What Gives?
A Hybrid Algorithm for Error Trace Explanation

Vijayaraghavan Murali[1], Nishant Sinha[2], Emina Torlak[3], and Satish Chandra[4]

[1] NUS, Singapore [2] IBM Research, India [3] UC Berkeley, USA [4] Samsung Electronics, USA

Abstract. When a program fails, the cause of the failure is often buried in a long, hard-to-understand error trace. We present a new technique for automatic error localization, which formally unifies prior approaches based on computing interpolants and minimal unsatisfiable cores of failing executions. Our technique works by automatically reducing an error trace to its essential components—a minimal set of statements that are responsible for the error, together with key predicates that explain how these statements lead to the failure. We prove that our approach is sound, and we show that it is useful for debugging real programs.

1 Introduction

Understanding why a program failed is the first step toward repairing it. But this first step is also time-consuming and tedious. It involves examining the *error trace* of a failing execution (typically generated by print statements), reducing that trace to statements relevant to the error, and figuring out how the relevant statements transform program state to cause the observed failure (e.g., an assertion violation). Although debuggers aid this process by providing watchpoints and breakpoints, it is still a mostly manual task that relies heavily on programmer intuition about the code. In particular, classic debuggers cannot remove irrelevant statements from the error trace to help the programmer.

1	**`x=3;`**	$x = 3$
2	`y=5;`	
3	**`z1=y+x;`**	$z_1 = y + 3, x = 3$
4	**`z2=y-x;`**	$z_1 = z_2 + 6$
5	**`assert(z2>z1);`**	*false*

Fig. 1. Explaining error traces using abstract labels after each statement

Dynamic slicing [1, 36] was introduced as a way to automatically remove irrelevant statements from the trace. Slicing is done using dependency information (data or control), removing statements that do not impact the violated assertion via any chain of dependence. The main limitation of dynamic slicing is that it does not consider the *semantics* of the bug, which can result in irrelevant statement being retained. For example, in the error trace shown in the left column of Figure 1, dynamic slicing cannot rule out statement 2, whereas at a semantic level, the value of `y` is irrelevant.

In this paper, we propose a new semantics-aware technique for analyzing error traces and for helping programmers understand the cause of an error. Given an error trace, our algorithm produces a slice of the original trace annotated with abstract labels explaining the failure. For example, our algorithm produces the slice $\{1, 3, 4, 5\}$ for the trace in Figure 1, along with the explanatory labels highlighted in gray, which show why the assertion `z2>z1` fails. We call such an annotated slice an *error explanation*.

Briefly, given an error trace, our technique computes an error explanation in two steps. First, it computes an *interpolant* for each statement in the original trace. An interpolant is a formula that captures the effect of a given statement on the program state (as defined in Section 2), and in the context of error explanation, interpolants serve as explanatory labels. Next, the trace is sliced by eliding all statements that are surrounded by identical labels. Throughout this paper, we make use of two conventions: (i) the interpolant before the first statement of a trace is always $true$, which will not be shown, and (ii) if a particular statement does not have a label after it, the previous label is assumed to be present there. In our example (Figure 1), this approach would elide statement 2, which is semantically correct.

A slice produced by our algorithm is *sound* in that it fails for every binding of its free variables to values (such as the variable `y` for our example subtrace), and it is *minimal* in that it cannot be reduced any further without loss of soundness. Our technique guarantees both soundness and minimality by exploiting two central results of this paper. The first result is a theorem characterizing a class of interpolant labelings, which we call *inductive interpolant labelings* (IILs), that always lead to sound slices. The second result is a theorem relating slices induced by "maximally stationary" IILs, in which labels remain unchanged over a maximal set of statements (as defined in Section 3), to *minimal unsatisfiable cores* (MUC) of the formula that encodes the trace semantics. Informally, a MUC is an unsatisfiable fragment of a formula that becomes satisfiable if any of its constraints are removed. The formula for our example trace (Fig. 1) is shown below, and the constraints comprising its sole MUC are highlighted in gray:

$$x = 3 \wedge y = 5 \wedge z_1 = y + x \wedge z_2 = y - x \wedge z_2 > z_1$$

Our technique uses this MUC to produce the maximally stationary IIL shown in the figure, which leads to the sound and minimal slice $\{1, 3, 4, 5\}$.

1	*a[2]=0;*	$a[2] = 0$
2	**`i=h;`**	$a[2] = 0, h = 1, i = h$
3	**`i++;`**	$a[2] = 0, h = 1, i = 2$
4	**`v=a[i];`**	$h = 1, i = 2, v = 0, \mathbf{h \leq j, j \leq 1}$
5	**`j=j-h;`**	$h = 1, i = 2, v = 0, j = 0$
6	**`a[j]=v;`**	$a[0] = 0$
7	**`assert(a[0]==11 ∧ a[1]==14);`**	$false$

Fig. 2. Error explanation by [12] for shell-sort

Previous work [12] that uses interpolant-based slicing tries to increase stationariness (cf. Sec. 3.2) heuristically, without preserving the inductive property of labeling (see Sec 2.2 for a quick summary). This can lead to unsound slices. As an example of such unsoundness, consider the faulty shell-sort program from [12], which we will investigate in detail in Sec. 5.1 (Fig. 7). The technique in [12] returns the annotated sliced trace shown in Fig. 2 (taken from Fig. 4 in [12]). Although the slice annotation consists of interpolant labels, the slice itself is unsound because it *does not* violate the assertion on line 7 in the following environment:

$$\{\mathtt{h} \mapsto 0, \mathtt{j} \mapsto 2, \mathtt{a[0]} \mapsto 11, \mathtt{a[1]} \mapsto 14\}$$

The unsoundness is due to the omission of several statements that set the variables `h` and `j`. These statements are *necessary* for reproducing the failure, and they are included in the sound slice computed by our algorithm (Fig. 7).

This paper makes the following contributions:

1. We formally characterize error explanations using the notion of *inductive interpolant labelings*, which may be viewed as special kind of Hoare proofs.
2. We characterize two key properties of error explanations: *soundness* and *minimality*. We show that IILs form sound error explanations.
3. To characterize minimality, we introduce a new notion of maximally stationary labelings. We show that computing these labelings is equivalent to computing MUCs of the path formula of the failing trace.
4. Finally, we propose a new *hybrid* algorithm for computing sound and minimal error explanation that combines interpolant computation with a black-box MUC extraction procedure. We have implemented this algorithm and applied our prototype to two small case studies.

The rest of the paper is organized as follows. Section 2 reviews the background material on MUCs, interpolants, and the corresponding trace minimization methods. Section 3 establishes formal properties of a restricted, but sound, form of interpolant labeling, and the notion of minimality of slices. These properties are exploited in Section 4 to develop our new algorithm for error trace explanation. We evaluate the efficacy of the algorithm in Section 5, discuss related work in Section 6, and present concluding remarks in Section 7.

2 Preliminaries

Our approach draws on two previous techniques for error localization [20] and explanation [12]. We give a brief overview of these techniques and introduce the necessary terminology along the way. Both techniques work on a logical representation of a failing execution path π in a program P, which takes the form of a *path formula* Φ_π. The formula Φ_π is a conjunction $\iota \wedge \Phi \wedge \epsilon$, where ι is the input that triggered the failure of the assertion ϵ in P, and $\Phi = (\phi_1 \wedge \phi_2 \cdots \wedge \phi_n)$ is the SSA encoding of π. Because π violates ϵ, the path formula Φ_π is unsatisfiable. Both of the techniques reviewed in the section, as well as our algorithm, rely on analyzing unsatisfiable path formulas.

2.1 Unsatisfiable Core based Error Trace Explanation

Every unsatisfiable formula $\Phi = \phi_1 \wedge \ldots \phi_n$ contains one or more *unsatisfiable cores*, which are subsets of Φ whose conjunction is unsatisfiable. When every proper subset of a core is satisfiable, it is called a *minimal unsatisfiable core* (MUC) [32]. An unsatisfiable formula Φ also contains one or more *satisfiable subsets*. When every proper superset in Φ of such a subset is unsatisfiable, we call it a *maximal satisfiable subset* (MSS). The complement of an MSS is called a *minimal correcting subset* (MCS), which denotes a minimal subset of Φ whose removal from Φ will make the remainder satisfiable again. The set of MUCs and the set of MCSs of Φ are duals of each other [21]: one can be obtained by computing the irreducible hitting sets of the other.

Computing MCSs (or, dually, MUCs) of a path formula Φ_π identifies the relevant statements in a trace π [20]. For the example in Fig. 1, this method might first compute an MCS consisting of the statement on line 1, which would be flagged as a possible fix for the error. If the programmer wanted to see another repair candidate, the technique would compute another MCS, consisting of the statement on line 3, and so on. This process eventually flags all MCSs, covering all the statements in each MUC. It would not flag the statement on line 2, because it does not appear in any MUC of the trace.

In cases where multiple MUCs exist for a failing trace, where each MUC explains one aspect of the bug, one may typically employ a ranking heuristic to compare the MUCs, using say, the size of the MUC. In [20], a ranking mechanism highlights statements that occur most frequently in MCSs as having higher likelihood of being the cause. Our experiments, and those in [20], however, suggest that in practice most error traces contain only a small number of MUCs.

2.2 Interpolants-based Error Trace Explanation

Given a pair of formulas A and B, where $\neg(A \wedge B)$ holds, an *interpolant* of A and B [8], denoted by $Itp(A \mid B)$, is a formula I over the common symbols of A and B such that $A \Rightarrow I$ and $B \Rightarrow \neg I$. Given an indexed set (or sequence) of formulas $\Phi = [A_i]_{i=1}^{n}$, such that $\bigwedge \Phi = \mathit{false}$, let $\mathcal{A}_i = [A_k]_{k=1}^{i}$ and $\mathcal{B}_i = [A_k]_{k=i+1}^{n}$ for some $1 \leq i \leq n$.

An *interpolation sequence* for Φ is a sequence $\mathcal{I} = [I_i]_{i=0}^{n}$, such that the following holds: (i) $I_0 = \mathit{true}$ and $I_n = \mathit{false}$; (ii) $\bigwedge \mathcal{A}_i \Rightarrow I_i$ and $\bigwedge \mathcal{B}_i \Rightarrow \neg I_i$, where each I_i is an interpolant; and (iii) each I_i is over symbols common to the sets $\mathcal{A}_i$ and $\mathcal{B}_i$. The sequence $\mathcal{I}$ is said to be *inductive* if for each I_i, $I_i \wedge A_{i+1} \Rightarrow I_{i+1}$. An interpolant sequence for a path formula Φ_π (which is itself a sequence $[\iota, \phi_1, \ldots, \phi_n, \epsilon]$, with SSA subscripts dropped), shows the intermediate state abstractions that lead to the error, and hence constitutes an *explanation* for why π failed.

Ermis et al [12] compute interpolant labelings in the following way. First, they obtain a sequence of *candidate* interpolant labels from a theorem prover for each location along the failing path. This initial set of labels is then *minimized* by a greedy procedure which substitutes an interpolant at a given location, say I_j, by one from another location, say I_k, and checks if I_k is an interpolant at location j. If this greedy substitution succeeds, then all statements between the two program points are deemed irrelevant and sliced away. We show that this greedy technique may produce *unsound* labelings (defined below) and hence unsound error explanations. (We caution the reader that the term ‘inductive’ is used in [12] in a different sense than in this paper.)

2.3 Labeling and Sound Slices

Given a program trace $\pi = [S_i]_{i=1}^n$, a labeling L for π is the sequence of labels $[I_i]_{i=0}^n$. An *error labeling* L for a failing path π consists of labels that form an interpolant sequence for π. We say that L is *stationary* across S_i, denoted $st(L,i)$ iff $I_{i-1} \equiv I_i$. A labeling L for π induces a *slice* $\rho = \{S_i \in \pi | \neg st(L,i)\}$ that excludes statements in π across which L is stationary.

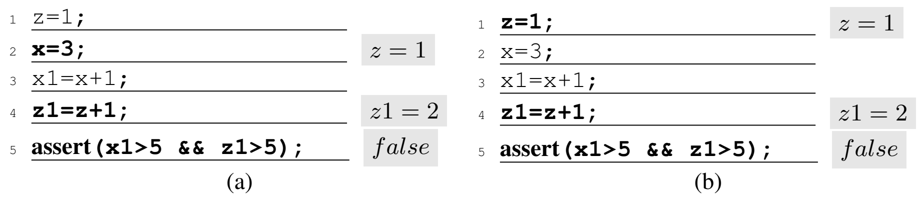
We say that ρ is a *sound* slice of π iff the path formula for ρ is unsatisfiable. Intuitively, this means that ρ is also a failing path. Instead of saying that the path formula for π is unsatisfiable, for simplicity we say that π is unsatisfiable or that π contains an unsatisfiable core.

3 Desired Properties of Error Explanations

In this section, we discuss two key properties for error explanations: soundness and minimality. We first show that explanations based on unrestricted interpolant labelings are not guaranteed to be sound. We then introduce the defining properties of sound and minimal explanations.

3.1 Sound Error Explanations

Figure 3(a) shows an example of an unsound error explanation, consisting of an error trace with a valid interpolant labeling (i.e., every label is a valid interpolant at its location). According to this explanation, statements 1 and 3 are irrelevant to the error, because both are surrounded by stationary labels ($true$ and $z = 1$, respectively). But removing statements 1 and 3 from the trace leaves the variables `z` and `x1` unconstrained, which renders the rest of the trace satisfiable.



```
1 z=1;
2 x=3;
3 x1=x+1;
4 z1=z+1;
5 assert(x1>5 && z1>5);
```

```
1 z=1;
2 x=3;
3 x1=x+1;
4 z1=z+1;
5 assert(x1>5 && z1>5);
```

Fig. 3. Problem with interpolants as labels for error explanation

The basic problem of general interpolant labelings is that a labeling as a whole may be unsound, even when the individual labels are valid interpolants. To ensure soundness (Theorem 1), we restrict the space of admissible sequences of interpolant labels to include only *inductive interpolant labelings* (IILs). An IIL is a sequence of interpolants that satisfies the inductive property (cf. Sec. 2)—each interpolant in an IIL must result from the conjunction of the preceding interpolant and the intervening program statement. That is, if I_1 and I_2 are the interpolant labels before and after a statement S, then $I_1 \wedge S \implies I_2$.

It is easy to see that the inductive property forbids the labeling sequence in Fig. 3(a), as $true \wedge x = 3 \not\Longrightarrow z = 1$. A possible IIL for this trace is given in Fig. 3(b), and it is the labeling computed by our method (Section 4). To the best of our knowledge, none of the previous methods for interpolant-based error explanation (including [12, 31]) guarantee the inductive property, nor any alternative soundness condition (see Sec. 6).

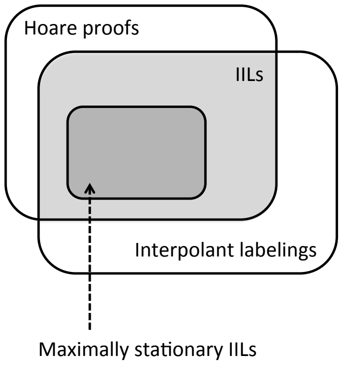


Fig. 4. Relating Hoare proofs, Interpolant labelings and IILs.

Remark. IILs correspond to a Hoare proof for a failing path π. In particular, $\{I_{j-1}\}S_j\{I_j\}$ is a valid Hoare triple for each $1 \leq j \leq n$. In an arbitrary Hoare proof for failure of π (with $true$ and $false$ as the first and last labels respectively), the assertion labels are inductive but they are not necessarily interpolants, i.e., they may contain symbols local to either the prefix or the suffix. Similarly, arbitrary interpolant error labelings do not form a Hoare proof, because they may not be inductive. IILs are restricted Hoare proofs that contain labels only over symbols common to the prefix and the suffix. Fig. 4 shows the general relationship between Hoare proofs, interpolant labelings and IILs visually (maximally stationary IILs are explained in Sec. 3.2).

Theorem 1. (Sound Error Labelings)
If an error labeling L for π is an inductive interpolant labeling (IIL) then the sliced error path π' induced by L is sound.

Proof. Without loss of generality, assume that exactly two statements S_j and S_k ($j < k$) are elided from the error path π to obtain π'. We then know the following:

(1) I_j is the same as I_{j-1} and I_k is the same as I_{k-1}, because S_j and S_k are sliced away
(2) $\bigwedge\{S_1, \ldots, S_{j-1}\} \implies I_{j-1}$ (Interpolant property)
(3) $I_k \wedge \bigwedge\{S_{k+1}, \ldots S_n\}$ is unsatisfiable (Interpolant property)
This means $I_{k-1} \wedge \bigwedge\{S_{k+1}, \ldots S_n\}$ is unsatisfiable (1)
(4) $I_j \wedge \bigwedge\{S_{j+1}, \ldots, S_{k-1}\} \implies I_{k-1}$ (Inductive property)
This means $I_{j-1} \wedge \bigwedge\{S_{j+1}, \ldots, S_{k-1}\} \implies I_{k-1}$ (1)

From (2) and (4), we have
(5) $\bigwedge\{S_1, \ldots, S_{j-1}, S_{j+1}, \ldots, S_{k-1}\} \implies I_{k-1}$

From (3) and (5), we have
(6) $\bigwedge\{S_1, \ldots, S_{j-1}, S_{j+1}, \ldots, S_{k-1}, S_{k+1}, \ldots, S_n\}$ is unsatisfiable.

This set contains all statements in the error path π except S_j and S_k and so represents the sliced error path π'. Hence, the path formula for π' is unsatisfiable, and π' is sound (cf. Sec. 2). The proof can be directly extended to an arbitrary number of elided statements. □

3.2 Minimal Error Explanations

Our goal is to compute not only sound but also *minimal* error explanations, which succinctly explain the fault to the programmer. To characterize the minimality of explanations, we propose a new formal criterion based on their stationariness.

Maximally Stationary Labeling. An IIL L is maximally stationary for a failing path π iff there exists no IIL L' (with induced slices π_L and $\pi_{L'}$ respectively), such that $\pi_L \subset \pi'_L$. In order to compute maximally-stationary IILs efficiently, we show that they correspond exactly to the MUCs of the path formula for π (Theorem 2). To prove this result, we need the following lemma which says that for *any* unsatisfiable core M of π, there exists an IIL stationary across all the statements $\mathcal{S}$ excluded from the core.

Lemma 1. *Let M be an unsatisfiable core for a failing path π and $\mathcal{S} \subset \pi$. If $\mathcal{S} \cap M = \emptyset$, then there exists an IIL L which is stationary across each statement in $\mathcal{S}$.*

PROOF. Let $|M| = l$. Let $\pi' = [S_{i_k}]_{k=1}^{l}$ be the *projection* of $\pi = [S_j]_{j=1}^{n}$ on to M such that each $S_{i_k} \in M$ and $1 \leq i_1 < i_2 \cdots < i_l \leq n$. Because M is unsatisfiable, there exists an IIL $L' = [I_{i_k}]_{k=1}^{l+1}$ for π', where for each i_k,
(1) $I_{i_k} \wedge S_{i_k} \Rightarrow I_{i_{k+1}}$ (2) $I_{i_k} = Itp(\iota \wedge S_{i_1} \wedge \ldots \wedge S_{i_{k-1}} \mid S_{i_k} \wedge \ldots \wedge S_{i_l} \wedge \epsilon)$.
Given a statement $S \in \mathcal{S}$, we will show how L' can be extended to an IIL for $M \cup \{S\}$. Assuming S occurs at position p in π, where $i_{k-1} < p < i_k$, we extend L' to L'' by copying I_{i_k} across S: $(I_{i_1})S_{i_1} \ldots S_{i_{k-1}}(I_{i_k})S(I_{i_k})S_{i_k}(I_{i_{k+1}}) \ldots S_{i_l}(I_{i_{l+1}})$.
It follows from (1) and $I_{i_k} \wedge S \Rightarrow I_{i_k}$ that L'' is inductive. To prove that each label is an interpolant, consider labels I_{i_m} where $m < p$. It follows from (2) that $(\iota \wedge S_{i_1} \wedge \ldots \wedge S_{i_{m-1}}) \Rightarrow I_{i_m}$. Also, $(S_{i_m} \wedge \ldots S_{i_l} \wedge \epsilon) \Rightarrow \neg I_{i_m}$. Hence, $(S \wedge S_{i_m} \wedge \ldots S_{i_l} \wedge \epsilon) \Rightarrow \neg I_{i_m}$. Also, each I_{i_m} is defined only over the shared symbols at position m in L''. The proof is similar for $m > p$. By extending L' for each $S \in \mathcal{S}$ iteratively, we obtain an IIL for π. □

Using Lemma 1, we can now prove our main theorem.

Theorem 2. *An IIL L for a failing trace π is maximally stationary iff the induced slice of L forms a MUC.*

PROOF. We show that ($\Rightarrow$) the slice induced by a maximally stationary IIL forms a MUC, and that ($\Leftarrow$) for every MUC M of π, there exists a maximally stationary IIL whose induced slice is equivalent to M. Both proofs are by contradiction.
($\Rightarrow$) Suppose the slice M induced by L is not a MUC. Then there exists an $S \in M$, such that $M \setminus \{S\}$ is unsatisfiable. So, by Lemma 1, there is an L' whose induced slice is $M \setminus \{S\} \subset M$. So, L is not maximally stationary.
($\Leftarrow$) By Lemma 1, there exists an IIL L whose induced slice is exactly M. Suppose L is not maximal. So there exists IIL L' with induced slice $M' \subset M$. Because L' is an IIL, it follows from Theorem 1 that the slice M' is sound, i.e., M' is also unsatisfiable. So M is not a MUC. □

In the next section, we exploit Thm. 2 to propose a new error explanation algorithm that computes maximally stationary IILs by using a MUC computation engine as a black-box. Our algorithm thus benefits directly from the advances in techniques for computing MUCs.

4 A Hybrid Algorithm for Error Trace Explanation

We now present a new algorithm for error trace understanding that combines the benefits of MUCs and interpolants. Fig. 5 shows the pseudocode for GENLABELS, the main procedure of our algorithm, to which the error path $\pi \equiv \{S_1, S_2, \ldots, S_n\}$ is provided. Without loss of generality, we assume that the input ι and the violated assertion ϵ are included in π. GENLABELS computes an IIL L for the failing trace π.

```
GENLABELS ($\pi \equiv S_1, S_2 \ldots, S_n$)
1:  **let** $\mathcal{C}$ := GETCORE ($\pi$) be $\{S_{c_1}, S_{c_2} \ldots, S_{c_m}\}$ s.t. $c_1 < c_2 \ldots < c_m$
2:  $I_0$ := *true*
3:  **for** i := 1 to $n$ **do**
4:      **if** $S_i$ occurs in $\mathcal{C}$ **then**
5:          **let** $k$ be such that $c_k == i$
6:          $\mathcal{V}$ := vars $(I_{i-1} \wedge S_i)$
7:          $\mathcal{V}'$ := vars $(S_{c_{k+1}} \wedge S_{c_{k+2}} \ldots \wedge S_{c_m})$
8:          $I_i$ := $\exists\, \overline{(\mathcal{V} \cap \mathcal{V}')}.(I_{i-1} \wedge S_i)$ (eliminate irrelevant variables)
9:      **else**
10:         $I_i$:= $I_{i-1}$
11: **endfor**
12: **return** $\{I_0, I_1 \ldots, I_n\}$
```

Fig. 5. Generating IILs from unsatisfiable cores

GENLABELS starts by obtaining an unsatisfiable core of π by calling GETCORE (line 1). GETCORE is a procedure that returns an unsatisfiable core of the given formula, while maintaining the relative ordering of constraints (statements) in the formula. Our algorithm then computes an IIL for $\mathcal{C}$, which induces a sound slice of π (by Thm. 1). If $\mathcal{C}$ is a MUC, the resulting IIL is maximally stationary (by Thm. 2), and the corresponding slice is minimal.

Line 2 initializes I_0 to *true*, the first label for any error trace. Then, for each statement S_i, GENLABELS performs the following steps. If S_i occurs in the core $\mathcal{C}$ (line 4), it computes the set $\mathcal{V}$ of variables that appear in the conjunction of the previous label I_{i-1} and the current statement S_i (line 6). It also computes the set $\mathcal{V}'$ of variables that appear in the core statements following S_i (line 7). The set $\mathcal{V} \cap \mathcal{V}'$ consists of variables that appear in both $I_{i-1} \wedge S_i$ and in the core statements following S_i. Its complement, $\overline{\mathcal{V} \cap \mathcal{V}'}$, represents the variables that are *not* common to $I_{i-1} \wedge S_i$ and the core statements following S_i.

On line 8, all variables in the set $\overline{\mathcal{V} \cap \mathcal{V}'}$ are existentially quantified and eliminated from the formula $I_{i-1} \wedge S_i$. The resulting formula is the label I_i after S_i. This step results in I_i being the "projection" of $I_{i-1} \wedge S_i$ onto the variables of the statements in $\mathcal{C}$ that follow S_i—that is, I_i is a formula only over the variables in $\mathcal{V} \cap \mathcal{V}'$. Finally, if the current statement S_i is not in the core $\mathcal{C}$, line 10 sets I_i to be the same as I_{i-1}. In Sec. 4.2, we prove that the labels generated in this way form an IIL for π.

As an example, consider the trace in Fig 1. Given $\pi \equiv \{x = 3, y = 5, z1 = y + x, z2 = y - x, z2 > z1\}$, assume that GETCORE returns the MUC $\mathcal{C} \equiv \{x =$

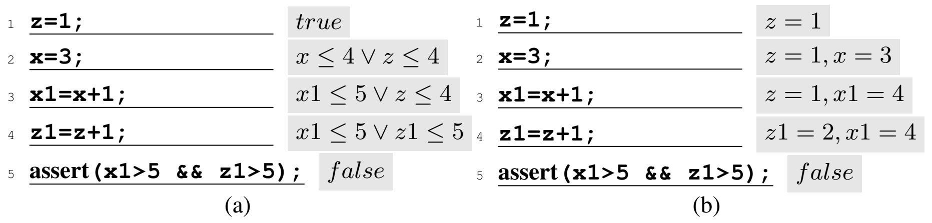
$3, z1 = y + x, z2 = y - x, z2 > z1\}$. Our algorithm first initializes I_0 to *true*. For the first statement $S_1 \equiv x = 3$, it sets $\mathcal{V}$ to be the variables in $I_0 \wedge S_1$, i.e., $\{x\}$, and $\mathcal{V}'$ to be the variables in $S_3 \wedge \ldots \wedge S_5$, i.e., $\{x, y, z1, z2\}$. Now, I_1 is the formula $\exists\{y, z1, z2\}.(true \wedge x = 3)$, which yields $x = 3$ after quantifier elimination. The second statement, y=5, is not in $\mathcal{C}$ so we generate the same label $x = 3$ after it. For $S_3 \equiv z1 = y + x$, the sets $\mathcal{V}$ and $\mathcal{V}'$ would be $\{x, y, z1\}$ and $\{x, y, z1, z2\}$ respectively. Thus I_3 is the formula $\exists\{z2\}(x = 3 \wedge z1 = y + x)$, which is equivalent to the quantifier-free $x = 3 \wedge z1 = y + 3$.

The final label I_4 is interesting: $\mathcal{V}$ is the set of variables in $I_3 \wedge S_4$, i.e., $\{x, y, z1, z2\}$, and $\mathcal{V}'$ is the set $\{z1, z2\}$. Therefore I_4 is $\exists\{x, y\}(z1 = y + 3 \wedge x = 3 \wedge z2 = y - x)$. Eliminating the quantifier yields $z1 = z2 + 6$, a predicate that is not obvious from the program. This shows that regardless of the value of y, z1 is 6 more than z2, which is why the assertion $z2 > z1$ failed. It can be seen that each label is indeed an interpolant and the sequence is inductive, as we will formally prove in Sec. 4.2.

4.1 Discussion of the Algorithm

Our algorithm is a variation on the strongest postcondition (SP) and weakest precondition (WP) computation [9]. In general, SP and WP labels may not be interpolants—particularly, they may carry variables not common to the prefix and suffix at a program point—because they only utilize information from "one direction" (forward for SP and backward for WP). Our algorithm fixes this by using MUCs to obtain, in the forward direction, the set of relevant statements so far and, in the backward direction, the set of variables to project away. With this knowledge, our algorithm is able to combine the benefits of forward and backward reasoning.

Nevertheless, even if the restriction of interpolants to be only on common variables is relaxed, SP and WP rarely keep the labels stationary. Fig. 6 shows the same example from Fig. 3 but labeled with (a) WP and (b) SP. It can be seen that neither is stationary at any point along the trace, hence no statement is removed from their induced slices.



`z=1;`	$true$
`x=3;`	$x \leq 4 \vee z \leq 4$
`x1=x+1;`	$x1 \leq 5 \vee z \leq 4$
`z1=z+1;`	$x1 \leq 5 \vee z1 \leq 5$
`assert(x1>5 && z1>5);`	$false$

(a)

`z=1;`	$z = 1$
`x=3;`	$z = 1, x = 3$
`x1=x+1;`	$z = 1, x1 = 4$
`z1=z+1;`	$z1 = 2, x1 = 4$
`assert(x1>5 && z1>5);`	$false$

(b)

Fig. 6. WP and SP may keep irrelevant statements in the explanation

Our algorithm uses the GETCORE procedure to project out variables unnecessary to *an* explanation that corresponds to an unsatisfiable core (for which we typically use a MUC). In Fig. 3(b), having prior knowledge of a particular MUC that excludes the constraint $x = 3$ allows our algorithm to project away x, keeping the interpolant stationary across the statement `x=3`.[1] Without the knowledge of such a core—e.g., if GETCORE were to return the entire trace—the algorithm would have no basis to project away `x`. It is this lack of "global knowledge" that makes explanations based purely on an interpolant computation (using WPs, SPs, or similar methods) less powerful than those seeded with a MUC.

We remark that we made the design choice of using quantifier-elimination in our algorithm for two reasons. First, we do not need to depend on an interpolating theorem prover or construct a refutation proof of the formula. Second, this approach exposes an interesting connection to SP and WP, as seen above. Having said that, our algorithm can be easily extended to use proof-based interpolation procedures [24, 11] by first obtaining the refutation proof p corresponding to an unsatisfiable core of π and then computing inductive interpolants from p.

4.2 Properties of the Algorithm

We present formal proofs that our algorithm generates IILs for every error trace π, thus inducing sound slices (Theorem 1).

Lemma 2. (Labels are inductive)
If I_{i-1} and I_i are the labels generated by our algorithm before and after a statement S_i respectively, then $I_{i-1} \ \wedge \ S_i \Rightarrow I_i$

Proof. Assume that GETCORE returned some unsatisfiable core $\mathcal{C}$ of the error path. If S_i did not appear in $\mathcal{C}$, then according to the algorithm (line 10), I_i is the same as I_{i-1}, and we are done since $I_{i-1} \wedge S_i \Rightarrow I_{i-1}$.

If S_i did appear in $\mathcal{C}$, then $I_i \equiv \exists \mathcal{V}''.(I_{i-1} \wedge S_i)$ where $\mathcal{V}''$ is as defined in the algorithm (lines 6-8). Assuming the theory of $I_{i-1} \wedge S_i$ supports quantifier elimination, $I_{i-1} \wedge S \Rightarrow \exists \mathcal{V}''.(I_{i-1} \wedge S_i)$ since quantifier elimination from a formula entails abstraction. Therefore $I_{i-1} \wedge S_i \Rightarrow I_i$. □

Note that by transitive closure of the inductive property for a sequence $S_i \ldots S_j$ of statements, we have that $I_{i-1} \wedge \bigwedge\{S_i, S_{i+1} \ldots S_j\} \Rightarrow I_j$.

Lemma 3. (Labels are interpolants)
Let the error path $\pi \equiv S_1, S_2 \ldots, S_i, \ldots, S_n$. If I_i is the label generated by our algorithm after S_i, then I_i is an interpolant. That is,

(a) $\bigwedge\{S_1, \ldots, S_i\} \Rightarrow I_i$
(b) $I_i \wedge \bigwedge\{S_{i+1}, \ldots, S_n\}$ *is unsatisfiable*
(c) I_i *is a formula only on the common variables of* $\{S_1, \ldots, S_i\}$ *and* $\{S_{i+1}, \ldots, S_n\}$

Proof. Base case: *true* is an interpolant before S_1 (or after an implicit empty S_0) since it satisfies (a), (b) and (c).

Assume that I_{i-1} is an interpolant after S_{i-1}.

[1] We could have also produced another IIL corresponding to the other MUC, removing the statement `z=1` in that case.

(a) $\bigwedge\{S_1, \ldots, S_{i-1}\} \Rightarrow I_{i-1}$ (hypothesis), and $I_{i-1} \wedge S_i \Rightarrow I_i$ (Lemma 2). Therefore, $\bigwedge\{S_1, \ldots, S_i\} \Rightarrow I_i$

(b) $I_{i-1} \wedge \bigwedge\{S_i, \ldots, S_n\}$ is unsatisfiable (hypothesis).

Consider the case S_i not occurring in $\mathcal{C}$. Then, $I_{i-1} \wedge \bigwedge\{S_{i+1}, \ldots, S_n\}$ is unsatisfiable (from hypothesis). The algorithm in this case sets I_i to be I_{i-1}. Therefore, $I_i \wedge \bigwedge\{S_{i+1}, \ldots, S_n\}$ is unsatisfiable.

Consider the case S_i occurring in $\mathcal{C}$. Then, $I_{i-1} \wedge \bigwedge\{S_{c_k}, S_{c_{k+1}}, \ldots, S_{c_m}\}$ is unsatisfiable, where $i = c_k$ and $\{S_{c_{k+1}}, S_{c_{k+2}}, \ldots, S_{c_m}\}$ is a subset of $\{S_{i+1}, \ldots, S_n\}$. We assume that the quantifier elimination is such that $I_i \equiv \exists\overline{\mathcal{V} \cap \mathcal{V}'}(I_{i-1} \wedge S_i)$ is the strongest formula implied by $I_{i-1} \wedge S_i$ on the variables $\mathcal{V} \cap \mathcal{V}'$. That is, I_i is *equivalent* to $I_{i-1} \wedge S_i$ on $\mathcal{V}$ and $\mathcal{V}'$. We also know that $\mathcal{V}' \equiv \mathsf{vars}\ (S_{c_{k+1}} \wedge S_{c_{k+2}} \ldots \wedge S_{c_m})$. This entails $I_i \wedge \bigwedge\{S_{c_{k+1}}, S_{c_{k+2}}, \ldots, S_{c_m}\}$ is unsatisfiable. Therefore, $I_i \wedge \bigwedge\{S_{i+1}, S_{i+2}, \ldots, S_n\}$ is unsatisfiable.

(c) We in fact prove a stronger version of (c): that I_i is only on the common variables of $\{S_{c_1}, S_{c_2}, \ldots, S_{c_k}\}$ and $\{S_{c_{k+1}}, S_{c_{k+2}}, \ldots, S_{c_m}\}$ for some k where the former is a subset of $\{S_1, S_2, \ldots, S_i\}$ and the latter is a subset of $\{S_{i+1}, S_{i+2}, \ldots, S_n\}$. The induction hypothesis here is that I_{i-1} is only on the common variables of $\{S_{c_1}, S_{c_2}, \ldots, S_{c_{k-1}}\}$ and $\{S_{c_k}, S_{c_{k+1}}, \ldots, S_{c_m}\}$ where the former is a subset of $\{S_1, S_2, \ldots, S_{i-1}\}$ and the latter is a subset of $\{S_i, S_{i+1}, \ldots, S_n\}$.

Consider the case S_i not occurring in $\mathcal{C}$, which implies $i \neq c_k$. Let $j = k - 1$. Then, I_i (the same as I_{i-1}, as set by the algorithm) is only on the common variables of $\{S_{c_1}, S_{c_2}, \ldots, S_{c_j}\}$ and $\{S_{c_{j+1}}, S_{c_{j+2}}, \ldots, S_{c_m}\}$. From the hypothesis, the former is a subset of $\{S_1, S_2, \ldots, S_{i-1}, S_i\}$ and the latter is a subset of $\{S_{i+1}, S_{i+2}, \ldots, S_n\}$, since $i \neq c_k$ (or $i \neq c_{j+1}$).

Consider the case S_i occurring in $\mathcal{C}$, which implies $i = c_k$. Then, $I_{i-1} \wedge S_i$ will be on $\mathsf{vars}(I_{i-1}) \cup \mathsf{vars}(S_i)$. Now, if all variables in $\mathsf{vars}(S_i)$ occur in $S_{c_{k+1}}, \ldots, S_{c_m}$ then $\mathcal{V} \cap \mathcal{V}'$ is simply $\mathsf{vars}(I_{i-1}) \cup \mathsf{vars}(S_i)$. Hence I_i is on the common variables of $\{S_{c_1}, \ldots, S_{c_k}\}$ and $\{S_{c_{k+1}}, \ldots S_{c_m}\}$. If there is a variable $v \in \mathsf{vars}(S_i)$ not occurring in $S_{c_{k+1}} \ldots S_{c_m}$, then v will not appear in $\mathcal{V} \cap \mathcal{V}'$. Hence it will appear in $\overline{\mathcal{V} \cap \mathcal{V}'}$ and will be quantified and eliminated from I_i.

Thus, I_i is an interpolant after S_i. □

5 Experimental Evaluation

We implemented our algorithm on the TRACER [17] framework for symbolic execution. The GETCORE procedure was implemented to return a MUC. We computed all MUCs using the method presented in [2], and generated an IIL for each MUC. We found that the method scales poorly for large programs, so one can also implement algorithms such as [21, 26, 4]. Our target programming language was C.

```
shell_sort(int a[], int size)
   int h=1, i, j;
   do
      h=h*3;
   while (h <= size);
   do {
      h /= 3;
      for (i=h; i<size; i++) {
         int v = a[i];
         for (j=i; j>=h &&
            a[j-h]>v; j-=h)
            a[j] = a[j-h];
         if (i != j)
            a[j] = v;
      }
   } while (h != 1);
```

(a)

```
1  a[0]=11
2  a[1]=14                    true
3  a[2]=0                     a[2] = 0
4  size=3

5  h=1;                       a[2] = 0, h = 1
6  i=h;                       a[2] = 0, h = 1, i = 1
7  v=a[i];
8  j=i;
9  i++;                       a[2] = 0, h = 1, i = 2
10 v=a[i];                    v = 0, h = 1, i = 2
11 j=i;                       v = 0, h = 1, j = 2
12 a[j]=a[j-h];
13 j=j-h;                     v = 0, h = 1, j = 1
14 a[j]=a[j-h];
15 j=j-h;                     v = 0, j = 0
16 a[j]=v;                    a[0] = 0
17 i++;
18 assert(a[0]==11);∧
   assert(a[1]==14);          false
```

(b)

Fig. 7. (a) The faulty shell sort program and (b) its error trace for input [11, 14] with labels

We provided input traces manually for our case studies, but they were automatically converted to SSA form by TRACER. The underlying constraint solver is CLP($\mathcal{R}$) [16], which uses the Fourier-Motzkin procedure for quantifier elimination over reals. We modeled program variables in the theory of linear real arithmetic due to our choice of solver, but any theory with quantifier elimination would work. Arrays were modeled using uninterpreted functions, and the McCarthy axioms [23] were applied to obtain a symbolic expression for each array reference. The heap was modeled as an array.

We now describe two case studies that serve as proof-of-concept that our algorithm works well in practice. The first case study uses the faulty sorting example from [12], and the second case study uses a more realistic program from the SIR repository [33]. We emphasize that the goal of this paper is mainly to provide a formal unification of MUC-based and interpolant-based error explanation, and so user studies regarding which approach is more "intuitive" for debugging are out of scope of this paper.

5.1 Shell Sort

Figure 7(a) shows the faulty program from [12], which is supposed to sort a given array of integers. When applied to the already sorted input `[11,14]`, it returns `[0,11]` instead of the input itself. Our safety property therefore asserts that the output should be `[11,14]`. The corresponding error trace is shown in Fig. 7(b), annotated with the labels computed by our algorithm, where bolded statements constitute the slice. We do not show the assume statements as they do not change the program state, and hence do not affect the interpolants. Note that we have “grounded” `h` to 1 instead of executing `h=h*3` and `h=h/3` because our underlying solver does not reason about integer arithmetic.

The initial label $a[2] = 0$ immediately suggests that there is a problem, because we are only sorting an array of two integers and should not be accessing `a[2]`. The rest of the labels capture how the value in `a[2]` gets propagated to `a[0]`. The variables `h` and `i` are initialized to 1. Then, `i` is incremented to 2, causing `a[i]` (now 0) to be stored in `v` (line 10). Next, `j` is initialized to `i`, and decremented by `h` twice to result in `j` being 0. Finally, line 16 stores `v` (which is 0) into `a[0]`, causing the violation. Our algorithm computed these labels within 2-3 seconds.

The inductiveness of labels is key to the quality of the explanation. Each label is implied by the conjunction of the previous label and the intermediate (bolded) statement, enabling local reasoning about the bug flow. Since each label is an interpolant, it only captures variables that are relevant to the bug at each point (see, e.g., line 10, where `a[2]` stops being relevant to the bug and `v` becomes relevant).

5.2 schedule2

For our second case study, we used the schedule2 program from the SIR repository [33]. It implements a priority scheduler for a given sequence of processes and their priorities (1, 2, or 3). The program's distributors seeded a bug, which sets the default priority (`prio`) of a process with no priority to be 1 (instead of -1). The error trace for an input of one process with no priority is shown in Fig. 8.

We do not show the entire trace for space reasons. However, it is worth noting that we applied dynamic slicing [27] on the original trace, which reduced its size from 129 to 58 statements, on which we applied our algorithm. In Fig. 8, the initial statement `prio=1` is the seeded bug which our labeling has captured. At line 37, a new process is created, which is captured by the label $new\_process \neq \texttt{NULL}$.

At line 44, the process is added to the priority queue data structure, as shown by the label $prio\_queue[1].head \neq \texttt{NULL}$. This indicates to the programmer that something is wrong, because the process should not have been added to the queue. At line 50, the process is retrieved from the head of the queue and set as the current job to be executed, as captured by the label $current\_job \neq \texttt{NULL}$ at line 51. The assertion states that when the `finish` function is called to execute the processes, there should be no jobs available, but there is one due to the bug, and therefore the assertion is violated. Our algorithm computed these labels in about 2-3 seconds.

Ultimately the dynamically sliced trace of 58 statements was reduced to just 16 statements through our algorithm. Together with the explanatory labels, it presents a much better explanation of the trace compared to dynamic slicing.

6 Related Work

BugAssist [20] analyzes proofs of unsatisfiability of the path formula to localize errors; a minimal set of statements which, on removal, makes the formula satisfiable again

30	**prio=1;**	$prio = 1$
31	**prio**$_{\text{new\_job}}$**=prio;**	$prio_{\text{new\_job}} = 1$
37	**new_process=malloc(sizeof(struct process));**	$prio_{\text{new\_job}} = 1, new\_process \neq \texttt{NULL}$
41	**prio**$_{\text{enqueue}}$**=prio**$_{\text{new\_job}}$**;**	$prio_{\text{enqueue}} = 1, new\_process \neq \texttt{NULL}$
42	**prio**$_{\text{put\_end}}$**=prio**$_{\text{enqueue}}$**;**	$prio_{\text{put\_end}} = 1, new\_process \neq \texttt{NULL}$
43	**process**$_{\text{put\_end}}$**=new_process;**	$prio_{\text{put\_end}} = 1, process_{\text{put\_end}} \neq \texttt{NULL}$
44	**prio_queue[prio**$_{\text{put\_end}}$**].head=process**$_{\text{put\_end}}$**;**	$prio\_queue[1].head \neq \texttt{NULL}$
45	**prio**$_{\text{get\_current}}$**=3;**	$prio_{\text{get\_current}} = 3, prio\_queue[1].head \neq \texttt{NULL}$
46	**prio**$_{\text{get\_current}}$**=prio**$_{\text{get\_current}}$**-1;**	$prio_{\text{get\_current}} = 2, prio\_queue[1].head \neq \texttt{NULL}$
47	**prio**$_{\text{get\_current}}$**=prio**$_{\text{get\_current}}$**-1;**	$prio_{\text{get\_current}} = 1, prio\_queue[1].head \neq \texttt{NULL}$
48	**prio**$_{\text{get\_process}}$**=prio**$_{\text{get\_current}}$**;**	$prio_{\text{get\_process}} = 1, prio\_queue[1].head \neq \texttt{NULL}$
49	**job**$_{\text{get\_process}}$**=&current_job;**	$job_{\text{get\_process}} = \&current\_job, prio_{\text{get\_process}} = 1, prio\_queue[1].head \neq \texttt{NULL}$
50	**next**$_{\text{get\_process}}$**=&prio_queue[prio**$_{\text{get\_process}}$**].head;**	$job_{\text{get\_process}} = \&current\_job, *next_{\text{get\_process}} \neq \texttt{NULL}$
51	***job**$_{\text{get\_process}}$**=*next**$_{\text{get\_process}}$**;**	$current\_job \neq \texttt{NULL}$
54	**job**$_{\text{finish}}$**=current_job;**	$job_{\text{finish}} \neq \texttt{NULL}$
58	**assert(job**$_{\text{finish}}$**==NULL);**	$false$

Fig. 8. Inductive interpolant labeling computed by our algorithm for schedule2

is marked as containing potential causes of the error. Instead, our approach computes error-explaining labels on the trace and exploits MUCs to compute a path slice relevant to the error. Ermis et al. [12] proposed the idea of computing error-explaining labels using interpolants. Because they compute individual labels independently, followed by a post-processing step to improve stationariness, the resulting sequence of labels may be unsound (cf. Sec. 3).

Popular methods for fault localization include Delta debugging [35, 7] and Darwin [28], which compare failing and successful program states and executions; DIDUCE [14], which computes likely invariants from good runs and checks for their violations on other runs; statistical methods [19], which compute suspiciousness of statements based on the frequency of their occurrences in passing and failing runs; and methods based on dynamic slicing [1, 36], which consider dependency flows in failing runs. Other approaches use symbolic techniques to explain counterexamples obtained by model checking [3, 13]. Symbolic techniques have also been used for program repair [6, 22]. Sahoo et al. [30] combine likely invariants, delta debugging and dynamic slicing techniques for scalable root cause analysis on real-world programs. Error-explaining labels may be viewed as likely invariants along the failing path; they assist the developer in pinpointing the root cause.

Interpolant computation [24] is widely used to enable convergence of SAT/SMT-based bounded model checking of both hardware [24] and software [15, 25]. Interpolants of different strengths may be derived from the same proof of unsatisfiability; D'Silva et al. [11] present a unified lattice-based framework for ordering the interpolants computed by different methods. Like our method, CEGAR-based software verification techniques [15, 5], which use interpolants for refinement, also require that the interpolant sequence is inductive [18, 34, 29]. Dillig et al. use abductive inference for assisting developers in classifying erroneous analysis reports [10].

7 Conclusion

Interpolant-based error explanations are attractive in their ability to convey the essence of an error trace to a programmer. In this paper, we examined their ability to filter out statements that are guaranteed to be irrelevant to the error. Since the goal of minimal unsatisfiable core computation is also similar, we examined whether the two techniques have any formal relationship. We found that, in general, interpolant-based error explanations are weaker than minimal cores in their ability to exclude irrelevant statements. More importantly, we identified sufficient conditions on interpolant sequences so that statements are not unsoundly ruled out as irrelevant; previous work is vulnerable to this pitfall. We also pinpoint reasons why it is difficult to arrive at sound interpolant labeling that matches minimal unsatisfiable cores in their power to exclude irrelevant statements.

References

1. H. Agrawal and J. R. Horgan. Dynamic program slicing. In *PLDI '90*, pages 246–256, 1990.
2. J. Bailey and P. J. Stuckey. Discovery of minimal unsatisfiable subsets of constraints using hitting set dualization. In *PADL*, 2005.
3. T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: localizing errors in counterexample traces. In *POPL '03*, pages 97–105, 2003.
4. A. Belov and J. Marques-Silva. MUSer2: An efficient MUS extractor. *JSAT*, 2012.
5. D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker BLAST: Applications to software engineering. *Int. J. Softw. Tools Technol. Transf.*, 9(5), 2007.
6. S. Chandra, E. Torlak, S. Barman, and R. Bodik. Angelic debugging. ICSE, 2011.
7. H. Cleve and A. Zeller. Locating causes of program failures. ICSE, pages 342–351, 2005.
8. W. Craig. Three uses of Herbrand-Gentzen theorem in relating model theory and proof theory. *Journal of Symbolic Computation*, 22, 1957.
9. E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1997.
10. I. Dillig, T. Dillig, and A. Aiken. Automated error diagnosis using abductive inference. PLDI, 2012.
11. V. D'Silva, D. Kroening, M. Purandare, and G. Weissenbacher. Interpolant strength. VMCAI, 2010.
12. E. Ermis, M. Schäf, and T. Wies. Error invariants. In *FM*, 2012.
13. A. Groce, S. Chaki, D. Kroening, and O. Strichman. Error explanation with distance metrics. *Software Tools for Technology Transfer (STTT)*, 8:229–247, 2006.
14. S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. ICSE '02.

15. T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *POPL*, 2004.
16. J. Jaffar, S. Michaylov, P. J. Stuckey, and R. H. C. Yap. The CLP($\mathcal{R}$) language and system. 14(3):339–395, 1992.
17. J. Jaffar, Vijayaraghavan Murali, J. Navas, and A. Santosa. TRACER: A Symbolic Execution Tool for Verification. In *CAV 2012*, pages 758–766, 2012.
18. R. Jhala and K. L. McMillan. A practical and complete approach to predicate refinement. TACAS, 2006.
19. J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *ICSE '02*, pages 467–477, 2002.
20. M. Jose and R. Majumdar. Cause clue clauses: Error localization using maximum satisfiability. *SIGPLAN Not.*, 46(6):437–446, 2011.
21. M. H. Liffiton and K. A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *J. Autom. Reasoning*, 40(1), 2008.
22. F. Logozzo and T. Ball. Modular and verified automatic program repair. OOPSLA, 2012.
23. J. McCarthy. A Basis for a Mathematical Theory of Computation. In *Computer Programming and Formal Systems*, 1963.
24. K. L. McMillan. Interpolation and sat-based model checking. In *CAV*, 2003.
25. K. L. McMillan. Applications of craig interpolants in model checking. In *TACAS*, 2005.
26. A. Nadel. Boosting minimal unsatisfiable core extraction. FMCAD, 2010.
27. G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proceedings of the 11th International Conference on Compiler Construction*, 2002.
28. D. Qi, A. Roychoudhury, Z. Liang, and K. Vaswani. Darwin: An approach to debugging evolving programs. *ACM Trans. Softw. Eng. Methodol.*, 21(3):19, 2012.
29. S. F. Rollini, O. Sery, and N. Sharygina. Leveraging interpolant strength in model checking. CAV'12.
30. S. K. Sahoo, J. Criswell, C. Geigle, and V. Adve. Using likely invariants for automated software fault localization. ASPLOS, 2013.
31. M. Schäf, D. Schwartz-Narbonne, and T. Wies. Explaining inconsistent code. In *ESEC/SIGSOFT FSE*, pages 521–531, 2013.
32. J. P. M. Silva. Minimal unsatisfiability: Models, algorithms and applications (invited paper). In *ISMVL*, 2010.
33. Software-artifact infrastructure repository (SIR). http://sir.unl.edu/portal/index.html, August 2010.
34. Y. Vizel and O. Grumberg. Interpolation-sequence based model checking. In *FMCAD*, 2009.
35. A. Zeller. Isolating cause-effect chains from computer programs. In *FSE '02*, pages 1–10, 2002.
36. X. Zhang, N. Gupta, and R. Gupta. A study of effectiveness of dynamic slicing in locating real faults. *Empirical Softw. Engg.*, 12(2).