	22	6
systemc11	69	237	31	84	120	39	14
systemc12	108	361	25	80	120	36	5

most iterations an error trace was considered. We note that we have not modified the choice of the abstract counterexample implemented in Ultimate Automizer.

6 RELATED WORK

In the following we discuss the works that are related to this paper.

6.1 Fault localization

Fault localization is a big topic in the field of program analysis with a vast abundance of works. Most approaches try to identify statements that "have something to do with the error". The exact notion of what this means differs quite a lot; we adapt the terminology to ours (e.g., we say "relevant statements") for convenience. There certainly does not exist a *best* fault localization technique. Many approaches have complementary strengths and weaknesses and can be combined.

6.1.1 Major research lines. A recent survey collects 331 research papers and theses ranging from 1977 to late 2014 which shows that this area has appealed the interest of many researchers [41]. The authors classify the works into seven major categories (spectrum-based, slice-based, statistics-based, program state-based, machine learning-based, data mining-based, and model-based) and several minor ones. We next discuss these major categories. Our approach does not fall into any of these categories.

Most fault localization techniques are *spectrum-based*. They compare passing and failing executions obtained from a test suite (usually satisfying a certain coverage criterion) and then rank the statements according to their relevance [1, 2, 21, 28, 40]. The basic idea is that the more often a statement appears in failing executions, the more relevant it is. *Mutation-based* techniques [27, 31] extend this idea by also taking into account how often a statement mutation (i.e., replacing a statement by another statement) affects the test outcome. We also replace statements, but we do that symbolically, contrary to performing a concrete replacement and reexecuting the program. Another variant is to use a distance measure between traces to identify the correct trace that is most similar to the feasible error trace [35]. Our approach only analyzes a single feasible error trace and hence does not depend on the variation of the test suite. However, the lifting to programs follows similar ideas: We also sample (finitely many) feasible error traces and the outcome of the analysis depends on the particular choice of traces. We have shown one way to increase the variety in a concrete implementation using the notion of danger automata. Recently, the above-mentioned spectrum- and mutation-based ranking techniques have been evaluated in a large-scale comparison [32]. The authors of the survey found that most of the results from previous comparisons were reproducible on programs with artificial bugs. However, the results were not reproducible (with statistical relevance) on programs with real bugs. In our experiments we have considered programs from SV-COMP which contained both artificial and real bugs. At the moment we do not know whether our approach is biased for artificial or real bugs.

Slice-based techniques delete parts from a program such that it still retains the original behavior w.r.t. a certain specification [39]. For example, Zhang *et al.* evaluated several dynamic slicing techniques for the purpose of fault localization [42]. A statement is relevant if it occurs in the slice. Slices that only contain DEF-USE related statements are usually too small, while slices that additionally contain control flow related statements often contain all bug-related statements. By contrast, we do not remove statements but rather replace them with new statements. Since slicing techniques often only remove irrelevant statements, they can be used as a preprocessing to other, more precise and more expensive fault localization techniques such as ours.

Statistics-based techniques add predicates (e.g., `x > 0`) to program points and then run tests to compute the conditional probability of the predicate being satisfied (or just being visited) and an error [26]. The predicates can then be ranked by these probabilities in order

to find possible causalities. Obtaining predicates that hint at causal relationships can be a valuable information for the developer that is orthogonal to pointing out relevant statements. In our approach there is no testing involved and the predicates that we compute play a completely different role.

Program state-based techniques, as the name suggests, compare the program states of several executions, e.g., via delta debugging [43]. We have also characterized our notion of relevance via program states (cf. Definition 3.1). However, our algorithm for fault localization is symbolic (i.e., we consider sets of program states) and static (i.e., we do not execute the program).

In the past years *machine learning* has become a hot topic, and several approaches that try to learn relevant statements have been proposed [6, 38]. Such approaches automatically learn models of failures from input data, e.g., statement coverage, and are quite different from our single trace-based formal analysis. This abstract view also subsumes *data mining-based* techniques where one tries to identify a pattern, e.g., of statements in error traces [29]. Informally, a human-invented fault localization technique often works well for programs with a certain structure that it was designed for. Learning algorithms can be seen as an automatic method to find such structures, even those that humans would not detect.

Model-based techniques assume a correct model of the program and compare the model to the erroneous program [34]. Such an assumption can be exploited in many different flavors. Our approach does not make this assumption and thus cannot be compared to model-based approaches.

6.1.2 Other approaches. In the following we discuss those approaches that are most related to ours.

Ball *et al.* localize faults in the context of model checking by comparing feasible error traces to "correct" traces (i.e., traces that terminate normally) [4]. They call a statement relevant if it does not occur on a correct trace (thus this approach can be seen as spectrum-based). In contrast, we need only a single trace for our analysis. When analyzing several feasible error traces, our definition is rather dual: We call a statement relevant if it is relevant on all traces. The authors also show how their approach can be used to generate different feasible error traces with the model checker. After identifying relevant statements, one can remove the respective transition from the transition system and continue the analysis normally. Every feasible error trace produced in this way has different relevant statements. Our automata-based difference may look similar, but in fact we only subtract a set of traces, which does not necessarily mean that any transition is removed from the minuend. Our subtraction is also not based on relevant statements (at this point we only collect feasible error traces); in fact, a danger automaton may not contain a single relevant statement.

Jose *et al.* triggered a line of formula-based fault localization works akin to ours [22]. Given a feasible error trace, they construct a Boolean *unsatisfiable trace formula* consisting of an input I that triggers the error, the trace formula π, and the negated error condition E:

$$I \wedge \pi \wedge \neg E$$

Passing the formula to a partial MAX-SAT solver, where φ and $\neg E$ are considered as so-called hard clauses, the solver returns a maximal number of clauses (from π) that can still be satisfied. All other clauses are then considered relevant because removing them all would make the trace formula satisfiable, which corresponds to that the error condition may hold. The authors also proposed to analyze several feasible error traces and report those statements that occur more often with higher priority. On the other hand, our approach

works on first-order formulas, and we query the theory solver several times with smaller formulas. When there are several possible error causes, the partial MAX-SAT solver will choose one with the fewest clauses. The result of our algorithm is unique.

Ermis *et al.* adapted the idea to so-called *error invariants* [12] which are defined similarly to Craig interpolants. Error invariants are predicates, one between each statement, that overapproximate the reachable states but are still strong enough to imply the error condition. If a given predicate is an error invariant at positions i and $j > i$, the authors say that the statements in between are irrelevant and can be dropped, or equivalently, replaced by a `skip` statement. Given a set of predicates, the authors find a smallest such covering of the error trace with error invariants. Instead, we replace assigning statements by an assignment statement if a statement is relevant. Error invariants are in general not unique, and thus two implementations may differ in their output. As another difference, the authors used the weakest precondition function (**wp**) of the trace for I in the formula above. We use the precondition function (**pre**) which allows us to handle traces that contain nondeterminism. This is a general limitation of techniques based on the unsatisfiable trace formula because the formula needs to be unsatisfiable. Error invariants, unlike danger invariants [10], are not inductive. The error invariants approach was later extended to "flow-sensitive" analysis that supports error conditions inside `if-endif` blocks [7], to concolic testing [30], and to concurrent traces [19].

6.2 Error invariant automata

Schäf *et al.* generalize the idea of error invariants to *error invariant automata* [37]. The locations are annotated with error invariants (see the discussion above) [12]. In its basic form, the automaton accepts a single feasible error trace where no error invariant repeats, i.e., the trace where all irrelevant statements have been dropped. The automata can then be mapped back to the original program they were constructed for and be interpreted as smaller versions of this program. In another view, only those statements that remain in the error invariant automaton are relevant. The approach requires that all traces of the program are "inconsistent", which means that every error trace formula is unsatisfiable. We have no such assumption, i.e., our danger automaton construction is universally applicable. The traces that the automata represent are inherently different. While both automata are constructed from a feasible error trace π, a danger automaton still accepts π, while an error invariant automaton constructs an unsatisfiable formula from π and hence somehow accepts an infeasible version of π.

6.3 Proof of feasibility

Inductive safe invariants represent a proof of program safety [20]. For unsafety, one usually considers a whole feasible error trace as a proof. Motivated by the fact that deep bugs have long feasible error traces, and thus long proofs, David *et al.* introduced the notion of a danger invariant [10]. Danger invariants give a compact representation of a feasibility proof, and we have already discussed them in Section 4.3. They can be seen as one form of *must* summaries [3, 13, 14] where a state *can* reach all of its successors.

The doomed program point paradigm by Hoenicke *et al.* is more restrictive than danger invariants and requires that all executions starting from that point inevitably lead to the error [18]. Danger invariants only ensure that there exists at least one such execution.

6.4 Program repair

While reporting relevant statements to the developer can be a tremendous help in debugging, the even better solution would be to automatically repair the program. Several recent approaches to this challenging task have been proposed [9, 36]. At first glance our fault localization technique with replacing a relevant assigning statement may seem similar, but we do not have a guarantee that the new assignment removes the bug for all executions (nor that it does not introduce new bugs).

6.5 Fault correlation

Le and Soffa propose a technique to correlate different faults [25]. Two faults are correlated if they occur together along a feasible error trace. This information can be used to prioritize bug fixes because sometimes fixing one bug fixes another one. In our analysis we do not relate relevant statements, and our notion of simple bug fixes and the computation of relevant statements is independent of other statements.

6.6 Bug classification

Podelski *et al.* classify feasible error traces according to a semantic characterization [33]. The concept is a natural extension or error invariants (see the discussion above) [12] and, intuitively speaking, two error traces are similar if there exists a sequence of predicates that are error invariants for both traces. Our danger automata can also be seen as (an equivalence class of) a classification of feasible error traces. We are, however, not aware of a similar description of when two feasible error traces belong to the same danger automaton.

REFERENCES

[1] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J. C. van Gemund. 2009. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software* 82, 11 (2009), 1780–1792. https://doi.org/10.1016/j.jss.2009.06.035

[2] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. 2009. Spectrum-Based Multiple Fault Localization. In *ASE*. IEEE Computer Society, 88–99. https://doi.org/10.1109/ASE.2009.25

[3] Thomas Ball, Orna Kupferman, and Greta Yorsh. 2005. Abstraction for Falsification. In *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings (LNCS)*, Kousha Etessami and Sriram K. Rajamani (Eds.), Vol. 3576. Springer, 67–81. https://doi.org/10.1007/11513988_8

[4] Thomas Ball, Mayur Naik, and Sriram K. Rajamani. 2003. From symptom to cause: localizing errors in counterexample traces. In *POPL*, Alex Aiken and Greg Morrisett (Eds.). ACM, 97–105. http://doi.acm.org/10.1145/640128.604140

[5] Dirk Beyer. 2016. Reliable and Reproducible Competition Results with BenchExec and Witnesses (Report on SV-COMP 2016). In *TACAS (LNCS)*, Marsha Chechik and Jean-François Raskin (Eds.), Vol. 9636. Springer, 887–904. https://doi.org/10.1007/978-3-662-49674-9_55

[6] Martin Chapman, Hana Chockler, Pascal Kesseli, Daniel Kroening, Ofer Strichman, and Michael Tautschnig. 2015. Learning the Language of Error. In *ATVA (LNCS)*, Vol. 9364. Springer, 114–130.

[7] Jürgen Christ, Evren Ermis, Martin Schäf, and Thomas Wies. 2013. Flow-Sensitive Fault Localization. In *VMCAI (LNCS)*, Vol. 7737. Springer, 189–208.

[8] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2000. Counterexample-Guided Abstraction Refinement. In *CAV (LNCS)*, E. Allen Emerson and A. Prasad Sistla (Eds.), Vol. 1855. Springer, 154–169. https://doi.org/10.1007/10722167_15

[9] Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. 2016. Qlose: Program Repair with Quantitative Objectives. In *CAV (2) (LNCS)*, Vol. 9780. Springer, 383–401. http://dx.doi.org/10.1007/978-3-319-41540-6_21

[10] Cristina David, Pascal Kesseli, Daniel Kroening, and Matt Lewis. 2016. Danger Invariants. In *FM (LNCS)*, Vol. 9995. 182–198.

[11] Edsger W. Dijkstra. 1975. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM* 18, 8 (1975), 453–457. http://doi.acm.org/10.1145/360933.360975

[12] Evren Ermis, Martin Schäf, and Thomas Wies. 2012. Error Invariants. In *FM (LNCS)*, Dimitra Giannakopoulou and Dominique Méry (Eds.), Vol. 7436. Springer, 187–201. https://doi.org/10.1007/978-3-642-32759-9_17

[13] Patrice Godefroid. 2014. May/Must Abstraction-Based Software Model Checking for Sound Verification and Falsification. In *Software Systems Safety*, Orna Grumberg, Helmut Seidl, and Maximilian Irlbeck (Eds.). NATO Science for Peace and Security Series, D: Information and Communication Security, Vol. 36. IOS Press, 1–16. https://doi.org/10.3233/978-1-61499-385-8-1

[14] Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and SaiDeep Tetali. 2010. Compositional may-must program analysis: unleashing the power of alternation. In *POPL*, Manuel V. Hermenegildo and Jens Palsberg (Eds.). ACM, 43–56. http://doi.acm.org/10.1145/1706299.1706307

[15] Matthias Heizmann, Yu-Wen Chen, Daniel Dietsch, Marius Greitschus, Alexander Nutz, Betim Musa, Claus Schätzle, Christian Schilling, Frank Schüssele, and Andreas Podelski. 2017. Ultimate Automizer with an On-Demand Construction of Floyd-Hoare Automata - (Competition Contribution). In *TACAS (LNCS)*, Axel Legay and Tiziana Margaria (Eds.), Vol. 10206. 394–398. https://doi.org/10.1007/978-3-662-54580-5_30

[16] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. 2009. Refinement of Trace Abstraction. In *SAS (Lecture Notes in Computer Science)*, Vol. 5673. Springer, 69–85.

[17] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. 2013. Software Model Checking for People Who Love Automata. In *CAV (LNCS)*, Natasha Sharygina and Helmut Veith (Eds.), Vol. 8044. Springer, 36–52. https://doi.org/10.1007/978-3-642-39799-8_2

[18] Jochen Hoenicke, K. Rustan M. Leino, Andreas Podelski, Martin Schäf, and Thomas Wies. 2009. It's Doomed; We Can Prove It. In *FM (LNCS)*, Ana Cavalcanti and Dennis Dams (Eds.), Vol. 5850. Springer, 338–353. https://doi.org/10.1007/978-3-642-05089-3_22

[19] Andreas Holzer, Daniel Schwartz-Narbonne, Mitra Tabaei Befrouei, Georg Weissenbacher, and Thomas Wies. 2016. Error Invariants for Concurrent Traces. In *FM (LNCS)*, John S. Fitzgerald, Constance L. Heitmeyer, Stefania Gnesi, and Anna Philippou (Eds.), Vol. 9995. 370–387. https://doi.org/10.1007/

978-3-319-48989-6_23

[20] Ranjit Jhala, Andreas Podelski, and Andrey Rybalchenko. to appear in 2017. Predicate abstraction for program verification. In *Handbook of Model Checking*, Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem (Eds.). Springer International Publishing, Chapter 15.

[21] James A. Jones and Mary Jean Harrold. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *ASE*, David F. Redmiles, Thomas Ellman, and Andrea Zisman (Eds.). ACM, 273–282. http://doi.acm.org/10.1145/1101908.1101949

[22] Manu Jose and Rupak Majumdar. 2011. Cause clue clauses: error localization using maximum satisfiability. In *PLDI*, Mary W. Hall and David A. Padua (Eds.). ACM, 437–446. http://doi.acm.org/10.1145/1993498.1993550

[23] Saurabh Joshi, Shuvendu K. Lahiri, and Akash Lal. 2012. Underspecified harnesses and interleaved bugs. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, John Field and Michael Hicks (Eds.). ACM, 19–30. https://doi.org/10.1145/2103656.2103662

[24] Daniel Kroening and Ofer Strichman. 2016. *Decision Procedures - An Algorithmic Point of View, Second Edition*. Springer. https://doi.org/10.1007/978-3-662-50497-0

[25] Wei Le and Mary Lou Soffa. 2010. Path-based fault correlations. In *FSE*, Gruia-Catalin Roman and Kevin J. Sullivan (Eds.). ACM, 307–316. http://doi.acm.org/10.1145/1882291.1882336

[26] Ben Liblit, Mayur Naik, Alice X. Zheng, Alexander Aiken, and Michael I. Jordan. 2005. Scalable statistical bug isolation. In *PLDI*, Vivek Sarkar and Mary W. Hall (Eds.). ACM, 15–26. http://doi.acm.org/10.1145/1065010.1065014

[27] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. 2014. Ask the Mutants: Mutating Faulty Programs for Fault Localization. In *ICST*. IEEE Computer Society, 153–162. https://doi.org/10.1109/ICST.2014.28

[28] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. 2011. A model for spectra-based software diagnosis. *ACM Trans. Softw. Eng. Methodol.* 20, 3 (2011), 11:1–11:32. http://doi.acm.org/10.1145/2000791.2000795

[29] Syeda Nessa, Muhammad Abedin, W. Eric Wong, Latifur Khan, and Yu Qi. 2008. Software Fault Localization Using N-gram Analysis. In *WASA (LNCS)*, Yingshu Li, Dung T. Huynh, Sajal K. Das, and Ding-Zhu Du (Eds.), Vol. 5258. Springer, 548–559. https://doi.org/10.1007/978-3-540-88582-5_51

[30] Chanseok Oh, Martin Schäf, Daniel Schwartz-Narbonne, and Thomas Wies. 2014. Concolic Fault Abstraction. In *SCAM*. IEEE Computer Society, 135–144.

[31] Mike Papadakis and Yves Le Traon. 2015. Metallaxis-FL: mutation-based fault localization. *Softw. Test., Verif. Reliab.* 25, 5-7 (2015), 605–628. https://doi.org/10.1002/stvr.1509

[32] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and improving fault localization. In *ICSE*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE / ACM, 609–620. http://dl.acm.org/citation.cfm?id=3097441

[33] Andreas Podelski, Martin Schäf, and Thomas Wies. 2016. Classifying Bugs with Interpolants. In *TAP (LNCS)*, Bernhard K. Aichernig and Carlo A. Furia (Eds.), Vol. 9762. Springer, 151–168. https://doi.org/10.1007/978-3-319-41135-4_9

[34] Raymond Reiter. 1987. A Theory of Diagnosis from First Principles. *Artif. Intell.* 32, 1 (1987), 57–95. https://doi.org/10.1016/0004-3702(87)90062-2

[35] Manos Renieris and Steven P. Reiss. 2003. Fault Localization With Nearest Neighbor Queries. In *ASE*. IEEE Computer Society, 30–39. https://doi.org/10.1109/ASE.2003.1240292

[36] Roopsha Samanta, Oswaldo Olivo, and E. Allen Emerson. 2014. Cost-Aware Automatic Program Repair. In *SAS (LNCS)*, Vol. 8723. Springer, 268–284. http://dx.doi.org/10.1007/978-3-319-10936-7_17

[37] Martin Schäf, Daniel Schwartz-Narbonne, and Thomas Wies. 2013. Explaining inconsistent code. In *ESEC/SIGSOFT FSE*. ACM, 521–531.

[38] Westley Weimer and George C. Necula. 2005. Mining Temporal Specifications for Error Detection. In *TACAS (LNCS)*, Vol. 3440. Springer, 461–476.

[39] Mark David Weiser. 1979. *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method*. Ph.D. Dissertation. Ann Arbor, MI, USA. AAI8007856.

[40] W. Eric Wong, Vidroha Debroy, Ruizhi Gao, and Yihao Li. 2014. The DStar Method for Effective Software Fault Localization. *IEEE Trans. Reliability* 63, 1 (2014), 290–308. https://doi.org/10.1109/TR.2013.2285319

[41] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Trans. Software Eng.* 42, 8 (2016), 707–740.

[42] Xiangyu Zhang, Haifeng He, Neelam Gupta, and Rajiv Gupta. 2005. Experimental Evaluation of Using Dynamic Slices for Fault Location. In *AADEBUG*. ACM, 33–42. http://doi.acm.org/10.1145/1085130.1085135

[43] Thomas Zimmermann and Andreas Zeller. 2001. Visualizing Memory Graphs. In *Software Visualization (LNCS)*, Stephan Diehl (Ed.), Vol. 2269. Springer, 191–204. https://doi.org/10.1007/3-540-45875-1_15

A APPENDIX

A.1 Program Names

locks1 locks/test_locks_14_false-unreach-call_true-valid-memsafety_false-termination.c.
locks2 locks/test_locks_15_false-unreach-call_true-valid-memsafety_false-termination.c.
loop-ac11 loop-acceleration/underapprox_false-unreach-call1_true-termination.i.
loop-ac12 loop-acceleration/underapprox_false-unreach-call2_true-termination.i.
loop-ac5 loop-acceleration/multivar_false-unreach-call1_true-termination.i.
loop-lit1 loop-lit/gcnr2008_false-unreach-call_false-termination.i.
loops10 loops/sum04_false-unreach-call_true-termination.i.
loops12 loops/terminator_03_false-unreach-call_true-termination.i.
loops3 loops/count_up_down_false-unreach-call_true-termination.i.
loops5 loops/for_bounded_loop1_false-unreach-call_true-termination.i.

ntdrivers1 ntdrivers-simplified/cdaudio_simpl1_false-unreach-call_true-valid-memsafety_true-termination.cil.c.
ntdrivers2 ntdrivers-simplified/floppy_simpl3_false-unreach-call_true-valid-memsafety_true-termination.cil.c.
ntdrivers3 ntdrivers-simplified/floppy_simpl4_false-unreach-call_true-valid-memsafety_true-termination.cil.c.
ntdrivers4 ntdrivers-simplified/kbfiltr_simpl2_false-unreach-call_true-valid-memsafety_true-termination.cil.c.

plines10 product-lines/minepump_spec3_product10_false-unreach-call_false-termination.cil.c.
plines11 product-lines/minepump_spec3_product29_false-unreach-call_false-termination.cil.c.
plines12 product-lines/minepump_spec3_product40_false-unreach-call_false-termination.cil.c.
plines9 product-lines/minepump_spec3_product03_false-unreach-call_false-termination.cil.c.

ssh1 ssh-simplified/s3_clnt_1_false-unreach-call.cil.c.
ssh10 ssh-simplified/s3_srvr_1_false-unreach-call.cil.c.
ssh11 ssh-simplified/s3_srvr_2_false-unreach-call.cil.c.
ssh2 ssh-simplified/s3_clnt_2_false-unreach-call_true-termination.cil.c.
ssh3 ssh-simplified/s3_clnt_3_false-unreach-call.cil.c.
ssh4 ssh-simplified/s3_clnt_4_false-unreach-call.cil.c.
ssh5 ssh-simplified/s3_srvr_10_false-unreach-call.cil.c.
ssh6 ssh-simplified/s3_srvr_11_false-unreach-call.cil.c.
ssh7 ssh-simplified/s3_srvr_12_false-unreach-call.cil.c.
ssh8 ssh-simplified/s3_srvr_13_false-unreach-call.cil.c.
ssh9 ssh-simplified/s3_srvr_14_false-unreach-call.cil.c.
systemc1 systemc/kundu1_false-unreach-call_false-termination.cil.c.
systemc10 systemc/transmitter.01_false-unreach-call_false-termination.cil.c.
systemc11 systemc/transmitter.02_false-unreach-call_false-termination.cil.c.
systemc12 systemc/transmitter.03_false-unreach-call_false-termination.cil.c.
systemc2 systemc/kundu2_false-unreach-call_false-termination.cil.c.
systemc3 systemc/pc_sfifo_2_false-unreach-call_false-termination.cil.c.
systemc4 systemc/pipeline_false-unreach-call_false-termination.cil.c.
systemc5 systemc/token_ring.01_false-unreach-call_false-termination.cil.c.
systemc6 systemc/token_ring.02_false-unreach-call_false-termination.cil.c.
systemc7 systemc/token_ring.03_false-unreach-call_false-termination.cil.c.

systemc8 systemc/toy1_false-unreach-call_false-termination.cil.c.
systemc9 systemc/toy2_false-unreach-call_false-termination.cil.c.