Fault Localization for Formal Verification: An Implementation and Evaluation of Algorithms based on Error Invariants and UNSAT-cores

Bachelor's Thesis in Computer Science

01.07.2020

Ludwig-Maximilians-Universität München

Matthias Kettl

Supervisor: Prof. Dr. Dirk Beyer
Mentor: Thomas Lemberger

Abstract

Debugging is a very time intensive aspect of software development. To find the root cause of a bug earlier, techniques like fault localization are used. Speeding up the detection of faults and improving the correctness of fixes saves valuable time. The reduction of programs to a few error-prone statements minimizes the locations to look at. Such a minimal subset of locations in the program indicates where fixes are necessary. Therefore, we implement and improve three fault localization algorithms based on error invariants and UNSAT-cores in the CPAchecker-framework. These algorithms reduce the number of locations to look at and search for explanations why the faults arise. At the beginning, we let CPAchecker compute a counterexample and build a trace formula which is a Boolean formula representing a failing execution through a program. With its help the algorithms enclose error-prone locations in the source code of the program. Since the algorithms need to call a SMT-solver several times during the computation of the fault, the run time is bad. Hence, we apply improvements such as memoization to reduce the number of calls to the solver. Moreover, we extend the standard algorithms with useful options, which yields more freedom in influencing and guiding the results towards the desired direction. We attach additional information to the found locations either based on the computations of the algorithm or on patterns and schemes. We evaluate the run time, effectiveness and the usability of the mentioned techniques. Additionally, we implement a data structure for arbitrary fault localization algorithms in the CPAchecker-framework including a generic and interactive visualization. Our user study shows that fault localization boosts the performance of programmers in debugging on unknown and faulty tasks. Furthermore, the visual report improves the correctness and the needed time to solve a task. The qualitative analysis shows that fault localization sensibly reduces locations to look at although there exist cases where the results are misleading.

Contents

List of Algorithms

List of Figures

List of Tables

List of Programs

1 Introduction

Debugging is an essential part of a programmers everyday life. It takes a considerable amount of the overall time spent on a project [14]. Therefore many techniques for finding bugs developed over the recent years. Currently, most of the programmers use a debugger with breakpoints and watches to go through the program step by step and analyze the current states of the program. This requires the developer to analyze every position and to decide whether a certain statement can cause an error. Additionally, the programmer must know for what inputs and edge-cases the program will fail eventually. In small and simple programs the developer might be able to observe all states and work out all edge-cases, but in larger projects this is difficult.

Formal verification can help to tackle this problem as it analyzes the whole program and its reachable states with all possible variable assignments. CPAchecker [2] is an excellent tool for formal verification of ANSI-C programs [1]. It has a variety of already implemented analysis methods such as symbolic execution [17] or predicate analysis [11]. Furthermore, it can handle non-deterministic variables. This is especially handy for finding specific cases with faulty behavior. Moreover, the tool can create counterexamples consisting of edges along the control flow automaton (CFA). The obtained list of edges describes an error path through the program which already limits the selection of edges to track.

To create an error path we need a post-condition that should never be violated. Program 1 reaches the error label for `x := 2`. Now we let CPAchecker generate the counterexample. The result is a list of edges and a failing variable assignment. The counterexample is a subset of all possible transitions. Program statements that do not operate on `x` have no impact on the fault and can be discarded. We are left with the observation of lines 5, 8, 9 and 13. In line 13, the if-statement checks if the resulting `x` has an valid value. Whenever this last assertion - the post-condition - is violated the program fails. The precondition, in this case, equals `x := 2` because for all other assignments the program calculates a correct index `x`. The precondition is either the initial variable assignment or a failing variable assignment for all

```
1  int array[3] = {1,2,3};
2
3  int find(int x) {
4     /** many operations */
5     if (x < 0) {
6        x = 0;
7        /** many operations */
8     } else if (x >= 2) {
9        x = x + 1;
10       /** many operations */
11    }
12    /** many operations */
13    if (x > 2 || x < 0)
14       error();
15    return array[x];
16 }
```

Program 1: Index Calculation

non deterministic variables, like `x` here. Now that we have a small selection of edges that can potentially fix the error we either want to find an abstraction of the error trace which simultaneously provides an explanation of the arising fault or reduce the selection by a significant amount. The SingleUnsatCore algorithm calculates a single, not necessarily minimal set of edges that is unsatisfiable when conjuncted with the pre- and post-condition whereas our adaption of the MaxSat algorithm [16] detects all minimal subsets of edges for which the precondition conjugated with the minimal set and the post-condition is unsatisfiable. Lastly, the ErrInv algorithm [9] produces an alternating sequence of interpolants and edges, called *abstract error trace*. The interpolants abstract the error path because they cover the execution of multiple edges with one formula that summarizes the semantics of all the covered edges.

This thesis covers the implementation of these three algorithms that abstract or refine the selection of edges provided by the counterexample. We compare the performance and evaluate the usability of the visual report and effectiveness of the results. Finally, we apply optimizations to improve the run time and add useful options such as an alternative precondition.

2 Related Work

Tarantula [15]. Test suites provide data about the source code, inter alia the coverage of executed parts of the code. Never reached code indicates the need of new test cases. The inputs for the algorithm are several test cases with a flag stating whether the test case failed and a list of covered lines during their execution. After running the algorithm, lines are colored according to their meaning based on the overall coverage in the test cases. Discrete coloring, for example, colors statements that are exclusively executed during failing tests in red. However, the continuous approach yields much better results by adding a brightness component and a continuous function for coloring lines. The computation now relies not only on simple conditions but involves the relative frequency of certain locations in failed and succeeding test cases.

Dynamic slicing [19, 22]. There exist two types of dynamic slices, backward dynamic slices and forward dynamic slicing. A backward slice of a variable maintains a sequence of executed statements that influenced the value of the variable at a certain position in the execution trace whereas a forward slice maintains a sequence of statements that are influenced by this variable at this position. In general, a dynamic slice describes a faulty subset of all executed statements. Presented are three slicing algorithms, whose goal it is to find a subset of executed program statements that had an impact on the faulty value. The three algorithms *data slicing, full slicing* and *relevant slicing* pursue different approaches in obtaining such subsets.

Distance metrics [12, 13]. In few steps this technique acquires descriptive information about the error. The algorithm relies on loop unrolling. First, a bounded model checker transforms a buggy program to a Boolean satisfiability problem with correct static single assignment indices (c.f. Section 3.5) and calculates a counterexample. The target is to obtain a satisfying assignment for the program that does not reach an ERROR-label while simultaneously producing an execution as close to the counterexample as possible.

A distance metric Δd calculates the difference between the executions of the failing and the satisfying algorithm. After a slicing step which reduces Δd the differences are returned to the user as promising locations to look at. As the metric must comply to the four properties of a distance metric, a unique representation of such executions has to be found.

Delta debugging [21]. A different approach compared to the previous techniques is called "Delta Debugging". Finding minimal failing test cases is often desirable to detect the root cause of a bug. Usually this is done manually. Delta debugging describes the process of automating the search for minimal test cases. Based on a failing test case, program statements are systematically removed and readded, depending on the error still occurring or not. This method easily can be applied to arbitrary programs of different programming languages. The technique requires a *minimizing delta debugging function* that finds a minimal failing test case from a given failing test case. An extension of this algorithms computes the difference between failing and passing tests.

3 Background

3.1 UNSAT Cores and Models

SMT solvers [18] resolve Boolean satisfiablity problems of first-order logic. CPAchecker has access to multiple SMT solvers and an integrated prover. The solver is able to generate UNSAT cores and to check whether a formula is unsatisfiable whereas the prover calculates models of formulas.

A formula is unsatisfiable if there does not exist any variable assignment that makes the formula true. For instance, we cannot find a valid value for x in $x > 0 \wedge x < 0$ because no number is smaller and greater than 0 at the same time. In this work we mainly focus on Boolean formulas in CNF (conjunctive normal form), meaning that clauses are separated by logical ands ($\wedge$). We are especially interested in minimal UNSAT cores (MIN-UNSAT cores). Consider an unsatisfiable Boolean formula f_B with n clauses. A UNSAT core is a selection of $m \leq n$ clauses of f_B where their conjunction remains unsatisfiable. Consequently, a MIN-UNSAT core is an UNSAT core with a minimal number of clauses meaning, that an UNSAT core with fewer clauses does not exist. Concomitant a MAX-SAT set includes the maximal number of clauses that are simultaneously satisfiable. For example the clauses $\{(x < 0), (y > 3), (x > 0)\}$ of the formula

$$f_B \Leftrightarrow x < 0 \wedge y = 5 \wedge y > 3 \wedge x > 0 \wedge y < 4$$

are an UNSAT core of f_B and the only two MIN-UNSAT cores are given by the clauses $\{(x < 0), (x > 0)\}$ and $\{(y = 5), (y < 4)\}$. A MAX-SAT set is, for example, given by $\{(x < 0), (y = 5), (y > 3)\}$. Note that all elements of the complement of this set are elements of MIN-UNSAT cores, too.

The prover used by CPAchecker can also calculate a model of a formula. A model M assigns concrete values to each occurring variable in a Boolean formula f_B in a way that $M \models f_B$ (M satisfies f_B)[4].

3.2 Control Flow Automaton

Before the analysis starts, CPAchecker parses the program to a control flow automaton (CFA). The CFA is a directed, not necessarily circular graph from the first executed program statement to the exit point. The children of every node are all reachable states from the current state.

Formally, a CFA is a triple (L, l_0, G) where L is the set of all reachable locations, $G \subseteq L \times O \times L$ and l_0 denotes the start location. An element $g \in G$ is called a transition. O is the set of operations. A path $\Pi = (l_0, o_0, l_1), (l_1, o_1, l_2), \ldots, (l_{n-1}, o_{n-1}, l_n)$ $(n \in \mathbb{N})$ through the CFA is expressed as a sequence of transitions.

After processing a program we obtain a graph showing all possible executions starting at l_0. The nodes symbolize the current location and the edges represent a transition, i.e., a program statement that leads from one node to one of its child nodes.

We now extend the definition of a CFA by error locations and we get a quadruple $CFA_E = (L, l_0, G, E)$. $E \subseteq L$ contains all reachable error locations. An error path Π_E can now be defined as path from l_0 to an $e \in E$. With $\Pi[i]$ we refer to the i-th $(i \in [0; n])$ transition of path Π. A subpath containing all transitions form position i to $j - 1$ is indicated as $\Pi[i, j]$ with $0 \leq i < j \leq n$. The path Π is called feasible if there exists an execution that covers all of the transitions in the path in the correct order.

3.3 Counterexample

A counterexample is found by the CEGAR algorithm [6] whenever a program has a reachable error location. CPAchecker generates a counterexample which can be understood as a minimized proof of incorrectness of a program. The proof consists of a sequence of edges along the CFA ending in an error state. All of the three fault localization algorithms are based on the resulting counterexample.

3.4 Trace

Every edge can be represented as a Boolean formula. For simplicity we reduce the possible operations to declarations (`int x = 5;`), statements (`x = x + 1;`) and assumes (`x ≥ 6`) in this chapter. Constructs like arrays, structs and pointer are handled by the implementation. Every edge $g \in G$ can be transformed to a Boolean formula based on its operation $o \in O$. Furthermore

the last edge before reaching $e \in E$ is guaranteed to be an assume edge. Of course, in reality this is not applicable for every program. The handling of these cases will be discussed in Chapter 6.

```
1  int test(int x, int b) {
2      if (x == 0) {
3          x = x + 1;
4      } else {
5          b = b - 1;
6      }
7      x = 5 + b;
8      if (x == 14) {
9          error();
10     }
11     return 0;
12 }
```

Program 2: Simple Example

We define the trace formula based on [9, 16]. The trace $\tau = (\psi, \pi, \phi)$ of an error path Π_E with $n+1$ transitions consists of the triple (ψ, π, ϕ). The first part ψ is a failing variable assignment or more generally a model of $\pi \wedge \neg\phi$. In Program 2 the execution calls `error()` in line 9 if `x` := `1` and `b` := `10` so consequently $\psi \Leftrightarrow (x = 1) \wedge (b = 10)$. Secondly, π describes the executed path to the error location. Hence, it consists of the first n transitions of Π_E. The last transition becomes the post-condition ϕ by $\phi = \neg\Pi_E[n+1]$. This is the reason why the last edge has to be an assume edge. The trace τ has length n. The trace formula `TF` of a trace τ is expressed as

$$\texttt{TF}(\tau) \Leftrightarrow \psi \wedge \bigwedge_{i=0}^{n-1} \pi[\texttt{i}] \wedge \phi$$

and therefore, the trace formula for Program 2 equals

$$\begin{aligned}\texttt{TF}(\tau) \Leftrightarrow & \underbrace{(x = 1) \wedge (b = 10)}_{\psi} \\ & \wedge \underbrace{\neg(x == 0) \wedge (b' = b - 1) \wedge (x' = 5 + b')}_{\text{“}\pi\text{”}} \\ & \wedge \underbrace{\neg(x' == 14)}_{\phi}.\end{aligned}$$

From now on, we use $\texttt{TF}(\pi) = \texttt{TF}(\pi[\texttt{0};\texttt{n}]) = \texttt{TF}((\top, \pi, \top))$ as a shortened notation for the conjunction of the actual execution path π.

As mentioned above the post-condition of a feasible error path is the negation of the last assume operation before reaching the error label. This implies that $\texttt{TF}(\tau)$ has to be unsatisfiable by construction.

3.5 SSA-Map

After updating a variable we add a prime to the variable to keep track of which changes may be error-prone. Let X be the set of variables, then for every $x \in X : x^{\langle j \rangle}$ denotes the variable at time j of the path π of length n with $0 \leq j \leq n$. Static single assignment maps (SSA-maps) store the current static single assignment index (SSA-index) of all variables in X. The index is equal to the number of primes added to a variable. We represent such a map by extending the notation to sets. Therefore, $X^{\langle j \rangle}$ contains all variables at time j. For Program 2, $X^{\langle 2 \rangle} = \{x^{\langle 2 \rangle}, b^{\langle 2 \rangle}\} = \{x, b'\}$. It follows, that the SSA-index of x equals 0 and the SSA-index of b equals 1. In addition, for a Boolean formula f_B, $f_B^{\langle j \rangle}$ replaces all its occurrences of a variable $x \in X$ by the corresponding $x \in X^{\langle j \rangle}$. For instance, if $f_B \Leftrightarrow x = 0 \wedge b = 3$, then $f_B^{\langle 2 \rangle} \Leftrightarrow x = 0 \wedge b' = 3$ with $X^{\langle 2 \rangle} = \{x, b'\}$ as above. Until now we implicitly assumed that the SSA-indices are correctly assigned when creating the trace formula but for the algorithms it is important to explicitly indicate them. Thus, the correct formulation of the trace formula is expressed as:

$$\mathtt{TF}(\tau) = \psi \wedge \bigwedge_{i=0}^{n-1} \pi[\mathtt{i}]^{\langle \mathtt{i} \rangle} \wedge \phi^{\langle n \rangle}.$$

3.6 Single-UNSAT-Core Algorithm

Algorithm 1 depicts the core concept of the following algorithms: MaxSat and ErrInv. They will later extend and refine (SingleUnsatCore) to achieve better results and to give better explanations.

Algorithm 1: SingleUnsatCore

Input: ψ, π, ϕ: Boolean Formula
Output: C: candidate set of Boolean Formulas
Result: Subset of clauses of π where adapting the corresponding lines/locations in the program can fix the bug.

1 $\mathtt{tf} = \psi \wedge \bigwedge_{i=0}^{n-1} \pi[\mathtt{i}]^{\langle \mathtt{i} \rangle} \wedge \phi^{\langle n \rangle}$;
2 `C = solver.unsatCore(tf);`
3 **return** `C`;

Algorithm 1 shows the basic implementation for calculating a single UNSAT core. We obtain a subset $C \subseteq \{\psi, \phi^{\langle n \rangle}\} \cup \bigcup_{i=0}^{n-1} \{\pi[i]^{\langle i \rangle}\}$ of clauses where $\psi \wedge$

```
1  int simple (int input) {
2     int x = input;
3     if (x > 0) {
4        if (x < 5) {
5           if (x > 1) {
6              if (x != -1 && x != 6) {
7                 error();
8              }
9           }
10       }
11    }
12 }
```

Program 3: Simple checks

$\left(\bigwedge_{c \in C} c\right) \wedge \phi^{\langle n \rangle}$ already is unsatisfiable. C can be interpreted as a reduction of possible bug locations.

To explain the behavior of SingleUnsatCore, we start with Program 3. Setting `input` := `2` forces the program to call the function `error`() which means the program has a bug. To figure out promising locations where changes fix the program, we run Algorithm 1 with the inputs:

ψ: $\texttt{input} = 2$

π: $\{\texttt{x} = \texttt{input}, x > 0, x < 5, x > 1\}$

ϕ: $\neg(x \neq -1 \wedge x \neq 6) \Leftrightarrow (x = -1 \vee x = 6)$.

The trace formula `tf` is equivalent to the conjunction of ψ, all elements of π and ϕ (Algorithm 1, line 1). We do not need SSA-indices here since the values of x or `input` are never updated. Next, we let the solver calculate one of the possible UNSAT cores of `tf` and return it to the user. For Program 3 the set $\texttt{C} = \{(\texttt{x} < \texttt{5}), (\texttt{x} > \texttt{1})\}$ is returned. With this two constraints the possible values for `x` are $2, 3$ and 4 which contradicts the post-condition that `x` either must have the value -1 or 6. Now, the task for the developer is to make the correct adaptions to the given and reduced result set. Replacing line 4 with $\texttt{if}(\texttt{x} < \texttt{2})$ resolves the bug which, in this constructed example is a valid fix but in real world programs it, of course, might not be suitable because we ignore the semantics. This particular problem will be part of Chapter 6. Calculating MIN-UNSAT cores only works if $\psi \wedge \phi^{\langle n \rangle}$ is satisfiable. Otherwise, the MIN-UNSAT core in terms of clauses of π would result in the empty set. The algorithm exactly determined the error-causing clauses. Additionally, we gained the information that, whenever the program reaches line 6 there

is no way to satisfy the post-condition and therefore an adaption of either line 4 or line 5 or both lines is needed. Here, the proposed solution for line 4 fixes the bug.

3.7 MAX-SAT Algorithm

3.7.1 Selector

For calculating *maximal satisfiable subsets* (MSS) we introduce the concept of selector variables labeled λ_i for every clause in π [16]. Hence, we enhance the construction of the trace formula as follows:

$$\mathtt{TF}_{\Lambda}(\tau) \Leftrightarrow \psi \wedge \bigwedge_{i=0}^{n-1} (\lambda_i \Rightarrow \pi[\mathtt{i}]^{\langle i \rangle}) \wedge \phi^{\langle n \rangle},$$

offering the opportunity to switch on and off specific clauses of π. This can be done by assigning either `true` or `false` to a selector variable λ_i.

If λ_i is `true`, then $(\lambda_i \Rightarrow \pi[\mathtt{i}]) \Leftrightarrow (\mathtt{true} \Rightarrow \pi[\mathtt{i}]) \Leftrightarrow \pi[\mathtt{i}]$ and in consequence $\pi[i]$ must be considered when the solver runs the satisfiability check. Otherwise, if λ_i is `false`, $(\lambda_i \Rightarrow \pi[\mathtt{i}]) \Leftrightarrow (\neg\mathtt{false} \vee \pi[i]) \Leftrightarrow (\mathtt{true} \vee \pi[i])$ simplifies to `true` and $\pi[i]$ will be ignored. As long as no concrete value is assigned to a selector, we call it "free". Trace formulas using selectors are indicated by adding Λ as an index to $\mathtt{TF}_{\Lambda}$. Every selector can be mapped to the line and location of the clause it implies. For proofing a formula satisfiable the solver tries to find a proper assignment of all free variables. Thus, the absence of such an assignment states that the formula is unsatisfiable. Note that $\mathtt{TF}_{\Lambda}$ always is satisfiable if $\psi \wedge \phi^{\langle n \rangle}$ is satisfiable, whereas $\mathtt{TF}_{\Lambda}(\tau) \wedge \bigwedge_{\lambda \in M} \lambda$ is unsatisfiable if M is an UNSAT core. The problem we want to solve now is to find a MSS of selectors to which we can assign the value `true`, i.e., $\mathtt{TF}_{\Lambda}(\tau) \wedge \bigwedge_{\lambda \in MSS} \lambda$ is satisfiable.

3.7.2 Soft and Hard Clauses

The clause $\lambda_i \Rightarrow \pi[i]$ automatically becomes satisfiable because - while checking for satisfiability - the solver only needs to assign `false` to the free selector. This works every time since a selector only appears once in $\mathtt{TF}_{\Lambda}$ and will therefore not inflict problems somewhere else. Clauses with free selectors are called soft clauses because the solver can choose the fitting assignment for this clause.

Clauses that should be regarded are labeled as hard clauses. For instance the precondition as well as the post-condition do not get selectors and consequently are always marked as hard. To mark free selectors of the index set $\mathbb{I}$ as hard, it is sufficient to append "$\bigwedge_{i\in\mathbb{I}} \lambda_i$" to the trace formula $\mathtt{TF}_\Lambda$. The solver now has no other choice than setting all $\lambda_i, i \in \mathbb{I}$ to `true` because otherwise the formula $\mathtt{TF}_\Lambda \wedge \lambda_i \Leftrightarrow \mathtt{TF}_\Lambda \wedge \mathtt{false} \Leftrightarrow \mathtt{false}$ becomes unsatisfiable. Hence, the solver requires the selector to have the value `true` which is because of the implication equivalent to mark the clause as hard.

In summary we now have a extended trace formula $\mathtt{TF}_\Lambda$ where the pre- and the post-condition are marked as hard and the selectors are marked as soft initially.

3.7.3 The Algorithm

Algorithm 2: MaxSat (adapted from [16])

Input: ψ, π, ϕ: Boolean Formula
Output: H: Set of sets of Boolean Formulas
Result: Subsets of clauses of π where adapting the corresponding lines/locations in one of the sets can fix the bug.

1 $S = \{\lambda_0, \cdots, \lambda_{n-1}\}$; `// soft set`
2 $H = \{\}$; `// stores all found CoMSS`
3 $\mathtt{tf} = \psi \wedge \bigwedge_{i=0}^{n-1} (\lambda_\mathtt{i} \Rightarrow \pi[\mathtt{i}]^{\langle \mathtt{i} \rangle}) \wedge \phi^{\langle n \rangle}$;
4 **while** *true* **do**
5 $\overline{M} = \mathtt{CoMMS}(\mathtt{tf}, S)$;
6 **if** $\overline{M} == \emptyset$ **then**
7 `break`;
8 $h = \bigvee_{m\in\overline{M}} m$;
9 $\mathtt{tf} = \mathtt{tf} \wedge \mathtt{h}$;
10 $H = H \cup \{\overline{M}\}$;
11 $S = S \setminus \overline{M}$;
12 **return** H ;

MaxSat [16] calculates every possible MSS of selectors of $\mathtt{TF}_\Lambda$ starting with the largest set and ending with the smallest one. All found sets share one common property: the complement of the MSS, namely $\overline{M} = S \setminus MSS$ is a subset of an UNSAT core. The advantage of finding all possible $\overline{M}$ lies in the fact that the user has access to several promising and minimal sets of locations. The complements $\overline{M}$ are minimal because MSS is maximal [16].

Every set provides locations where changes can fix the bug although some of them might not be suitable for a proper fix, e.g, if the programmer does not want to change this line. Thus, having multiple selections of locations gives access to other possible approaches on fixing the bug. Algorithm 2 shows the basic implementation of MaxSat using a *p-MAX-SAT-solver*. Such a solver allows to pass hard and soft clauses as arguments and it returns a MSS of soft clauses. The letter p abbreviates the word partial as it only returns subsets of soft set clauses and no hard clauses. In this setting this means, we search for a *MSS* with maximal cardinality such that $\texttt{TF}_\Lambda \wedge \bigwedge_{m \in MSS} m$ remains satisfiable. The complement $\overline{M}$ of MSS has to be a subset of an UNSAT core otherwise elements of $\overline{M}$ could have been added to the **satisfiable** set *MSS*. $\overline{M}$ is calculated by a `CoMSS` solver which implements a p-MAX-SAT-solver and returns the complement. `CoMSS` translates to "complement of maximal satisfiability subset".

Now we take a look at the algorithm itself. First, we create all selectors and store them in the set S. We proceed to calculate the first part (`tf`) of the trace formula, which exactly matches $\texttt{TF}_\Lambda$ and pass it together with the conjunct of all available selectors S to the `CoMSS`-solver. The selectors are marked as soft. Note that $\texttt{tf} \wedge \bigwedge_{\texttt{s} \in \texttt{S}} \texttt{s}$ is unsatisfiable. The while-loop ends as soon as the newly returned minimal set $\overline{M}$ is empty. In line 5 we compute the first MSS with respect to the selectors S and the already found complements contained in H. We do not pass H as argument as the subsets are also added to the trace formula in line 9. Appending h to the trace formula ensures that we cannot compute the same complement with elements of S twice. To proof the correctness of this we take a look at the functionality of the `CoMSS`-solver. Assume we have an arbitrary program where λ_0, λ_1 and λ_3 are elements of $\overline{M}$ in the first iteration. We update `tf` with h resulting in $\texttt{tf}' = \texttt{tf} \wedge (\lambda_0 \vee \lambda_1 \vee \lambda_3)$. Together with the soft clauses this formula is passed to the `CoMSS`-solver which tries to calculate the next MSS. We see that the solver now has to set at least one of the three selectors to `true` to ensure satisfiability. The next found MSS now contains a non empty subset of $\{\lambda_0, \lambda_1, \lambda_3\}$ together with other selectors and thus the complement of the set can never again be equal to $\{\lambda_0, \lambda_1, \lambda_3\}$. In line 11 we remove all selectors in $\overline{M}$ from the set of selectors S because otherwise we produce an endless loop. If we remove line 11, then every selectors is always marked as soft. After we added all possible $\overline{M}$ to the formula we might be at an point where the formula is unsatisfiable no matter which Boolean values are assigned to the selectors. The complement of the empty set would always be S. Thus, we would add the clause $\bigvee_{s \in S} s$ infinitely many times to `tf` and never exit the loop.

3.8 Error Invariants Algorithm

3.8.1 Error Invariants and Craig Interpolants

We define error invariants based on the idea of Craig interpolants [7]. To obtain an error invariant, we firstly specify a Craig interpolant I for an unsatisfiable formula $f_C \Leftrightarrow f_A \wedge f_B$ by the following three properties:

1) $f_A \Rightarrow I$ is valid, meaning $\neg(f_A \Rightarrow I)$ is unsatisfiable,

2) $I \wedge f_B$ is unsatisfiable,

3) free variables in I are also free in f_A and f_B.

We know that $\mathtt{TF}(\tau)$ of length n of an error path is unsatisfiable, enabling us to set $f_C = \mathtt{TF}(\tau)$. Splitting the trace formula on position $i \in [0; n] \cap \mathbb{N}$ yields the formulas for f_A and f_B by $f_A \Leftrightarrow \psi \wedge \mathtt{TF}(\pi[\mathtt{0}; \mathtt{i}])$ and $f_B \Leftrightarrow \mathtt{TF}(\pi[\mathtt{i}; \mathtt{n}]) \wedge \phi^{\langle n \rangle}$. If I_i is a Craig interpolant on position i with this definition of f_A and f_B, the interpolant is considered to be an error invariant [9]. In this work we often refer to *error invariants* with *inductive interpolants*.

For example, the trivial interpolant for position $i = 0$ directly emerges from the precondition ψ. Let $i = 0$, then $I_0 \Leftrightarrow \psi$ because consequently $f_A \Leftrightarrow \psi \wedge \mathtt{TF}(\pi[\mathtt{0}; \mathtt{0}]) \Leftrightarrow \psi$ and $f_B \Leftrightarrow \mathtt{TF}(\pi[\mathtt{0}; \mathtt{n}]) \wedge \phi^{\langle n \rangle}$. The implication $f_A \Rightarrow I_0$ holds because $\psi \Rightarrow \psi$ and $I_0 \wedge f_B$ being, in this case, equivalent to $\mathtt{TF}(\tau)$ is unsatisfiable by construction. We can always find an interpolant for unsatisfiable first-order logical formulas.

3.8.2 Abstract Error Trace

A big advantage of the *error invariants algorithm* (ErrInv) [9] compared to the other presented algorithms is, that it produces explanations for every location in the program on the fly by computing an abstract error trace. We first sketch the idea to then formalize the process.

In a first step, the algorithm asks the solver to return an interpolant for every $\pi[i], 0 \leq i < n$ in the trace τ of length n. The interpolants can be understood as reasons of why this certain location eventually leads to the error. We will see that some interpolants hold for multiple locations in the trace. The traget is to find the maximal interval where the interpolant holds, i.e., is inductive. For instance, if I is an interpolant for position i and the last time for position $k \geq i$, I is an inductive interpolant in the interval $[i; k]$. I now abstracts all locations from i to k. In an informal way we can simply express whatever happens between locations i and k as I. This leads to

an alternating sequence of interpolants and actual locations of the program building the abstract error trace $\tau^{\#}$.

We adopt the following definition of $\tau^{\#}$ from [9]. Formally, the abstract error trace $\tau^{\#} = (\psi, \pi^{\#}, \phi)$ is an abstraction of $\tau = (\psi, \pi, \phi)$ where $\pi^{\#}$ precisely explains why π is failing in combination with the pre- and the post-condition. It must hold that for every n-step execution σ of τ we can find an m-step execution $\sigma^{\#}$ of $\tau^{\#}$ such that $\sigma^{\#}[0] = \sigma[0]$, $\sigma^{\#}[m] = \sigma[n]$ and $\sigma^{\#} \preccurlyeq \sigma$. The operator $\preccurlyeq$ denotes the *subsequence ordering* of two sequences defined as:

$$(a_i)_{i \in [0;m] \cap \mathbb{N}_0} \preccurlyeq (b_j)_{j \in [0;n] \cap \mathbb{N}_0} \Leftrightarrow a_0 = b_{i_0}, \ldots, a_m = b_{i_m}$$

for some indices $0 \leq i_0 < \cdots < i_m \leq n$.

Algorithm 3: ErrInv (adapted from [9])

Input: ψ, π, ϕ: Boolean Formula
Output: $\pi^{\#}$: Abstract error trace of $\tau = (\psi, \pi, \phi)$
Result: Alternating sequence of elements of π and interpolants concisley explaining the cause of the error.

```
1  𝕀 = interpolate(TF(τ)) ;    // interpolant for every position
2  interval_max = {(l ↼ searchL(0, j, I_j), r ↼
     searchR(j, n, I_j), errinv ↼ I_j) | I_j ∈ 𝕀};
3  interval_max.sort((a, b) → a.l ≤ b.l) ;    // sort ascending by l
4  π# = [];
5  maxInterval = interval_max[0];
6  prevEnd = 0;
7  for currInt : interval_max do
8      if currInt.l > prevEnd then
9          π#.next = maxInterval.errinv;
10         if maxInterval.r < n then
11             π#.next = π[maxInterval.r];
12         prevEnd = maxInterval.r;
13         maxInterval = currInt;
14     else if currInt.r > maxInterval.r then
15         maxInterval = currInt;
16 return π# ;
```

With "$x \leftharpoonup y$" we denote the assignment of value y to a variable named x. Algorithm 3 computes an abstraction $\pi^{\#} = I_0, T_1, I_1, \ldots, T_k, I_k$ ($T_l \in \pi$) where an interpolant I_j can be understood as transitions that summarize the

meaning between the surrounding two transitions T_j and T_{j+1}. Note that the SSA-indices have to be compliant. The details of ErrInv are described in the next section.

3.8.3 The Algorithm

Algorithm 3 shows a basic version for a procedure to find the abstract error trace. Based on the calculated interpolants in line 1 we try to find the ones that are inductive for many positions in π. In line 2 we calculate the borders of every interpolant, i.e., the region where it is inductive. We proceed by sorting the intervals ascending by their attribute `l`, representing the left border. Afterwards we initialize `maxInterval` with the interval starting at position 0. Now we loop through every interval to yield the actual abstract error trace as follows: If the current interval starts right of `prevEnd` we add it as an interpolant to the abstract trace. Furthermore, if the end is within the boundaries of the actual trace, we add the corresponding transition of π to $\pi^{\#}$. This is always the case if the interpolant has a `r`-value less than n. In this way we produce the alternating sequence of interpolants and transitions. If the `r`-value of the interpolant equals n it finishes the trace. An interval with boundaries $\texttt{l} = i < \texttt{r} = j$ is inductive from i to $j - 1$. Next, `prevEnd` is updated to the value of the end of `maxInterval` since the variable is overridden by `currInt` in the following line. As a summary the then-block produces the alternating sequence of transitions and interpolants if the current interval starts right of the maximal interval. Otherwise, if the interval starts before and ends after the end of the current maximal interval, we override it by the current processed interval because it represents more clauses. As we can see the most important part of the algorithm is to find the boundaries of the interval, implemented with the methods $\texttt{searchL}(\texttt{0}, \texttt{j}, \texttt{I}_\texttt{j})$ and $\texttt{searchR}(\texttt{j}, \texttt{n}, \texttt{I}_\texttt{j})$. Indeed, these two functions are the most expensive ones in terms of run time in Algorithm 3. In a naive approach `searchL` implements a for-loop through every position returning the position i for which I_k is an interpolant for the first time. Analogous, `searchR` loops backwards from $n - 1$ to 0 and returns the position for which I_k is an interpolant for the first time. For large programs this means that we have many calls to the solver for determining the boundaries of the interpolants. Therefore, prior work [9] proposes an implementation of these two methods as a fast binary search reducing the run time in terms of calls to the solver to $O(log(n))$. The better the implementation of the boundary search for interpolants the better the algorithm performs. Thus, the search for start and end of an interval are implemented as a guided binary search based on the interpolant. The algorithm takes a minimal number `start` and a maximal number `end` to

Algorithm 4: Binary search for inductive interpolants [9].

Input: `start, end`: Integer, `incLow`: Boolean Function
Output: `position`: Integer
Result: Find boundaries of the interval.

```
1 if end < start then
2     return start;
3 mid = ⌊(start+end)/2⌋;
4 if incLow(mid) then
5     return search(mid + 1, end, incLow) ;      // recursive call
6 else
7     return search(start, mid − 1, incLow) ;    // recursive call
```

calculate the actual boundaries of the interpolant. The current interpolant is contained in the function `incLow`, which guides the search in the correct direction by either increasing the `start`-variable or decreasing the value of `end`. If end is smaller than start we return the value of start, otherwise we calculate the mid and let `incLow` decide where to look next. Algorithm 4 shows an implementation of the binary search.

The method `incLow` takes an interpolant I_j and a time stamp i to determine if the interpolant of position j is an error invariant (c.f. Section 3.8.1) for position i, too. In Algorithm 3 we used the functions `searchL` and `searchR` to determine the boundaries. Now we can define them by using the search method. Hence,

- $\texttt{searchL}(0, \texttt{j}, \texttt{I}_\texttt{j}) = \texttt{search}(0, \texttt{j}, \texttt{i} \rightarrow \neg\texttt{incLow}(\texttt{I}_\texttt{j}, \texttt{i}))$
- $\texttt{searchR}(\texttt{j}, \texttt{n}, \texttt{I}_\texttt{j}) = \texttt{search}(\texttt{j}, \texttt{n}, \texttt{i} \rightarrow \texttt{incLow}(\texttt{I}_\texttt{j}, \texttt{i})) - 1$

for a trace of length n. The expression $i \rightarrow \texttt{func}(\dots)$ symbolizes a lambda expression known from several programming languages, including Java. We will now discuss the correctness of `searchL` and `searchR`. To find the left bound of the interval we apply the binary search together with the negation of the Boolean function $\texttt{incLow}(\texttt{I}_\texttt{j}, \texttt{i})$. If I_j turns out to be an interpolant on position `mid` as well, the function `incLow` returns true. As its negation is false, we execute line 7 in Algorithm 4 and therefore let the value `start` unchanged. The search moves from right to left. Since the initial call of `searchL` has the lowest possible value 0 as argument for `start` the search ends on the left border. Otherwise, if I_j is not an interpolant, we restart the search from the midpoint of the beginning and the actual position of the interpolant. If an interpolant happens to be inductive up to `mid` we can

```
1  int main (void) {
2     int x = 5;
3     x = x + 5;
4     x = x + 1;
5     x = x - 1;
6     if (x == 10) {
7        goto ERROR;
8     }
9  EXIT:
10    return 0;
11 ERROR:
12    return 1;
13 }
```

Program 4: Error Invariants Example

restart the search for values between `mid` and `end`, as we know it is already inductive at `mid`. Searching the right border with `searchR` works analogous. Finally, we run through the algorithm with Program 4 as input and obtain following values for the trace τ:

ψ: $x = 5$

π: $x' = x + 5 \wedge x'' = x' + 1 \wedge x''' = x'' - 1$

ϕ: $x''' \neq 10$.

Table 1: Interpolants and boundaries of Program 4

line	interpolant	borders
2	$x = 5$	[2;2]
3	$x = 10$	[3;3]
4	$x = 11$	[4;4]
5	$x = 10$	[3;5]

First, the interpolants for each position are calculated and then are extended by the inductive boundaries as seen in Table 1. The algorithm proceeds to sort the intervals ascending by their left boundary to then calculate the abstract error trace. Applying Algorithm 3 yields the abstract error trace shown in Table 2.

The algorithm abstracts the error trace by summarizing lines 4 and 5 to $x = 10$ since we effectively do not change the value of x within these two lines.

Table 2: Abstract error trace of Program 4

Abstract trace	
Interval[2;2]:	$x = 5$
Statement[3]:	$x = x + 5$
Interval[3;5]:	$x = 10$

This means that the error condition is already violated at line 3. We can now either change line 3 or the behavior within the interval $[3; 5]$. For better readability we used line numbers to indicate the boundaries. Of course, the algorithm uses indices instead of line numbers.

4 Theoretic Contributions

4.1 Adaptions for MAX-SAT

Algorithm 5: Adapted MaxSat (original [16])

Input: ψ, π, ϕ: Boolean Formula
Output: H: Set of sets of Boolean Formulas
Result: Subsets of clauses of π where adapting the corresponding lines/locations in one of the sets can fix the bug.

1 $S = \{\lambda_0, \cdots, \lambda_{n-1}\}$; // soft set
2 $\mathtt{tf}_1 = \psi \wedge \bigwedge_{i=0}^{n-1} (\lambda_\mathtt{i} \Rightarrow \pi[\mathtt{i}]^{\langle \mathtt{i} \rangle}) \wedge \phi^{\langle n \rangle}$;
3 $H = \{\}$; // set of all found cores
4 $M = \{\}$; // current MIN-UNSAT core
5 **while** $|M| \neq |S|$ **do**
6 $\quad M = \mathtt{minUnsatCore}(S, tf_1, H)$;
7 $\quad$ **if** $|M| == 1$ **then**
8 $\quad\quad S = S \setminus M$;
9 $\quad$ **if** $|M| \neq n$ **then**
10 $\quad\quad H = H \cup \{M\}$;
11 **return** H ;

In Chapter 3 we worked with the `CoMSS`-solver which is not available in the CPAchecker-framework. As a consequence we adapted the presented algorithm with a few changes and implemented an own method based on the existing solver which guarantees to return all MIN-UNSAT cores as a set of selectors instead. We already mentioned that every element of the complement $\overline{M}$ of a MSS is a subset of a MIN-UNSAT core. That is the reason why we are interested in the MIN-UNSAT cores from now on. The advantage of MIN-UNSAT cores compared to the complements of MSS is

that the MIN-UNSAT cores directly display the contradicting locations in the program instead of just indicating locations. The goal is to calculate and return every MIN-UNSAT core of selectors. Note that this ensures that we do not lose any information compared to the presented approach in 3.7 since every complement of any MSS will be element of an UNSAT core. However, the combination of the selectors in the resulting sets may differ. Algorithm 5 shows the adaptations.

The first two lines remain the same. Now we introduce two new sets, the hard set H and the set M containing the MIN-UNSAT core of the current iteration. In this section, when we refer to MIN-UNSAT core or a minimal set we mean that no nonempty, real subset of the core or set forms an UNSAT core, too. Next, we enter the while-loop as long as the size of the current UNSAT core M does not equal the number of selectors in S. Afterwards, we calculate the next MIN-UNSAT core based on the already calculated sets in H and the remaining selectors S. In case the returned set M has size 1, we remove the selector from S in line 8. To understand why this is important, we have to define the function $\texttt{minUnsatCore}(S, tf_1, H)$. Its task is to find a minimal subset of selectors $\mathbb{L} \subseteq S$ such that the following two conditions hold:

a) $\overbrace{tf_1 \wedge \underbrace{\left(\bigwedge_{\lambda \in \mathbb{L}} \lambda\right)}_{\text{soft-set-formula } \texttt{SSF}(\mathbb{L})}}^{tf_2(\mathbb{L})}$ is unsatisfiable,

b) $\forall\, \mathbb{H} \in H : \mathbb{H} \nsubseteq \mathbb{L} \wedge \mathbb{L} \nsubseteq \mathbb{H}$.

To find the smallest set $\mathbb{L}$ of selectors, the algorithm iterates over every possible non empty subset of S and checks if tf_2 with the current subset $\mathbb{L}$ is unsatisfiable. Whenever this is the case and no other selector can be removed from the set, still guaranteeing unsatisfiability, we acquired a new UNSAT core. If we conjunct the soft-set-formula to $\texttt{TF}_\Lambda$ we automatically mark all elements of $\mathbb{L}$ as hard while all other selectors must not be regarded. As seen above SSF is defined as $\texttt{SSF}(\mathbb{L}) = \left(\bigwedge_{\lambda \in \mathbb{L}} \lambda\right)$ for any subset $\mathbb{L}$ of selectors. On condition that tf_2 is unsatisfiable and $\mathbb{L}$ is minimal, it can be returned (condition a)).

To prevent returning an already found set $M \in H$, we store them in the hard set, as seen in Algorithm 5, line 10. Checking if the size of M does not equal n in line 9 is optional but returning all inputted transitions to the user yields no information because he already knows that there is a fault somewhere in the whole program. An important side note is that we do not

prevent appending cores of the size of S, we only prohibit appending sets of size equal to the initial number of selectors. Now, the modified hard-set

Algorithm 6: MIN-UNSAT core computation

Input: S: Set of selectors, tf_1: trace formula, H: all found cores
Output: M: new minimal set of UNSAT cores.
Result: Returns a new minimal set of selectors that makes the trace formula unsatisfiable.

```
1  result = S;
2  do
3      changed = false;
4      for λ in result do
5          currCore = S \ {λ};
6          if check(currCore, H) then
7              if isUnsat(TF_Λ ∧ SSF(currCore)) then
8                  changed = true;
9                  result = result \ {λ};
10                 break;
11 while changed;
12 return result;
```

is passed as argument to the function `minUnsatCore` in the next iteration. After finding a new potential subset $\mathbb{L}$ of selectors, the function first checks if this set is contained by another set of H or if it contains another set of H (condition b)). If this is the case we do not return the subset because it either is not minimal or already found. Algorithm 6 sketches the calculation of MIN-UNSAT cores. We systematically remove selectors from the set of all remaining selectors S. Under the assumption that $tf_2(S)$ is unsatisfiable we remove selectors from S as long as $tf_2(S')$ remains unsatisfiable where S' denotes the reduced selector set originating from S. Therefore, every selector whose removal does not make tf_2 satisfiable has no impact on the root cause of the bug and hence is not needed. This does not mean that the same selector cannot be part of another UNSAT core, it only states that this selector is not needed in the current combination of selectors to yield unsatisfiability. In order to compute a new MIN-UNSAT core we create a copy of the remaining selectors and initialize the Boolean variable `changed` with false. Since we use a do-while construct we ensure entering the iteration-body at least once. Now after entering the for-loop in line 4, we remove one selector after another and check if the formula is still unsatisfiable. If so, we remove the selector

from the result set, assign true to `changed` and restart the procedure with a smaller result set. Otherwise, if we cannot remove anything from `result` anymore, we return it as a new MIN-UNSAT core. The function `check` in line 6 checks if condition b) from above holds and returns true whenever `currCore` is neither a subset nor a superset of any set in H. As soon as a subset of $\mathbb{L}$ is contained in H we can skip the calculation of $\mathbb{L}$ and all its subset. We illustrate the idea with the help of an example. Let the set $\mathbb{H} = \{\lambda_0, \lambda_1\} \in H$ be an element of H and the set $\mathbb{L} = \{\lambda_0, \lambda_1, \lambda_2, \lambda_4\}$ the currently checked candidate for a new MIN-UNSAT core. Condition b) is violated as $\mathbb{L}$ is a superset of $\mathbb{H}$ and therefore will be skipped. Since $\mathbb{L}$ obviously is not minimal MaxSat is able to remove further selectors and will eventually end up with $\mathbb{H}$ either way. In a few cases the set $\{\lambda_2, \lambda_4\}$ might be an undetected UNSAT core which we seemingly will skip and ignore now. However, this core will also be found in latter iterations. In some point of the execution the algorithm will remove selector 1 first and thus the whole set cannot be a superset of element in H. However, this set still contains the selectors 2 and 4 and thus will return the MIN-UNSAT core $\{\lambda_2, \lambda_4\}$.

We can now see the improvement that removing cores of size 1 yields (line 7 and 8 in Algorithm 5). Removing just one element from the set of selectors reduces the number of possible subsets by half. A set with n elements has 2^n subsets, removing just one element leads to 2^{n-1} subsets. The difference of both values equals $2^{n-1} = {}^{2^n}/_2$. Furthermore, we spot the reason, why this method does not return the cores in correct order: Imagine the singleton set of selector $\{\lambda_0\}$ is one of the UNSAT cores but the only one of size 1. Now, after entering the for-loop for the first time, we immediately remove λ_0 in line 5 from the set of considered selectors. The methods `check` and `isUnsat` will return true since other UNSAT cores with a combination of the still available selectors are possible and we have not found another core yet. Algorithm 6 will proceed and return a set not containing λ_0 as first UNSAT core, although it is not the smallest one.

The original algorithm handles this differently. The usage of a partial maximum satisfiability solver (p-MAX-SAT-solver) allows to directly mark clauses as hard and soft to then obtain a maximum set P of selectors such that tf_2 is satisfiable. Instead of checking for sub- and super-sets as we proposed in b), they append a new hard clause `h` given by $\mathtt{h}(\mathbb{H}) = \bigvee_{\mu \in \mathbb{H}} \mu$ to the hard clauses and modify the soft set accordingly [16]. This prevents the p-MAX-SAT solver to return identical cores twice.

Table 3: Run of Algorithm 5 on Program 3

Run	M	H	S.`size`
#0	{}	{}	4
#1	$\{\lambda_2,\lambda_3\}$	$\{\{\lambda_2,\lambda_3\}\}$	4
#2	$\{\lambda_0\}$	$\{\{\lambda_0\},\{\lambda_2,\lambda_3\}\}$	3
#3	$\{\lambda_1,\lambda_2\}$	$\{\{\lambda_0\},\{\lambda_1,\lambda_2\},\{\lambda_2,\lambda_3\}\}$	3
#4	$\{\lambda_1,\lambda_2,\lambda_3\}$	$\{\{\lambda_0\},\{\lambda_1,\lambda_2\},\{\lambda_2,\lambda_3\},\{\lambda_1,\lambda_2,\lambda_3\}\}$	3

To conclude this section, we let Algorithm 5 analyze Program 3. The inputs are:

ψ: `input` $= 2$

π: $\{$`x` $=$ `input`$, x > 0, x < 5, x > 1\}$

ϕ: $\neg(x \neq -1 \wedge x \neq 6)$.

Table 3 shows the results after each iteration in the while-loop. At the beginning all sets but the selector set are empty. S has a size of 4 because we have 4 transitions before we reach the error label. In run #1 we find the UNSAT core $\{\lambda_2, \lambda_3\}$ that was found by SingleUnsatCore, too. We just have to add the set to H and continue to calculate all remaining cores, if available. The second iteration returns an UNSAT core of size 1 containing the selector λ_0. The reason is that $\psi \wedge (\lambda_0 \Rightarrow (x =$ `input`$)) \wedge \phi^{\langle 4 \rangle} \wedge \lambda_0$ is unsatisfiable since (`input` $= 2$) $\wedge$ (`x` $=$ `input`) $\wedge \neg(x \neq -1 \wedge x \neq 6)$ is not feasible. We now remove λ_0 from S to reduce the possible subsets of S in the next iteration. None of the following UNSAT cores can contain λ_0. Next, the solver finds another different UNSAT core of size 2. As stated above returning the same UNSAT core twice is prevented by checking for sub- and super-sets (Algorithm 6, line 6). Finally, the algorithm returns all left selectors as the new UNSAT core, which means that we break out of the while loop and return H to the user.

Table 4: Possible fixes for each found set

Set	Fix
$\{\lambda_0\}$	line 2: `int x` $= -$|`input`|
$\{\lambda_1, \lambda_2\}$	line 4: `if (x < 2) {`
$\{\lambda_2, \lambda_3\}$	line 4: `if (x < 2) {`
$\{\lambda_1, \lambda_2, \lambda_3\}$	line 4: `if (x < 2) {`

Adjustments to any of the found sets can fix the bug. See Table 4 for possible fixes. Once again, not every change in Table 4 might be appropriate for the situation.

Whenever we write MaxSat in the following chapters and sections, we refer to our adaption presented here (Algorithm 5).

4.2 Memoization of Error-Invariants

Before we start the search for inductive interpolants we first obtain a list of interpolants for every element of π. In our benchmarks we noticed that many interpolants occur multiple times in this list. Subsequently, the method `incLow` repeatably checks if equal interpolants hold on the same position i. We achieved improvements by adding memoization. Before we check if an interpolant holds on position i we look it up in a table, which maps an interpolant to all already processed positions. If the interpolant has a mapping to i it is inductive on position i. If the interpolant has a mapping to $-i$ it is not inductive on position i. If there exist neither a mapping to i nor $-i$ the interpolant has not been proven inductive or not inductive on position i. This means that we only have to call the solver for every interpolant together with a certain position once.

4.3 Information Extraction

For additional information we further process the found faults and look for common error patterns like faulty handling of iteration variables. In our implementation, the methods of class `InformationProvider` search for iteration variables and calculations within the array brackets. CPAchecker allows to access the actual program statement of a `CFAEdge` which gives us the chance to look for such patterns on string basis. To find out which variables are iteration variables we look for multiple occurrences of the pattern "varname = varname <operator> 1" or "varname<operator><operator>" with + and - as operators. The method marks all variables matching this pattern more than 3 times. Even if it is not part of a loop, manually and frequently subtracting or adding one to a variable acts like an iteration variable, too. Afterwards, we tell the user that an iteration variable is involved in causing the bug. Secondly, we search for the pattern "name[a op b]", which matches operations within the array subscript. If we find faults containing a line with such a pattern, we notify the user with the message that there may be a suspicious calculation within the array subscript.

Moreover, our data structure (c.f. Chapter 5) provides the interface `FaultExplanation` to map a fault to an explanation. Furthermore, it allows to append additional information (called appendables) to every found subset of error-prone transitions (called fault). Whenever appendables are added, the constructor of them demands a descriptive text which is meant to be generated by `FaultExplanations`. The advantage lies in the option of a uniformed way to create explanations or descriptions for possible fixes. We implemented the class `NoContentExplanation` using this interface. If the passed fault has a size of 1, it creates a description of a possible fix based on the contained edge. The proposed fix relies only on the edge type without the context of what happened before or will happen afterwards. For instance, the proposed fix for assume edges (e.g., `if (x < 5)`) is to replace the used operator by any of the other Boolean operators like "$<, >, \leq, \geq, =, \neq$". For more implementation details see Chapter 5.

4.4 Rankings for Faults

MaxSat tends to compute many different faults as it returns every possible MIN-UNSAT core. With the rankings we want to highlight especially promising faults by displaying them at the top of the visual report (c.f. Chapter 5). By default, every ranking assigns a normalized score to each fault in the obtained set. The score attribute of faults is implemented as a double value. Our rankings assign scores from 0 to 1 to each of the faults with the restriction that the sum of the assigned scores equals 1. This allows different rankings to be more comparable to others. If for example a ranking scores faults based on their set size, faults with a minimal set size get a higher score. Formally, we create a scoring function $h(\mathcal{F}, F)$ and a normalization function $n(x, F)$ with $\mathcal{F} \in F$ and $x \in \mathbb{R}$. F equals the obtained set of faults. For ranking r the following properties hold for h and n:

1. $h^{(r)} : (F, 2^F) \mapsto \mathbb{R}$
2. $n^{(r)} : (\mathbb{R}, 2^F) \mapsto [0; 1]$
3. $\sum_{\mathcal{F} \in F} n^{(r)}(h^{(r)}(\mathcal{F}, F), F) = 1$
4. $\forall \mathcal{F}, \mathcal{F}' \in F : h_1 = h^{(r)}(\mathcal{F}, F) > h^{(r)}(\mathcal{F}', F) = h_2 \Rightarrow n^{(r)}(h_1) > n^{(r)}(h_2)$
5. $\forall \mathcal{F}, \mathcal{F}' \in F : h_1 = h^{(r)}(\mathcal{F}, F) = h^{(r)}(\mathcal{F}', F) = h_2 \Rightarrow n^{(r)}(h_1) = n^{(r)}(h_2)$

Setting $n(x, F) = \frac{x}{\sum_{\mathcal{F} \in F} h(\mathcal{F}, F)}$ is applicable most of the time. The function h maps a fault of the set F to a numerical score. The properties ensure

that n maintains the monotony of h. As n is a normalization function it has to map all possible values to the same range of values from 0 to 1. To set the values into a comparable relation the normalized values have to sum up to 1 for each applied ranking. A higher value of h results in a higher value after normalization. Additionally, if two faults get exactly the same score the normalization function has to yield the same score, too. As a result every ranking assigns a weighted score between 0 and 1 to every fault making it more comparable to other rankings. As already explained it is possible and useful to concatenate different rankings for better results. For the final arrangement based on all the already applied rankings it is necessary to have normalized values. However, our implementation does not limit the scoring system to the presented one. One can assign arbitrary scores to every fault but the weighting and final ranking must then be manually adapted. Still, the scoring system above works for every future ranking without the need to adapt anything. If the ranking follows the convention the newly created ranking can be concatenated to the existing ones without further steps. The final scoring to yield the ordered list of faults averages the previously assigned scores of all used rankings. Let R be the set of used rankings, then

$$s_f = \frac{\sum_{r \in \mathrm{R}} n^{(r)}(h^{(r)}(\mathcal{F}, F), F)}{|\mathrm{R}|}$$

equals the final score for fault $\mathcal{F}$. We will now discuss the most important rankings and their functionality as well as fitting scenarios. The concatenation and concrete implementation of the Rankings will be covered in Chapter 5.

Identity Ranking. The simplest of all rankings assigns an equal score to every fault and sorts them in the order the set-iterator returns them. The ranking is useful for testing and achieving fast results without additional steps. It should not be used in cooperation with other rankings because it decreases the differences of the scores of the faults. Let $F = \{\mathcal{F}_1, \mathcal{F}_2\}$ the

Table 5: Average values without identity ranking

	$n^{(1)}$	$n^{(2)}$	$\varnothing$
$\mathcal{F}_1$	0.3	0.4	0.35
$\mathcal{F}_2$	0.7	0.6	0.65

set of found faults. Table 5 shows the normalized values of two rankings for each fault in F and their mean values which are used to rank the faults.

Table 6: Average values with identity ranking

	$n^{(1)}$	$n^{(2)}$	$n^{(\mathrm{id})}$	$\varnothing$
$\mathcal{F}_1$	0.3	0.4	0.5	0.4
$\mathcal{F}_2$	0.7	0.6	0.5	0.6

Currently, $\mathcal{F}_2$ leads by 0.3 points. If we now apply the identity ranking as seen in Table 6 we notice that the lead of $\mathcal{F}_2$ decreases by 0.1 points. The identity ranking will not disturb the order of the faults but it will assimilate the scores although meaningful rankings actually calculated a more noticeable gap between the two faults. Note that the sum of each column equals 1.

- $h(\mathcal{F}, F) = 1$
- $n(x, F) = {}^{x}/_{|F|}$

Line Distance Rankings. We now present two rankings that evaluate the score based on the absolute line-distance to the error location. The first ranking assigns a higher score to faults with a minimal distance to the error locations whereas the second ranking assigns a higher score to faults further away. Both rankings make sense in different applications. Usually, declaration of variables takes place in the upper part of the program. Assuming that the declarations are correct, sorting the faults by minimal distance to the error locations yields better results because less error prone steps will be executed earlier. However, standard C programs require a strict ordering of functions. If function a calls function b, b has to be declared above a, meaning that important execution steps are far away from the error location when referring to the line numbers. Let l be the line number of the error location, then for the second ranking the function h and n can be defined as (remember that a fault $\mathcal{F} \in F$ is, general speaking, a set of edges):

- $h(\mathcal{F}, F) = |l - \min_{e \in \mathcal{F}}(e.\texttt{lineNumber})|$
- $n(x, F) = {}^{x}/_{\sum_{\mathcal{F} \in F} h(\mathcal{F}, F)}$

We use a different procedure to calculate the score in our implementation but the idea remains the same.

Overall Occurrence Ranking. Taking MaxSat as an example, we see that same selectors can be included in multiple subsets (c.f. Table 4). Many

of them contain the same selector, indicating that this selector maps to an important location in the program that might be very likely to cause the bug. If one set contains many of the frequently appearing selectors it should be ranked higher than other sets.

- $h(\mathcal{F}, F) = \sum_{\mathcal{F}' \in (F \setminus \{\mathcal{F}\})} |\mathcal{F} \cap \mathcal{F}'|$
- $n(x, F) = {}^{x}/_{\sum_{\mathcal{F} \in F} h(\mathcal{F}, F)}$

Set Size Ranking. The ranking above ignores that selectors contained in singleton sets only appear once overall because no subset of higher cardinality containing the singleton can be minimal. These sets indicate that the bug can be fixed within one line although they would get a lower score. Adapting a minimal number of locations in the program is desirable. Therefore, we propose a ranking based on the set size. It can be used in combination with the *overall occurrence ranking* to neutralize the weakness from the latter.

- $h(\mathcal{F}, F) = \max_{\mathcal{F}' \in F}(|\mathcal{F}'|) - |\mathcal{F}| + 1$
- $n(x, F) = {}^{x}/_{\sum_{\mathcal{F} \in F} h(\mathcal{F}, F)}$

Call Hierarchy Ranking. We already discussed the advantages and disadvantages of the *line distance rankings*. The *call hierarchy ranking* represents a optimization to the problems of the *line distance rankings*. Whereas these rankings measure the distance to the error in lines this ranking returns the number of execution steps between the corresponding edge and reaching the error label. We now surely obtain the location closest to the error label in terms of the execution order. The function `indexOf`$(\mathcal{F})$ returns the position of an element of $\mathcal{F}$ in the execution order closest to the error label.

- $h(\mathcal{F}, F) = \max_{\mathcal{F}' \in F}(\texttt{indexOf}(\mathcal{F}')) - \texttt{indexOf}(\mathcal{F}) + 1$
- $n(x, F) = {}^{x}/_{\sum_{\mathcal{F} \in F} h(\mathcal{F}, F)}$

Edge Type Ranking. The last ranking differs from the others because it is based on an experimental heuristic. In our benchmarks we figured that some types of edges are more likely to fix a bug than others. We can determine the type of an edge easily because every selector in the implementation maps to exactly one `CFAEdge` with an assigned type calculated by CPAchecker

before running our algorithm. The heuristic $H(e), e \in \mathcal{F}$ is defined as:

$$H(e) = \begin{cases} 1 & \text{if } e \text{ is an assume edge} \\ 0.5 & \text{if } e \text{ is a statement edge} \\ 0.25 & \text{if } e \text{ is a return statement edge} \\ 0.125 & \text{if } e \text{ is a function-return/-call or call-to-return edge} \\ 0 & \text{otherwise} \end{cases}$$

Using the heuristic we can define h and n as:

- $h(\mathcal{F}, F) = \sum_{e \in \mathcal{F}} H(e)$
- $n(x, F) = {}^{x}/_{\sum_{\mathcal{F} \in F} h(\mathcal{F}, F)}$

4.5 Options

We added 9 options that can be set to refine and improve the results of the algorithms. All available options will be explained now.

faultlocalization.type
The option accepts three inputs: "UNSAT", "MAXSAT" and "ERRINV". Via this option the algorithm for the further analysis is set. "UNSAT" executes SingleUnsatCore, "MAXSAT" executes MaxSat and lastly "ERRINV" executes ErrInv. The results of every algorithm are ranked by the individually best combination of rankings based on experimental results. If this option is not set, we run SingleUnsatCore by default.

faultlocalization.maintainhierarchy
Enable this option to maintain the original ordering in the execution path, i.e., the resulting faults are not sorted by the average score received by the application of multiple rankings. Whenever we run ErrInv the option is recommended to be enabled since the algorithm produces an alternating sequence of interpolants and transitions in an predefined order. Sorting by the highest score destroys the ordering.

faultlocalization.memoization
This option also only affects ErrInv. If enabled, the algorithm stores already processed interpolants at a certain position and maps them to a Boolean value depending on whether the interpolant is inductive on that position or not.

traceformula.altpre and **traceformula.filter**
Enable the alternative precondition to make use of the filter. The filter is a comma-separated string that should contain function names of the underlying program. The alternative precondition then adds every initial variable assignment of the form `int x = 5;` to the precondition. It will only regard variables that are initialized in one of the functions contained in *filter*. Note that we do not add statements like `int y = x + c` with $x \in X$ and $c \in \mathbb{N} \cup X$ because the line already might cause the bug. The same rule applies for other data types. Arrays with an initialization like `int a[] = {1,2}` are part of the alternative precondition, too. By default, *filter* only filters variables of the main function. We enable the alternative precondition automatically as soon as the model equals `true`, i.e., there are no nondeterministic variables used.

traceformula.ignore
This is also a comma-separated string variable that takes variable names of the form `function::variable` or just the `variable`. If no prefix with a function name is given for a variable the program ignores all variables with the same name regardless of the scope. Ignored variables will not be part of the alternative precondition. The option is mainly useful to ignore iteration variables in for-loops.

faultlocalization.ban
This option is another comma-separated string for variables, either with a function as prefix like `function::variable` or without. If a variable is banned all faults that contain an edge using the variable will not be printed. This is especially useful to filter sets containing important variables for the post-condition. This option is not available when running ErrInv as every transition is important for the abstract error trace.

traceformula.uniqueselectors
If enabled, duplicate lines received by loop enrolling get the same selector. The advantage is that we drastically reduce the amount of selectors and therefore the amount of subsets we have to check. The drawback is that the result is not that precise anymore. Seeing that, for instance, only the first three iterations are part of the minimal subset yields more information than just seeing that the iteration variable is of interest. In the original MaxSat algorithm [16] the option is enabled. Due to a reasonable improvement of the obtained results we disabled it by default.

exprconv.niceexpr
If enabled, formulas obtained by the solver are transformed from prefix to infix notation for better readability. The amount of brackets is reduced but not minimal. We implemented the class `ExpressionConverter` to transform Boolean formulas of arbitrary length to infix notation as long as all operations are unary or binary. Problems arise whenever arrays are part of the formula. This has to be fixed. This is also the reason why this options is disabled by default. Because of the CPAchecker-internal implementation of Boolean formulas the `ExpressionConverter` works on the string representation of them and thus is not perfect yet. Nevertheless, for small programs the readability of the reports is highly increased.

Example Configurations:
Finally, we give example configurations for each of the implemented algorithms. The option *alwaysStoreCounterexample* is required to run fault localization. Activate options by appending "`-setprop <name as in the descriptions above>=<possible values>`". The following commands allow to run fault localization with the implemented algorithms:

MAXSAT:
```
-preprocess
-predicateAnalysis
-setprop analysis.algorithm.FaultLocalization=true
-setprop analysis.alwaysStoreCounterexamples=true
-setprop faultlocalization.type=MAXSAT
-setprop traceformula.altpre=true
<path to program>
```

ERRINV:
```
-preprocess
-predicateAnalysis
-setprop analysis.algorithm.FaultLocalization=true
-setprop analysis.alwaysStoreCounterexamples=true
-setprop faultlocalization.type=ERRINV
-setprop faultlocalization.maintainhierarchy=true
-setprop traceformula.altpre=true
<path to program>
```

UNSAT:
```
-preprocess
-predicateAnalysis
```

```
-setprop analysis.algorithm.FaultLocalization=true
-setprop analysis.alwaysStoreCounterexamples=true
-setprop faultlocalization.type=UNSAT
<path to program>
```

5 Data Structure for Fault Localization

5.1 Concept of the Data Structure

The algorithms are embedded in a data structure explicitly created for any fault localization algorithm. We designed the structure to be easily usable and extendable in every possible way granting a simple adaptation to all kinds of fault localization algorithms. Whenever a counterexample is found, CPAchecker creates a visual report using HTML, JavaScript and CSS. More details on the graphical report can be found in the upcoming section (c.f. 5.2). The report already contains a visual representation of the CFA, gives access to the source code and lists all relevant edges from the counterexample in the correct order from the beginning of the program to the error.

The class `CounterexampleInfo` is the interface for counterexamples and their visualization as HTML-report. `FaultLocalizationInfo` extends this class and enhances the existing report by an additional view for the `Faults` and adds a rank to each edge. `FaultLocalizationInfo` is a simple class that either takes an already sorted list of `Faults` or a set of `Faults` and a `FaultRanking`. The purpose of `FaultLocalizationInfo` is to transform the list to a JSON-format applicable for the report and to transform the attached descriptions to HTML-format. For the transformation of them the class `FaultReportWriter` is used.

The `FaultReportWriter` implements methods for transforming objects into HTML-format that fits for most fault localization algorithms. If one wants to change the HTML-output this class can simply be extended and designed by need. An instance of this extended class can then replace the default `FaultReportWriter` in the class `FaultLocalizationInfo`. A `FaultRanking` is an interface that transforms a set of `Faults` to a sorted list. Usually, the `Faults` are sorted descending by their score. `Faults` are

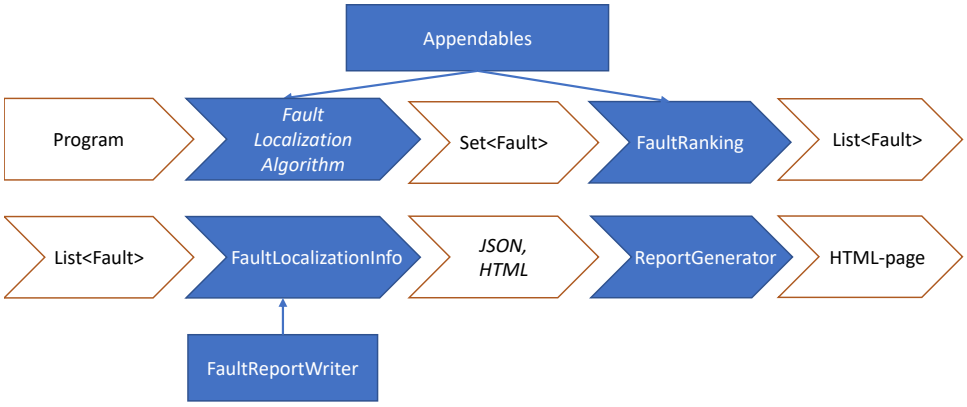


Figure 1: Flow chart for the fault localization data structure

a set of `FaultContributions`. Every `FaultContribution` maps to exactly one `CFAEdge` and extends it with the possibility to assign a score and to attach explanations, called "appendables", for why this edge is part of a `Fault`. `Faults` have a score as well as the possibility to attach explanations, too. Figure 1 shows the simple linear steps to get a visual representation without further modifications. Input and outputs are outlined in orange and the classes that handle them are colored in blue. As seen in the figure, any fault localization algorithm returning a set of `Faults` can use the data structure. Subsequently, we have to rank the faults, i.e., transform the set into a list that determines which `Fault` will be printed on first place in the resulting HTML-page. This can be done by one of many provided and useful rankings (c.f. Section 4.4) or by an own arbitrary implementation. It is recommended to implement the interface `FaultRanking` for own rankings because it opens up an easy way to concatenate multiple of them. The list can be passed along with an instance of the class `CounterexampleInfo` to an instance of the class `FaultLocalizationInfo`. With the help of the `FaultReportWriter` the list is translated to JSON and HTML such that the `ReportGenerator` can write the information to an interactive HTML-page. As already mentioned the `FaultReportWriter` can be adapted by need and replace the default report writer. To sum up the process, the user needs to go through four steps:

- obtain an instance of `CounterexampleInfo` from a target state in the ARG (Abstract Reachability Graph),
- run a fault localization algorithm that returns a set of `Faults`,
- transform the set to a list and create the `FaultLocalizationInfo`-object `flInfo` by calling the constructor with the `CounterexampleInfo`

and the list as arguments,

- call the method `apply()` on `flInfo`.

We will now explain more details to the most important classes of the data structure:

FaultInfo. The class `FaultInfo` gives access to four types of additional information that can be printed to the user. The four types are listed below:

Reason: Whenever a fault localization algorithm finds a candidate set of possibly error-prone `CFAEdges` it can justify the decision by attaching the reason for choosing this set to the created `Fault`. For example, in our implementation, ErrInv adds the describing interpolant as reason to the `Fault`. There is no obligation to add a reason to a `Fault` but it will help the user to better understand the results.

Potential Fix: A potential fix suggests an adaptation that might fix the bug. We implemented two classes producing potential fixes, e.g., based on the edge-type. Assume that a `Fault` contains a return edge, then we can draw attention to the function probably returning a wrong value by stating that the function might have an unwanted return value.

Rank Info: `FaultRankings` rank the `Faults` and add a `FaultInfo` with a likelihood and a description why it assigned this score. Each of the implemented `FaultRankings` adds this information.

Hint: A hint adds additional information that does not belong to any of the previous category as for instance the failing variable assignment.

In Figure 1 we refer to them as "appendables". On demand, chosen categories are hidden in the final report by calling the method `hideTypes` of the `FaultReportWriter`. As parameters the method expects types of the enum `InfoType`, namely: `REASON, FIX, HINT` and `RANK_INFO`. All passed types will not be shown in the final report.

FaultContribution. A `FaultContribution` has to be initialized with a `CFAEdge` as argument. For every edge in the counterexample, exactly one instance of this class should be created. The class allows to add a score to all instances, indicating how likely adapting the operation of the edge will fix the bug. It is recommended to calculate the score by averaging all likelihoods of the attached `FaultInfos`. We implemented the helper class `FaultRankingImpl` to give access to the methods `assignScoreTo(Fault)`

and `assignScoreTo(FaultContribution)` which implement a default way for scoring `Faults` and `FaultContributions`. The methods average all scores of `FaultInfos` of the type `RankInfo`.

Fault. The class `Fault` represents a set of `FaultContributions`. `Faults` have a score and maintain a list of additional `FaultInfos`. The final report displays all of these information in a readable way to the user sorted by their category.

FaultRanking. This interface offers a method that simply transforms a set of `Faults` to a list. There is no directive on how to create the list but we recommend to sort the list by likelihood. For instance, we implemented a `FaultRanking` that assigns a higher score to `Faults` with less elements and consequently sorts it ascending by the size of the `Faults`.

FaultRankingUtils. While creating `FaultRankings` we repeatably needed the same bits of functionality, like assigning scores to `Faults`, concatenate different `FaultRankings`, accessing standard rankings and obtaining a *fault-to-score* map. Therefore, we decided to make them publicly accessible in this class. It may be useful to rank the same set of `Faults` by the average value of different `FaultRankings`. The concatenation of multiple rankings is handled by the method `concatHeuristicsWithDefaultFinalScoring`. The concatenation of multiple rankings works by applying every ranking on the set of faults and returning a list sorted by the average ranking-score. The method `assignScoreTo` has already been covered.

FaultLocalizationInfo. This class has two constructors. The first one only expects a list of `Faults` and an instance of `CounterexampleInfo` whereas the second one awaits a set of `Faults` combined with a ranking and the `CounterexampleInfo`. Both constructors lead to same result if the passed ranking equals the ranking used to obtain the list for the first constructor. The second one applies the ranking on the set of `Faults` and proceeds to prepare it for the HTML-page. Furthermore, we implemented a method that transforms a set of `CFAEdges` to a set of `FaultContributions`. This enables the possibility to use the report at minimal cost.

In our implementation the class `Selector` extends `FaultContribution` because a selector maps to exactly one element of π, too. This property becomes convenient when we have to map the obtained sets of Boolean formulas back to the original edges. A `Selector` stores the formula and the corresponding `CFAEdge` and is uniquely identifiable. Therefore, we let ErrInv and Sin-

gleUnsatCore return instances of the class `Selector`. The uniqueness of the selectors allows to maintain a hashed map that yields the corresponding edge of an clause in an expected run time of $O(1)$.

5.2 Visualization

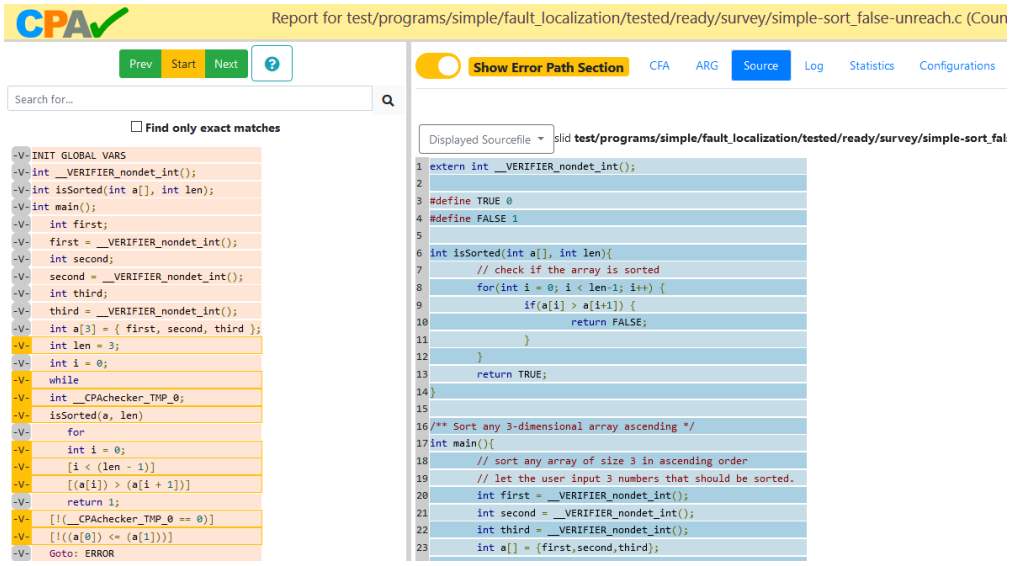
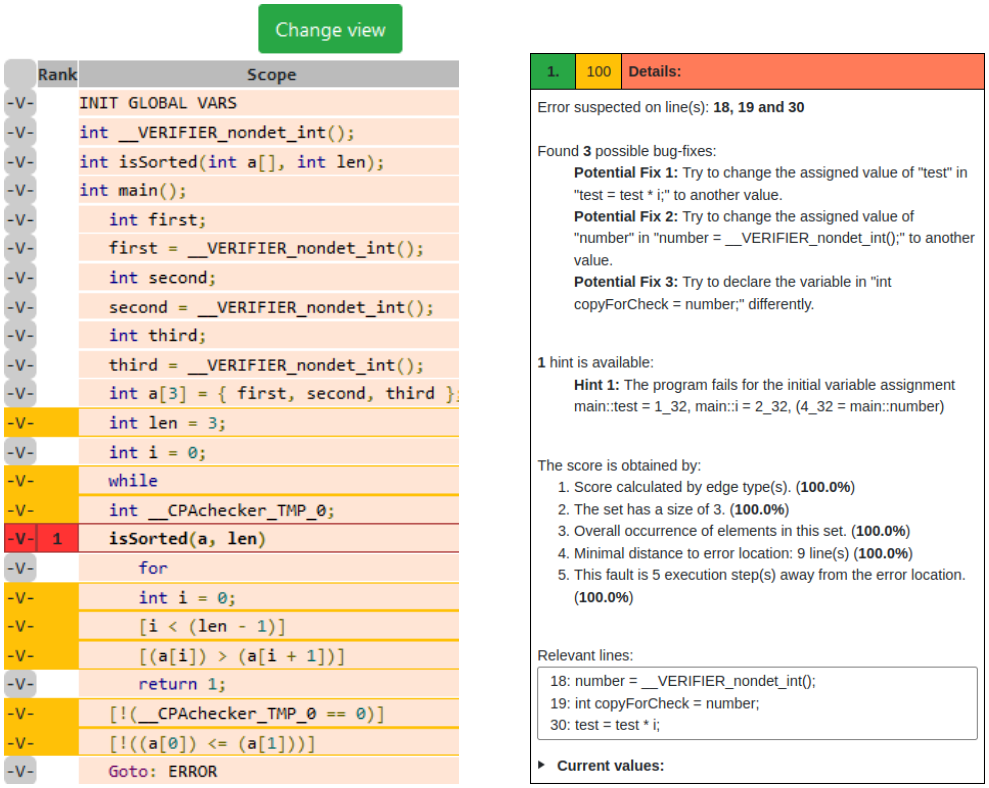


Figure 2: Counterexample report

We enhanced the existing graphical report shown in Figure 2 for counterexamples in CPAchecker by adding a fault localization section and ranks in front of the edges. Figure 2 shows the computed counterexample on the left hand side. A click on an entry marks the corresponding edge in the CFA, ARG and the corresponding line in the source code depending on the selected view. To toggle the views the navigation bar on top can be used. The source code or the graphs are then shown next to the counterexample. On the left hand side, Figure 3 shows the enhanced counterexample after fault localization was used. It shows all relevant edges and marks the important edges for the counterexample in yellow. We inserted an additional column rank. Every edge of the counterexample that is also part of one or multiple faults has its best rank written in the new column. Clicking on the rank shows all the information the algorithm gathered for this edge. Remember that attaching information is possible not only for faults but for fault contributions, too. The user can access this additional information here. The button “Change view” is the second addition. On click, the counterexample is replaced by the



(a) Counterexample (b) Fault

Figure 3: Counterexample and fault description

ranked HTML-representation of every fault. This view gives an interactive overview about all important edges. A click on the faults marks the edges of the graphs or the lines in the source code.

On the right hand side, Figure 3 shows the description of a fault obtained by running MaxSat. The first line shows all related lines in the source code. The second block provides potential fixes for each of the lines. All potential fixes are created with an instance of the class "`NoContextExplanation`". The hint-section here has only one hint showing the precondition, i.e., a failing variable assignment. The fourth block shows the rankings and their score. Since the algorithm found just one fault all scores are equal to 100. The yellow cell on the top left represents the overall score, i.e., the average of all scores listed in the fourth block. Next to it, in the green cell, we can see the resulting rank. The last block shows the relevant lines and statements in the order of execution. A click on the arrow next to "Current values" shows a table with every variable and its value. Variables of high importance to the fault are marked red. The table depicts the values after the execution of the

most recent line shown in the section “relevant lines”.

Whenever the ErrInv algorithm is used, there will be an additional block of information in Figure 3 b) labeled “reason”. This section contains the previous inductive interpolant, i.e., the interpolant before the displayed transition in the abstract error trace

Currently, we encounter one problem: Some edges are excluded from the counterexample section because they are not important for the understanding. However, these edges can be part of a fault. Whenever this is the case, we neither can map the relevant lines nor query the current values of the edge(s). Consequently, theses 2 blocks will be empty in the report for faults containing such edges.

6 Implementation

We integrated the algorithms SingleUnsatCore, MaxSat and ErrInv in the CPAchecker-framework, which is written in Java. For some explanations we use existent data structures of the Java-API. This chapter gives an insight into the implementation of each algorithm and describes the most important functionalities of CPAchecker we used.

6.1 Overview

The implementation of the fault localization techniques provides a variety of options and enhancements to the original algorithms. As already discussed, we achieved improvements in the run time with respect to the number of calls to a solver and found ways to alleviate some of the weaknesses. The process of generating faults can be described in 6 steps:

1) Choose the algorithm

2) Generate counterexamples for every $e \in E$ (error labels)

3) Build the trace formula for the current error path based on the counterexample

4) Execute the chosen algorithm on the trace formula

5) Rank the obtained locations and provide additional information

6) Output the results as an interactive error report embedded in the existing counterexample report of CPAchecker

We already covered the theoretic background of all these steps. This chapter focuses on the implementation of the algorithms in the CPAchecker-framework.

6.2 External Functions for CPAchecker

To simulate nondeterministic variables within the C program we want to analyze, we have to add an external function to the inputted program. Insert `extern <datatype> __VERIFIER_nondet_<datatype>();` at the top of it to use declarations like `int x = __VERIFIER_nondet_int();` for assigning a value to `x` that is not known at compile time. As stated in Chapter 3, we need a failing variable assignment for the precondition which can, for instance, be computed with a model. Afterwards, we have to assign this value to the nondeterministic variable and make sure that we can identify this value. Therefore, CPAchecker creates, for example the formulas `__VERIFIER_nondet_int!2` $= 5$, `__VERIFIER_nondet_int!3` $= 0$, ... where the number after the exclamation mark equals the unique ID.

Semantic faults have to be indicated by an ERROR-label as seen in Program 5 on the next page. Whenever the analysis of CPAchecker reaches such a label it concludes that the program is buggy and hence creates the counterexample which we use for further analysis.

6.3 Counterexamples in CPAchecker and Preprocessing

In Chapter 3 we assumed that every error path ends with an assume statement that becomes the post-condition ϕ. In real world programs this is not applicable because there may be statements or logging before the program reaches the error label. The counterexample contains these statements because they have to be executed before reaching the ERROR-label. Thus, we have to make sure to take the last known assume edge as post-condition. CPAchecker produces a list of `CFAEdges` representing the error path from $l_0 \in L$ to $e \in E$. The class `CFAEdge` provides several useful methods and attributes like the line number in the original input file or the type of the edge. Possible edge types are *return edge, assume edge, statement edge, declaration edge, blank edge* or *function return edge*. Every type indicates the kind of transition. On a blank edge nothing happens, a declaration edge indicates a variable declaration, a statement edge indicates an operation on a variable and the assume edge is equal to a condition like `x` > 2. The different return edges refer to the return value of a function.

A peculiarity of CPAchecker is the handling of if-statements whenever it consists of a sequence of Boolean expressions (e.g., `if`$(x > 0 \wedge y > 10)\{\dots\}$). CPAchecker will generate two `CFAEdges` ((Node 1: $x > 0$) and

(Node 2: $y > 10$)) for this if-statement. To determine that they are components of the same if-statement we have to compare the type and their starting line in the original file. The generated counterexample is the foundation to the trace formula.

6.4 Trace Formula

The trace formula is the core component of all three algorithms. It computes the precondition, $\texttt{TF}(\pi)$, $\texttt{TF}_\Lambda(\pi)$ and the post-condition. In a preprocessing step we remove all unnecessary edges like blank edges and store the line of the last *assume edge* in `lastAssume` for the post-condition.

First, we want to compute a Boolean formula for each edge including the creation of its selector and its SSA-map for this point of execution. Afterwards, we extract the SSA-map and the formula for the edge in each step and create a unique selector. The post-condition is created on the fly. We iterate over the list of `CFAEdges` and conjunct edge by edge to an instance of the class `PathFormula`, an existing class for creating Boolean formulas with correct SSA-maps based on `CFAEdges`. The correctly initialized formulas are added to a list called `atoms`. The list contains the Boolean formula for all edges such that $\texttt{TF}(\pi) = \bigwedge_{\texttt{a} \in \texttt{atoms}} \texttt{a}$. If an edge has the type assume edge and additionally the starting line in the origin is equal to `lastAssume` we instead add it to the list `negated`. Therefore, $\phi = \neg \left(\bigwedge_{\texttt{p} \in \texttt{negated}} \texttt{p} \right)$.

The selectors are stored in a `HashMap` as a key pair value of a Boolean formula mapping to its selector. The class `Selector` has the static method `of(BooleanFormula formula)` to query the selector for a given formula if present.

```
1  extern int __VERIFIER_nondet_int();
2
3  int main (void) {
4        int x = __VERIFIER_nondet_int();
5        int y = 0;
6        if (x == y) {
7           goto ERROR;
8        }
9  EXIT:
10    return 0;
11 ERROR:
12    return 1;
13 }
```

Program 5: Model

Via a prover we calculate the precondition as a model of $\texttt{TF}(\pi) \wedge \neg\phi$ and extract the non deterministic variables. As mentioned before the model can be understood as a variable assignment that satisfies $\texttt{TF}(\pi) \wedge \neg\phi$. The model for Program 5 equals $m \Leftrightarrow \texttt{__VERIFIER\_nondet\_int!2} = 0 \wedge x = 0 \wedge y = 0$. Thus, our precondition evaluates to $\psi \Leftrightarrow \texttt{__VERIFIER\_nondet\_int!2} = 0$ whereas $\texttt{TF}(\pi) \Leftrightarrow x = \texttt{__VERIFIER\_nondet\_int!2} \wedge y = 0$ and $\phi \Leftrightarrow x \neq y$. We compute $\texttt{TF}_\Lambda$ analogously to the calculation of $\texttt{TF}(\pi)$ by $\texttt{TF}_\Lambda(\pi) \Leftrightarrow \bigwedge_{\texttt{a} \in \texttt{atoms}} (\texttt{selector(a)} \Rightarrow \texttt{a})$.

6.5 Single-UNSAT-Core Algorithm

We start with the implementation of SingleUnsatCore (Algorithm 1). With the help of the included tools of CPAchecker and the `TraceFormula` object we were able to translate the pseudo code nearly one-to-one into the CPAchecker-framework. With the use of the solver we compute a list of clauses whose conjunct is an UNSAT core. Afterwards, we map each element of the list to the previously created selector and return one `Fault` containing the `Selectors`. Remember that they inherit from the class `FaultContributions`.

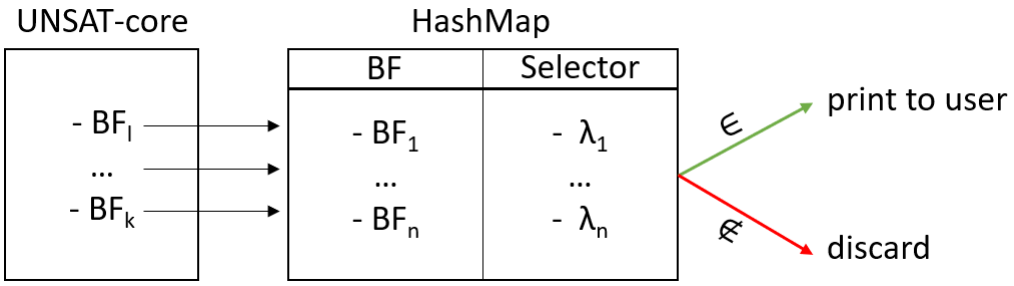


Figure 4: Mapping of UNSAT core to selectors

Figure 4 illustrates the mapping of Boolean formulas (BF) to selectors (λ). We obtain a list of clauses whose conjunct is an UNSAT core. The indices l and k must not be element of the interval $[1, \ldots, n]$. In a next step we look up if a Boolean formula of the list maps to a known selector, i.e., a location in the program. Whenever this is the case we print the location to the user, otherwise we discard the entry. This only happens on condition that the precondition or the post-condition is part of the UNSAT core. In our implementation we do not add the precondition beforehand to avoid the precondition being part of the UNSAT-core. We only add the precondition if the trace formula would be satisfiable otherwise.

6.6 MAX-SAT Algorithm

Since we have no access to a `CoMSS`-solver we had to modify the algorithm that slightly differs from the theoretic aspects presented in [16]. In our implementation we successfully used Algorithm 5 and Algorithm 6 to replace the `CoMSS`-solver and obtain MIN-UNSAT cores. Once again, we were able to translate the pseudo code with minor changes to Java. To check for super- and subsets we used the method `containsAll` provided by the `HashMaps` of the standard Java library. In conclusion MaxSat enhances SingleUnsat-Core by computing all possible cores. As already mentioned, some cores might not contain the positions where the programmer is willing to make adaptations but now he has access to a whole set of possible minimal sets of locations where adaptations might be suitable.

6.7 Error Invariants Algorithm

The algorithm demands using the correct SSA-maps on interpolants to find an interpolant inductive at several positions in the trace formula. Therefore, we maintain a list of SSA-maps for every position, i.e., for every time stamp in the trace formula. This eases the correct shifting of interpolants. The original algorithm [9] requires many shifts of the interpolants and the post-condition. Our implementation reduced this to just a single shift of the interpolant. Instead of restarting with the SSA-indices set to 0 for every new conjunct of formulas we just shift the interpolant once.

To store the boundaries we implemented the hashable class `Interval` with the attributes `start, end` and `interpolant`. This class together with the class `Selector` implement the marker interface `AbstractTraceElement`, allowing us to maintain an alternating list of interpolants and transitions (selectors). Since the interpolants are unreadable and the resulting abstract error trace cannot be depicted in the visual report for fault localization algorithms we had to make minor adaptions. We had to introduce a new option (`maintainhierarchy`), post process the intervals to improve readability and find a way to represent the abstract error trace. We realized the last point by appending the formula of the previous and the following interval to the description of the fault. Every `Fault` consists of exactly one `FaultContribution` which maps to an edge simultaneously contained in π and $\pi^{\#}$.

Apart from this, the implementation equals an one to one translation of the pseudo code which we presented earlier. We combined Algorithm 3 with the suggested binary search in Algorithm 4 and enhanced the checks

of `incLow` with the memoization. This is realized with a hash map, so the expected run time for a lookup is element of $O(1)$. We store every processed interpolant on a certain position together with the result of its validity check. Whenever we find an entry in the hash map, we must not shift formulas and call the solver again. We can skip calculating an already solved problem.

7 Evaluation

7.1 Comparison of the Algorithms

In this section we compare all three algorithms and outline their strengths and weaknesses on specific tasks. We evaluate the precision and the needed time of each algorithm. We consider an algorithm to be precise if it sensibly reduces the selection of lines to look at.

7.1.1 Qualitative Analysis

Implementing and running benchmarks pointed to flaws of the implemented algorithms, on the one hand limited by CPAchecker and on the other hand by the used techniques itself.

Neither does CPAchecker currently support proof splitting nor recursion. Hence, it cannot analyze every program. In a few cases this holds true even for simple programs, for example the recursive factorial-function. Once these features enrich CPAchecker our implementation should work without any changes since it exclusively relies on the correct translation of the list of `CFAEdges` to Boolean formulas. Under special circumstances the verification process may time out, does not terminate or the creation of the counterexample is not possible. In case of this happening we cannot continue to run fault localization. Another problem emerges from the automatic simplification of Boolean formulas through the solver. Too simple programs cannot be analyzed because the solver will simplify the formula to true or false. Imagine a program that consists of 2 transitions, the declaration `int x = 0` and the if-statement "`if (x == 0)`" enabling the reachability of the ERRRO-label. Based on the assumption that the post-condition never is faulty, changing the initial value of `x` fixes the program but our analysis will not return this fix. The simplification of $x = 0 \wedge x \neq 0$ to false causes loss of information. Even though the formula is unsatisfiable the result will be empty. Note that enabling the option "traceformula.altpre" also yields the same problem without the simplification because $x = 0$ will be part of

```
1  int main (void) {
2     int x = 0;
3     x++;
4     if (x == 1) {
5        x--;
6        goto ERROR;
7     }
8  EXIT: return 0;
9  ERROR:    return 1;
10 }
```

Program 6: Problem with post-condition I

the precondition and $x \neq 0$ will become the post-condition. In this case no algorithm can return transitions because π does not contain any transitions.

As already discussed, our implementation takes the last assume edge as the post-condition, so we recommend to deliberately place a if-statement containing the post-condition before the execution reaches the ERROR-label otherwise this can cause problems and unwanted behavior. For that consider Program 6 above. Although, the statement `x--` in line 5 has no effect at all to the error, it will be part of the trace formula and get a selector. This means that we have more selectors to loop through and more subsets to check for unsatisfiability when we run MaxSat. Switching line 5 and line 6 has no semantic effect but will reduce the amount of selectors. Using MaxSat, it is worth looking for this kind of optimizations to reduce computation time. Nevertheless, the calculated results will remain valid since the algorithm ignores all latter changes of x because the post-condition has a lower SSA-index and thus is not connected to changes after the if-statement.

```
1  int main (void) {
2     int x = 0;
3     x++;
4     if (x == 1) {
5        int i = 0;
6        int j = 2;
7        while (i != j)
8           i++;
9        goto ERROR;
10    }
11 EXIT: return 0;
12 ERROR:    return 1;
13 }
```

Program 7: Problem with post-condition II

Nonetheless, we can adapt Program 6 with a seemingly uninteresting change and obtain a completely different result. We replace `x++` with a meaningless while-loop and obtain Program 7. We can see that the while-loop in line 7 - not even using the variable `x` - has no impact on the reachability of the ERROR-label in line 12 through the goto statement in line 9. However, the computed post-condition equals $\phi \Leftrightarrow (i \neq j) \wedge (i' \neq j) \wedge (i'' = j)$ since the post-condition always consists of all previous and nearest assumes on the same line. The analysis now returns the set $\{i' = i+1, i'' = i'+1\}$ as possible locations for fixes which has nothing to do with the root cause of the bug. In fact these locations are only reachable because the bug occurred. To solve this problem we need to find out which assume edge is the direct parent of the `goto ERROR` call instead of always taking the last assume edge. For now the user should avoid the execution of irrelevant assumes before reaching the ERROR-label.

```
1 /** ... code ... */
2 if (a) { if (b && c) {
3     goto ERROR;
4 }}
5 /** ... code ... */
```

Program 8: Problem with post-condition III

A related problem arises by the computation of post-conditions if many if-statements are in the same line, like in Program 8. Although, the program has two separate if-statements the post-condition ϕ will read $\phi_1 \Leftrightarrow \neg(a \wedge (b \wedge c))$. Under certain circumstances this might not equal the intended post-condition $\phi_2 \Leftrightarrow \neg(b \wedge c)$ since ϕ_1 does not allow changes on the first if-statement during the analysis. Lastly, we will discuss the problems of the algorithms themselves. Calling functions for important computations within the function that contains an ERROR-label leads to weaker results when we use MaxSat.

```
1  extern int __VERIFIER_nondet_int();
2
3  int isErr (int x) {
4      if (x != -1 && x != 6) {
5          return 1;
6      }
7      return 0;
8  }
9
10 int main (void) {
11     int x = __VERIFIER_nondet_int();
12     if (x > 0) {
```

```
13          if (x < 5) {
14              if (x > 1) {
15                  int error = isErr(x);
16                  if (error == 1) {
17                      goto ERROR;
18                  }
19              }
20          }
21      }
22      EXIT:
23          return 0;
24      ERROR:
25          return 1;
26  }
```

Program 9: Problem with function calls

Program 9 is an adaption of Program 3 transformed into the syntax of CPAchecker. Without the external post-condition check in function `isErr`, our implementation returns exactly the expected sets of selectors shown in Table 3. Instead of checking if "`x != -1 && x != 6`" directly before we call "`goto ERROR`" in line 16, we outsource the computation in to the function `isErr`. Since the post-condition now reads `main::error != 1` instead of `main:x == -1 || main:x == 6` we will not find any of the sets in Table 3. Analyzing Program 9 yields the following formulas for $\psi, \mathtt{TF}_\Lambda(\pi)$ and ϕ:

ψ: $\mathtt{\_\_VERIFIER\_nondet\_int!2 = 2}$

$\mathtt{TF}_\Lambda(\pi)$:

$$\begin{aligned}(\lambda_0 \Rightarrow (\mathtt{main} :: \mathtt{x} = \mathtt{\_\_VERIFIER\_nondet\_int!2}))\wedge\\ (\lambda_1 \Rightarrow (\mathtt{main} :: \mathtt{x} > 0))\wedge\\ (\lambda_2 \Rightarrow (\mathtt{main} :: \mathtt{x} < 5))\wedge\\ (\lambda_3 \Rightarrow (\mathtt{main} :: \mathtt{x} > 1))\wedge\\ (\lambda_4 \Rightarrow (\mathtt{isErr} :: \mathtt{x} = \mathtt{main} :: \mathtt{x}))\wedge\\ (\lambda_5 \Rightarrow (\mathtt{isErr} :: \mathtt{x} \neq 1))\wedge\\ (\lambda_6 \Rightarrow (\mathtt{isErr} :: \mathtt{x} \neq 6))\wedge\\ (\lambda_7 \Rightarrow (\mathtt{isErr} :: \mathtt{retval} = 1))\wedge\\ (\lambda_8 \Rightarrow (\mathtt{main} :: \mathtt{error} = \mathtt{isErr} : \mathtt{retval}))\end{aligned}$$

ϕ: $\mathtt{main} :: \mathtt{error} \neq 1$

CPAchecker uniquely identifies variables by putting the function name in front of them. To make the trace formula $\mathtt{TF}_\Lambda((\psi, \pi, \phi))$ unsatisfiable the

```
1  /** ... code ... */
2  int input = __VERIFIER_nondet_int();
3  int copy = input;
4  int test = 1;
5  if (input <= 0)
6     goto EXIT;
7  for (int i = 2; i < input; i++) {
8     if (isPrimefactor(i, input)) {
9        test = test * i;
10       input = input / i;
11       i = 2;
12    }
13 }
14 if (test != copy) {
15    goto ERROR;
16 }
17 /** ... code ... */
```

Program 10: Problems with understanding the semantics

set $\{\lambda_7, \lambda_8\}$ suffices. The analysis will now state that we have to change the return value of the function `isErr` but since it represents the outsourced post-condition we do not want to make changes there. Note that every possible UNSAT-core has to contain selector 7 and selector 8 because otherwise we have no connection to the variable `main :: error`. This means that at least these two selectors are always part of the UNSAT-cores. In addition we see that they already make the formula unsatisfiable when marked as hard so in fact we only obtain one result set containing these two selectors. We frequently encountered this problem in our tests. ErrInv performs better in these situations because we have access to the describing interpolants.

So that the algorithms understand the meaning of the program we need to define the pre- and the post-condition accordingly. In some cases this is not sufficient to guarantee a reliable and satisfying analysis. That becomes clear in case of the wrong handling of the variable `i` in Program 10. From line 7 to line 13 we want to calculate all prime factors of the number `input` if it belongs to the natural numbers. The implementation of `isPrimefactor(int factor, int number)` is bug-free. The faulty behavior can be observed if we input the number 4. We will find the first prime factor 2 in the first iteration. Since we reset i to 2 instead of 1 in line 11 we cannot find the second factor 2 as the loop starts with $i = 3$ because we increment i after resetting. Changing line 11 to $i = 1$ fixes the bug. However, the analysis states that we should fix the program using lines 2, 3 and 9. Semantically this makes no sense since it only contains inputs and the test variable `test`

but actually changing these lines can indeed prevent reaching line 15. For instance, replacing line 9 with the statement `test = copy` fixes the program for all possible inputs in terms of making the ERROR-label unreachable. In this case ErrInv exclusively contains exactly the same edges in the abstract error trace $\pi^{\#}$. Hence, all algorithms have weaknesses in case of usage of such actually useful test variables. Every time a program uses such variables the algorithms will at least suggest at one point to change the value of the test variable to the correct one in the post-condition. Thus, designing the post-condition has a massive impact on the quality of the result. We experienced that replacing the nondeterministic variables with a concrete failing input yields slightly better results. On condition that the inputted program has no assume edges the analysis will return nothing and log an error message explaining that no relevant edges could be found.

7.1.2 Limits of the Algorithms

```
1  /**
2    * Check if f1 and f2 are prime factors of number
3    * @param f1: factor 1
4    * @param f2: factor 2
5    * @return: are f1 and f2 prime factors of number?
6  **/
7  int test(int f1, int f2, int number) {
8     //isPrimeFactor is implemented correctly.
9     if (isPrimeFactor(f1, number)) {
10       return 1; //true
11    }
12    return 0; //false
13 }
```

Program 11: Problems with missing method-calls

In this subsection we want to outline problems of the algorithms that are not connected to the implementation per se.

None of the algorithms is capable of determining missing method calls. Program 11 illustrates this problem. The developer forgot to also check if `f2` is a prime factor of `number`. The inputs $3, 5, 12$ let the program return 1 instead of 0 as 5 is not a prime factor of 12. All of the algorithms will now mark locations in the correct implementation of `isPrimeFactor` as they cannot conclude that the user forgot another check and operate on the trace formula only.

Furthermore, all algorithms tend to mark to the root cause of the error unimportant variables. The initial variable assignment has, of course, a great

impact on the verification result. Variables that are not part of the precondition can be marked as potential bug sources. Hence, such variables will be marked more often when they are used frequently, even if the cause of the error is located elsewhere. The participants of the survey also confirmed this assumption and found the marking of these variables distracting. Disabling the option `altpre` only allows nondeterministic variables to be part of the precondition. Meaning, that variables with known values will most certainly be part of faults which decreases the efficiency.

7.1.3 Run Time and Precision

Unsurprisingly, SingleUnsatCore turns out to be the weakest but the fastest one. Calculating only a single UNSAT-core of arbitrary length is not precise enough. ErrInv outperforms MaxSat. On average the run time is lower because the binary search for the interval boundaries reduces the costly calls to the solver whereas MaxSat loops through many possible subsets. Additionally, the abstraction of error traces is more powerful than returning a selection of locations. Looking at Program 4 illustrates this. Whereas ErrInv is able to abstract the error trace stating that lines 4 and 5 do not add anything to the code, MaxSat returns nothing. MaxSat cannot find a MIN-UNSAT core since all edges are relevant to satisfy the post-condition and it will not return all locations as an UNSAT-core. The big advantage of ErrInv over MaxSat lies in the adequate shifting of the interpolant depending on its position. MaxSat uses the trace formula and does not adapt the SSA-maps which means, for example, in Program 4 every transition must be considered.

To evaluate the precision and the run time we created 17 faulty programs of varying length and executed the benchmark with the BenchExec tool [3]. We run the benchmark on a Intel Quad Core N4200 1.1GHz processor with 8 gigabyte DDR3 RAM to measure the CPU-time until the creation of the final report finished. Hence, the parsing and the analysis is part of the measured time.

The tasks cover different types of bugs and test edge cases. CPAchecker can create a counterexample for all tasks and verifies the programs correctly. Additionally, we activated the option `altpre` for all algorithms.

All benchmarks were executed with revision 34003 of the CPAchecker repository[1].

[1]Repository: https://svn.sosy-lab.org/software/cpachecker/!svn/bc/34003/

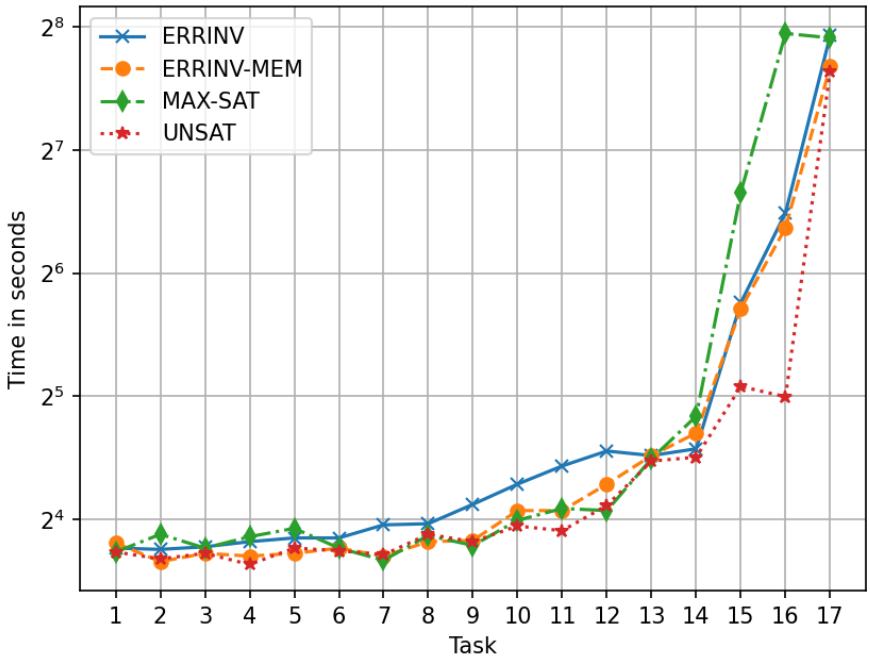


Figure 5: CPU-time per task

Figure 5 shows the results for each task on the x-axis. The y-axis, showing the CPU-time in seconds, has a logarithmic scale. On average the SingleUnsatCore (red line) has the best run time over all tasks followed by ErrInv with memoization (orange line). MaxSat and ErrInv (without memoization) have the worst run time. On long counterexamples the run time of ErrInv is better than on smaller counterexamples compared to MaxSat. Both algorithms often call the solver. Whereas MaxSat loops through every possible and promising subset of selectors, ErrInv has to make $2 \cdot \log_2(n)$ calls per interpolant to the solver, so $2n \cdot \log_2(n)$ calls in total. For every UNSAT-core MaxSat has to call the solver $n + n|m| - |m|^2$ times where n denotes the number of selectors and $|m|$ denotes the size of the found UNSAT-cores. In total $\sum_{m \in M} (n + n|m| - |m|^2)$ calls to the solver are needed where M is the set of all available UNSAT-cores. We sketch the proof with an example. Let S be the set of n selectors and $|m|$ the size of the first UNSAT-core $U = \{\lambda_0, \ldots, \lambda_{|m|}\}$, consisting of the first $|m|$ selectors in the set S. Remember that we remove selector by selector from S to determine if the reduced set still is unsatisfiable. In our example the first time unsatisfiability is guaranteed when we remove a selector that is not an

element of U. For this we need $|m|+1$ calls to the solver. After removing this selector, we rerun the procedure with the reduced set. This will be repeated $n-|m|$ times until we cannot remove another selector anymore. In conclusion we have $(n-|m|)(|m|+1)$ calls to the solver. If we now add the missing $|m|$ calls to verify that the current core is minimal, we need to make $(n-|m|)(|m|+1)+|m| = n+n|m|-|m|^2$ calls for each possible core. The run time of MaxSat improves if the cardinality of M is small. The presented run times are rough estimates of the worst case run times. These factors are the reason why both algorithm intersect several times in the graph of Figure 5.

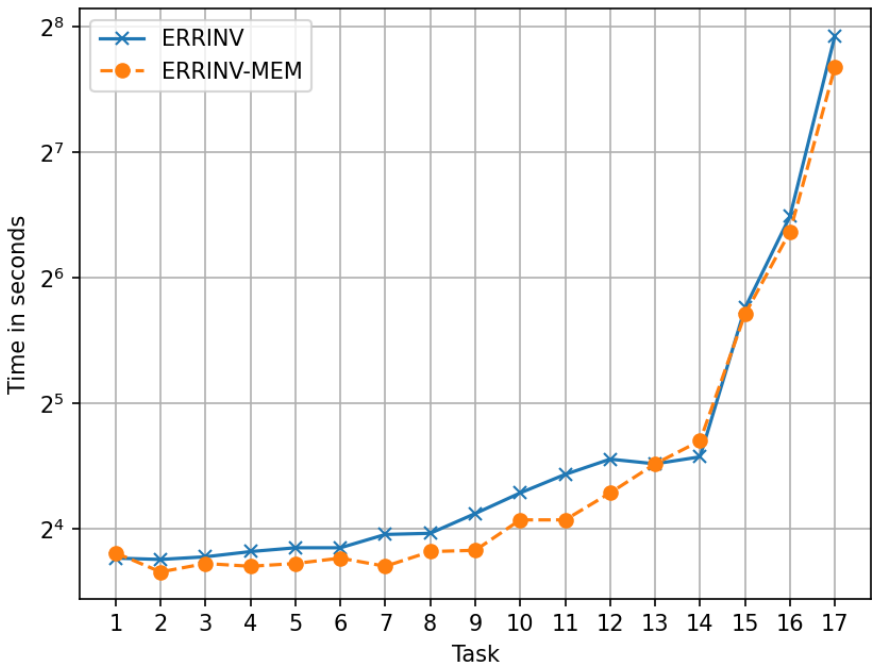


Figure 6: Comparison of ErrInv with and without memoization

Figure 6 shows that the addition of memoization improves the run time in our benchmark-set. Depending on the task and the size of the inductive intervals the run time decreases. The more interpolants are stored and reused while computing the error trace, the higher is the benefit of memoization. Finally, we have to look at the precision of the algorithms and set the needed time into perspective with the effectiveness.

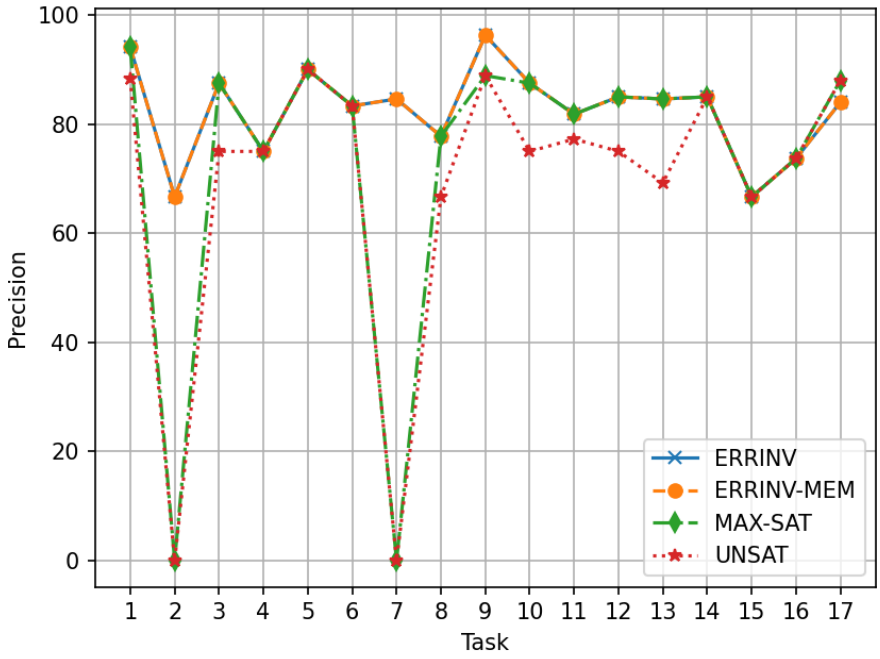


Figure 7: Precision of the algorithms

Figure 7 shows the number of unmarked lines relative to the number of total lines in percent on the y-axis. We only count lines that are part of the actual source code. Comments and empty lines are not counted. SingleUnsatCore tends to return larger subsets where the other algorithms ensure a minimal result. Logically, it always is a lower bound of MaxSat. The results of ErrInv with memoization matches the results of the standard algorithm. We see that the abstraction of ErrInv not only provides additional information and an explanation but also yields a higher precision. Combined with a reasonable run time ErrInv provides the most useful information.

7.1.4 Fitting Scenarios

SinleUnsatCore only requires to create the trace formula and call the solver once. Thus, the run time in terms of the cost-intensive solver calls will always be the best compared to the other algorithms. The usage of this algorithm becomes especially handy for getting first clues on the root cause of the error. The algorithm noticeably reduces the selection of error prone locations. However, only one core is returned. Possibly, this single core

exclusively contains locations where adaptions are unwanted. Still, for small programs the results are sufficient to make first observations on important variables.

Consequently, MaxSat is the better choice for larger programs. It shows all possible UNSAT-cores. Since the user often has an intuition which part of the program is particularly prone to errors he knows what UNSAT-cores are more important to look at than others. Additionally, the newly introduced options allow to easily filter relevant information. Contrary to ErrInv it guarantees to return locations where fixes can be made. Nevertheless, the locations might not be suitable places to make adaptions in terms of semantic correctness.

ErrInv yields the most information. Instead of returning subsets of locations it explains the faulty behavior through an abstract error trace. Understanding the reduced abstract error trace allows conclusions on why the program is failing. Thus, the algorithm is very useful for large programs, too. In addition, the programmer does not have to know or understand the program to debug in advance. Unnecessary variables are removed and in the best case complicated code is expressed as a single error invariant. Through the abstraction the faulty behavior can be detected without knowing the whole program.

In the next section we evaluate our survey that simulated this setting. We created reports with the results of ErrInv and gave it to 18 participants. Their task was to find and fix a bug in a to them unknown program.

7.2 Results of the Survey

7.2.1 Setting

Process of the survey. At the beginning, we gave a very detailed introduction on how to use the visual report. Every participant then had the chance to try out the new knowledge and test the report on an example task. This task was designed similar to the upcoming tasks but we told them that we will not evaluate it to make sure that they take their time to experiment with the report. After the introduction we split the participants into two groups. Every group had to find and fix a semantic bug in 2 programs with and 2 programs without the support of fault localization (FL). On condition that fault localization was disabled the contenders still had access to the standard report of CPAchecker (c.f. Figure 2). In every task they had to select the line(s) in which the participants suspected the error and add a description on what to change. Finally, they had to type in the needed time in seconds.

Before submitting the survey we asked them to grade all features and give feedback on the report and their experience. In our benchmarks and qualitative analysis above we discovered that ErrInv yields the best results so we used it to generate the reports for the survey.

Table 7: Usage of fault localization

Group	Task 1	Task 2	Task 3	Task 4
1	no FL	no FL	FL	FL
2	FL	FL	no FL	no FL

Table 7 shows which group was supported by fault localization on the different tasks. The survey was anonymous. We appended all tasks of the survey at the end as appendix together with a possible bug fix for each task.

Participants. We sent the survey to a variety of people with knowledge in computer science. In total 18 participants consisting of students (fifth semester or higher), scientists, junior and senior developers took part. The junior and senior developers are employed in a company for embedded software engineering in C. We collected 10 data samples for group 1 and 8 data samples for group 2.

Rules. We gave a small ruleset to every participant shown below:

1. All programs were able to be compiled and they terminated with no run time errors.
2. All programs had a bug (the calculated result does not match the expected result) and reached the ERROR-label (i.e., the line "goto ERROR;" was called eventually).
3. Swapping or deleting lines was prohibited.
4. Changing one or multiple lines could fix the bug.
5. A program is considered to be correct if it never reaches the ERROR-label. This means that the line: "goto ERROR" must never be executed. Any changes to any lines preserving the semantics are allowed. The participants were asked to ensure this.
6. All lines below the comment //POST-CONDITION were not allowed to be changed.

7. Checking/validating the answers with external tools beforehand was prohibited.

7.2.2 Practical Performance of Fault Localization

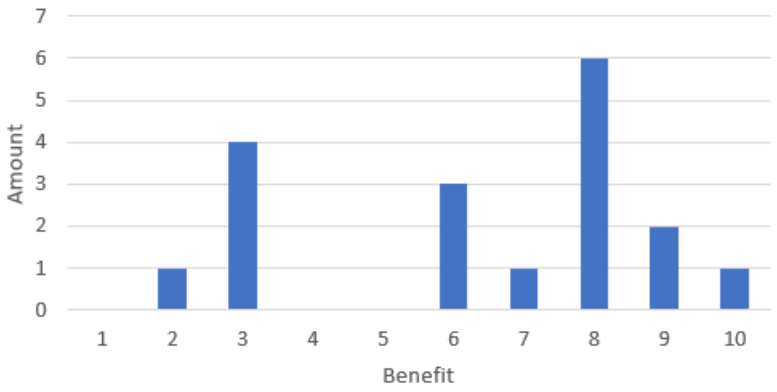


Figure 8: Estimated benefit of fault localization

After finishing all 4 tasks, we asked the participants to estimate how munch they think they benefited from fault localization. Figure 8 shows the results where the x-axis shows the estimates and the y-axis shows the number of votes. A first analysis shows that the mean equals 6.4 and the median 7.5. In conclusion most participants felt an improvement when using fault localization.

We now want to evaluate if the estimates coincide with the actual results. Therefore, we analyzed the two key features *needed time* for and *correctness* of the fix. We compare the needed time with and without fault localization and put it into context with the correctness.

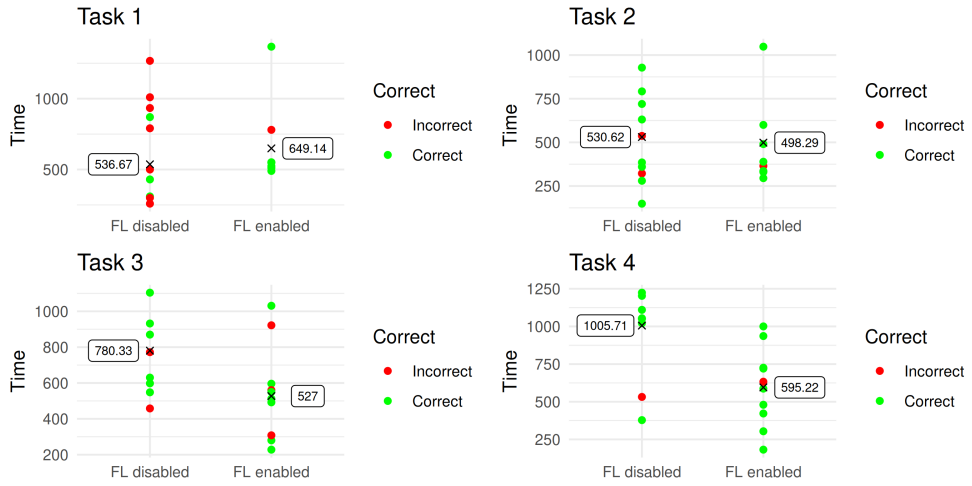


Figure 9: Time and correctness with and without fault localization

Figure 9 shows the performance of the participants with and without help on every task. The y-axis describes the needed time. The red and green points represent the participants where a green point means that the suggested fix of the participant indeed fixes the bug. Red points denote incorrect fixes. The cross (×) indicates the average time needed to find a **correct** solution. General speaking: the lower the cross on the y-axis the better the performance on the task on average. Note that the cross ignores incorrect fixes. Looking at the position of the crosses shows big improvements on task 3 and 4. Fault localization on tasks 1 and 2 seems to be not useful at first glance but regarding the correctness fault localization increased the quality of the fixes. Without fault localization people did not fix the bug twice as often. In fact, not using fault localization caused 67% of all wrong answers.

Table 8: Correctness with and without fault localization

	no FL	FL	Σ
Incorrect	12 (16.7%)	6 (8.3%)	18 (25.0%)
Correct	24 (33.3%)	30 (41.7%)	54 (75.0%)
Σ	36 (50.0%)	36 (50.0%)	72 (100.0%)

Table 8 contrasts the correctness with the usage of fault localization. In sum, 72 fixes, 18 for each tasks, were submitted. With fault localization, 30 out of 36 people made correct fixes which are 6 more than without its usage. Especially task 1 seems difficult to solve without help. Only 30%

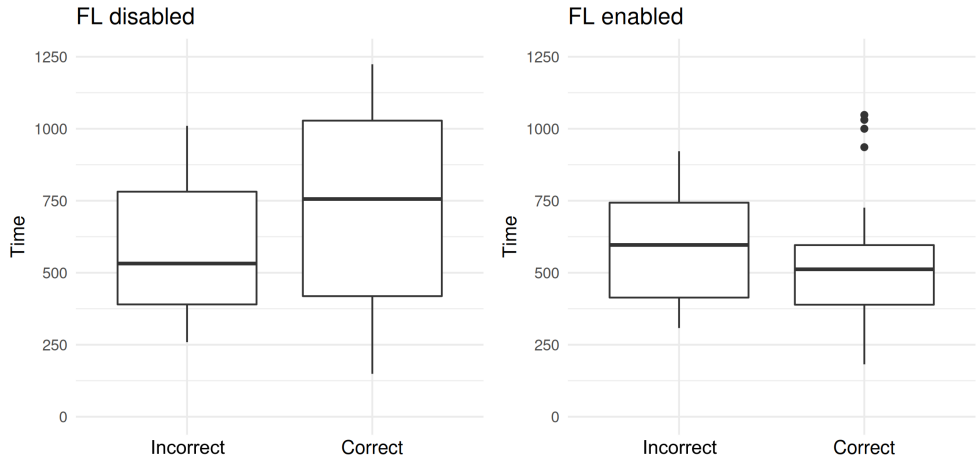


Figure 10: Connection between time and correctness with and without using fault localization

of all participants without help could solve the problem while 87.5% of all partakers using fault localization found a fix. Back to Figure 9 we adhere that fault localization improved the *needed time* in nearly all tasks. Still, if fault localization did not improve the time, it still improved the correctness of the fixes. Task 3 remains the only task where the percentage of correct fixes is less with than without fault localization. However, looking at the boxplot [20] in Figure 10 gives reason to assume a connection between improved correctness and needed time. The horizontal line within the boxes is the median of the needed time of all tasks. The rectangular box encapsulates the central 50% of the data points separated by the horizontal line. The so called whiskers reach to the last data point that is at max 1.5 times the size of the box away from it. The dots reassemble the outliers, i.e., data points that are not within the underlying indicated distribution. A first observation shows that people without support came faster to the incorrect answer because they had less clues where to look at. Furthermore the lack of support increased the needed time to find a correct solution for most of the participants as seen in the left subplot of Figure 10. The distribution of needed time for the middle 50% decreases when using fault localization. As seen in the right subplot, the box for correct fixes is smaller and lower than the box in the left subplot. Hence, over 50% of supported people have already fixed the program correctly at a time where most of the unsupported people are still busy figuring out the problem. All but 4 correct fixes of the supported participants are made before about 30% of the unsupported will submit an answer. The boxes for incorrect answers are similar to each

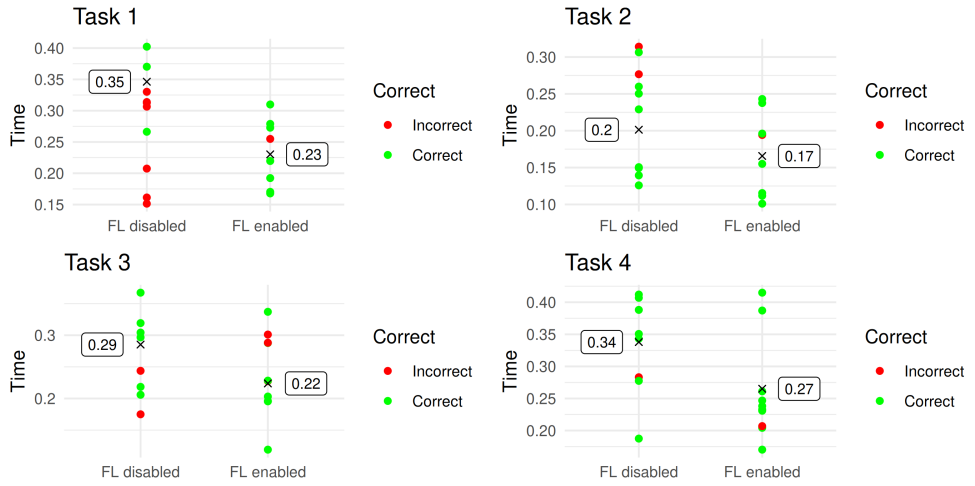


Figure 11: Normalized time and correctness with and without fault localization

other. However, a majority of people without help tend to quit working on the problems earlier. Until now we exclusively considered the raw data and ignored that different participants have different strengths and approaches to solve these kind of tasks. A participant who takes exactly 2000 seconds for each task does not benefit from fault localization but will distort the mean values. To make the data less prone to such cases we normalized the performance of each participant individually by calculating the percentage of time spent on a specific task compared to the summed time of all tasks. The results can be seen in Figure 11. The two axis and the labels are equal to the plots before. We see that the normalization does not have an effect on the end result, fault localization still helps. In fact the average, normalized time spent per task without help now always is higher than the mean values with fault localization for each task (c.f. the crosses "×" in Figure 11). Generally, people saved up to 12% of their overall debugging time on a single task to find a correct solution using fault localization. Additionally, the number of correct fixes increases.

7.2.3 Grading of the Features

The fault localization report contains a lot of information, like the score, hints, fixes, reasons, current values and relevant lines (c.f. Figure 3 b)). We asked the participants to grade them from "not useful at all" (0 points) to "very useful" (4 points). The results can be seen in Table 9.

As expected, *current values* and *relevant lines* are the most liked features

Table 9: Grades of the features

Feature	Mean	Median
Values	3.2	3
Lines	3.2	3
Reason	2.6	3
Hints	2.5	3
Fix	2.4	3
Score	1.6	2

with an average score of 3.2. They replace the long search for the crucial and faulty state including the current variable assignment when using a casual debugger. Instead of figuring out the faulty state step by step fault localization immediately displays it. The partakers consider *hints* and *reasons* to be useful in general. In our survey *hints* disclosed the precondition and the following invariant to the user whereas the *reasons* revealed the invariant that holds up to the current time point. In other words, the user knows what is responsible for the error up to this point, what happens now and what will happen from now on. Tracing these information allows to precisely detect important variables.

Potential fixes are, in the opinion of the contenders not absolutely necessary but looking at the median most people still found it useful. For the first task (Appendix A) the proposed fix stated that there is a fishy calculation within the array index using the variable k and indeed, adapting the for-loop iterating over variable k fixes the bug. The fix for the third task (Appendix C) explained that the function *isSorted* may have a wrong return value or unwanted side effects which was correct. The variables `TRUE` and `FALSE` were declared wrongly. Since fault localization cannot connect a variable name to its implied value (`TRUE` $= 1$ and `FALSE` $= 0$ in this case) it comes to the result that the return values are wrong. Hence, either swapping the initialization of the two variables or removing the exclamation mark (negation) in front of the function-call removes the faulty behavior. We see that the potential fixes are not directly connected to the root cause of the errors which might have caused confusion and therefore reduced the usefulness.

As expected the *scores* received the worst grading because the strength of ErrInv lies in the abstraction. The scores did not even have an impact on the ordering of the faults in the final report because we activated the option `faultlocalization.maintainhierarchy=true`. Perhaps the grading would have been better if the reports were generated after executing MaxSat.

Finally we asked the partakers to describe their feeling about the amount,

the accessibility and the clearness of the additional information compared to the standard report of CPAchecker. They agreed that the additional information neither is misleading nor too much. Furthermore they acknowledged the accessibility of the information. Overall the clearness of the complete report with a mean score of 7.9 out of 10 has been highly appreciated.

7.2.4 Feedback

In total we asked 5 free text answer questions. In this section we will discuss the most outstanding opinions. Right before the first real free text answer, we asked how the participants would proceed when they have to fix a bug and access to fault localization. 11 out of 18 people said that they would use it if they could not find the bug easily. Another 3 people would use fault localization immediately after noticing a bug in their program. However, 2 people would not use fault localization at all. The remaining 2 participants gave own answers indicating that they still want to use a casual debugger and switch to fault localization as a last resort.

In the feedback section we gave the participants the opportunity to suggest features and improvements. The answers can be reduced to two key statements. First, the representation of the data within the fault description should be reorganized and redundant information should be avoided. Additionally, some participants were confused by the visualization of the interpolants and formulas and found them to be unreadable. Improving and extending the class `ExpressionConverter` enables us to tackle this problem in the future. Secondly, most participants preferred seeing the actual values over the textual descriptions. This interpretation can be proven with regard to Table 9 as the values were graded as *(very) useful* with a mean value of 3.2. Instead of the current description, people wished for a description of the causality chain, for example: "If $x = 5$ on line 13 then x will get the value 10 on line 13 which eventually causes the error". In conclusion the report should become more consistent in the reasoning. Enabling the user to make changes to the source code within the report has been a suggested feature which will indeed increase the usability. Another suggestion was to avoid marking input variables as possible sources of bugs. For MaxSat we would have been able to activate the option "ban" to accomplish this but we used ErrInv which does not have access to this option instead. Unfortunately, this will remain a weakness of this technique.

However, looking at the average score of 7.9 in clearness these suggestions are useful improvements for even more convenience and proves us to be on the right way in terms of visualizing the results. The most liked features of the report were the interactivity and the simple access to relevant information

and important scenarios.

7.2.5 Threads to Validity

External Validity

Since the participants did not write the programs by themselves the debugging time increases. We do not know which percentage of the time a participant spent on understanding the program and which percentage were spent on the actual debugging process. Improvements in debugging may be more significant if the user wrote the program because the user can bring the additional information into context much faster.

Furthermore, the tasks were especially created for this survey, meaning that they are not part of real world applications. The usefulness of fault localization for real world applications may differ from the usefulness on our synthethic tasks.

Another thread to validity is given by the tasks themselves. We designed the tasks to be solvable in a reasonable amount of time and hence they are not too complex. In other words, experienced programmers may rapidly fix the bug just by looking and trying to understand the task supported by the fact that we implemented well known algorithms like sorting and prime number detection. Additionally, 18 participants in total forms just a small group of people. The central message may change with more participants. Still, a trend is recognizable.

Internal Validity

Unfortunately, the software used for the creation of the survey does not support time measurement. The participants were asked to input and measure the used time on their own which leaves doubts about the validity of the data, for example because of typos. Moreover, the participants could have double checked their answers with external tools like online compilers although it was prohibited.

The different tasks are based on widely known problems. This means that participants were able to solve tasks without looking at the report. People believing that they did not benefit from fault localization might actually not even have used it.

To not confuse the participants we used the best options for each task, e.g., we maintained the hierarchy to not mix up the abstract error trace. Knowing what which option does, the users can easily figure out the best options by themselves, too. As seen in the first subsection of this chapter,

the run time of the algorithms can be bad. We did not add the time needed to create the report to the time needed to solve a task with enabled fault localization. However, the needed time for task 2, 3 and 4 is less than 3 seconds. We can only find a measurable difference in task 1, since fault localization needed 50 seconds.

8 Future Work

There are multiple possible extensions and improvements for the algorithms. MaxSat returns different sets of possible error-prone locations. Based on the edge type that has to be executed to reach these locations we can propose actual fixes for the most common mistakes programmers make, as, for example, off-by-one errors or wrong indexing. Wrong indexing occurs whenever a program uses iteration variables that additionally are used to get access to an entry of an array or some other kind of collection. Off-by-one errors are mistakes where the programmer missed to add or subtract 1 of a variable, for instance in a conditional statement. Looping through a list of size n can lead to an error if one forgets that indexing starts at 0. The for-loop $\texttt{for}(\texttt{int i} = 1; \texttt{i} \leq \texttt{n}; \texttt{i} = \texttt{i} + 1)$ in a sense contains two off-by-one errors because we can fix the bug by adapting the loop to $\texttt{for}(\texttt{int i} = 1 - 1; \texttt{i} \leq \texttt{n} - 1; \texttt{i} = \texttt{i} + 1)$. The process of figuring out the existence of an off-by-one error in a program can be automized [16]. Assume that MAX-SAT found an UNSAT-core containing $\{(z' = z+1), (x <= y)\}$ as subset and $\{x, y, z\} \subseteq X$. To check if this fault is an off-by-one error we can simply rerun the verification algorithm on the same program but before we change X to $(X \setminus \{x\}) \cup \{x \pm 1\}$. We repeat this for every variable in $\{x, y, z\}$ and track the behavior. If the program is now feasible without reaching an error label $e \in E$ we successfully fixed an off-by-one error and we output this to the user. This extension would not be applicable for ErrInv since the actual locations in the abstract error trace are not necessarily the locations where changes have to be made in practice. In a latter step we can even output the corrected file to the user.

Nevertheless, there is an extension that is especially useful for ErrInv, called *flow sensitive trace formula* (`FSTF`) [5]. Until now all algorithms relied on a a trace formula based on the CNF of ψ, π and ϕ. The idea behind `FSTF` is to alter the construction of π: instead of a conjunction of all statements, assertions of if-statements imply all the statements that are part of the if-block. The `FSTF` is implemented but cannot used for now because we are missing one important part. Although there exists a dependence graph [8,

10], CPAchecker does not provide sufficient information about edges to tell whether the edge is part of an if-block especially when it comes to nested ifs. As soon as determining the end of an if-block is possible the `FSTF` can be integrated and tested. For Program 1 with the failing input `x = 2`, the flow-sensitive trace formula equals $(x = 2) \wedge (\neg(x < 0) \Rightarrow (x \geq 2 \Rightarrow (x' = x + 1))) \wedge \phi^{\langle 3 \rangle}$. Flow-sensitive trace formulas can determine whether certain if-statements are important for an error because of the implications, which is just one of their advantages.

The evaluation of the survey showed, that the usability of the report can be improved in terms of the representation of the data. In fact the used rankings can be optimized and adapted to special cases and the combination of them may be changed, too. The `InformationProvider` only knows a few patterns. Adding new ones is another target for future work. At the moment `NoContextExplanation` is the most used class for providing fixes. As the name indicates, the fixes are exclusively based on the edge type. For an *assume edge* the class advises to replace the operator (e.g., $<$) with any of the other possible operators ($>, \leq, \neq, = \ldots$). This is good for giving the user a first idea but not helpful in fixing the bug. In the future this can be refined if we, for example, think of the extension of MaxSat to fix off-by-one errors. Internally, CPAchecker handles arrays in a cryptic - for new user - not readable way. Currently this is bypassed by just telling the user that the values of a certain array are responsible for an error instead of the detailed information. We began to implement the class `ExpressionConverter` that converts Boolean formulas from the unreadable prefix notation in infix notation while replacing unimportant information. Still, the `ExpressionConverter` is not able to deal with all kinds of formulas yet and hence should not be used by default.

Also suggested by the participants of the survey, another convenient feature to improve the usability would be the addition of a selection of possible fixes to every computed and promising location, implemented as a drop down menu. The fixes can be proposed based on the edge type or a more complex analysis in a latter step. This complies our goal of making the report more interactive. By clicking a marked line the report will show a selection of fixes and apply it on a copy of the analyzed program. The user can proceed to rerun the analysis with the copy and after a successful rerun he can adopt the fix to the original file. After an unsuccessful run he can look at other potential fixes or figure out the fix by himself. For this scenario, allowing modifications of the program within the report are mandatory. Consequently, the user does not have to leave the report until he found the bug.

Lastly, we want to implement another approach for calculating the alternative precondition. Instead of removing *declaration edges* and directly

adding it to the precondition we want to replace the value of the statement by a new variable and add the new one to the precondition. This complies the concept of the nondeterministic variables. For example we would change `a = 5` to `int a = newVariable` and add `newVariable = 5` to the precondition. This means that the variables still get a selector and either are part of the fault or not relevant which automatically means that we gain more information.

9 Conclusion

We implemented three fault localization algorithms based on error invariants and UNSAT-cores and optimized them in their run time. Moreover, we created an interactive and clear presentation as a HTML-report. Additionally, we boosted the quality of the results and the report by describing potential fixes and the addendum of related hints. Instead of printing the formulas in the string representation of the implemented solver, we optionally present a clear and readable version in the final report. We even look for common faulty patterns, like calculation within the array subscript and the suspicious use of iteration variables. After noticing that ErrInv often runs checks on equal interpolants on the same splitting location, we extended ErrInv by using memoization. The benchmarks showed improvements in the run time, i.e., the calls to the solver are noticeably reduced. In larger programs the time reduction will be more significant. Of course, the algorithms are limited to the strength of CPAchecker and its analysis. We saw that the algorithm fail on spurious counterexamples and recursion is not supported by now. The usage of arrays and character can lead to inconsistencies, too.

We let 18 participants fix bugs in 4 programs with and without the assistance of fault localization. The results of the survey indicate that most people rapidly got used to the graphical report. Apart from that, fault localization improved the needed time and the correctness of the participants on several programs with different types of bugs. In a next step, we evaluated the effectiveness of three algorithms in a qualitative analysis and showed their strengths and weaknesses. SingleUnsatCore gives us quick access to a reduced set of locations whereas MaxSat calculates all minimal positions in the program but with increased run time. We think seeking the assistance of fault localization decreases the time for the localization of challenging bugs since we have access to the current variable assignment without further investigation. In addition we obtain a limitation of possible locations. Even if ErrInv is not necessarily on point in terms of finding the exact bug locations the abstract error trace gives a concise and helpful explanation. The results of the survey underline this statement. Most participants felt that

they benefited from fault localization and the comparison of time needed per task with and without using fault localization verifies this.

In summary the algorithms turn out to be a useful alternative to classic debugging although it has drawbacks. For most of the disadvantages we at least added helpful features to overcome the issues. Test variables containing the expected result are often marked as potential fault locations because changing their value would indeed make the ERROR-label unreachable. However, it would not fix the bug by any means because of semantic incorrectness. For MaxSat we introduced the option `ban` to prevent taking the variable into account. As a result we only show less confusing faults without this variable in the report. By exactly specifying variables that should be contained in the precondition we can refine the results, too. Most of the time, the user will have an intuition on which variables may be important. Our options then provide some degree of freedom and the programmer may optimize the results by giving the algorithms additional information that are not obvious to them.

We successfully implemented three fault localization algorithms based on error invariants and UNSAT cores. We evaluated the effectiveness and the usability and helpfulness of the visual report embedded in a specially developed data structure. The results show that fault localization significantly supports finding bugs in programs by minimizing the locations and providing additional information. Additionally, we enhanced the algorithms with different options and introduced memoization of interpolants for ErrInv to decrease the run time.

10 Bibliography

[1] D. Beyer. Advances in automatic software verification: Sv-comp 2020. In A. Biere and D. Parker, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 347–367, Cham, 2020. Springer International Publishing.

[2] D. Beyer and M. E. Keremoglu. Cpachecker: A tool for configurable software verification. In *International Conference on Computer Aided Verification*, pages 184–190. Springer, 2011.

[3] D. Beyer, S. Löwe, and P. Wendler. Reliable benchmarking: requirements and solutions. 2017.

[4] R. Brummayer and A. Biere. Boolector: An efficient smt solver for bit-vectors and arrays. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 174–177. Springer, 2009.

[5] J. Christ, E. Ermis, M. Schäf, and T. Wies. Flow-sensitive fault localization. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 189–208. Springer, 2013.

[6] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *International Conference on Computer Aided Verification*, pages 154–169. Springer, 2000.

[7] W. Craig. Linear reasoning. a new form of the herbrand-gentzen theorem. *The Journal of Symbolic Logic*, 22(3):250–268, 1957.

[8] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[9] E. Ermis, M. Schäf, and T. Wies. Error invariants. In *International Symposium on Formal Methods*, pages 187–201. Springer, 2012.

[10] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[11] S. Graf and H. Saïdi. Construction of abstract state graphs with pvs. In *International Conference on Computer Aided Verification*, pages 72–83. Springer, 1997.

[12] A. Groce. Error explanation with distance metrics. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 108–122. Springer, 2004.

[13] A. Groce, S. Chaki, D. Kroening, and O. Strichman. Error explanation with distance metrics. *International Journal on Software Tools for Technology Transfer*, 8(3):229–247, 2006.

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

[15] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering.*, pages 467–477. IEEE, 2002.

[16] M. Jose and R. Majumdar. Cause clue clauses: error localization using maximum satisfiability. *ACM SIGPLAN Notices*, 46(6):437–446, 2011.

[17] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[18] R. Sebastiani. Lazy satisfiability modulo theories. *Journal on Satisfiability, Boolean Modeling and Computation*, 3(3-4):141–224, 2007.

[19] M. Weiser. Program slicing. *IEEE Transactions on software engineering*, (4):352–357, 1984.

[20] D. F. Williamson, R. A. Parker, and J. S. Kendrick. The box plot: a simple visual method to interpret data. *Annals of internal medicine*, 110(11):916–921, 1989.

[21] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.

[22] X. Zhang, H. He, N. Gupta, and R. Gupta. Experimental evaluation of using dynamic slices for fault location. In *Proceedings of the sixth international symposium on Automated analysis-driven debugging*, pages 33–42, 2005.

Appendices

A Survey Task I

```
1  /**
2    * Calculate the maximum scoring sequence.
3    * The input is an array of integers.
4    * The output is the highest sum of consecutive
       elements in the array.
5    * Example: for array = [2, -1, 7, -5, 2] the
       output is 8 because 2 + (-1) + 7 = 8
6    * There is no other partial sequence wich sums up
       to a higher value.
7  */
8  int main(){
9
10   // The result is 8 because 5 + (-2) + 5 = 8
11   int a[] = {-2, 5, -2, 5};
12   // Will store the overall maxscore
13   int test = 0;
14
15   // 4 = length of array
16   // Loop through every possible consecutive
       sequence
17   for (int i = 1; i <= 4; i++) {
18      for (int j = i - 1; j < 4; j++) {
19         int s = 0;
20         // Calculate the sum in the given interval
21         for (int k = i; k <= j; k++) {
22            s = s + a[k - 1];
23         }
24         // Is the sum greater than the current
              maxscore?
25         if (s > test) {
26            test = s;
27         }
28      }
29   }
30
31   //POST-CONDITION check if the result is equal to
       8
```

```
32      if(test != 8) {
33          goto ERROR;
34      }
35
36  EXIT: return 0;
37  ERROR: return 1;
38  }
```

Program 12: Maximal Scoring Subsequence

Possible fix: change *line 21* to `for (int k = i; k <= j + 1; k++)`

B Survey Task II

```
1  int __VERIFIER_nondet_int();
2
3  int isPrime(int check){
4      if(check <= 1){
5          return 0;
6      }
7      // check/2 + 1 for fewer checks.
8      for(int i = 2; i <= check/2+1; i++){
9          if(check % i == 0){
10             return 0;
11         }
12     }
13     return 1;
14 }
15
16 /** Check if a number is a prime number */
17 int main() {
18
19     int input = __VERIFIER_nondet_int();
20     int check = input % 10;
21     int result = isPrime(check);
22
23     /* POST-CONDITION check if the program was able
           to identify all primes
24         from 0 to 10. The checks below are correct!
               */
25     if(       (result == 0 && check <= 0) || (result
           == 0 && check == 1)
26         || (result == 1 && check == 2) || (result ==
               1 && check == 3)
27         || (result == 0 && check == 4) || (result ==
               1 && check == 5)
28         || (result == 0 && check == 6) || (result ==
               1 && check == 7)
29         || (result == 0 && check == 8) || (result ==
               0 && check == 9)){
30         goto EXIT;
```

```
31      } else {
32          goto ERROR;
33      }
34
35      EXIT: return 0;
36      ERROR: return 1;
37  }
```

Program 13: Identifying Prime Numbers

Possible fix: change *line 8* to `for (int i = 2; i < check/2 + 1 ; i++)`

C Survey Task III

```
1  extern int __VERIFIER_nondet_int();
2
3  #define TRUE 0
4  #define FALSE 1
5
6  int isSorted(int a[], int len){
7      // check if the array is sorted
8      for(int i = 0; i < len-1; i++) {
9          if(a[i] > a[i+1]) {
10             return FALSE;
11         }
12     }
13     return TRUE;
14 }
15
16 /** Sort any 3-dimensional array ascending */
17 int main(){
18     // sort any array of size 3 in ascending order
19     // let the user input 3 numbers that should be
           sorted.
20     int first = __VERIFIER_nondet_int();
21     int second = __VERIFIER_nondet_int();
22     int third = __VERIFIER_nondet_int();
23     int a[] = {first,second,third};
24
25     // length of array
26     int len = 3;
27
28     // current position.
29     int i = 0;
30
31     while(!isSorted(a,len)) {
32         // swap entries if not sorted
33         int buff = a[i];
34         a[i] = a[i+1];
35         a[i+1] = buff;
36         i++;
```

```
37          if (i == len-1) {
38              i = 0;
39          }
40      }
41
42      //POST-CONDITION check if the array is sorted?
43      if (a[0] <= a[1] && a[1] <= a[2]) {
44          goto EXIT;
45      } else {
46          goto ERROR;
47      }
48
49
50  EXIT: return 0;
51  ERROR: return 1;
52
53  }
```

Program 14: Sorting

Possible fix: change *line 31* to `while(isSorted(a, len))` since `TRUE` and `FALSE` are initialized incorrectly in lines 3 and 4.

D Survey Task IV

```
1  extern int __VERIFIER_nondet_int();
2
3  int isPrime(int n){
4      for(int i = 2; i < n/2 + 1; i++){
5          if(n % i == 0) return 0;
6      }
7      return 1;
8  }
9
10 /**
11   Calculate all prime factors of a given number.
12   Example: prime factors of 420 are {2, 2, 3, 5, 7}
         because
13             2 * 2 * 3 * 5 * 7 = 420 and 2,3,5,7 are
                 prime.
14   */
15 int main(){
16
17     // Calculate prime factors of number;
18     int number = __VERIFIER_nondet_int();
19     int copyForCheck = number;
20     if (number <= 0) {
21         // Tell user that a positive number is
               required.
22         // This is not considered to be an error.
23         goto EXIT;
24     }
25
26     int test = 1;
27     for(int i = 2; i <= number; i++){
28         if (number % i == 0 && isPrime(i)) {
29             // Multiply all prime factors to test
30             test *= i;
31             // Reset i to restart computation with new
                   number
32             number = number / i;
33             i = 2;
```

```
34          }
35      }
36
37      // POST-CONDITION check if test equals number
38      // (test should equal the product of all found
            prime factors)
39      if(test != copyForCheck) {
40          goto ERROR;
41      }
42
43  EXIT: return 0;
44  ERROR: return 1;
45  }
```

Program 15: Prime Factors

Possible fix: change *line 33* to `i = 1`

E Survey Data

The data can be found on the following pages. On this page we describe the meaning of the column names. (**X** represents the number of the task.)

TaskX_Min: This column indicates if the participant fixed the bug with changing a minimal number of lines. Since most people were able to find the minimal fix, we did not include it in the evaluation. Possible values are 1 for true and 0 for false.

TaskX_Correct: Indicates if the participant was able to fix the bug. Possible values are 1 for true and 0 for false.

TaskX_Time: This column contains the needed time in seconds per participant.

TaskX_Help: If the value equals 1, the participant had to use fault localization. Otherwise, the participant had only access to the standard report.

Benefit_FL: We asked the participants to estimate how well they benefited from fault localization. Possible values are 1 to 10 where 10 is the highest score.

Clearness: The participants had to grade the clearness of the report with a value between 1 and 10 (best score).

Additional Information: The next 3 columns starting with "Info_" represent the data collected about the additional information compared to the standard report. Possible values lie between 0 for "not agree at all" and 4 for "totally agree".

Grading of the features: All columns starting with "Ben_" symbolize the felt benefit of all available features: *hints, potential fixes, justifications of the algorithm, score in the yellow cell, relevant lines* and *current values* (sorted by appearance).

ID	Task1_Min	Task1_Correct	Task1_Time	Task1_Help	Task2_Min	Task2_Correct	Task2_Time
1	1	0	259	0	1	0	537
2	1	1	310	0	0	0	322
3	1	1	870	0	1	1	720
4	1	1	430	0	1	1	149
5	1	0	1267	0	1	1	928
6	1	0	791	0	1	1	631
7	1	0	934	0	1	1	792
8	0	0	300	0	1	1	280
9	1	0	1010	0	1	1	385
10	1	0	500	0	1	1	360
11	1	1	550	1	1	1	330
12	1	1	524	1	1	0	365
13	1	1	1367	1	1	1	1048
14	1	1	490	1	1	1	295
15	1	1	512	1	1	1	336
16	1	1	551	1	1	1	389
17	1	0	780	1	1	1	600
18	1	1	550	1	1	1	490

Task2_Help	Task3_Min	Task3_Correct	Task3_Time	Task3_Help	Task4_Min	Task4_Correct
0	1	1	492	1	1	1
0	0	1	228	1	1	1
0	1	1	280	1	1	1
0	1	0	308	1	1	1
0	1	0	922	1	1	1
0	1	1	512	1	1	1
0	1	1	596	1	1	1
0	1	0	560	1	1	1
0	0	1	1031	1	0	0
0	1	1	550	1	1	1
1	0	1	870	0	1	1
1	1	0	458	0	1	0
1	1	0	772	0	1	1
1	1	1	932	0	1	1
1	0	1	1104	0	1	1
1	0	1	548	0	1	1
1	0	1	630	0	1	1
1	1	1	598	0	1	1

Task4_Time	Task4_Help	Benefit_FL	Clearness	Info_Misleading	Info_Accessible	Info_TooMuch
422	1	8	9	0	4	1
304	1	7	9	1	4	3
480	1	9	10	0	3	0
182	1	8	8	1	3	0
936	1	3	8	3	4	2
587	1	8	8	1	4	0
726	1	3	8	1	3	1
720	1	6	8	0	3	1
633	1	2	3	1	2	2
1000	1	9	7	1	3	1
1110	0	10	10	0	3	0
532	0	3	8	3	1	0
1224	0	3	8	3	2	0
1203	0	8	9	0	3	0
1054	0	8	10	1	4	1
1021	0	6	4	1	1	2
1050	0	8	6	2	3	2
378	0	6	9	1	3	0

Ben_Hints	Ben_Fix	Ben_Reason	Ben_Score	Ben_Lines	Ben_Values	Group
3	3	2	3	3	4	1
4	1	0	2	4	2	1
3	4	3	2	4	3	1
4	3	3	2	3	4	1
1	2	1	2	3	4	1
2	4	4	0	3	3	1
3	1	3	1	3	3	1
4	3	3	1	4	4	1
1	1	1	3	2	2	1
3	2	3	2	3	4	1
4	3	4	2	4	4	2
1	1	1	3	3	2	2
1	1	1	0	4	4	2
3	3	4	1	2	3	2
2	3	4	1	2	3	2
1	3	3	1	4	3	2
3	3	3	1	2	3	2
2	3	3	2	4	2	2

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich diese Bachelorarbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind, sowie dass ich die Bachelorarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

Matthias Kettl	Datum