Error Invariants

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. Localizing the cause of an error in an error trace is one of the most time-consuming aspects of debugging. We develop a novel technique to automate this task. For this purpose, we introduce the concept of *error invariants*. An error invariant for a position in an error trace is a formula over program variables that over-approximates the reachable states at the given position while only capturing states that will still produce the error, if execution of the trace is continued from that position. Error invariants can be used for slicing error traces and for obtaining concise error explanations. We present an algorithm that computes error invariants from Craig interpolants, which we construct from proofs of unsatisfiability of formulas that explain why an error trace violates a particular correctness assertion. We demonstrate the effectiveness of our algorithm by using it to localize faults in real-world programs.

1 Introduction

A central element of a programmer's work routine is spending time on debugging. Particularly time-consuming (and often the most frustrating part of debugging) is the task of *fault localization* [1, 3, 9, 10, 13, 14, 18, 20, 21], i.e., isolating the cause of an error by inspecting a failed execution of the program. This task encompasses, for instance, the identification of the program statements that are relevant for the error, and determining the variables whose values should be tracked in order to understand the cause of the error.

In this paper, we present a novel technique that enables automated fault localization and the automatic generation of concise error explanations. The input to our technique is a an *error trace* of the program, which consists of the sequence of program statements whose execution produced an error, and formulas describing the initial states of the trace and the expected output states (i.e., the assertion that was violated). Such error traces can be obtained either from failing test cases or from counterexamples produced by static analysis tools. Our technique is based on the new concept of *error invariants*. An invariant for a given position in a trace is a formula satisfied by all states reaching that position in an execution of the trace. An error invariant is an invariant for a position in an error trace that only captures states that will still produce the error, if execution of the trace is continued from that position. Hence, an error invariant provides an explanation for the failure of the trace at the given position. We observe that *inductive* error

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

invariants, which are those error invariants that hold for consecutive positions in an error trace, characterize statements in the trace that are irrelevant for the error. That is, if an error invariant holds for an interval of consecutive positions, no relevant changes have occurred to error relevant variables in that interval. A statement that is enclosed by an inductive error invariant can thus be replaced by any other statement that preserves the invariant, without changing the nature of the error. Hence, inductive error invariants can be used to compute slices of error traces that contain only relevant statements and information about reachable states that helps to explain the cause of an error. Moreover, error invariants characterize the relevant variables whose values should be tracked along the execution of the error trace.

To compute inductive error invariants, we build on the idea of *extended trace formulas* [14] to obtain an unsatisfiable formula from an error trace. We then compute Craig interpolants for each position in the trace from the proof of unsatisfiability of this formula. These Craig interpolants serve as candidate error invariants which we subsequently propagate through the trace to check their inductiveness. Thus, we build on existing techniques for synthesizing inductive invariants in program verification [12] to compute inductive error invariants. We implemented our technique in a prototype tool and evaluated it on error traces taken from the literature as well as real-world examples. For the error traces that we have considered, the error invariants computed by our technique capture the precise cause of the error.

Related Work. Minimizing error traces to aid debugging is an active area of research. Recently, static techniques for identifying relevant fragments of error traces have been developed. Closest to our approach is Bug-Assist [13, 14], which uses a MAX-SAT based algorithm to identify a maximal subset of statements from an error trace that cannot be responsible for the failing of the execution. The remaining statements then form an error trace such that removing any statements from this trace will result in a trace that has normally terminating executions. One benefit of this approach is that a compact error trace can be computed with a single MAX-SAT query while our approach requires several calls to a theorem prover. On the other hand, our approach can further simplify the error traces obtained by Bug-Assist because it may replace some of the remaining statements with error invariants. For instance, if a relevant variable is incremented several times in a row (e.g., in a loop), our approach may replace all these statement by one invariant stating that the incremented variable is within a certain bound. Also, error invariants identify variables that should be tracked during debugging and that highlight the relevant changes to the program state. This is particularly useful for *dense errors*, where the length of the error trace cannot be reduced significantly. A common limitation of our approach and Bug-Assist is that control-relevant variables might not be considered relevant. This, however, depends on the way error traces are encoded as formulas.

Another way to minimize error traces is to compare failing with successful executions (e.g., [1, 9, 10, 17, 18, 20]). Ball et al. [1] present an algorithm for isolating parts of an error trace that do not occur on feasible traces. Groce et al. [9–11] use distance metrics for program executions to find minimal abstractions of error traces. For a given counterexample, they find a passing execution that is as similar to the counterexample as possible. The deviations between the passing and failing executions are then presented

```
int y=0;

void testFoo() {
   int res = foo(0,-2,1);
   CU_ASSERT(res>=0);
}
```

```
int foo(int a, int b, int x) {
  x = x + a;
  x = x + b;
  y = y + a;
  return x;
}
```

Fig. 1. Example of a failing unit test

as an explanation for the error. The major difference of these approaches to ours is that they require passing executions that are similar to the failing execution as an additional input. Hence, these approaches are limited to cases where it is possible to find adequate passing runs that cover large portions of the original error trace.

Dynamic approaches can be used to reduce the cognitive load for the programmer. Delta Debugging (e.g., [3]) compares failing executions with passing executions to identify relevant inputs that can be blamed for the failing of the execution. Dynamic slicing (see, e.g., [21] for an overview) removes irrelevant fragments from error traces based on dynamic control and data dependencies. Both approaches return a compact representation of the original error trace. Hence, our approach of using error invariants is orthogonal to delta debugging and dynamic slicing. Similar to Bug-Assist, dynamic techniques can be used to compute a compressed error trace that serves as input to our static analysis algorithm, which then computes further information about the error.

2 Overview and Illustrative Example

We demonstrate how error invariants can help to produce a more compact representation of an error trace using the illustrative example in Figure 1. We call this compact representation an *abstract error trace*. The figure shows a procedure `foo` and a unit test `testFoo`, which checks if `foo` returns a certain value when it is called on a particular input. The tested procedure `foo` adds the variables `a` and `b` to `x` before returning the new value of `x`. Further, `foo` increments the global variable `y`. The unit test `testFoo` calls `foo` on an initial state where `a=0`, `b=-2`, and `x=1`, and then checks if `foo` returns a value greater or equal to 0. However, this is not the case and the unit test fails. From the failing test case we can derive the following error trace (ψ, π, ϕ), where the path π is the sequence of statements executed on the trace:

$$\psi \equiv (a = 0 \wedge b = -2 \wedge x = 1 \wedge y = 0) \qquad \phi \equiv x \geq 0$$
$$\pi = \ell_0 \colon x = x + a; \;\; \ell_1 \colon x = x + b; \;\; \ell_2 \colon y = y + a; \;\; \ell_3 \colon$$

To understand why the post-condition $x \geq 0$ of our unit test is violated, we first compute a *trace formula* for the error trace, which is a conjunction of the pre-condition ψ, the post-condition ϕ, and a formula representation $\mathbf{TF}(\pi)$ of the path π, such that the satisfying assignments of the trace formula exactly correspond to the possible executions of the path π that satisfy the pre and post-condition. The resulting formula

$$(a=0) \wedge (b=-2) \wedge (x=1) \wedge (y=0) \wedge (x'=x+a) \wedge (x''=x'+b) \wedge (y'=y+a) \wedge (x'' \geq 0)$$

is unsatisfiable, as the execution of the test case violates the assertion at the end of the error trace. From the proof of unsatisfiability, we compute a sequence of formulas $I_0, \ldots, I_3$, where each I_i is an error invariant for position ℓ_i of the error trace. This means that for each i, the formula $\psi \wedge \mathbf{TF}(\pi[0,i]) \Rightarrow I_i$ is valid and the formula $I_i \wedge \mathbf{TF}(\pi[i,3]) \wedge \phi$ is unsatisfiable, where $\pi[0,i]$ is the prefix of the trace up to position i and $\pi[i,3]$ the corresponding suffix. For our example, a possible sequence of error invariants is as follows:

$$I_0 \equiv (x = 1) \wedge (a = 0) \wedge (b = -2) \qquad I_2 \equiv (x = -1)$$
$$I_1 \equiv (x = 1) \wedge (b = -2) \qquad I_3 \equiv (x = -1)$$

An error invariant I_i can be seen as an abstraction of the set of post states of the prefix trace $\pi[0,i]$ such that any execution of the suffix trace $\pi[i,3]$ that starts in a state satisfying I_i will still fail. Thus, for each point of the error trace, the error invariant can provide a concise explanation why the trace fails when the execution is continued from that point. In particular, error invariants provide information about which variables are responsible for an execution to fail and which values they have at each point. In our example, I_0 is a summary of the initial state and indicates that the variable `a`, `b`, and `x` are responsible for the error. After executing $x = x + b$, we can see from I_2 that `a` and `b` are no longer relevant and only `x` has to be considered. Further, we can see that the value of variable `y` does not matter at all. Error invariants also help to identify irrelevant statements in an error trace. Note that the formula I_1 is a valid error invariant for both positions ℓ_2 and ℓ_3. That is, I_2 is *inductive* with respect to the statement `y = y + a` and any execution of the suffix trace $\pi[2,3]$ that starts in a state satisfying I_2 will still fail. The fact that I_2 is an error invariant for both ℓ_2 and ℓ_3 implies that the formula $\psi \wedge \mathbf{TF}(\pi) \wedge \phi$ will remain unsatisfiable even if `y = y + a` is removed from the trace π. Thus, this statement is irrelevant for the error.

In the following sections, we formalize the concept of error invariants and present an algorithm that synthesizes error invariants to compute concise abstractions of error traces.

3 Preliminaries

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

For a variable $x \in X$ and $i \in \mathbb{N}$, we denote by $x^{\langle i \rangle}$ the variable 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 $x^{\langle i \rangle}$, which we also extend to formulas. A *transition formula* T is

a first-order constraint over free variables from $X \cup X'$, where the variables X' denote the values of the variables from X in the next state. A transition formula T represents a binary relation on states and we write $s, s' \models T$ to denote that the pair of states (s, s') is in the relation represented by T.

A *program* $\mathcal{P}$ over variables X is simply a finite set of transition formulas over $X \cup X'$. Each transition formula $T \in \mathcal{P}$ models the semantics of a single program statement. Note that control can be model implicitly via a dedicated program variable $\mathsf{pc} \in X$ for the program counter. The correctness assertion of a program can be stated as a relation between the pre and the post states of the program's executions. A witness of the incorrectness of the program is given by an error trace. An error trace consists of a state formula, describing the initial states from which a failed execution can start, a path of the program describing the statements of the failed execution, and a state formula, which describes the violated correctness assertion.

Formally, a *path* π of a program $\mathcal{P}$ is a finite sequence of transition formulas in $\mathcal{P}$. Let $\pi = T_0, \ldots, T_{n-1}$ be a path. For $0 \leq i \leq j \leq n$, we denote by $\pi[i, j]$ the subpath $T_i, \ldots, T_{j-1}$ of π that goes from position i to position j. We use $\pi[i]$ to represent the i-th transition formula T_i of path π. A *trace* τ of a program $\mathcal{P}$ is a tuple (ψ, π, ϕ) where π is a path of $\mathcal{P}$ and ψ and ϕ are state formulas. We say that τ has length n if π has length n. An *execution* of a trace (ψ, π, ϕ) is a sequence of states $\sigma = s_0 \ldots s_n$ such that (1) $s_0 \models \psi$, (2) $s_n \models \phi$, and (3) for all $0 \leq i < n$, $s_i, s_{i+1} \models \pi[i]$. The *trace formula* $\mathbf{TF}(\tau)$ of a trace $\tau = (\psi, \pi, \phi)$ of length n is the formula $\psi \wedge (\pi[0])^{\langle 0 \rangle} \wedge \ldots \wedge (\pi[n-1])^{\langle n-1 \rangle} \wedge \phi^{\langle n \rangle}$. Thus, there is a one-to-one correspondence between the executions of τ and the models of $\mathbf{TF}(\tau)$. For a path π we write $\mathbf{TF}(\pi)$ as a shorthand for the trace formula $\mathbf{TF}((\top, \pi, \top))$. A trace τ is called *feasible*, if its trace formula $\mathbf{TF}(\tau)$ is satisfiable. A trace is called *error trace* if it is infeasible.

4 Error Invariants

An error trace provides sufficient information to repeat the program's behavior that violates the correctness assertion. There are many ways to obtain error traces, e.g., from a failing test case, from a counterexample returned by a static analysis tool [15], or when debugging, by manually marking a particular state as violation of the correctness assertion. By limiting the scope to only one control-flow path, error traces can help the programmer to detect the cause of the unexpected behavior. However, the error trace itself does not give any insight into which transitions on the path of the trace are actually responsible for the incorrect behavior. Further, the trace does not say which variables on this path should be tracked to identify the cause of the error. This is particularly challenging for error traces of large programs, where the number of transitions and variables might become intractable for the programmer.

In this paper, we propose the concept of *error invariants* as a means to rule out irrelevant transitions from an error trace, to identify the program variables that should be tracked along the path in order to understand the error, and to obtain a compact representation of the actual cause of an error.

Definition 1 (Error Invariant). *Let $\tau = (\psi, \pi, \phi)$ be an error trace of length n and let $i \leq n$ be a position in π. A state formula I is an* error invariant *for position i of τ, if the following two formulas are valid:*

1. $\psi \wedge \mathbf{TF}(\pi[0, i]) \Rightarrow I^{\langle i \rangle}$
2. $I \wedge \mathbf{TF}(\pi[i, n]) \wedge \phi^{\langle n-i \rangle} \Rightarrow \bot$

An error invariant for a position in an error trace can be understood as an abstract representation of the *reason* why the execution will fail if it is continued from that position. We next explain how error invariants can be used to identify transitions and program variables that are relevant for the fault in the error trace.

Using Error Invariants for Fault Localization. In the following, let τ be an error trace. We say that a state formula I is an *inductive* error invariant for positions $i \leq j$, if I is an error invariant for both i and j. Given such an inductive error invariant, we can argue that the execution of the path of the error trace will still fail for the same reason, even if the transitions between positions i and j are not executed. Thus, we can use inductive error invariants to identify irrelevant transitions in error traces. Inductive error invariants further help to identify the relevant program variables that should be tracked while debugging an error trace. Namely, if I is an inductive invariant for positions $i < j$, then only the program variables appearing in I should be tracked for the trace segment between the positions i and j. We can make these observations formal.

Abstract Error Traces. We say that a trace $\tau^\#$ *abstracts* a trace τ if for every n-step execution σ of τ there exists an m-step execution $\sigma^\#$ of $\tau^\#$ such that $\sigma^\#[0] = \sigma[0]$, $\sigma^\#[m] = \sigma[n]$, and $\sigma^\# \preceq \sigma$. Here, $\preceq$ denotes the *subsequence ordering*, i.e., $a_0 \dots a_m \preceq b_0 \dots b_n$ iff $a_0 = b_{i_0}, \dots, a_m = b_{i_m}$ for some indices $0 \leq i_0 < \dots < i_m \leq n$. The problem we are attempting to solve in this paper, is to find for a given error trace $\tau = (\psi, \pi, \phi)$ an error trace $\tau^\# = (\psi, \pi^\#, \phi)$ such that $\tau^\#$ abstracts τ and $\pi^\#$ concisely explains why π is failing for (ψ, ϕ). We use inductive error invariants to define such abstract error traces.

Let $\pi^\# = I'_0, T_1, I'_1, \dots, T_k, I'_k$ be an alternating sequence of primed state formulas I'_j and transition formulas T_j. Note that a primed state formula I' can be interpreted as a transition formula that models transitions in which all program variables are first non-deterministically updated and then assumed to satisfy the formula I. Thus, $\pi^\#$ is also a path. We call $(\psi, \pi^\#, \phi)$ an *abstract error trace* for (ψ, π, ϕ) if there exist positions $i_0 < \dots < i_{k+1}$ such that $i_0 = 0$, $i_{k+1} = n + 1$, for all j with $1 \leq j \leq k$, $T_j = \pi[i_j]$, and for all j with $0 \leq j \leq k$, I_j is an inductive error invariant for i_j and $i_{j+1} - 1$.

Theorem 2. *If $\tau^\#$ is an abstract error trace for an error trace τ, then $\tau^\#$ abstracts τ.*

In the next section, we show how abstract error traces can be computed using Craig interpolants, which we obtain automatically from the unsatisfiability proof of the extended path formula.

5 Error Invariants from Craig Interpolants

There are different ways to obtain error invariants for error traces. For instance, given an error trace $\tau = (\psi, \pi, \phi)$ of length n, the weakest error invariant for position i in

π is given by the weakest (liberal) precondition of the negated post-condition and the suffix of the path starting from i: $\mathsf{wlp}(\pi[i,n], \neg\phi)$. However, weakest error invariants are typically not inductive, nor do they provide compact explanations for the cause of an error. What we are really interested in are *inductive* error invariants.

Error invariants are closely related to the concept of *Craig interpolants* [5]. Let A and B be formulas whose conjunction is unsatisfiable. A formula I is a *Craig interpolant* for A and B, if the following three conditions hold: (a) $A \Rightarrow I$ is valid, (b) $I \wedge B$ is unsatisfiable, and (c) all free variables occurring in I occur free in both A and B. Thus, given a position i in an error trace, we can split the unsatisfiable path formula $\psi \wedge \mathbf{TF}(\pi) \wedge \phi^{\langle n \rangle}$ into two conjuncts $A = \psi \wedge \mathbf{TF}(\pi[0,i])$ and $B = \mathbf{TF}(\pi[i,n])^{\langle i \rangle} \wedge \phi^{\langle n \rangle}$, for which we can then obtain interpolants[4].

Proposition 3. *Let (ψ, π, ϕ) be an error trace of length n, let i be a position in π, and 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 i.*

Interpolants are always guaranteed to exist for first-order logical formulas A and B whose conjunction is unsatisfiable [5]. In fact, for many first-order theories there always exist quantifier-free interpolants that can be directly constructed from the proof of unsatisfiability of the conjunction $A \wedge B$ [2,16]. Interpolants constructed in this way often give concise explanations for the infeasibility of a trace. For this reason, they have shown to be useful for finding inductive invariants in program verification [12]. We argue that the same is true for error traces and show how to find interpolants that are, both, inductive error invariants and useful for fault localization.

Computing Abstract Error Traces. In the following, let (ψ, π, ϕ) be an error trace with $\pi = (T_i)_{0 \leq i < n}$. The problem we want to solve is to compute an abstract error trace for (ψ, π, ϕ), i.e., an alternating sequence of inductive error invariants and transitions $I_0, T_{i_1}, I_1, \ldots, T_{i_k}, I_k$ where the T_{i_j} are the relevant transition formulas in π and each I_j abstracts the intermediate sequences of irrelevant transition formulas between the T_{i_j}. Given the unsatisfiable trace formula $\psi \wedge T_0 \wedge \ldots \wedge T_{n-1}^{\langle n-1 \rangle} \wedge \phi^{\langle n \rangle}$, we can use a single call to an interpolating theorem prover to obtain a sequence of interpolants $I_0, \ldots, I_n$ such that each I_i is an error invariant for position i in the error trace. The basic idea underlying our algorithm is to use these interpolants as candidates for the inductive error invariants that occur in the computed abstract error trace. A naive algorithm to obtain the abstract error trace from the computed interpolants is to compute an $(n+1) \times (n+1)$ matrix $\mathcal{I}$ where an entry $\mathcal{I}_{ij}$ indicates whether interpolant I_i is an error invariant for position j of the error trace. The matrix $\mathcal{I}$ can then be used to obtain an abstract error trace by replacing maximal sequences of transition formulas $T_{i_1}, \ldots, T_{i_2}$ in the error trace by interpolant I_j, if I_j was found to be inductive for $i_1 < i_2$.

We have applied this naive algorithm to a number of example error traces and have found that it produces abstract error traces that concisely explain the cause of the error. The only problem with this naive algorithm is that it can be expensive: the number of theorem prover calls that is needed to compute the matrix $\mathcal{I}$ is quadratic in the length of the error trace. We have therefore developed an algorithm that obtains a better running

[4] Note that whenever we say *interpolant* we always mean *Craig interpolant*.

time. This improved algorithm is based on an observation that we made during the evaluation of the naive algorithm: if an interpolant I is an inductive error invariant for positions $i < j$ of an error trace, then it is also often an error invariant for all intermediate positions between i and j. Thus, instead of checking for each position i and interpolant I_j, whether I_j is an error invariant for i, we instead compute for each I_j the end positions of the interval of π on which I_j holds. Using a binary search, this can be done with fewer theorem prover calls than required by the naive algorithm.

Algorithm 1: Algorithm for computing abstract error traces.

Input: error trace $\tau = (\psi, \pi, \phi)$ of length n
Output: abstract error trace for τ

def $search(low : \mathsf{Int}, high : \mathsf{Int}, incLow : \mathsf{Int} \to \mathsf{Boolean}) : \mathsf{Int} = \{$
  if $(high < low)$ **return** low
  val $mid = (low + high)/2$
  if $(incLow(mid))$ $search(mid + 1, high, incLow)$
  else $search(low, mid - 1, incLow)$ $\}$
def $isErrInv(I : \mathsf{Formula}, i : \mathsf{Int}) : \mathsf{Boolean} =$
  $\text{valid}(\psi \wedge \mathbf{TF}(\pi[0, i]) \Rightarrow I^{\langle i \rangle}) \wedge \text{valid}(I \wedge \mathbf{TF}(\pi[i, n] \wedge \phi^{\langle n-i \rangle} \Rightarrow \bot)$
var $interpolants = \text{interpolate}(\mathbf{TF}(\tau))$
var $intervals = interpolants$ map $(\lambda I_j. \{ start = search(0, j, (\lambda i. \neg isErrInv(I_j, i)))$
  $end = search(j, n, (\lambda i. isErrInv(I_j, i))) - 1$
  $inv = I_j$ $\})$
var $sortedIntervals = intervals$ sortWith $(\lambda(a, b). a.start \leq b.start)$
var $maxInterval = sortedIntervals[0]$
var $prevEnd = 0$
for $(currInterval \leftarrow sortedIntervals)$ $\{$
  if $(currInterval.start > prevEnd)$ $\{$
    yield $maxInterval.inv$
    if $(maxInterval.end < n)$ **yield** $\pi[maxInterval.end]$
    $prevEnd = maxInterval.end$
    $maxInterval = currInterval$
  $\}$ **else if** $(currInterval.end > maxInterval.end)$ $maxInterval = currInterval$
$\}$

Algorithm 1 shows the pseudo code for our improved algorithm in a syntax akin to the Scala programming language. The algorithm takes an error trace τ as input and returns an abstract error trace for τ. It first computes a sequence of candidate error invariants *interpolants* by calling the interpolating theorem prover. It then computes for each interpolant I_j in *interpolants* a maximal interval on which I_j is inductive. Each interval is represented as a record with fields *start* and *end* storing the start and end position of the interval, and field *inv* storing the actual interpolant I_j. The interval boundaries are computed using a binary search that is implemented by the function *search*. The binary search is parameterized by a function *incLow*, which guides the search depending on which of the two interval bounds is to be computed. In either case, the function *incLow* is implemented using the function *isErrInv*, which checks

whether the given formula I is an error invariant for the given position i. The algorithm then computes a minimal subset of the intervals that cover all positions of the error trace. This is done by first sorting the computed intervals according to their start time and then selecting maximal intervals to cover all positions. The latter step is implemented in the final for-comprehension, which directly yields the error invariants and relevant transition formulas of the resulting abstract error trace.

Note that each binary search requires at most $\mathcal{O}(\log n)$ theorem prover calls, which gives $\mathcal{O}(n \log n)$ theorem prover calls in total (as opposed to $\mathcal{O}(n^2)$ for the naive algorithm). Also the sorting of the intervals can be done in time $\mathcal{O}(n \log n)$, which gives total running time $\mathcal{O}(n \log n)$, if we factor out the actual running time of the theorem prover calls.

6 Evaluation

We have implemented a prototype of Algorithm 1 on top of the interpolating theorem prover SMTInterpol [2] and applied it to compute abstractions of several error traces that we obtained from real-world programs. In the following, we present two of these examples in detail.

6.1 Faulty Sorting

Our first example is a faulty implementation of a sorting algorithm that sorts a sequence of integer numbers. This program is taken from [3] and shown in Figure 2. The program takes an array of numbers as input and is supposed to return a sorted sequence of these numbers. An error is observed when the program is called on the sequence $11, 14$. Instead of the expected output $11, 14$, the program returns $0, 11$. The corresponding error trace consists of the precondition $\psi \equiv (a[0] = 11 \wedge a[1] = 14)$, the post-condition $\phi \equiv (a'[0] = 11 \wedge a'[1] = 14)$ and the path π containing the sequence of 27 statements shown in Figure 3.

We translated each statement in path π into a corresponding transition formula. While there exist interpolation procedures for reasoning about arrays, SMTInterpol does not yet provide an implementation of such a procedure. We therefore encoded arrays using uninterpreted function symbols and added appropriate axioms for the array updates. Before calling the theorem prover, we instantiated all axioms with the ground terms occurring in the path formula. This resulted in an unsatisfiable quantifier-free formula which we used as input for our algorithm.

The theorem prover computed 28 interpolants, one for each position in the error trace. Table 1 shows the error invariant matrix for these interpolants. The matrix indicates for each computed interpolant I_i at which positions I_i is a valid error invariant. Note that the matrix is not actually computed by our algorithm. Instead, Algorithm 1 only computes for each I_i the boundaries of the interval of positions for which I_i is a valid error invariant. The marked interpolants I_1, I_{10}, I_{12}, I_{19}, I_{22}, and I_{26} are the ones that our algorithm selects for the computation of the abstract error trace. Thus, the only relevant statements for the error are the statements at positions 6, 11, 13, 20, 23, as well as the post-condition.

```c
static void shell_sort(int a[], int size) {
  int i, j;
  int h = 1;
  do {
    h = h * 3 + 1;
  } 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);
}
```

```c
int main(int argc, char *argv[]) {
  int i = 0;
  int *a = NULL;

  a = (int*)malloc((argc-1) *
                   sizeof(int));
  for (i = 0; i < argc - 1; i++)
    a[i] = atoi(argv[i + 1]);
  shell_sort(a, argc);

  for (i = 0; i < argc - 1; i++)
    printf("%d", a[i]);
  printf("\n");

  free(a);
  return 0;
}
```

Fig. 2. Faulty implementation of a sort algorithm taken from [3]. The faulty behavior can be observed, e.g., for the input value sequence $11, 14$

```
 0  int i,j, a[];
 1  int size=3;
 2  int h=1;
 3  h = h*3+1;
 4  assume !(h<=size);
 5  h/=3;
 6  i=h;
 7  assume (i<size);
 8  v=a[i];
 9  j=i;
10  assume !(j>=h && a[j-h]>v);
11  i++;
12  assume (i<size);
13  v=a[i];
14  j=i;
15  assume (j>=h && a[j-h]>v);
16  a[j]=a[j-h];
17  j-=h;
18  assume (j>=h && a[j-h]>v);
19  a[j]=a[j-h];
20  j-=h;
21  assume !(j>=h && a[j-h]>v);
22  assume (i!=j);
23  a[j]=v;
24  i++;
25  assume !(i<size);
26  assume (h==1);
```

Fig. 3. Error path π of the program in Figure 2 for the input sequence $11, 14$

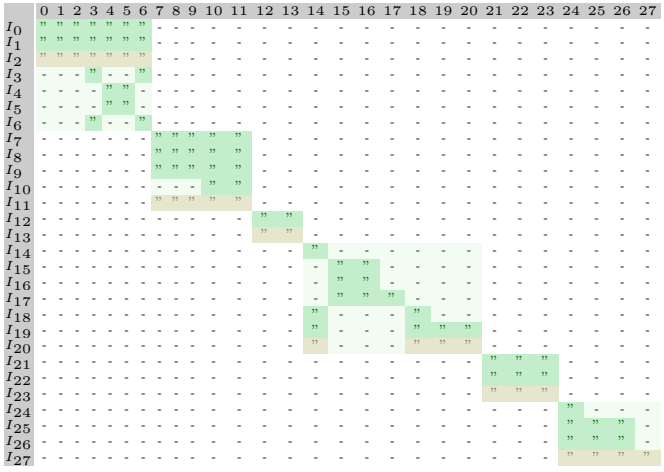

Table 1. Error invariant matrix for the error trace of the program in Figure 2

```
   a[2]=0
6: i=h;
   a[2]=0 ∧ h=1 ∧ i=h
11: i = i + 1;
   a[2]=0 ∧ h=1 ∧ i=2
13: v = a[i];
   h=1 ∧ i=2 ∧ v=0 ∧ h≤j ∧ j≤1
20: j = j - h;
   h=1 ∧ i=2 ∧ v=0 ∧ j=0
23: a[j] = v;
   a[0]=0
27: assert (a[0]=11 ∧ a[1]=14);
```

Fig. 4. Abstract error trace for the error path in Figure 3

The resulting abstract error trace is shown in Figure 4. We use boxed code such as `a[2]=0` to highlight the error invariants. From the first error invariant `a[2]=0` we can see that the only information we need to track until position 6 of our error trace is the value at index 2 of the array `a`. This error invariant is also the summary of the relevant part of the failing initial state ψ. Hence, we do not need to mention ψ explicitly in our abstract error trace. At position 6, the variable `i` is initialized. The error invariant no longer holds after this statement. The new error invariant now also states that `h=1` and `i=h` hold up to position 11. The next statement, which cannot be rendered irrelevant, is `i = i + 1` and sets `i` to 2. This statement already indicates the problem, as `i` should always be strictly smaller than the array bounds of `a`. The next error invariant now states that `i=2`. This error invariant holds up to position 13, where the value `a[2]` is stored in the local variable `v`. The new error invariant I_{20} keeps track of `v=0`, while the current content of the array `a` is completely irrelevant for the following parts of the error trace. The error invariant I_{20} also states that `j=1`. This property is temporarily violated from positions 14 to 17 but reestablished at position 18. The context of the variable `j` is abstracted away as all necessary information about `j` is provided by the error invariants. One could also think of a different algorithm that does not allow error invariants to be temporarily violated. Such an algorithm would further add the statements ℓ_{14} `:j=i` and ℓ_{17} `:j=j-h` to the error trace because we cannot find an invariant that holds at the enclosing positions of these statements. However, the relevant information about the variable `j` is provided by the error invariant. Hence, these statements can be omitted. The new error invariant holds up to position 20, when `j` is set to 0. This is recorded in the new error invariant, which holds up to position 23, when `a[0]` is finally set to `0`. This is the only information we need to keep track of for the remainder of the error trace, as it contradicts the post-condition. Hence, Algorithm 1 is able to reduce the error trace from its original 27 statements to only six statements. The computed error invariants further highlight the information about the state that is crucial for the error at each point of the error trace.

```
0  int OLEV = 600;
1  int MAXALTDIFF = 600;
2  int MINSEP = 300;
3  int NOZCROSS = 300;
4  int NO_INTENT = 0;
5  int DO_NOT_CLIMB = 1;
6  int DO_NOT_DESCEND = 2;
7  int TCAS_TA = 1;
8  int OTHER = 2;
9  int UNRESOLVED = 0;
10 int UPWARD_RA = 1;
11 int DOWNWARD_RA = 2;
12 int Positive_RA_Alt_Thresh = 740;
13 bool enabled = High_Confidence && (Own_Tracked_Alt_Rate <= OLEV) &&
       (Cur_Vertical_Sep > MAXALTDIFF);
14 bool tcas_equipped = (Other_Capability == TCAS_TA);
15 bool intent_not_known = (Two_of_Three_Reports_Valid&&(Other_RAC==NO_INTENT));
16 int alt_sep = UNRESOLVED;
17 assume(enabled && ((tcas_equipped && intent_not_known) || !tcas_equipped));
18 assume(Climb_Inhibit);
19 inhibitBiasedClimb = Up_Separation + NOZCROSS;
20 ownBelowThreat = (Own_Tracked_Alt < Other_Tracked_Alt);
21 ownAboveThreat = (Other_Tracked_Alt < Own_Tracked_Alt);
22 upward_preferred = (inhibitBiasedClimb > Down_Separation);
23 assume(upward_preferred);
24 nonCrossingBiasedClimb = !(ownBelowThreat) || ((ownBelowThreat) &&
                              (!(Down_Separation > Positive_RA_Alt_Thresh)));
25 need_upward_RA = nonCrossingBiasedClimb && ownBelowThreat;
26 upward_preferred = inhibitBiasedClimb > Down_Separation;
27 assume(upward_preferred);
28 nonCrossingBiasedDescend = ownBelowThreat && (Cur_Vertical_Sep >= MINSEP) &&
       (Down_Separation >= Positive_RA_Alt_Thresh);
29 need_downward_RA = nonCrossingBiasedDescend && ownAboveThreat;
30 assume !(need_upward_RA && need_downward_RA);
```

Fig. 5. Error path π for faulty TCAS produced by the model checker ULTIMATE

For comparison with the state of the art, we have also applied the tool Bug-Assist [13, 14] to the error trace in Figure 3. Bug-Assist returns a set of *potential bugs*, each of which is a statement in the input error trace. If ordered by their location, these statements form a reduced error trace. For our example, this reduced error trace still contains 18 statements.

6.2 Faulty TCAS

Our second example is a faulty implementation of the Traffic Alert and Collision Avoidance System (TCAS). TCAS is an aircraft collision detection system used by all US commercial aircraft. The TCAS example can be found in [8] and has been used in many papers to test algorithms that explain error traces (e.g., [10, 11, 13, 17, 19]). The error in this TCAS implementation is inflicted by a wrong inequality in the function `Non_Crossing_Biased_Climb()`. On some inputs, the error causes the Boolean variable `need_upward_RA` to become `true`. The effect is that the controlled aircraft will eventually rise even though its altitude is lower than the other aircraft's altitude. This may potentially lead to a collision.

To obtain an appropriate error trace for this error we applied the software model checker ULTIMATE [7] to the faulty TCAS implementation. The correctness condition

that exposes the error in the implementation has been taken from [4] and is as follows:

$$\texttt{need\_upward\_RA} \Rightarrow (\neg(\texttt{Up\_Separation} < \texttt{Positive\_RA\_Alt\_Thresh}) \wedge (\texttt{Down\_Separation} \geq \texttt{Positive\_RA\_Alt\_Thresh}))$$

This property is also the post-condition ϕ of our error trace (ψ, π, ϕ). When we checked this property with ULTIMATE, the model checker produced the error path π shown in Figure 5.

Note that the statement at position 24 is the problematic statement from function `Non_Crossing_Biased_Climb()` that causes the error. The strict inequality > should be replaced by >= for the implementation to be correct.

In order to obtain a suitable precondition ψ for our error trace, we used the SMT solver Z3 [6] to produce a model for the formula $\mathbf{TF}(\pi) \wedge \neg\phi$. We then encoded this model in a corresponding formula ψ. Applying our algorithm to the resulting error trace produces the abstract error trace shown in Figure 6. The error invariant matrix in Table 2 highlights the five interpolants that are selected for the abstract error trace.

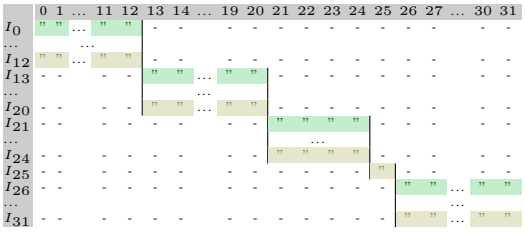

Table 2. Error invariant matrix for the error trace of the TCAS example

The abstract error trace shows how the infliction at position 24 affects the value assigned to `need_upward_RA` at position 25 and eventually leads to the error. The last error invariant forces the execution to take the `then` branch of the conditional, which is encoded as an implication in the post condition ϕ. The algorithm reduces the error trace from 31 to 4 statements. These statements are sufficient to understand the causality between the erroneous line and the error. The abstract error trace depends only on 7 instead of 37 variables. The number of input variables is reduced from 12 to 5. The abstract error trace thus significantly simplifies the search for the erroneous statement at position 24. For the purpose of comparison, we also ran Bug-Assist on the TCAS example. The reduced error trace thus obtained still contained 14 statements. We therefore believe that error invariants provide a valuable instrument that improves upon the state of the art.

7 Conclusion

We have introduced the concept of error invariants for reasoning about the relevancy of portions of an error trace. Error invariants provide a semantic argument why certain

```
Up_Separation = 441 ∧ Down_Separation = 740 ∧ Own_Tracked_Alt = −1 ∧
Other_Tracked_Alt = 0
12:Positive_RA_Alt_Thresh = 740;
Positive_RA_Alt_Thresh = 740∧ Up_Separation = 441 ∧Down_Separation = 740 ∧
Own_Tracked_Alt = −1 ∧ Other_Tracked_Alt = 0
20:ownBelowThreat = Own_Tracked_Alt < Other_Tracked_Alt;
ownBelowThreat ∧Positive_RA_Alt_Thresh = 740 ∧Up_Separation = 441 ∧
Down_Separation = 740 ∧ Own_Tracked_Alt = −1 ∧ Other_Tracked_Alt = 0
24:nonCrossingBiasedClimb = !ownBelowThreat ||
    (ownBelowThreat && (!(Down_Separation > Positive_RA_Alt_Thresh))));
nonCrossingBiasedClimb ∧ownBelowThreat ∧ Positive_RA_Alt_Thresh =740 ∧
Up_Separation = 441 ∧ Down_Separation = 740
25:need_upward_RA = nonCrossingBiasedClimb && ownBelowThreat;
need_upward_RA ∧ Positive_RA_Alt_Thresh = 740 ∧ Up_Separation = 441 ∧
Down_Separation = 740
31:assert (need_upward_RA ==> !(Up_Separation < Positive_RA_Alt_Thresh) &&
          (Down_Separation >= Positive_RA_Alt_Thresh));
```

Fig. 6. Abstract error trace of the TCAS example

portions of an error trace are irrelevant to the search for the cause of an error. Removing those irrelevant portions from the error trace will not alter the observable error. This is in contrast to related static approaches for slicing error traces that are based on computing unsatisfiable cores of extended path formulas. We have presented an algorithm that synthesizes error invariants from Craig interpolants and uses them to obtain compact abstractions of error traces. Our evaluation has shown that our algorithm can indeed help programmers understand the cause of an error more easily. We therefore believe that our algorithm will be a useful component in future debugging tools.

We see many opportunities to further improve the performance of the presented algorithm, which will be subject to our future work. For instance, our approach can be used on already reduced error traces, to further compress them. Also, many theorem prover calls during the binary search can be avoided by first syntactically checking whether a candidate invariant speaks about variables that do not occur in both the prefix and suffix of the trace. In this case, it is not necessary to invoke the theorem prover because the candidate invariant cannot be a Craig interpolant. Further optimizations are possible if the theorem prover is not treated as a black box. In particular, we will explore different approaches to compute Craig interpolants from unsatisfiable path formulas. If we have more control over the structure of the computed interpolants, this will allow us to build more efficient algorithms for computing inductive error invariants.

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. J. Christ and J. Hoenicke. Instantiation-based interpolation for quantified formulae. In *SMT Workshop Proceedings*, 2010.

3. H. Cleve and A. Zeller. Locating causes of program failures. In *ICSE'05*, pages 342–351, 2005.
4. A. Coen-Porisini, G. Denaro, C. Ghezzi, and M. Pezze. Using symbolic execution for verifying safety-critical systems, 2001.
5. W. Craig. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *The Journal of Symbolic Logic*, pages 269–285, 1957.
6. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS'08*, pages 337–340. Springer, 2008.
7. E. Ermis, J. Hoenicke, and A. Podelski. Splitting via interpolants. In *VMCAI'12*, pages 186–201. Springer, 2012.
8. T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. *ACM Trans. Softw. Eng. Methodol.*, pages 184–208, 2001.
9. A. Groce. Error explanation with distance metrics. In *TACAS'04*, pages 108–122, 2004.
10. A. Groce and D. Kroening. Making the Most of BMC Counterexamples. *ENTCS*, pages 67–81, 2005.
11. A. Groce, D. Kroening, and F. Lerda. Understanding counterexamples with explain. In *CAV'04*, pages 453–456, 2004.
12. R. Jhala and K. L. Mcmillan. Interpolant-based transition relation approximation. In *CAV'05*, pages 39–51. Springer, 2005.
13. M. Jose and R. Majumdar. Bug-Assist: Assisting Fault Localization in ANSI-C Programs. In *CAV'11*, pages 504–509, 2011.
14. M. Jose and R. Majumdar. Cause clue clauses: error localization using maximum satisfiability. In *PLDI '11*, pages 437–446. ACM, 2011.
15. K. R. M. Leino, T. Millstein, and J. B. Saxe. Generating error traces from verification-condition counterexamples. *Sci. Comput. Program.*, pages 209–226, 2005.
16. K. L. McMillan. An interpolating theorem prover. *Theor. Comput. Sci.*, pages 101–121, 2005.
17. D. Qi, A. Roychoudhury, Z. Liang, and K. Vaswani. Darwin: an approach for debugging evolving programs. In *ESEC/SIGSOFT FSE*, pages 33–42, 2009.
18. M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *ASE*, pages 30–39, 2003.
19. C. Wang, Z. Yang, F. Ivancic, and A. Gupta. Whodunit? causal analysis for counterexamples. In *ATVA'06*, pages 82–95, 2006.
20. A. Zeller. Isolating cause-effect chains from computer programs. In *SIGSOFT FSE*, pages 1–10, 2002.
21. X. Zhang, H. He, N. Gupta, and R. Gupta. Experimental evaluation of using dynamic slices for fault location. In *AADEBUG'05*, pages 33–42. ACM, 2005.