Flow-sensitive Fault Localization

Jürgen Christ[1], Evren Ermis[1], Martin Schäf[2]*, and Thomas Wies[3]

[1] University of Freiburg
[2] United Nations University, IIST, Macau
[3] New York University

Abstract. Identifying the cause of an error is often the most time-consuming part in program debugging. Fault localization techniques can help to automate this task. Particularly promising are static proof-based techniques that rely on an encoding of error traces into *trace formulas*. By identifying irrelevant portions of the trace formula, the possible causes of the error can be isolated. One limitation of these approaches is that they do not take into account the control flow of the program and therefore miss common causes of errors, such as faulty branching conditions. This limitation is inherent to the way the error traces are encoded. In this paper, we present a new flow-sensitive encoding of error traces into trace formulas. The new encoding enables proof-based techniques to identify irrelevant conditional choices in an error trace and to include a justification for the truth value of branching conditions that are relevant for the localized cause of an error. We apply our new encoding to the fault localization technique based on error invariants and show that it produces more meaningful error explanations than previous approaches.

1 Introduction

Debugging programs is tedious. Often the most time consuming part in debugging is the task of *fault localization*. To localize the fault of an error, the programmer has to trace the program execution, e.g., starting from a malicious test input, and identify the *relevant* statements that explain the error. Using this information the programmer can then devise a solution to fix the program. Automated fault localization tools [1, 6–8, 11, 14, 15, 17, 19, 20] that reduce the manual effort involved in this task can significantly improve the programmer's work experience.

Particularly promising are proof-based fault localization techniques [6, 11]. These techniques rely on an encoding of error traces into so-called *error trace formulas*. An error trace formula is an unsatisfiable logical formula. A proof of unsatisfiability of the error trace formula captures the reason why an execution of the error trace fails. By applying an automated theorem prover to obtain such proofs of unsatisfiability, the relevant statements for the error can be automatically identified. One advantage of this approach is that the proof of

* Supported in part by the projects SAFEHR and COLAB, funded by Macau Science and Technology Development Fund.

unsatisfiability provides additional valuable information that can help the programmer understand the error, such as information about the program states at different points of execution.

A key limitation of existing proof-based fault localization techniques is that they only consider statements in the explanation of an error that have side-effects on the program's data. Information about the control flow of the program is ignored. This makes it difficult for the programmer to understand why the statements that are reported as relevant are actually reachable. Existing proof-based techniques therefore fail to explain common classes of program errors, such as faulty branching conditions.

In this paper, we propose a *flow-sensitive* proof-based fault localization technique. The result of a flow-sensitive fault localization not only explains why the error occurred, but also justifies why the statements leading to the error were executed. We give a basic algorithm for flow-sensitive fault localization based on our previous work on error invariants [6]. This algorithm applies fault localization recursively to explain the truth values of branching conditions along the error trace. We then observe that already the non-recursive algorithm from [6] can produce a flow-sensitive error explanation if one uses a simple modification in the encoding of the error trace formula. We refer to this new encoding of an error trace as the *flow-sensitive error trace formula.*

We discuss a number of examples that demonstrate the usefulness of flow-sensitive fault localization. We compare the results to the original fault localization based on error invariants and show that the new technique precisely explains why an error occurs.

Related Work. BugAssist [10, 11] minimizes a given error trace obtained from bounded model checking using a Max-SAT algorithm. Similar to our previous work [6], BugAssist encodes the error in an unsatisfiable formula. The fault is localized by identifying fragments of the error trace that are not needed to prove unsatisfiability of the trace formula. This results in the limitation that only executable statements can be part of the minimized error trace. Branch conditions, and those statements that explain why they hold, are always omitted as the violation of the assertion causing the error can be proven even without considering them. The main contribution of our work is a new way to encode error traces as formulas such that branch conditions, and statements affecting them, can appear in the result of the localization.

A common approach to fault localization is to compare failing with successful executions (e.g., [1, 7, 8, 14, 15, 17, 19, 20]). These approaches differ in the way the failing and successful executions are obtained, the way they compare executions, and in the information they report to the user. A detailed survey about the differences among these approaches can be found in [16, 18]. The main difference to our approach is that we do not execute the program and that we do not need a successful execution.

In [17], an approach is presented that compares similar failing and successful executions of a program and returns a bug report that contains only those branch conditions that have been evaluated differently in both executions. From a given

```
1 void xZero(int input) {
2   int x = 1;
3   int y = input - 42;
4   if (y<0) {
5     x = 0;
6   }
7   //@ assert x != 0;
8 }
```

Fig. 1. The error occurs only if the `then` block is taken. The condition and the derivation of its truth value are important to reproduce the error.

```
1 int yZero(int input) {
2   int x = 0;
3   int y = input - 42;
4   if (y<0) {
5     y = 0;
6   }
7   //@ assert x != 0;
8 }
```

Fig. 2. The error occurs no matter which branch of the conditional is taken.

failing execution of a program, they automatically construct a *similar* successful execution using a constraint solver. The evaluation of the branch conditions in both executions is recorded during execution, and the difference is reported to the user as a bug report. With this approach, we share the motivation that branch conditions not only provide valuable information for debugging but are often the reason for errors in a program and therefore are essential for fault localization. However, unlike the approach in [17], our approach reports not only branch conditions but also relevant statements, error invariants, and the variables that need to be tracked.

Similar to dynamic approaches there are static approaches that do fault localization by comparing feasible traces in the model of the program with counterexamples produced by a verifier. Ball et al. [1] present an algorithm to localize faults in counterexamples produced by the model checker SLAM. They isolate parts of a counterexample that do not occur on feasible traces. Groce et al. [7–9] use causal dependencies (see, e.g. [2]) and distance metrics for program executions to find minimal abstractions of error traces. In contrast to our approach, they use causal dependencies between variables and the difference between the failing and the successful execution to generate error reports.

2 Overview and Illustrative Example

We illustrate the benefits of flow-sensitive trace formulas on two simplified examples. Fig. 1 shows a procedure whose execution violates the assertion at line 7 if it is called with a value less than 42 for the parameter `input`. In this case, the assignment in line 5 is executed and the assigned value of `x` conflicts the assertion at line 7. An error is observable by executing an *error trace* starting from a state that satisfies an *error precondition*. In our example, the error trace is the sequence of statements obtained by restricting the program to the `then` branch of the conditional, i.e., including the statement in line 5. The error precondition is the formula `input < 42` (or any other formula that implies `input < 42`).

The problem of fault localization is to identify the statements in the error trace that are relevant for the error. Intuitively, a statement is relevant if the error no longer occurs after removing the statement from the trace. Various notions of relevancy have been considered in the literature depending on what it means to remove a statement from a trace.

In proof-based fault localization techniques [6, 11] relevancy of statements is determined by encoding the error into an unsatisfiable formula called the *extended trace formula*. This formula consists of a *trace formula* of the error trace in conjunction with the error precondition and the correctness assertion. A proof of unsatisfiability of the extended trace formula provides information about which statements are relevant. For our example, the extended trace formula is as follows:

$$(\mathit{input} < 42) \wedge (x = 1) \wedge (y = \mathit{input} - 42) \wedge (y < 0) \wedge \underline{(x' = 0)} \wedge \underline{(x' \neq 0)}.$$

The first conjunct is the error precondition, the last conjunct is the failing assertion, and the remaining conjuncts constitute the trace formula encoding of the error trace, e.g., the conjunct $(x' = 0)$ results from the statement in line 5. The conjuncts that are needed to prove the unsatisfiability of the formula are underlined, i.e., already $(x' = 0) \wedge (x' \neq 0)$ is contradictory. Thus, it appears as if only the statement in line 5 is relevant for the error. However, in order to reproduce the error we also need to know that `y<0` holds in line 4, otherwise the statement in line 5 will not be executed. Hence, we need the statement `y=input-42` and the precondition `input<42` to show that the condition in line 4 is true. Unfortunately, we cannot derive this information from the unsatisfiability proof because the values of `y` and `input` are irrelevant for the proof.

To overcome this problem, we introduce an alternative encoding of errors into so-called *flow-sensitive trace formulas*. A flow-sensitive trace formula keeps track of dependencies between statements and the branching conditions that are relevant for the reachability of these statements in the control flow graph of the program. A proof of unsatisfiability of such a formula includes a justification for the truth value of every branching condition on which a relevant statement depends. The flow-sensitive trace formula for the example in Fig. 1 is as follows:

$$\underline{(\mathit{input} < 42)} \wedge (x = 1) \wedge \underline{(y = \mathit{input} - 42)} \wedge \underline{(y < 0 \implies x' = 0)} \wedge \underline{(x' \neq 0)}$$

For a proof of unsatisfiability we still need the information that $x' = 0$ holds, but this information is now encoded in an implication $(y < 0 \implies x' = 0)$. The premise of the implication, $(y < 0)$, is the branching condition at line 4 that needs to be satisfied to reach the statement in line 5. The implication encodes that either the branching condition holds and the statement in line 5 is executed, or the branching condition does not hold and x' takes an arbitrary value (effectively over-approximating the `else` branch of the conditional). That is, either the implication $(y < 0 \implies x' = 0)$ is relevant for the proof or we can show that the value of x' is completely irrelevant. If the implication is relevant, then the branching condition $y < 0$ must also be relevant and so must be all the statements that affect its truth value. Hence, a statement that affects the reachability of a relevant statement is also considered relevant. In this example, the unsatisfiability of the flow-sensitive trace formula can only be proven if we can show that the implication $(\mathit{input} < 42) \wedge (x = 1) \wedge (y = \mathit{input} - 42) \implies (y < 0)$ is valid. The conjunct $(y = \mathit{input} - 42)$ resulting from line 3, and the precondition $(\mathit{input} < 42)$, which are both part of the premise, are sufficient for this. Hence, using the flow-sensitive trace formula, we can still identify the conflict between line 5 and line 7, but in addition, we can also explain why line 5 is reached. In the end, only the statement in line 2 is considered irrelevant.

Flow-sensitive trace formulas also distinguish between conditional choices that are relevant to reach the error and those that are irrelevant. We illustrate this using the procedure `yZero` shown in Figure 2. The flow-sensitive trace formula of `yZero` is either

$$(\mathit{input} < 42) \wedge \underline{(x = 0)} \wedge (y = \mathit{input} - 42) \wedge (y < 0 \implies y' = 0) \wedge \underline{(x \neq 0)}$$

if the trace to the error goes through the if-statement at line 4, or

$$(\mathit{input} < 42) \wedge \underline{(x = 0)} \wedge (y = \mathit{input} - 42) \wedge (y \geq 0 \implies y' = y) \wedge \underline{(x \neq 0)}$$

if it does not. Note that both traces are error traces, as the correctness assertion $x \neq 0$ in line 7 is violated in each case. We reuse the error precondition $(\mathit{input} < 42)$ from the previous example, though we might as well use any other constraint on `input` (as long as it is satisfiable). To prove the formula unsatisfiable, it is sufficient to prove the contradiction between the conjunct $(x = 0)$, resulting from line 2, and the assertion $(x' \neq 0)$. The conjunct $(y < 0 \implies y' = 0)$, respectively $(y \geq 0 \implies y' = y.)$, is not needed. We thus conclude that the conditional choice in line 4 is irrelevant to reproduce the error. Similarly, we observe that the constraints resulting from the statement `y=input-42` and the precondition `input<42` are not needed. Hence, we further conclude that neither the value of `input`, nor the value of `y` are relevant to reproduce the error.

3 Preliminaries

We use first-order logic formulas to define programs. We assume standard syntax and semantics of such formulas and use $\top$ and $\bot$ to denote the Boolean constants for *true* and *false*, respectively. For a set of variables X, we denote by $\mathsf{Expr}(X)$ the set of terms built from variables in X and we denote by $\mathsf{Preds}(X)$ the set of all quantifier-free formulas with free variables in X.

Programs and Statements. Let X be a fixed set of variables, which we call *program variables*, and let $\mathcal{L}$ be a set of *labels*. A *program statement st* is either a conditional choice, a while loop, a sequence of statements, an assignment, or a label:

$$\begin{aligned} &x \in X, \quad \ell \in \mathcal{L}, \quad e \in \mathsf{Expr}(X), \quad \mathit{cond} \in \mathsf{Preds}(X) \\ &\mathit{st} ::= \texttt{if } \mathit{cond} \texttt{ then } \mathit{st} \texttt{ else } \mathit{st} \mid \texttt{while } \mathit{cond} \texttt{ do } \mathit{st} \\ &\qquad \mid \mathit{st}; \mathit{st} \mid x := e \mid \texttt{label } \ell \end{aligned}$$

Labels have no operational meaning. They are only used to uniquely identify certain points during the execution of a program statement. We require therefore that each label ℓ occurs at most once in a statement. We use the short-hand notation $\ell : st$ for the program statement $\texttt{label}\ \ell; st$.

A program $\mathcal{P} = \langle st, \Phi \rangle$ consists of a program statement st, and an *assertion map* $\Phi : \mathcal{L} \to \mathsf{Preds}(X)$ which maps each label ℓ in st to a formula that should hold at the point of execution of st determined by ℓ.

Atomic Statements and Traces. We define the semantics of program statements and programs in terms of *atomic statements* (or simply statements), which are defined by the following grammar:

$$st^a ::= \texttt{if}\ cond\ \mid \texttt{endif}\ \mid x := e\ \mid \texttt{label}\ \ell\ \mid \texttt{havoc}\ cond$$

A *trace* π is a finite sequence of atomic statements. Let π be a trace of length n. For $0 \leq i < n$, we denote by $\pi[i]$ the i-th atomic statement of π, and for $0 \leq i < j < n$, we denote by $\pi[i,j]$ the sub-trace $\pi[i]; \ldots ; \pi[j-1]$ of π. Traces obtained from programs do not contain $\texttt{havoc}\ cond$ statements. We will use such statements later to define abstract traces.

A program statement st defines a possibly infinite, prefix-closed set of traces $Traces(st)$. The set $Traces(st)$ is obtained by unwinding the loops in st arbitrarily often. Formally, we define the set of *complete traces* $CTraces(st)$ of a statement st as the least fixed point of the following system of equations:

$$\begin{array}{l}
CTraces(x := e) = \{x := e\}\\
CTraces(\texttt{label}\ \ell) = \{\texttt{label}\ \ell\}\\
CTraces(st_1\ ;\ st_2) = \{\, \pi_1; \pi_2 \mid \pi_1 \in CTraces(st_1), \pi_2 \in CTraces(st_2)\,\}\\
CTraces(\texttt{if}\ cond\ \texttt{then}\ st_1\ \texttt{else}\ st_2) =\\
\quad \{\, \texttt{if}\ cond; \pi; \texttt{endif} \mid \pi \in CTraces(st_1)\,\} \cup\\
\quad \{\, \texttt{if}\ \neg cond; \pi; \texttt{endif} \mid \pi \in CTraces(st_2)\,\}\\
CTraces(\texttt{while}\ cond\ \texttt{do}\ st) =\\
\quad \{\texttt{if}\ \neg cond; \texttt{endif}\} \cup\\
\quad \{\, \texttt{if}\ cond; \pi; \texttt{endif}; \pi' \mid \pi \in CTraces(st), \pi' \in CTraces(\texttt{while}\ cond\ \texttt{do}\ st)\,\}
\end{array}$$

The set of traces $Traces(st)$ of a program statement st is then defined as the set of all prefixes of its complete traces $CTraces(st)$. The traces of a program $\mathcal{P} = \langle st, \Phi \rangle$ are the traces of its program statement, i.e., $Traces(\mathcal{P}) = Traces(st)$.

The purpose of using the above notation for the syntax of programs and traces, instead of more common notations such as guarded commands and passive programs, is that it allows to identify the scope of a branching condition from the syntax. We might as well use a more common syntax but then we need to recompute this scope in a preprocessing step.

Semantics of Traces and Programs. A program *state* s is a function that assigns a value $s(x)$ to each program variable $x \in X$. We call the formulas $\mathsf{Preds}(X)$ state formulas and we write $s \models F$ to denote that a state s satisfies a state formula F.

For a variable $x \in X$ and $i \in \mathbb{N}$, we denote by $x^{\langle i \rangle}$ the variable obtained from x by adding i primes to it. The variable $x^{\langle i \rangle}$ 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 $y^{\langle i \rangle}$. We denote by $x^{\langle -i \rangle}$ the inverse operation of $x^{\langle i \rangle}$.

The formulas $\mathsf{Preds}(X \cup X')$ are called *transition formulas*. We use transition formulas to represent the semantics of statements in a trace, where the variables X' denote the values of the variables from X in the next state. We write $s, s' \models T$ to denote that the states s and s' satisfy the transition formula T, where s' is used to interpret the variables in X'. We associate with each atomic statement st^a a transition formula $T[st^a]$ as shown in Figure 3. Here, $\mathit{frame}(Y)$ denotes the formula $\bigwedge_{y \in Y} y = y'$. The atomic statement `havoc` *cond* assigns non-deterministic values to all variables in $\mathit{vars}(\mathit{cond})$ and has no effect on the values of the remaining variables. Note that these statements do not occur in traces from the program as our programs are deterministic.

st^a	$T[st^a]$
`if` cond	$\mathit{cond} \wedge \mathit{frame}(X)$
`endif`	$\mathit{frame}(X)$
`label` ℓ	$\mathit{frame}(X)$
$x := e$	$x' = e \wedge \mathit{frame}(X \setminus \{x\})$
`havoc` cond	$\mathit{cond}' \wedge \mathit{frame}(X \setminus \mathit{vars}(\mathit{cond}))$

Fig. 3. The transition formulas describing the semantics of the statements in traces.

An *execution* of a trace π of length n is a sequence of states $s_0 \dots s_n$ such that for all $0 \leq i < n$, $s_i, s_{i+1} \models T[\pi[i]]$. We denote by $\mathit{Execs}(\pi)$ the set of all executions of π. The *trace formula* of π is the formula

$$\mathbf{TF}(\pi) := T[\pi[0]]^{\langle 0 \rangle} \wedge \dots \wedge T[\pi[n-1]]^{\langle n-1 \rangle} \ .$$

The trace formula is satisfiable if and only if π has a (feasible) execution. That is, if $\mathit{Execs}(\pi)$ is non-empty. In fact, there is a one-to-one correspondence between the executions of π and the models of $\mathbf{TF}(\pi)$. We call a trace *feasible* if its trace formula is satisfiable, otherwise we call it *infeasible*.

A program $\mathcal{P} = (st, \Phi)$ is *safe* if for every trace $\pi \in \mathit{Traces}(\mathcal{P})$ whose final statement is a label statement `label` ℓ and every execution σ of π, the final state of σ satisfies $\Phi(\ell)$. If a program is not safe, an *error* can be witnessed along a trace. The error corresponds to a set of executions that violate the assertion associated with the last label of the trace.

Definition 1. *An* error *of a program $\mathcal{P} = \langle st, \Phi \rangle$ is a tuple (ψ, π, ϕ) where ψ and ϕ are state formulas and $\pi = st_0^a; \dots st_{n-2}^a;$ `label` ℓ is a trace of $\mathcal{P}$ with $\Phi(\ell) = \phi$ such that the following conditions hold:*

1. $\psi \wedge \mathbf{TF}(\pi)$ *is satisfiable, and*
2. $\psi \wedge \mathbf{TF}(\pi) \wedge \phi^{\langle n \rangle}$ *is unsatisfiable.*

That is, for an error (ψ, π, ϕ), no execution of the trace π that starts in a state satisfying ψ ends in a state satisfying the postcondition ϕ. However, there must exist at least one execution of π that starts in a state satisfying ψ. We refer to the unsatisfiable formula $\psi \wedge \mathbf{TF}(\pi) \wedge \phi^{\langle n \rangle}$ as the *extended trace formula* of the error.

4 Flow-sensitive Fault Localization

Proof-based fault localization techniques such as the ones described in [11] and [6] use the unsatisfiability proof of the extended trace formula to identify the parts of the trace formula that are relevant for the error. However, the extended trace formula only encodes that the trace does not have any execution for the given pre and postcondition. That is, to show that this formula is unsatisfiable, it might be sufficient to identify a single statement in the trace that establishes a contradiction with the postcondition. Though, to understand why the execution is possible at all, and thus to understand why the undesired post-state is reachable on this execution, more statements might be necessary. Therefore we propose *flow-sensitive fault localization* which localizes the fault and further explains for each statement in the result of the localization why the conditions needed to reach this statement are satisfied.

In this section, we show how existing fault localization algorithms can be made flow-sensitive while using the underlying algorithm as a black box. We do this for the fault localization technique based on *error invariants* [6], but the same principle also applies to other algorithms.

Error Invariants. Given an error (ψ, π, ϕ) with trace π of length n, let $0 \leq i \leq n$ be a position in the trace[4]. An error invariant for position i is a state formula I such that (i) $\psi \wedge \mathbf{TF}(\pi[0, i]) \models I^{\langle i \rangle}$ and (ii) $I \wedge \mathbf{TF}\pi[i, n] \wedge \phi^{\langle n-i \rangle} \models \bot$. That is, I over-approximates the final states of the executions $\mathit{Execs}(\pi[0, i])$ that start in a state satisfying ψ. Furthermore, the final state of any execution of $\pi[i, n]$ that starts in a state satisfying the error invariant still violates ϕ. An error invariant is *inductive* for positions $i \leq j$ if it is an error invariant for both positions i and j.

Fault Localization. We formulate fault localization as the problem of finding an *abstract error* for a given error that describes the essence of why the error occurred. The abstract error comprises an abstract error trace that over-approximates the executions of the original error trace and fails for the same reason. We define these abstract error traces using inductive error invariants. If an error invariant I is inductive for positions $j > i$, then the statements between positions i and j are not needed to reproduce the error. We can drop

[4] Position n is the position where the assertion ϕ is supposed to hold.

these statements from the error trace and replace them by a *summary transition* that non-deterministically changes the values of variables, but preserves the error invariant I. The statements in the error trace for which we cannot find an encompassing inductive invariant is considered relevant and remains in the abstract error trace.

Definition 2 (Abstract error trace). *Given an error* (ψ, π, ϕ) *with trace* π *of length* n. *Let* $\pi^{\#} = \textit{\texttt{havoc}}\ I_0; st^a_1; \textit{\texttt{havoc}}\ I_1; \ldots; st^a_k; \textit{\texttt{havoc}}\ I_k$ *be a trace where all* I_i, $0 \leq i \leq k$, *are state formulas. We call* $\pi^{\#}$ *an* abstract error trace *for* (ψ, π, ϕ) *if there exist positions* $i_0 < \ldots < i_{k+1}$ *such that* $i_0 = 0$, $i_{k+1} = n + 1$, *for all* j *with* $1 \leq j \leq k$, $st^a_j = \pi[i_j]$, *and for all* j *with* $0 \leq j \leq k$, I_j *is an inductive error invariant for positions* i_j *and* $i_{j+1} - 1$. *We call* $(\psi, \pi^{\#}, \phi)$ *an* abstract error *associated with error* (ψ, π, ϕ).

Example 3. Consider again the example program from Fig. 1. We get an error for $input = 41$ and the trace $x := 1$; $y := input - 42$; `if` $y < 0$; $x := 0$; `endif`; `label` ℓ where the condition assigned to label ℓ is $x \neq 0$. One possible abstract error trace is `havoc` $\top$; $x := 0$; `havoc` $x = 0$.

Craig Interpolants. There is a close connection between error invariants and Craig interpolants that we exploit to compute error invariants using the extended trace formula associated with the error.

Given two formulas A and B whose conjunction is unsatisfiable, a Craig interpolant for (A, B) - hereafter called interpolant - is a formula I such that (a) $A \implies I$ is valid, (b) $B \wedge I$ is unsatisfiable, and (c) the free variables in I occur free in both A and B. This concept has been extended to *inductive sequences of interpolants* [13]. Given n formulas $F_1, \ldots, F_n$ whose conjunction is unsatisfiable, an inductive sequences of interpolants is a sequence $I_0, \ldots, I_n$ of $n + 1$ formula such that (a) I_0 is $\top$, (b) I_n is $\bot$, (c) $I_{i-1} \wedge F_i \implies I_i$ is valid for $0 < i \leq n$, and (d) the free variables in I_i occur free in both, the formulas with index less than or equal to i, and the remaining formulas. Such inductive sequences of interpolants can be computed automatically from proofs of unsatisfiability using an appropriate interpolation procedure (see, e.g., [12]). We assume that such an interpolation procedure is given.

For an error (ψ, π, ϕ) with trace π of length n and a position $0 \leq i \leq n$, let $A = \psi \wedge \mathbf{TF}(\pi[0, i])$ and $B = \mathbf{TF}(\pi[i, n])^{\langle i \rangle} \wedge \phi^{\langle n \rangle}$. Then for every interpolant I of (A, B), the formula $I^{\langle -i \rangle}$ is an error invariant for position i of π.

Flow-sensitivity. An abstract error trace $\pi^{\#}$ for an error (ψ, π, ϕ) is an abstraction of the concrete error trace π. Since the abstraction only has to preserve the error, it might lose information about the control flow that is vital to reproduce the error in the original program. For instance, the abstract error trace in Example 3 does not provide any information about how line 5, the assignment $x := 0$, is reached. In particular, the abstract error trace does not incorporate any information about the variable y that is used in the condition of the **if**-statement (line 4) surrounding this assignment. In general, an error might not be caused directly by the execution of a specific statement, but by the reachability of that statement under the given error precondition. For short error traces it is often easy to see why a certain statement is reachable. However, for longer error traces this is non-trivial. Hence, automation in form of fault localization that precisely captures the relevant control flow is desirable.

We introduce *flow-sensitive* abstract error traces to solve this problem. A flow-sensitive abstract error trace $\pi^\#$ is an abstract error trace with the property that for every statement st^a in $\pi^\#$, a prefix of $\pi^\#$ can be used to explain why st^a is reachable in the original trace. To formalize this concept, we introduce two helper functions *Conds* and *Prev*. We denote by $\mathit{Conds}(\pi)$ the conditions needed to reach $\pi[n]$, i. e., the conditions of the **if**-statements whose corresponding `endif`-statements are not part of π:

$$\begin{aligned}
\mathit{Conds}(\pi) &= \mathit{Conds}(\emptyset, \pi, n)\\
\mathit{Conds}(S, \epsilon, 0) &= S\\
\mathit{Conds}(S, \pi\ ;\ \texttt{endif}, i) &= \mathit{Conds}(S, \pi[0, \mathit{Prev}(\pi)], \mathit{Prev}(\pi))\\
\mathit{Conds}(S, \pi\ ;\ \texttt{if}\ \mathit{cond}, i) &= \mathit{Conds}(S \cup \{\mathit{cond}\}, \pi, i-1)\\
\mathit{Conds}(S, \pi\ ;\ \texttt{label}\ \ell, i) &= \mathit{Conds}(S, \pi\ ;\ x := e, i)\\
&= \mathit{Conds}(S, \pi\ ;\ \texttt{havoc}\ \mathit{cond}, i)\\
&= \mathit{Conds}(S, \pi, i-1)
\end{aligned}$$

Here, ϵ denotes the empty trace. If $\mathit{Conds}(\pi)$ is used within a formula, it is interpreted as the conjunction of its elements. The helper function $\mathit{Prev}(\pi)$ computes for a trace of length n the position of the last **if**-statement that does not have a corresponding `endif`-statement in π:

$$\begin{aligned}
\mathit{Prev}(\pi) &= \mathit{Prev}(1, \pi, n)\\
\mathit{Prev}(0, \pi, i) &= i+1\\
\mathit{Prev}(k, \pi\ ;\ \texttt{endif}, i) &= \mathit{Prev}(k+1, \pi, i-1)\\
\mathit{Prev}(k, \pi\ ;\ \texttt{if}\ \mathit{cond}, i) &= \mathit{Prev}(k-1, \pi, i-1)\\
\mathit{Prev}(k, \pi\ ;\ \texttt{label}\ \ell, i) &= \mathit{Prev}(k, \pi\ ;\ \texttt{havoc}\ \mathit{cond}, i)\\
&= \mathit{Prev}(k, \pi\ ;\ x := e, i)\\
&= \mathit{Prev}(k, \pi, i-1)
\end{aligned}$$

Definition 4 (Flow-sensitive). *Let P be a program. An abstract error $(\psi, \pi^\#, \phi)$ for an error (ψ, π, ϕ) of P is called* flow-sensitive *if for every statement st^a in $\pi^\#$ with $st^a = \pi[i] = \pi^\#[k]$, some prefix of $\pi^\#[0, k]$ is an abstract error trace for the error $(\psi, \pi[0, i], \neg\mathit{Conds}(\pi[0, i]))$ of the program that is obtained from P by inserting*[5] *the statement* ***label*** *ℓ before the statement $\pi[i]$ and mapping the fresh label ℓ to the assertion $\neg\mathit{Conds}(\pi[0, i])$.*

[5] Note that the insertion of labels into a program does not change the semantics of the program.

To make fault localization flow-sensitive, we need to know the scope of the statement, i.e., the conditions that have to hold for the statement to be reachable. With this knowledge, we can make any fault localization technique flow-sensitive by using recursive calls to find out why the conditions have to hold when reaching statements in conditional branches. As an example, we show in Algorithm 1 how to make the fault localization based on error invariants flow-sensitive. In general, this algorithm can easily be adapted to other fault localization techniques.

Algorithm 1: FSEI(ψ, π, ϕ): naive algorithm to compute error invariants for flow-sensitive fault localization.

```
Input: Pre-condition $\psi$, Trace $\pi$ of length $n$, and post-condition $\phi$
Output: Sequence of $n+1$ error invariants
$E_0, \dots, E_n \leftarrow ErrorInvariants(\psi, \pi, \phi)$   (1)
$Pos \leftarrow Changes(E_0, \dots, E_n)$   (2)
$Conditions \leftarrow \bigcup_{i \in Pos} Scope(i)$   (3)
foreach $(i, cond)$ in $Conditions$ do
    $S_0, \dots, S_i \leftarrow \mathbf{FSEI}(\psi, \pi[0,i], \neg cond)$   (4)
    foreach $0 \le j \le i$ do $E_j \leftarrow E_j \wedge S_j$   (5)
return $E_0, \dots, E_n$
```

In the line marked by (1) it uses an algorithm for fault localization based on error invariants [6] as a black box. This algorithm is supposed to return an error invariant for every position in the trace and one for the post-condition. Hence, for an input trace of length n the result consists of $n+1$ error invariants. The flow-sensitive localization then extracts, in (2), the positions in the sequence of error invariants where the invariant changes. These positions index the relevant statements in the trace. At this point, we get the result of the fault localization without considering flow-sensitivity. Hence, (1) and (2) could be substituted by any fault localization algorithm that computes a set of relevant statements for the error. From the list generated in (2) the algorithm extracts, in (3), the scope of every statement as a set of pairs of positions and conditions. Let j be the position of a statement in a trace π, then $Scope(j)$ returns the set of all pairs $(i, cond)$ such that (1) $i < j$, (2) $\pi[i]$ is `if` $cond$, and (3) the corresponding `endif`-statement is not in $\pi[0, j]$.

When this information is available, we can reduce flow-sensitive fault localization to a simpler subproblem. For every pair $(i, cond)$ we insert[5] a fresh label ℓ_i before the statement $\pi[i]$ and map the label to $\neg cond$. Then, in line (4), we recursively call the flow-sensitive fault localization for the error $(\psi, \pi[0,i], \neg cond)$ in the modified program. Essentially we ask the question "Why does $cond$ hold after executing $\pi[0,i]$ from states in ψ?"

To understand why this procedure works, we first note that the condition $cond$ has to hold for all execution of $\pi[0,i]$ that start in ψ since our programs are deterministic. Hence, the tuple $(\psi, \pi[0,i], \neg cond)$ is an error in the modified program.

In our algorithm, the recursive calls return error invariants that explain why every execution of the prefix up to the condition that starts in ψ must satisfy the condition. The results of the recursive calls have to be combined with the result of the initial call to the error invariant generation. We are only allowed to strengthen the error invariants. Otherwise we might introduce executions that do not violate the postcondition in which case the result would not be a flow-sensitive abstract error trace. Thus, we conjoin the invariants derived by the recursive call with the current invariants.

A binary search algorithm similar to the one presented in [6] can be used to find for each error invariant a maximal interval of positions for which this error invariant is inductive. We denote by **Localize** the procedure that takes as input a sequence of error invariants and an error (ψ, π, ϕ), and builds an abstract error $\textbf{Localize}(\langle E_0, \ldots, E_n \rangle, (\psi, \pi, \phi))$, by replacing each subsequence $\pi[i, j]$ of π by `havoc` E_k, if E_k is an inductive invariant for positions $i < j$, for some $0 \leq k \leq n$.

Theorem 5. *Let (ψ, π, ϕ) be an error and $E_0, \ldots, E_n = \textbf{FSEI}(\psi, \pi, \phi)$ a sequence of error invariants. Then, $\textbf{Localize}(\langle E_0, \ldots, E_n \rangle, (\psi, \pi, \phi))$ is a flow-sensitive abstract error.*

A downside of Algorithm 1 is that it might need one recursive call to fault localization per **if**-statement in the trace, which results in a quadratic worst case complexity. This will be inefficient for long traces. We therefore propose a new encoding of an error into a *flow-sensitive trace formula*. The benefit of this new encoding is that it yields a more efficient non-recursive flow-sensitive fault localization algorithm.

5 Flow-sensitive Trace Formulas

We denote the flow-sensitive trace formula of a trace π by $\textbf{FSTF}(\pi)$. The idea behind flow-sensitive trace formulas is that, in addition to encoding the executions of the trace π, they also encode an over-approximation of the executions of all other traces that only differ from π in conditional statements. That is, if we need a statement that is only reachable under certain conditions to prove $(\psi \wedge \textbf{FSTF}(\pi) \wedge \phi^{\langle n \rangle})$ unsatisfiable, we also prove that every execution starting in a state in ψ inevitably has to reach this condition.

Algorithm 2 shows how the flow-sensitive trace formula $\textbf{FSTF}(\pi)$ is computed for a given trace π. Similar to a trace formula, we compute a conjunction of (appropriately shifted) transition formulas for each statement of π. However, instead of adding conjuncts for branch conditions `if` *cond*, we memorize the branch conditions that have to hold to reach a particular position on the trace (see ℓ_1 and ℓ_3), and for every statement other than `if` *cond* and `endif`, we add a chain of implications for these branch conditions to the transition formula of this statement (see ℓ_5). Hence, whenever we want to show that a statement is needed to prove that the trace formula contradicts with pre and postcondition, we also have to prove that all conditions needed to reach this statement can be

Algorithm 2: FSTF(π): flow-sensitive trace formula.

```
    Input: Trace $\pi$ of length $n$
    Output: flow-sensitive trace formula $ret$
       Stack $conds$;
       Formula $ret = true$;
       for $(i = 0$ to $n - 1)\{$
         if $(\pi[i]$ is (if $cond$)) {
$\ell_1$       $conds$.push($cond^{\langle i\rangle}$);
$\ell_2$       $ret = ret \wedge frame(X)^{\langle i\rangle}$;                //stub for TF($\pi[i]$)
         } else if $(\pi[i]$ is (endif)) {
$\ell_3$       $conds$.pop();
$\ell_4$       $ret = ret \wedge frame(X)^{\langle i\rangle}$;                //stub for TF($\pi[i]$)
         } else {
$\ell_5$       $ret = ret \wedge \big((\bigwedge_{c \in conds} c) \implies (T[\pi[i]]^{\langle i\rangle})\big)$;
         }
       }
```

Algorithm 3: $FSErrInvs(\psi, \pi, \phi)$: algorithm to compute a flow-sensitive error invariants.

```
    Input: Precondition $\psi$, Trace $\pi$ of length $n$, and postcondition $\phi$
    Output: Sequence of $n + 1$ error invariants
    $I_0, \ldots, I_n \leftarrow InductiveInterpolants(\psi \wedge \mathbf{FSTF}(\pi) \wedge \phi)$   (1)
    return $I_0^{\langle -0\rangle}, \ldots, I_n^{\langle -n\rangle}$
```

satisfied. This guarantees that our fault localization will consider these branch conditions and all the statements affecting their truth values.

Algorithm 3 is our non-recursive flow-sensitive fault localization algorithm that computes the error invariants directly via Craig interpolation of the flow-sensitive trace formula. The resulting abstract error is still guaranteed to be flow-sensitive:

Theorem 6. *Let (ψ, π, ϕ) be an error with trace π of length n and $E_0, \ldots, E_n = FSErrInvs(\psi, \pi, \phi)$ be a sequence of error invariants computed by Algorithm 3. Then,* **Localize**$(\langle E_0, \ldots, E_n\rangle, (\psi, \pi, \phi))$ *is a flow-sensitive abstract error.*

Proof (sketch). Let the result of **Localize**$(\langle E_0, \ldots, E_n\rangle, (\psi, \pi, \phi))$ be $(\psi, \pi^{\#}, \phi)$ with $\pi^{\#} \equiv (\texttt{havoc } E_0, st_1^a; \texttt{havoc } E_1; \ldots; \texttt{havoc } E_k)$ where no st_i^a is a havoc statement. Since $\pi^{\#}$ is an abstract error trace by construction, we only have to show that it is also flow-sensitive. Note that **Localize** guarantees that for all $0 \leq i \leq k$ and $0 \leq j \leq k$, if $i \neq j$ then E_i is not an error invariant that can replace E_j.

Let j be a position in the abstract error trace with $0 < j \leq k$. Furthermore, let i_j denote the position of st_j^a in the concrete error trace π. If $Conds(\pi[0, i_j]) = \emptyset$, i. e., the statement does not occur under an **if**-statement, flow-sensitivity holds trivially for this statement. Otherwise, we denote by ξ the formula corresponding to $\mathit{Conds}(\pi[0,i_j])$. Note that $\mathbf{FSTF}(\pi[i_j]) \equiv \xi \rightarrow \mathbf{TF}(\pi[i_j])$.

We know that E_j is an error invariant at position i_j and E_{j-1} is an error invariant at position $i_j - 1$. Furthermore, we know that the following equivalences hold:

$$\begin{aligned} \psi \wedge \mathbf{FSTF}(\pi[0,i_j]) &\equiv \psi \wedge \mathbf{FSTF}(\pi[0,i_j-1]) \wedge \mathbf{FSTF}(\pi[i_j])^{\langle i_j - 1\rangle} \\ &\equiv \psi \wedge \mathbf{FSTF}(\pi[0,i_j-1]) \wedge \xi \rightarrow \mathbf{TF}(\pi[i_j])^{\langle i_j - 1\rangle} \end{aligned}$$

If ξ does not hold in E_{j-1} it is not relevant for the error. Hence, the statement st^a_j cannot appear in the abstract error trace. This contradicts the property guaranteed by **Localize** considering E_j and E_{j-1}. Hence, a prefix of the abstract error trace up to the error invariant E_{j-1} has to ensure that ξ has to hold.

$$\begin{array}{l l} \mathit{input} < 42 & \\ \hline & \mathit{input} < 42 \\ x' = 1 & \\ \hline & \mathit{input} < 42 \\ y' = \mathit{input} - 42 & \\ \hline & y' < 0 \\ y' < 0 \implies x'' = 0 & \\ \hline & x'' = 0 \\ x'' \neq 0 & \end{array}$$

(a) using flow-sensitive trace formula

$$\begin{array}{l l} \mathit{input} < 42 & \\ \hline & \top \\ x' = 1 & \\ \hline & \top \\ y' = \mathit{input} - 42 & \\ \hline & \top \\ y' < 0 \wedge x'' = 0 & \\ \hline & x'' = 0 \\ x'' \neq 0 & \end{array}$$

(b) using trace formula

Fig. 4. Interpolant derivation for the program from Figure 1

Example 7. Figure 4(a) shows the derived interpolants for the flow-sensitive trace formula of the error in our motivating example given in Fig. 1. Every conjunct of the trace formula is written on a single line. The first formula is the precondition of the error, the last formula is the postcondition, and the remaining formulas are the flow-sensitive trace formulas for the statements in the trace. The horizontal lines separating the different formula parts correspond to the positions where an interpolant is computed. Adjacent to each line, we annotate the computed interpolant. Note that each interpolant is unsatisfiable in conjunction with the formulas below the adjacent line[6]. Consider the last interpolant $x'' = 0$. This interpolant suffices to show that the postcondition is violated. The flow-sensitive encoding however forces the prover to justify why the condition $y' < 0$ holds. This justification is given in the preceding interpolants.

Figure 4(b) shows the derivation of interpolants for the same error encoded with the original extended trace formula. In this encoding, the fact $x'' = 0$ holds unconditionally. Therefore, only the last interpolant is non-trivial because no justification has to be given why the assignment statement $x = 0$ is actually reachable.

[6] The interpolants actually form an inductive sequence. We omit the initial $\top$ and final $\bot$ interpolant of this sequence.

```
1  void process() {
2    int x, y, z;
3    z = 0;
4    lock = 1;
5    if (x == 0) {
6      if (y == 0)
7        z = 1;
8    }
9    if (y != 0) {
10     z = y;
11   }
12   if (x != 0) {
13     z = 2;
14     lock = 0;
15   } else if (z > 0) {
16     z = 3;
17     lock = 0;
18   }
19   //@ assert lock == 0;
20 }
```

Fig. 5. Faulty locking with a simplified locking mechanism

6 Evaluation

For our prototype implementation we use the software model checker Kojak which is based on Ultimate [5] and the interpolating theorem prover SMT-Interpol [4]. Kojak implements the fault localization algorithm based on error invariants [6]. We modified the implementation to perform flow-sensitive fault localization using the flow-sensitive trace formula encoding.

In the following, we demonstrate the benefits of flow-sensitive trace formulas in fault localization on two examples taken from the literature to which we have applied our implementation. We prefix statements in abstract error traces with their corresponding line numbers in the original program. For frame conditions we use the line number of the associated conditional choice and add the subscript *fc*. For the sake of presentation we limit ourselves to small examples. Compared to previous approaches, our new method results in longer abstract error traces since it now provides additional information about the program's control flow.

Faulty Locking. The first example is taken from [3] and shown in Figure 5. The program uses a locking mechanism to protect the access to variables. For the purpose of demonstration we simplified the lock/unlock steps by introducing a simple Boolean variable *lock*. The program sets $lock = 1$, then starts operations on the variables x, y, z, and finally checks if *lock* has been set to zero at the end of the procedure using an assertion in line 19. We use the model checker Kojak to check the safety of this assertion. The model checker finds a counterexample trace π and provides an initial failing state $\psi \equiv x = 0 \wedge y = -1$. Conjoining the obtained information yields the extended trace formula

$$\begin{aligned} (x = 0 \wedge y = -1) \wedge (z' = 0) \wedge \underline{(lock' = 1)} \wedge \\ ((x = 0) \wedge (y \neq 0) \wedge (z'' = z')) \wedge ((y \neq 0) \wedge (z''' = y)) \wedge \\ ((x = 0) \wedge (z''' \leq 0) \wedge \underline{(lock'' = lock')}) \wedge \underline{(lock'' = 0)} \enspace . \end{aligned}$$

We can see that only the variable *lock* is relevant to prove this formula unsatisfiable. Hence, the vanilla algorithm from [6] provides us only two inductive error invariants, $\top$ and $lock = 1$. The corresponding abstract error trace is shown on the right side of Figure 6. The abstract error trace explains the direct cause of the error, but it does not show why the error is reachable.

$\texttt{havoc}\ x = 0 \wedge y = -1$	$\texttt{havoc}\ \top$
$4: lock := 1;$	$4: lock := 1;$
$\texttt{havoc}\ x = 0 \wedge y = -1 \wedge lock = 1$	$\texttt{havoc}\ lock = 1$
$10: z := y;$	
$\texttt{havoc}\ x = 0 \wedge lock = 1 \wedge z = -1$	
$18_{fc}: lock := lock;$	
$\texttt{havoc}\ lock = 1$	

Fig. 6. Abstract error traces of the example in Fig. 5 with and without flow-sensitive encoding of the error.

Using flow-sensitive trace formulas for this error we get a different sequence of interpolants. Again, we show the formula one conjunct above the other with the interpolants written adjacent to the separating line:

$x = 0 \wedge y = -1$	
	$x = 0 \wedge y = -1$
$z' = 0$	
	$x = 0 \wedge y = -1$
$lock' = 1$	
	$x = 0 \wedge y = -1 \wedge lock' = 1$
$(x = 0) \Rightarrow (y \neq 0) \Rightarrow (z'' = z')$	
	$x = 0 \wedge y = -1 \wedge lock' = 1$
$(y \neq 0) \Rightarrow (z''' = y)$	
	$x = 0 \wedge lock' = 1 \wedge z''' = -1$
$(x = 0) \Rightarrow (z''' \leq 0) \Rightarrow (lock'' = lock')$	
	$lock'' = 1$
$lock'' = 0$	

From this sequence of interpolants we obtain the flow-sensitive abstract error trace shown in Figure 6. The first error invariant only represents the error precondition. Note that, in general, this invariant might be more general than the concrete values given by the model checker for the error precondition.

The second error invariant summarizes the control flow from line 5 to line 9 and the effect of the program up to line 5 that is relevant for the error. In particular, this invariant describes that the then-branch of the **if**-statement in line 5, the else-branch of the **if**-statement in line 6, and the then-branch of the **if**-statement in line 9 are taken. Furthermore, no assignment statement and no frame for the code between lines 5 to 9 affects the occurrence of the error.

The statement in line 10, however, has such an effect which can be seen in the next error invariant. The value of y becomes irrelevant for the remainder of the trace, but the value of the variable z now becomes relevant. From this error invariant it is easy to see that both conditions in lines 12 and 15 are not satisfied. Hence, the **if**-statement is skipped and we have to insert appropriate frame conditions into our SSA-encoded error trace. This frame condition actually changes the error invariant. From this change we can conclude that the case distinction in lines 12 to 18 is incomplete. Note that this change is only enforced by the symbol condition for interpolants. To enforce this change in general, we introduce fresh auxiliary variables for the conditions needed to execute the statement. These variables are always local to exactly one interpolant and, hence, cannot be shifted. To improve readability we omitted these variables as they are not needed for the example.

```
1  int absValue(int input) {
2    int sign,abs;
3    abs = input;
4    if (input == 0)
5      return 0;
6    if (input<0) {
7      sign = -1;
8      printf("negative");
9    } else {
10     sign = 1;
11     printf("positive");
12   }
13   if(sign == -1) {
14     sign = input*-1;
15   } else {
16     abs = input;
17   }
18   //@ assert abs >= 0;
19   return abs;
20 }
```

Fig. 7. Example code of a faulty program that computes the absolute value and sign of the variable `input`.

Faulty Absolute Value. The second example program is shown in Fig. 7. It computes the absolute value of an input variable *input*. The procedure `absValue` takes a variable `input` as input. If this variable is 0, the procedure returns without further computations (line 4-5). Otherwise the procedure sets `sign` to -1 if `input` is negative and to 1 if not (line 6-12) and prints a corresponding message to the console. Then, it computes the absolute value of `input` and writes it to `abs` (line 3-17). However, there is an error in the computation of the absolute value in line 14. The absolute value is written to `sign` instead of `abs`, which causes `abs` to have the original (negative) value of `input` that was assigned to it in line 3. This violates the assertion in line 18 which expects `abs` to be greater or equal to zero. The challenge here is that the error occurs because the variable `abs` is *not* modified in line 14. The above error is detected by KOJAK and the error is witnessed using the example input `input=-1`. For this input, we can then extract an error trace π of the procedure `absValue`. The right column of Fig. 8 shows the abstract error trace obtained by analyzing π using the approach from [6] with non flow-sensitive encoding of trace formulas. To reproduce the failing execution it is sufficient to know that `input` is initially -1, and that in line 3 `abs` is set to the value of `input`. Since `abs` is not changed after that, these statements, together with the postcondition constitute an execution that fails the same way as the original error trace. However, what we are actually

$\boxed{\texttt{havoc}\ input \leq -1}$

3 :$abs := input;$

$\boxed{\texttt{havoc}\ input \leq -1 \wedge abs \leq -1}$

7 :$sign := -1;$

$\boxed{\texttt{havoc}\ abs \leq -1 \wedge sign = -1}$

15_{fc} :$abs := abs;$

$\boxed{\texttt{havoc}\ abs \leq -1}$

$\boxed{\texttt{havoc}\ input \leq -1}$

3 :$abs := input;$

$\boxed{\texttt{havoc}\ abs \leq -1}$

Fig. 8. Abstract error traces of the example in Fig. 7.

interested in is to see that the problem occurs because `sign` is modified instead of `abs` in line 14. This information is missing in the abstract error trace.

We now consider the abstract error trace produced by the flow-sensitive algorithm. Again, we show the flow-sensitive trace formula and the sequence of interpolants computed for it:

$$\begin{array}{l}
input = -1 \\ \hline
abs' = input \\ \hline
(input \neq 0) \Rightarrow \ldots \\ \hline
(input < 0) \Rightarrow (sign' = -1) \\ \hline
(sign' = -1) \Rightarrow (sign'' = input * -1) \\ \hline
(sign' = -1) \Rightarrow (abs'' = abs') \\ \hline
abs'' \geq 0
\end{array}
\qquad
\begin{array}{l}
input \leq -1 \\
input \leq -1 \wedge abs' \leq -1 \\
input =\leq 1 \wedge abs' \leq -1 \\
abs' \leq -1 \wedge sign' = -1 \\
abs' \leq -1 \wedge sign' = -1 \\
abs'' \leq -1
\end{array}$$

From this sequence of interpolants we obtain the abstract error trace shown in Fig. 8.

The first error invariant in the abstract error trace is a generalization of the error precondition produced by Kojak. It shows that the error also occurs if the value of `input` is any value less than or equal to −1. The next error invariant incorporates the effect of the assignment in line 3. Furthermore, it states that the **if**-statement in line 4 can be ignored when analyzing the error and that the then-branch of the **if**-statement in line 6 is important. The statement at line 7 which is contained in the then-branch is the next statement to change the error invariants. From the next error invariant we realize that, from line 7 to the end of the program, the value of the variable `input` is irrelevant, but the value of the variables `abs` and `sign` should be tracked. Furthermore, this invariant states that the then-branch of the **if**-statement in line 13 is taken. In this branch, the variable `abs` is not assigned, but the else-branch assigns it. Hence, we get a frame condition which then changes the error invariant again. From this change we can see that the missing assignment to `abs` is a potential reason for the failing assertion.

7 Conclusion

We have introduced the concept of flow-sensitive trace formulas. This new encoding of error traces into logical formulas enables proof-based fault localization methods to explain why relevant statements are reached in an erroneous execution of a program. Flow-sensitive trace formulas encode conditional choices of the error trace in such a way that the theorem prover can argue about the validity of the condition in the context of its prefix on the error trace. The resulting proof of unsatisfiability provides control-flow-related information on why the error occurred. We applied the flow-sensitive trace formula encoding to our fault localization technique based on error invariants. The resulting method identifies irrelevant portions of the code and finds a justification for the reachability of the remaining portions. Our evaluation shows that, while the produced abstract error traces are longer, they provide more useful error explanations. We therefore believe that the new encoding helps the programmer considerably to understand the faulty code fragment.

The application of the flow-sensitive trace formula encoding is not restricted to fault locations based on error invariants but can also be used in other methods. We will study such applications to other methods in our future work.

References

1. T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: localizing errors in counterexample traces. *SIGPLAN Not.*, pages 97–105, 2003.
2. I. Beer, S. Ben-David, H. Chockler, A. Orni, and R. Trefler. Explaining counterexamples using causality. In *CAV'09*, pages 94–108. Springer, 2009.
3. S. Chaki, A. Groce, and O. Strichman. Explaining abstract counterexamples. *SIGSOFT Softw. Eng. Notes*, 29(6):73–82, Oct. 2004.
4. J. Christ, J. Hoenicke, and A. Nutz. SMTInterpol: An interpolating SMT solver. In *SPIN*, pages 248–254, 2012.
5. E. Ermis, J. Hoenicke, and A. Podelski. Splitting via interpolants. In *VMCAI'12*, pages 186–201. Springer, 2012.
6. E. Ermis, M. Schäf, and T. Wies. Error Invariants. In *FM'12*, pages 338–353. Springer, 2012.
7. A. Groce. Error explanation with distance metrics. In *TACAS'04*, pages 108–122, 2004.
8. A. Groce and D. Kroening. Making the Most of BMC Counterexamples. *ENTCS*, pages 67–81, 2005.
9. A. Groce, D. Kroening, and F. Lerda. Understanding counterexamples with explain. In *CAV'04*, pages 453–456, 2004.
10. M. Jose and R. Majumdar. Bug-Assist: Assisting Fault Localization in ANSI-C Programs. In *CAV'11*, pages 504–509, 2011.
11. M. Jose and R. Majumdar. Cause clue clauses: error localization using maximum satisfiability. In *PLDI '11*, pages 437–446. ACM, 2011.
12. K. L. McMillan. An interpolating theorem prover. *Theor. Comput. Sci.*, pages 101–121, 2005.
13. K. L. McMillan. Lazy abstraction with interpolants. In *CAV'06*, pages 123–136, 2006.

14. D. Qi, A. Roychoudhury, Z. Liang, and K. Vaswani. Darwin: an approach for debugging evolving programs. In *ESEC/SIGSOFT FSE*, pages 33–42, 2009.
15. M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *ASE*, pages 30–39, 2003.
16. F. Tip. A survey of program slicing techniques. *JOURNAL OF PROGRAMMING LANGUAGES*, 3:121–189, 1995.
17. T. Wang and A. Roychoudhury. Automated path generation for software fault localization. In *ASE*, pages 347–351. ACM, 2005.
18. W. E. Wong and V. Debroy. Software fault localization, 2009.
19. A. Zeller. Isolating cause-effect chains from computer programs. In *SIGSOFT FSE*, pages 1–10, 2002.
20. X. Zhang, N. Gupta, and R. Gupta. Locating faults through automated predicate switching. In *ICSE*, pages 272–281, New York, NY, USA, 2006. ACM.