CFaults: Model-Based Diagnosis for Fault Localization in C with Multiple Test Cases

Pedro Orvalho[1][0000−0002−7407−5967](✉), Mikoláš Janota[2][0000−0003−3487−784X], and Vasco Manquinho[1][0000−0002−4205−2189]

[1] INESC-ID, IST, Universidade de Lisboa, Portugal
{pmorvalho, vasco.manquinho}@tecnico.ulisboa.pt
[2] Czech Technical University in Prague, Czechia
mikolas.janota@cvut.cz

Abstract. Debugging is one of the most time-consuming and expensive tasks in software development. Several formula-based fault localization (FBFL) methods have been proposed, but they fail to guarantee a set of diagnoses across all failing tests or may produce redundant diagnoses that are not subset-minimal, particularly for programs with multiple faults. This paper introduces a novel fault localization approach for C programs with multiple faults. CFaults leverages Model-Based Diagnosis (MBD) with multiple observations and aggregates all failing test cases into a unified MaxSAT formula. Consequently, our method guarantees consistency across observations and simplifies the fault localization procedure. Experimental results on two benchmark sets of C programs, TCAS and C-Pack-IPAs, show that CFaults is faster than other FBFL approaches like BugAssist and SNIPER. Moreover, CFaults only generates subset-minimal diagnoses of faulty statements, whereas the other approaches tend to enumerate redundant diagnoses.

Keywords: Fault Localization · Model-Based Diagnosis · Formula-based Fault Localization · Debugging · Maximum Satisfiability.

1 Introduction

Localizing system faults has always been one of the most time-consuming and expensive tasks. Given a buggy program, *fault localization (FL)* involves identifying locations in the program that could cause a faulty behaviour (bug).

Given a faulty program and a test suite with failing test cases, current *formula-based fault localization (FBFL)* methods encode the localization problem into several optimization problems to identify a minimal set of faulty statements (diagnoses) within a program. Typically, these methods find a minimal diagnosis considering each failing test case individually rather than simultaneously with all failing test cases. Moreover, these FBFL methods enumerate all *Minimal Correction Subsets (MCSes)* [22] to cover all diagnoses.

For instance, BugAssist [17,18], a prominent FBFL tool, implements a ranking mechanism for bug locations. For each failing test, BugAssist enumerates all

Listing 1.1: Faulty program example. Faulty lines: {5,8,11}.

```
1  int main(){
2    // finds maximum of 3 numbers
3    int f,s,t;
4    scanf("%d%d%d",&f,&s,&t);
5    if (f < s && f >= t)
6      // fix: f >= s
7      printf("%d",f);
8    if (f > s && s <= t)
9      // fix: f < s and s >= t
10     printf("%d",s);
11   if (f > t && s > t)
12     // fix: f < t and s < t
13     printf("%d",t);
14
15   return 0;
16 }
```

	Input			Output
t_0	1	2	3	3
t_1	6	2	1	6
t_2	-1	3	1	3

Table 1: Test-suite.

	BugAssist	SNIPER
#Diagnoses t_0	8	8
#Diagnoses t_1	21	21
#Diagnoses t_2	9	9
#Total Unique Diagnoses	32	1297
Final Diagnosis	{4,13}	{5,8,11}

Table 2: Number of diagnoses (faulty statements) generated by BugAssist [17] and SNIPER [21] per test.

diagnoses of a Maximum Satisfiability (MaxSAT) formula corresponding to bug locations. Subsequently, BugAssist ranks diagnoses based on their frequency of appearance in each failing test. Other FBFL tools, like SNIPER [21], also enumerate all diagnoses for each failing test. However, the set of SNIPER's diagnoses is obtained by taking the Cartesian product of the diagnoses gathered using each failing test. As a result, while FBFL methods can determine minimal diagnoses per failing test, BugAssist cannot guarantee a minimal diagnosis considering all failing tests, and SNIPER may enumerate a significant number of redundant diagnoses that are not minimal [16]. These limitations may pose challenges for programs with multiple faulty statements, as shown in Example 1.

Example 1 (Motivation). Consider the program presented in Listing 1.1, which aims to determine the maximum among three given numbers. However, based on the test suite shown in Table 1, the program is faulty, as its output differs from the expected. The set of minimally faulty lines in this program is {5, 8, 11}, as all three `if`-conditions are incorrect according to the test suite. Fixing any subset of these lines would be insufficient to repair the program. One possible fix is to replace all these conditions with the suggested fixes in lines {6, 9, 12}.

In a typical FBFL approach, the minimal set of statements identified as faulty might include, for example, lines 4 and 5. Removing the `scanf` statement and an `if`-statement would allow an FBFL tool to assign any value to the input variables in order to always produce the expected output. However, considering an approach that prioritizes identifying faulty statements within the program's logic before evaluating issues in the input/output statements (such as `scanf` and `printf`), one might identify lines {5, 8, 11} as the faulty statements. When applying BugAssist's and SNIPER's approach on the program in Listing 1.1 with the described optimization criterion and utilizing the inputs/outputs detailed in Table 1 as specification, distinct sets of faults are identified for each failing test. Table 2 presents the diagnosis (set of faulty lines) produced by each tool, along with the number of diagnoses enumerated for each failing test case

and the total number of unique diagnoses after aggregating the diagnoses from all tests, using each tool's respective method.

In the case of BugAssist, diagnoses are prioritized based on their occurrence frequency. Consequently, BugAssist yields 32 unique diagnoses and selects {4, 13} since this diagnosis is identified in every failing test. In contrast, SNIPER computes the Cartesian product of all diagnoses, resulting in 1297 unique diagnoses. Note that BugAssist's diagnoses may not adequately identify all faulty program statements. Conversely, SNIPER's diagnosis {5, 8, 11} is minimal, even though it enumerates an additional 1296 diagnoses. Hence, existing FBFL methods do not ensure a minimal diagnosis across all failing tests (e.g., BugAssist) or may produce an overwhelming number of redundant sets of diagnoses (e.g., SNIPER), especially for programs with multiple faults.

This paper tackles this challenge by formulating the FL problem as a single optimization problem in Section 3. We leverage MaxSAT and the theory of *Model-Based Diagnosis (MBD)*, integrating all failing test cases simultaneously. This approach allows us to generate only minimal diagnoses to identify all faulty program components within a C program. Furthermore, we have implemented the MBD problem with multiple test cases in CFaults, a fault localization tool for ANSI-C programs, presented in Section 4. CFaults begins by unrolling and instrumentalizing C programs at the code-level, ensuring independence from the bounded model checker. Next, CFaults utilizes CBMC [5], a well-known bounded model checker for C, to generate a trace formula of the program. Finally, CFaults encodes the problem into MaxSAT to identify the minimal set of diagnoses corresponding to the buggy statements.

Experimental results presented in Section 5 on two benchmarks of C programs, TCAS [10] (industrial), and C-Pack-IPAs [30] (programming exercises), show that CFaults effectively detects minimal sets of diagnoses. In contrast, SNIPER and BugAssist either generate an overwhelming number of redundant diagnoses or fail to produce a minimal set required to fix each program.

To summarize, the contributions of this work are: (1) we tackle the fault localization problem in C programs using a Model-Based Diagnosis (MBD) approach considering multiple failing test cases, and formulating it as a unified optimization problem; (2) we implement this MBD approach in a publicly available tool called CFaults [31] [3] that unrolls and instrumentalizes C programs at the code level, making it independent of the bounded model checker used; (3) CFaults allows refinement of localized faults to pinpoint the bug's location more precisely; (4) we evaluate CFaults on two sets of C programs (TCAS and C-Pack-IPAs), showing that CFaults is fast and only produces subset-minimal diagnoses, unlike other state-of-the-art formula-based fault localization tools.

[3] `https://github.com/pmorvalho/CFaults`

2 Preliminaries

This section provides definitions and notations that are used throughout the paper. We start by presenting basic definitions of propositional logic and programs and then address standard *model-based diagnosis (MBD)* definitions.

The *Boolean Satisfiability (SAT)* problem is the decision problem for propositional logic [3]. A propositional formula in Conjunctive Normal Form (CNF) is a conjunction of clauses where each clause is a disjunction of literals. A literal is a propositional variable x_i or its negation $\neg x_i$. Given a CNF formula ϕ, the SAT problem corresponds to deciding if there is an assignment to the variables in ϕ such that ϕ is satisfied or prove that no such assignment exists. When applicable, set notation will be used for formulas and clauses. A formula can be represented as a set of clauses (meaning its conjunction) and a clause as a set of literals (meaning its disjunction).

The *Maximum Satisfiability (MaxSAT)* problem is an optimization version of the SAT problem. Given a CNF formula ϕ, the goal is to find an assignment that maximizes the number of satisfied clauses in ϕ. In partial MaxSAT, ϕ is split into hard clauses (ϕ_h) and soft clauses (ϕ_s). Given a formula $\phi = (\phi_h, \phi_s)$, the goal is to find an assignment that satisfies all hard clauses in ϕ_h while minimizing the number of unsatisfied soft clauses in ϕ_s. Moreover, in the weighted version of the partial MaxSAT problem, each soft clause is assigned a weight, and the goal is to find an assignment that satisfies all hard clauses and minimizes the sum of the weights of the unsatisfied soft clauses. Let $\phi = (\phi_h, \phi_s)$ be a partial MaxSAT formula. A Minimal Correction Subset (MCS) μ of ϕ is a subset $\mu \subseteq \phi_s$ where $\phi_h \cup (\phi_s \setminus \mu)$ is satisfiable and, for all $c \in \mu$, $\phi_h \cup (\phi_s \setminus \mu) \cup \{c\}$ is unsatisfiable. A dual concept of MCSes are *Minimal Unsatisfiable Subsets (MUSes)* [22,16].

Programs. A program is considered sequential, comprising standard statements such as assignments, conditionals, loops, and function calls, each adhering to their conventional semantics in C. A program is deemed to contain a bug when an assertion violation occurs during its execution with input I. Conversely, if no assertion violation occurs, the program is considered correct for input I. In cases where a bug is detected for input I, it is possible to define an error trace, representing the sequence of statements executed by program P on input I.

A Trace Formula (TF) is a propositional formula that is SAT iff there exists an execution of the program that terminates with a violation of an assert statement while satisfying all assume statements. For further information on TFs, interested readers are referred to [5,8].

Model-Based Diagnosis (MBD). The following definitions are commonly used in the *MBD* theory [34,16,24]. A system description $\mathcal{P}$ is composed of a set of components $\mathcal{C} = \{c_1, \ldots, c_n\}$. Each component in $\mathcal{C}$ can be declared healthy or unhealthy. For each component $c \in \mathcal{C}$, $h(c) = 0$ if c is unhealthy, otherwise, $h(c) = 1$. As in prior works [16,25], $\mathcal{P}$ is described by a CNF formula, where $\mathcal{F}_c$ denotes the encoding of component c:

$$\mathcal{P} \triangleq \bigwedge\nolimits_{c \in \mathcal{C}} (\neg h(c) \vee \mathcal{F}_c) \tag{1}$$

Observations represent deviations from the expected system behaviour. An observation, denoted as o, is a finite set of first-order sentences [34,16], which is assumed to be encodable in CNF as a set of unit clauses. In this work, the failing test cases represent the set of observations.

A system $\mathcal{P}$ is considered faulty if there exists an inconsistency with a given observation o when all components are declared healthy. The problem of model-based diagnosis (MBD) aims to identify a set of components which, if declared unhealthy, restore consistency. This problem is represented by the 3-tuple $\langle \mathcal{P}, \mathcal{C}, o \rangle$, and can be encoded as a CNF formula:

$$\mathcal{P} \wedge o \wedge \bigwedge\nolimits_{c \in \mathcal{C}} h(c) \vDash \bot \tag{2}$$

For a given MBD problem $\langle \mathcal{P}, \mathcal{C}, o \rangle$, a set of system components $\Delta \subseteq \mathcal{C}$ is a diagnosis iff:

$$\mathcal{P} \wedge o \wedge \bigwedge\nolimits_{c \in \mathcal{C} \setminus \Delta} h(c) \wedge \bigwedge\nolimits_{c \in \Delta} \neg h(c) \nvDash \bot \tag{3}$$

A diagnosis Δ is minimal iff no subset of Δ, $\Delta' \subsetneq \Delta$, is a diagnosis, and Δ is of minimal cardinality if there is no other diagnosis $\Delta'' \subseteq \mathcal{C}$ with $|\Delta''| < |\Delta|$.

A diagnosis is redundant if it is not subset-minimal [16].

To encode the Model-Based Diagnosis problem with one observation with partial MaxSAT, the set of clauses that encode $\mathcal{P}$ (1) represents the set of hard clauses. The soft clauses consists of unit clauses that aim to maximize the set of healthy components, i.e., $\bigwedge_{c \in \mathcal{C}} h(c)$ [36,24]. This MaxSAT encoding of MBD enables enumerating minimum cardinality diagnoses and subset minimal diagnoses, considering a single observation. Furthermore, a minimal diagnosis is a minimal correction subset (MCS) of the MaxSAT formula. Given an inconsistent formula that encodes the MDB problem (2), a minimal diagnosis Δ satisfies (3), thereby making Δ an MCS of the MaxSAT formula. BugAssist [18], SNIPER [21], and other model-based diagnosis (MBD) tools for fault localization in circuits [24,36,16] encode the localization problem with partial MaxSAT.

More recently, the MaxSAT encoding for MBD [16] has been generalized to multiple inconsistent observations. Let $\mathcal{O} = \{o_1, \ \ldots \ o_m\}$ be a set of observations. Each observation is associated with a replica $\mathcal{P}_i$ of the system $\mathcal{P}$. The system remains unchanged given different observations, where the components are replicated for each observation, but the healthy variables are shared. For a given observation o_i, a diagnosis is given by the following:

$$\mathcal{P}_i \wedge o_i \wedge \bigwedge\nolimits_{c \in \mathcal{C} \setminus \Delta} h(c) \wedge \bigwedge\nolimits_{c \in \Delta} \neg h(c) \nvDash \bot \tag{4}$$

The goal is to find a minimal diagnosis $\Delta \subseteq \mathcal{C}$, such that Δ is a minimal set of components when deactivated the system becomes consistent with all observations $\mathcal{O} = \{o_1, \ \ldots \ o_m\}$. Moreover, when considering multiple observations, an aggregated diagnosis is a subset of components that includes one possible diagnosis for each given observation.

3 Model-Based Diagnosis with Multiple Test Cases

This paper encodes the fault localization problem as a Model-Based Diagnosis with multiple observations using a single optimization problem. We simultaneously integrate all failing test cases (observations) in a single MaxSAT formula. This approach allows us to generate only minimal diagnoses capable of identifying all faulty components within the system, in our case, a C program.

Given m observations, $\mathcal{O} = \{o_1, \ldots, o_m\}$, a distinct replica of the system, denoted as $\mathcal{P}_i$, is required for each observation o_i. The hard clauses, ϕ_h, in our MaxSAT formulation correspond to each observation's encoding (o_i) and m system replicas, one for each observation, $\mathcal{P}_i$. Hence, $\phi_h = \bigwedge_{o_i \in \mathcal{O}} (\mathcal{P}_i \wedge o_i)$. Additionally, we aim to maximize the set of healthy components. Therefore, the soft clauses are formulated as: $\phi_s = \bigwedge_{c \in \mathcal{C}} h(c)$. Thus, given the MaxSAT solution of (ϕ_h, ϕ_s), its complement, i.e., the set of unhealthy components ($h(c) = 0$), corresponds to a subset-minimal aggregated diagnosis. This diagnosis is a subset-minimal of components that, when declared unhealthy (deactivated), make the system consistent with all observations, as follows:

$$\bigwedge\nolimits_{o_i \in \mathcal{O}} (\mathcal{P}_i \wedge o_i) \wedge \bigwedge\nolimits_{c \in \mathcal{C} \setminus \Delta} h(c) \wedge \bigwedge\nolimits_{c \in \Delta} \neg h(c) \not\models \bot \tag{5}$$

We assume that the system remains unchanged given different observations, where the components are replicated for each observation, but the healthy variables are shared. This is necessary because we analyze all observations jointly, which can affect the component's behaviour. In our work, the observations consist of a test suite containing failing test cases.

The HSD [16] algorithm was proposed to localize single faults in circuits given multiple observations. The HSD algorithm is based on hitting set dualization (HSD). For each observation o_i, this algorithm computes minimal unsatisfiable subsets (MUSes) of the MaxSAT formula encoded by (4). Next, the HSD algorithm computes a minimum hitting set $\mathcal{H}$ on the MUSes, and checks if $\mathcal{H}$ makes the system consistent with each observation individually. Hence, to compute all subset-minimal aggregated diagnoses of a faulty system $\mathcal{P}$, the algorithm performs at least m oracle calls for each minimum hitting set computed, where m is the number of observations. Each oracle call uses a different system replica (4).

Our approach encodes the problem into a single MaxSAT formula, while HSD [16] divides the problem into m MaxSAT formulas, one for each observation. Additionally, for each minimal hitting set computed in HSD, m oracle calls are needed to check if a diagnosis is consistent with all observations. However, in our case, we just need to perform a single MaxSAT call that returns a minimal diagnosis, which is, by definition, consistent with all observations since all observations are encoded into the formula. Furthermore, the HSD algorithm was solely evaluated using single faults in circuits given multiple observations, and it was not implemented to work with programs. A potential drawback is that our MaxSAT formula grows with the number of observations. This could result in a large formula and affect the performance of the MaxSAT solver. However, this scenario was not observed in our experimental results (see Section 5).

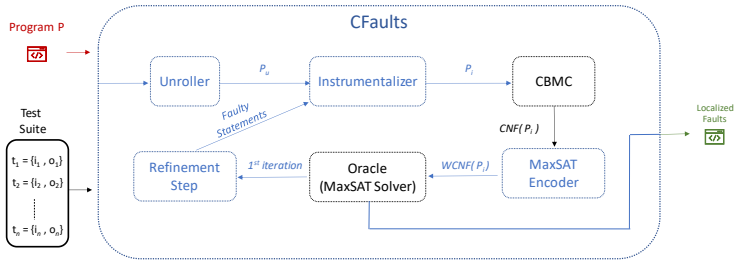


Fig. 1: Overview of CFaults.

4 CFaults: MBD with Multiple Observations for C

CFaults is a new model-based diagnosis (MBD) tool for fault localization in C programs with multiple test cases. Unlike previous works, CFaults uses the approach proposed in Section 3, and C programs are relaxed at the code level, enabling users to leverage other bounded model checkers effectively. Figure 1 provides an overview of CFaults consisting of six main steps: program unrolling, program instrumentalization, bounded model checking (CBMC), encoding to MaxSAT, an Oracle (MaxSAT solver), and a refinement step. Hence, CFaults formulates the MBD problem with multiple test cases as the 3-tuple $\langle \mathcal{P}, \mathcal{C}, \mathcal{O} \rangle$, where the observations $\mathcal{O}$ consist of failing test cases (inputs and assertions), the components $\mathcal{C}$ represent the set of program statements, and the system description $\mathcal{P}$ is a trace formula of the unrolled and instrumentalized program. The program is instrumented at the code level with relaxation variables corresponding to our *healthy variables*.

Program unrolling. CFaults starts the unrolling process by expanding the faulty program using the set of failed tests from the test suite. In this context, an unrolled program signifies the original program expanded m times (m program scopes), where m denotes the number of failed test cases. An unrolled program encodes the execution of all failing tests within the program, along with their corresponding inputs and specifications (assertions).

The unrolling process encompasses three primary steps. Initially, CFaults generates fresh variables and functions for each of the m program scopes, ensuring each scope possesses unique variables and functions. Subsequently, CFaults establishes variables representing the inputs and outputs for each program scope corresponding to the failing tests. Input operations, such as `scanf`, undergo translation into read accesses to arrays corresponding to the inputs, while output operations, such as `printf`, are replaced by write operations into arrays representing the program's output. Every exit point of the program (e.g., a `return` statement in the `main` function) is replaced with a `goto` statement directing the program flow to the next failing test's scope. Lastly, at the end of the unrolled program, CFaults embeds an assertion capturing all the specifications of the

Listing 1.2: The program from Listing 1.1 after being subjected to CFaults' unrolling process, using the test suite presented in Table 1. For simplicity, only the initial scope corresponding to test t_0 is displayed. The scopes `scope_1` and `scope_2` associated with failing tests t_1 and t_2 are omitted.

```
1  float _input_f0[3] = {1, 2, 3};
2  char _out_0[2] = "3";
3  int _ioff_f0 = 0, _ooff_0 = 0;
4  // ... inputs and outputs for the other tests
5  int main(){
6    scope_0:{
7      int f_0, s_0, t_0;
8      f_0 = _input_f0[_ioff_f0++];
9      s_0 = _input_f0[_ioff_f0++];
10     t_0 = _input_f0[_ioff_f0++];
11     if ((f_0 < s_0) && (f_0 >= t_0))
12         _ooff_0 = printInt(_out_0, _ooff_0, f_0);
13     if ((f_0 > s_0) && (s_0 <= t_0))
14         _ooff_0 = printInt(_out_0, _ooff_0, s_0);
15     if ((f_0 > t_0) && (s_0 > t_0))
16         _ooff_0 = printInt(_out_0, _ooff_0, t_0);
17     goto scope_1;
18   }
19   // ... scope_1 and scope_2
20   final_step:
21   assert(strcmp(_out_0, "3") != 0 || // other assertions);
22 }
```

failing tests. Consequently, the unrolled program encapsulates the execution of all failing tests within a single program.

Listing 1.2 exhibits a program segment generated through the unrolling process applied to Listing 1.1. CFaults establishes global variables to represent the inputs and outputs of each failing test (lines 1–3, Listing 1.2). For the sake of simplicity, the depicted listing illustrates solely the initial scope corresponding to test 0 from the test suite outlined in Table 1. Distinct variables are introduced for each failing test. Furthermore, the `scanf` function call is substituted with input array operations (lines 8–10), while the `printf` calls are replaced with CFaults' print functions, akin to `sprintf` functions, which direct output to a buffer. Lastly, the unrolled program concludes with an assertion representing the disjunction of the negation of all failing test assertions. For instance, suppose there are m failing tests, where A_i denotes the assertion of test t_i. In this scenario, CFaults injects the following assertion into the program: $\neg A_1 \vee \cdots \vee \neg A_m$.

Program Intrumentalization. After integrating all possible executions and assertions from failing tests during the unrolling step, CFaults proceeds to instrumentalize the unrolled C program by introducing relaxation variables for each program component (statement/instruction). Each relaxation variable activates (or deactivates) the program component being relaxed when assigned to true (or false) respectively. CFaults ensures that there are no conflicts between the names of the relaxation variables and the names of the program's original variables. For this step, CFaults needs to receive a maximum number of iterations that the program should be unwound.

Listing 1.3: Program statements.

```
1  int i;
2  int n;
3  int s;
4
5  s = 0;
6  n = _input_f0[_ioff_f0++];
7
8  if (n == 0)
9      return 0;
10
11 for (i=1; i < n; i++){
12     s = s + i;
13 }
```

Listing 1.4: Program statements relaxed.

```
1  //main scope
2  bool _rv1, _rv2, _rv3, _rv5;
3  bool _rv6[UNWIND],..., _rv8[UNWIND];
4  int _los; // loop1 offset
5
6  //test scope
7  bool _ev4;
8  int i,n,s;
9  _los=1;
10
11 if (_rv1) s = 0;
12 if (_rv2) n = _input_f0[_ioff_f0++];
13
14 if ( _rv3 ? (n == 0) : _ev4)
15     return 0;
16
17 for (_rv5 ? (i = 1) : 1;
18      !_rv6[_los] || (i<n);
19      _rv8[_los] ? i++ : 1, _los++){
20     if (_rv7[_los]) s = s + i;
21 }
```

The relaxation process introduces relaxation variables that deactivate or activate program components. This process involves four distinct relaxation rules for: (1) conditions of `if`-statements, (2) expression lists (e.g., an expression list executed at the beginning of a for-loop), (3) loop conditions, and (4) other program statements.

Example 2. Listings 1.3 shows a code snippet that sums all the numbers between 1 and `n`. Listings 1.4 depicts the same program statements after undergoing relaxation by CFaults. For the sake of simplicity, all relaxation variables' and offsets' names were simplified.

In more detail, the rule for relaxing a general program statement is to envelop the statement with an `if`-statement, whose condition is a relaxation variable. For example, consider lines 5 and 6 in the program on Listings 1.3. These lines are relaxed by CFaults using relaxation variables `_rv1` and `_rv2` respectively, appearing as lines 11 and 12 on Listings 1.4.

Furthermore, when relaxing `if`-statements, the statements inside the `then` and `else` blocks adhere to the previously explained relaxation rule. However, the conditions of `if`-statements are relaxed using a ternary operator, as shown in line 14 of Listings 1.4. Note that if the relaxation variable is assigned true, then the original `if` condition is executed. Otherwise, a different relaxation variable (e.g., `_ev4` in Listings 1.4) determines whether the program execution enters the `then`-block or the `else`-block (if one exists). These relaxation variables (`else`*'s relaxation variables*) are local to each failing test scope and enable different tests to determine whether to enter the `then` or `else`-block.

When handling expression lists, CFaults adopts a comparable strategy to that of generic program statements, enclosing each expression within a ternary operator instead of an `if`-statement. If the program component is deactivated,

the expression is replaced by `1`. For example, the initialization of variable `i` in line 11 of Listings 1.3 is relaxed into the ternary operation in line 17 of Listings 1.4.

Lastly, all relaxation variables inside a loop are Boolean vectors to relax statements within a loop. Each entry of these vectors relaxes the loop's statements for a given iteration. The maximum number of iterations of the loops is defined by the CFaults user. CFaults follows a similar approach for inner loops, creating arrays of arrays. Thus, for simple program statements within a loop, CFaults encapsulates them with `if`-statements, with the relaxation variables indexed to the iteration number. Line 20 of Listings 1.4 illustrates a relaxed statement inside a loop. The loop's condition is relaxed by implication of the relaxation variable, as demonstrated in line 18 of Listings 1.4. Furthermore, each loop has its own offsets to index relaxation variables. These offsets are initialized just before the loop and incremented at the end of each iteration (e.g., line 19 in Listing 1.4).

When handling auxiliary functions, CFaults declares the relaxation variables needed in the main scope of the program and passes these variables as parameters. Hence, CFaults ensures that the same variables are used throughout the auxiliary functions' calls.

Listing 1.5 depicts the program resulting from the instrumentalization process of Listing 1.2 performed by CFaults. The same program components (statements/instructions) across different failing test scopes are assigned the same relaxation variable declared in the main scope. Consequently, if a relaxation variable is set to 0, the corresponding program component is deactivated across all test executions. Additionally, the relaxation variables are left uninitialized, allowing CFaults to determine the minimal number of faulty components requiring deactivation. Note that relaxation variables are not declared as global variables but as local variables within the `main` scope. This is to prevent the C compiler from automatically initializing all these variables to 0.

CBMC. After unrolling and instrumentalizing the C program, CFaults invokes CBMC, a bounded model checker for C [5]. CBMC initially transforms the unrolled and relaxed program into *Static Single Assignment (SSA)* form, an intermediate representation ensuring that variables are assigned values only once and are defined before use [9]. SSA achieves this by converting existing variables into multiple versions, each uniquely representing an assignment. Next, CBMC translates the SSA representation into a CNF formula, which represents the trace formula of the program. During the CNF formula generation, CBMC negates the program's assertion $(\neg(\neg A_1 \vee \cdots \vee \neg A_m))$ to compute a counter-example. Moreover, the CNF formula, ϕ, encodes each failing test's input (I_i), assertion (A_i), and all execution paths of the unrolled and relaxed incorrect program encoded by the trace formula (P), i.e., $\phi = (I_1 \wedge \ldots \wedge I_m) \wedge P \wedge (A_1 \wedge \cdots \wedge A_m)$. Thus, if ϕ is SAT, an assignment exists that activates or deactivates each relaxation variable and makes all failing test assertions true. Hence, each satisfiable assignment is a diagnosis of the C program, considering all failing tests.

Listing 1.5: Instrumentalized program.

```
1  //global vars
2  int main(){
3    bool _rv1, _rv2, ..., _rv12;
4    scope_0:{
5      bool _ev5, _ev8, _ev11;
6      int f_0, s_0, t_0;
7      if (_rv1) f_0 = _input_f0[_ioff_f0++];
8      if (_rv2) s_0 = _input_f0[_ioff_f0++];
9      if (_rv3) t_0 = _input_f0[_ioff_f0++];
10     if (_rv4 ? ((f_0 < s_0) && (f_0 >= t_0)) : _ev5 ){
11         if (_rv6) _ooff_0 = printInt(_out_0, _ooff_0, f_0);
12     }
13     if (_rv7 ? ((f_0 > s_0) && (s_0 <= t_0)) : _ev8 ){
14         if (_rv9) _ooff_0 = printInt(_out_0, _ooff_0, s_0);
15     }
16     if (_rv10? ((f_0 > t_0) && (s_0 > t_0)) : _ev11 ){
17         if (_rv12) _ooff_0 = printInt(_out_0, _ooff_0, t_0);
18     }
19     goto scope_1;
20   }
21   // scope_1 and scope_2
22   final_step:
23   assert(strcmp(_out_0, "3") != 0 || ... // other assertions);
24 }
```

MaxSAT Encoder. Let ϕ denote the CNF formula generated by CBMC in the previous step. Next, CFaults generates a weighted partial MaxSAT formula $(\mathcal{H}, \mathcal{S})$ to maximize the satisfaction of relaxation variables in the program, aiming to minimize the necessary code alterations. The set of hard clauses is defined by CBMC's CNF formula (i.e., $\mathcal{H} = \phi$), while the soft clauses consist of unit clauses representing relaxation variables used to instrument the C program, expressed as $\mathcal{S} = \bigwedge_{c \in \mathcal{C}} (rv_c)$. Additionally, we assign a hierarchical weight to each relaxation variable based on the height of its sub-AST (Abstract Syntax Tree). For instance, in the case of an `if`-statement without an `else`-block, the relaxation variable for its condition will be assigned a weight equal to the sum of the weights of the relaxation variables within the `then`-block. Furthermore, to prioritize the identification of faulty statements within the program's logic over evaluating issues in the input/output, these statements (such as `scanf` and `printf`) are assigned a significantly higher cost compared to other program statements. Moreover, due to the use of hierarchical weights in the relaxation variables, CFaults enumerates all MaxSAT solutions to identify all subset-minimal diagnoses since there can be more than one MaxSAT solution (with the same cost) that differ in the number of relaxed program statements.

Oracle. CFaults invokes a MaxSAT solver to determine the program's minimal set of faulty statements, aligning with the principles of Model-Based Diagnosis (MBD) theory. By consolidating all failing tests into a unified, unrolled, and instrumentalized program, the MaxSAT solution identifies the minimum subset of statements requiring removal to fulfil the assertions of all failing tests.

Refinement. The standard Model-Based Diagnosis (MBD) theory focuses on faulty components (program statements) whose removal can rectify the system (program's assertions). However, addressing program faults in software may necessitate introducing, relocating, or replacing statements. Hence, CFaults incorporates a refinement step that introduces nondeterminism into the program, enabling the Oracle to simulate actions such as introducing, reallocating or replacing existing program statements. During the first iteration of CFaults, the refinement step is invoked to introduce non-determinism, with the aim of minimizing the number of faulty statements. This step can improve fault localization by conducting a more detailed analysis of previously identified faulty statements. For example, in the scenario outlined in Example 1, refining line 5 into

```
if ((_rv1? (f < s) : nondet_bool()) && (_rv2? (f >= t) : nondet_bool()))
```

enables CFaults to determine that only the left part of the binary operation (`f < s`) is faulty, while the right part remains unaffected. This fine-grained approach allows for more precise detection of program faults. When the refinement step is triggered, CFaults instrumentalizes the program again, introducing nondeterminism exclusively to the statements previously identified as faulty during the initial Oracle call. Through this process, CFaults aims to reduce the set of faulty program components by executing them or assigning them to nondeterministic functions. All remaining program components are executed, meaning their relaxation variables are activated during this step.

5 Experimental Results

All of the experiments were conducted on an Intel(R) Xeon(R) Silver computer with 4210R CPUs @ 2.40GHz running Linux Debian 10.2, using a memory limit of 32 GB and a timeout of 3600s, for each program. CFaults has been evaluated using two distinct benchmarks of C programs: TCAS [10] and C-Pack-IPAs [27]. TCAS stands out as a well-known program benchmark extensively utilized in the fault localization literature [18,21]. This benchmark comprises a C program from Siemens and 41 versions with intentionally introduced faults, with known positions and types of these faults. Conversely, C-Pack-IPAs is a set of student programs collected during an introductory programming course. For this evaluation, we used the first lab class of C-Pack-IPAs, which consists of ten programming assignments, comprising 486 faulty programs and 799 correct implementations. C-Pack-IPAs has proven successful in evaluating various works across program analysis [32], program transformation [29], and clustering [28].

CFaults uses `pycparser` [33] for unrolling and instrumentalizing C programs. Additionally, CBMC version 5.11 is used to encode C programs into CNF formulas. Furthermore, since the source code of BugAssist and SNIPER is either unavailable or no longer maintained (resulting in compilation and linking issues), prototypes of their algorithms were implemented. It is worth noting that the original version of SNIPER could only analyze programs that utilized a subset of ANSI-C, lacked support for loops and recursion, and could only partially handle global variables, arrays, and pointers. In this work, both SNIPER

Benchmark: **TCAS**			
	Valid Diagnosis	**Memouts**	**Timeouts**
BugAssist	41 (100.0%)	0 (0.0%)	0 (0.0%)
SNIPER	7 (17.07%)	34 (82.93%)	0 (0.0%)
CFaults	41 (100.0%)	0 (0.0%)	0 (0.0%)
CFaults-Refined	41 (100.0%)	0 (0.0%)	0 (0.0%)

Benchmark: **C-Pack-IPAs**			
	Valid Diagnosis	**Memouts**	**Timeouts**
BugAssist	454 (93.42%)	0 (0.0%)	32 (6.58%)
SNIPER	446 (91.77%)	4 (0.82%)	36 (7.41%)
CFaults	483 (99.38%)	1 (0.21%)	2 (0.41%)
CFaults-Refined	482 (99.18%)	1 (0.21%)	3 (0.62%)

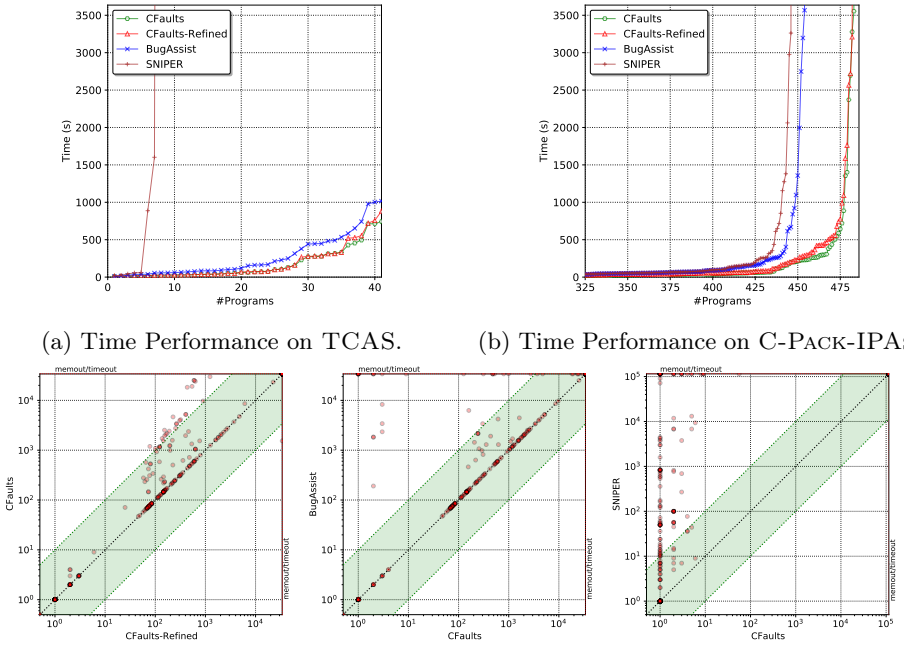
Table 3: BugAssist, SNIPER and CFaults fault localization results.

and BugAssist handle ANSI-C programs, as their algorithms are built on top of CFaults's unroller and instrumentalizer modules. For the MaxSAT oracle, RC2Stratified [15] from the `PySAT` toolkit [14] (v. 0.1.7.dev19) was used.

Furthermore, all three FBFL algorithms evaluated (CFaults, BugAssist, and SNIPER) consistently generate diagnoses that are consistent with (5), indicating that all proposed diagnoses undergo validation by CBMC once the algorithm provides a diagnosis. However, this validation primarily serves to verify diagnoses generated by BugAssist, as it has the capability to produce diagnoses that may not align with all failing test cases. In contrast, CFaults' MaxSAT solution, by definition, aligns with all observations, and SNIPER's aggregation method (Cartesian product) produces only valid diagnoses, although they may not always be subset-minimal. When considering BugAssist, we iterate through all computed diagnoses based on BugAssist's voting score, until we identify one diagnosis that is consistent with all observations, i.e., conforms to (5).

Table 3 provides an overview of the results obtained using SNIPER, BugAssist, and CFaults on the two benchmarks of C programs. The TCAS program comprises approximately 180 lines of code and has a maximum of 131 failing tests for each program. This leads SNIPER to reach the memory limit of 32GB for almost 83% of the programs when aggregating the sets of MCSes computed for each failing test. Additionally, a higher rate of timeouts is observed for SNIPER and BugAssist than for CFaults. Figures 2a and 2b depict cactus plots that present the CPU time spent on fault localization in each program (y-axis) versus the number of programs with all faults successfully localized (x-axis) using BugAssist, SNIPER, and CFaults (with and without refinement) on TCAS and C-Pack-IPAs, respectively. Notably, CFaults generally exhibits faster performance compared to BugAssist and SNIPER across both benchmarks. In Figure 2a, SNIPER's performance is due to its memout rate on TCAS.

In TCAS, CFaults, whether invoking the refinement step or not, identifies faults in the entire dataset. However, in C-Pack-IPAs, CFaults localizes faults in one additional program when the refinement step is not called. Even if the refinement step reaches the time limit, CFaults still possesses a subset-minimal diagnosis from the preceding step that has not undergone refinement. The refinement step slightly slows down CFaults, as shown in Figures 2a and 2b. Nonetheless, Figure 2c illustrates a scatter plot comparing the optimum costs (MaxSAT solution's cost) achieved by CFaults with and without calling the refinement step on C-Pack-IPAs. Each point on this plot represents a faulty program, where the x-value (resp. y-value) represents the optimum cost of CFaults' with refinement (resp. without refinement) diagnosis. If a point



(a) Time Performance on TCAS.

(b) Time Performance on C-Pack-IPAs.

(c) Costs of refined diagnoses on C-Pack-IPAs.

(d) Costs of diagnoses on C-Pack-IPAs.

(e) #Diagnoses generated on C-Pack-IPAs.

Fig. 2: Comparison between BugAssist's, SNIPER's and CFaults' diagnoses.

lies above the diagonal, it indicates that a non-refined diagnosis has a higher cost than a refined diagnosis for the same program. Therefore, while the refinement step may marginally slow down CFaults, it enables CFaults to identify smaller diagnoses at a reduced cost in approximately 16% of C-Pack-IPAs's programs. Moreover, this observation was not noted in the TCAS dataset, as each program contains a maximum of two faults, and the refinement step did not yield improved outcomes in this particular dataset.

Additionally, Figure 2d illustrates a scatter plot comparing the diagnoses' costs achieved by CFaults (x-axis) against BugAssist (y-axis) on C-Pack-IPAs. BugAssist fails to provide an optimal diagnosis in almost 6% of cases. In the TCAS benchmark, although BugAssist manages to localize faults in all programs, it yields a non-optimal diagnosis in 10% of the programs. Furthermore, Figure 2e depicts a scatter plot comparing the number of diagnoses generated by CFaults (x-axis) against SNIPER (y-axis). While CFaults needs to enumerate all MaxSAT solutions due to the weighted MaxSAT formula, it is evident that SNIPER generates significantly more diagnoses than CFaults. This discrepancy suggests that SNIPER overlooks the possibility of redundant diagnoses being computed. The number of such redundant diagnoses is much larger than the subset-minimal diagnoses generated by CFaults. Figure 2e il-

lustrates that in some instances, SNIPER may enumerate up to 100K diagnoses, whereas CFaults generates less than 10.

As a validation step for our implementation, we analyzed all three fault localization methods on the collection of 799 correct programs in C-Pack-IPAs. This was done to ensure that all methods yielded zero faults for all correct implementations of each programming exercise. Moreover, we conducted a comparison between CFaults and the HSD algorithm [16] (see Section 3) on the ISCAS85 dataset [13], which is a widely studied collection of single-fault circuits. It is worth noting that HSD's implementation currently only supports fault localization in circuits. We encountered no performance issues during this comparison, and both approaches successfully localized all faults within each circuit.

6 Related Work

Fault localization (FL) techniques typically fall into two main families: *spectrum-based (SBFL)* and *formula-based (FBFL)*. SBFL methods [1,38,26,39,40,2] estimate the likelihood of a statement being faulty based on test coverage information from both passing and failing test executions. While SBFL techniques are generally fast, they may lack precision, as not all identified statements are likely to be the cause of failures [23,35]. In contrast, FBFL approaches [17,18,21,11,20,12,41,42,19] are considered exact. FBFL methods encode the fault localization problem into several optimization problems aimed at identifying the minimum number of faulty statements within a program. Typically, these methods perform a MaxSAT call for each failing test, allowing them to individually identify a minimal set of faults for each failing test case rather than simultaneously addressing all failing test cases. *Program slicing* [37,35,43] has also emerged as a technique for localizing faults within programs. A more syntactic FBFL approach [35] is to use program slicing to enumerate all minimal sets of repairs for a given faulty program. Another method for identifying the causes of faulty program behaviour involves analyzing the variances between various versions of the software [43]. *Refinement* has a long-standing tradition in verification; particularly for refining abstractions of reachable states [7,6,4]. In that sense, our form of refinement is different because it enables us to more precisely pinpoint faults of the user, at the sub-expression level.

7 Conclusion

This paper introduces a novel formula-based fault localization technique for C programs capable of addressing any number of faults. Leveraging Model-Based Diagnosis (MBD) with multiple observations, CFaults consolidates all failing test cases into a unified MaxSAT formula, ensuring consistency in the fault localization process. Experimental evaluations on TCAS and C-Pack-IPAs, show that CFaults is faster than other FBFL approaches like BugAssist and SNIPER. Furthermore, CFaults only generates minimal diagnoses of faulty statements, while other methods tend to produce redundant diagnoses.

Acknowledgements

This work was partially supported by Portuguese national funds through FCT, under projects UIDB/50021/2020 (DOI: 10.54499/UIDB/50021/2020), PTDC/-CCI-COM/2156/2021 (DOI: 10.54499/PTDC/CCI-COM/2156/2021) and 2022.-03537.PTDC (DOI: 10.54499/2022.03537.PTDC) and grant SFRH/BD/07724/-2020 (DOI: 10.54499/2020.07724.BD). PO acknowledges travel support from the European Union's Horizon 2020 research and innovation programme under ELISE Grant Agreement No 951847. This work was also supported by the MEYS within the program ERC CZ under the project POSTMAN no. LL1902 and co-funded by the European Union under the project *ROBOPROX* (reg. no. CZ-.02.01.01/00/22_008/0004590). This article is part of the RICAIP project that has received funding from the EU's Horizon 2020 research and innovation program under grant agreement No 857306.

References

1. Abreu, R., Zoeteweij, P., van Gemund, A.J.C.: Spectrum-based multiple fault localization. In: ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering, Auckland, New Zealand, November 16-20, 2009. pp. 88–99. IEEE Computer Society (2009). https://doi.org/10.1109/ASE.2009.25, https://doi.org/10.1109/ASE.2009.25
2. Abreu, R., Zoeteweij, P., Golsteijn, R., van Gemund, A.J.C.: A practical evaluation of spectrum-based fault localization. J. Syst. Softw. **82**(11), 1780–1792 (2009). https://doi.org/10.1016/J.JSS.2009.06.035, https://doi.org/10.1016/j.jss.2009.06.035
3. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press (2009)
4. Clarke, E.M., Grumberg, O., Kroening, D., Peled, D.A., Veith, H.: Model checking, 2nd Edition. MIT Press (2018), https://mitpress.mit.edu/books/model-checking-second-edition
5. Clarke, E.M., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings. Lecture Notes in Computer Science, vol. 2988, pp. 168–176. Springer (2004). https://doi.org/10.1007/978-3-540-24730-2_15, https://doi.org/10.1007/978-3-540-24730-2_15
6. Clarke, E.M., Kroening, D., Sharygina, N., Yorav, K.: Predicate abstraction of ANSI-C programs using SAT. Formal Methods Syst. Des. **25**(2-3), 105–127 (2004). https://doi.org/10.1023/B:FORM.0000040025.89719.F3, https://doi.org/10.1023/B:FORM.0000040025.89719.f3
7. Clarke, E.M., Kroening, D., Sharygina, N., Yorav, K.: SATABS: sat-based predicate abstraction for ANSI-C. In: Halbwachs, N., Zuck, L.D. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3440, pp. 570–574. Springer (2005).

https://doi.org/10.1007/978-3-540-31980-1_40, https://doi.org/10.1007/978-3-540-31980-1_40
8. Clarke, E.M., Kroening, D., Yorav, K.: Behavioral consistency of C and verilog programs using bounded model checking. In: Proceedings of the 40th Design Automation Conference, DAC 2003, Anaheim, CA, USA, June 2-6, 2003. pp. 368–371. ACM (2003). https://doi.org/10.1145/775832.775928, https://doi.org/10.1145/775832.775928
9. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst. **13**(4), 451–490 (1991). https://doi.org/10.1145/115372.115320, https://doi.org/10.1145/115372.115320
10. Do, H., Elbaum, S.G., Rothermel, G.: Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. Empir. Softw. Eng. **10**(4), 405–435 (2005). https://doi.org/10.1007/S10664-005-3861-2
11. Feser, J.K., Chaudhuri, S., Dillig, I.: Synthesizing data structure transformations from input-output examples. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015. pp. 229–239 (2015)
12. Griesmayer, A., Staber, S., Bloem, R.: Automated fault localization for C programs. In: Bloem, R., Roveri, M., Somenzi, F. (eds.) Proceedings of the Workshop on Verification and Debugging, V&D@FLoC 2006, Seattle, WA, USA, August 21, 2006. Electronic Notes in Theoretical Computer Science, vol. 174, pp. 95–111. Elsevier (2006). https://doi.org/10.1016/J.ENTCS.2006.12.032, https://doi.org/10.1016/j.entcs.2006.12.032
13. Hansen, M.C., Yalcin, H., Hayes, J.P.: Unveiling the ISCAS-85 benchmarks: A case study in reverse engineering. IEEE Des. Test Comput. **16**(3), 72–80 (1999). https://doi.org/10.1109/54.785838, https://doi.org/10.1109/54.785838
14. Ignatiev, A., Morgado, A., Marques-Silva, J.: PySAT: A python toolkit for prototyping with SAT oracles. In: Beyersdorff, O., Wintersteiger, C.M. (eds.) Theory and Applications of Satisfiability Testing - SAT 2018 - 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10929, pp. 428–437. Springer (2018). https://doi.org/10.1007/978-3-319-94144-8_26, https://doi.org/10.1007/978-3-319-94144-8_26
15. Ignatiev, A., Morgado, A., Marques-Silva, J.: RC2: an efficient MaxSAT solver. J. Satisf. Boolean Model. Comput. **11**(1), 53–64 (2019)
16. Ignatiev, A., Morgado, A., Weissenbacher, G., Marques-Silva, J.: Model-based diagnosis with multiple observations. In: Kraus, S. (ed.) Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019. pp. 1108–1115. ijcai.org (2019). https://doi.org/10.24963/IJCAI.2019/155, https://doi.org/10.24963/ijcai.2019/155
17. Jose, M., Majumdar, R.: Bug-assist: Assisting fault localization in ANSI-C programs. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6806, pp. 504–509. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_40, https://doi.org/10.1007/978-3-642-22110-1_40
18. Jose, M., Majumdar, R.: Cause clue clauses: error localization using maximum satisfiability. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011. pp. 437–446. ACM (2011)

19. Könighofer, R., Bloem, R.: Automated error localization and correction for imperative programs. In: Bjesse, P., Slobodová, A. (eds.) International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011. pp. 91–100. FMCAD Inc. (2011), http://dl.acm.org/citation.cfm?id=2157671
20. Lamraoui, S., Nakajima, S.: A formula-based approach for automatic fault localization of imperative programs. In: Merz, S., Pang, J. (eds.) Formal Methods and Software Engineering - 16th International Conference on Formal Engineering Methods, ICFEM 2014, Luxembourg, Luxembourg, November 3-5, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8829, pp. 251–266. Springer (2014). https://doi.org/10.1007/978-3-319-11737-9_17, https://doi.org/10.1007/978-3-319-11737-9_17
21. Lamraoui, S., Nakajima, S.: A formula-based approach for automatic fault localization of multi-fault programs. J. Inf. Process. **24**(1), 88–98 (2016). https://doi.org/10.2197/IPSJJIP.24.88, https://doi.org/10.2197/ipsjjip.24.88
22. Liffiton, M.H., Sakallah, K.A.: Algorithms for computing minimal unsatisfiable subsets of constraints. J. Autom. Reason. **40**(1), 1–33 (2008). https://doi.org/10.1007/S10817-007-9084-Z, https://doi.org/10.1007/s10817-007-9084-z
23. Liu, K., Koyuncu, A., Bissyandé, T.F., Kim, D., Klein, J., Le Traon, Y.: You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems. In: 2019 12th IEEE conference on software testing, validation and verification (ICST). pp. 102–113. IEEE (2019)
24. Marques-Silva, J., Janota, M., Ignatiev, A., Morgado, A.: Efficient model based diagnosis with maximum satisfiability. In: Yang, Q., Wooldridge, M.J. (eds.) Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015. pp. 1966–1972. AAAI Press (2015), http://ijcai.org/Abstract/15/279
25. Metodi, A., Stern, R., Kalech, M., Codish, M.: A novel sat-based approach to model based diagnosis. J. Artif. Intell. Res. **51**, 377–411 (2014). https://doi.org/10.1613/JAIR.4503, https://doi.org/10.1613/jair.4503
26. Naish, L., Lee, H.J., Ramamohanarao, K.: A model for spectra-based software diagnosis. ACM Trans. Softw. Eng. Methodol. **20**(3), 11:1–11:32 (2011). https://doi.org/10.1145/2000791.2000795, https://doi.org/10.1145/2000791.2000795
27. Orvalho, P., Janota, M., Manquinho, V.: C-Pack of IPAs: A C90 Program Benchmark of Introductory Programming Assignments. CoRR **abs/2206.08768** (2022). https://doi.org/10.48550/arXiv.2206.08768, https://doi.org/10.48550/arXiv.2206.08768
28. Orvalho, P., Janota, M., Manquinho, V.: InvAASTCluster: On Applying Invariant-Based Program Clustering to Introductory Programming Assignments. CoRR **abs/2206.14175** (2022). https://doi.org/10.48550/ARXIV.2206.14175, https://doi.org/10.48550/arXiv.2206.14175
29. Orvalho, P., Janota, M., Manquinho, V.: MultIPAs: Applying Program Transformations to Introductory Programming Assignments for Data Augmentation. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022. pp. 1657–1661. ACM, Singapore (2022). https://doi.org/10.1145/3540250.3558931
30. Orvalho, P., Janota, M., Manquinho, V.: C-Pack of IPAs: A C90 Program Benchmark of Introductory Programming Assignments. In: International Workshop on Automated Program Repair, APR@ICSE 2024, Lisbon, Portugal, April 20, 2024. pp. – (2024). https://doi.org/10.1145/3643788.3648010, https://doi.org/10.1145/3643788.3648010

31. Orvalho, P., Janota, M., Manquinho, V.: CFaults: Model-Based Diagnosis for Fault Localization in C with Multiple Test Cases (Jun 2024). https://doi.org/10.5281/zenodo.12510220, https://github.com/pmorvalho/CFaults
32. Orvalho, P., Piepenbrock, J., Janota, M., Manquinho, V.M.: Graph neural networks for mapping variables between programs. In: ECAI 2023 - 26th European Conference on Artificial Intelligence. Frontiers in Artificial Intelligence and Applications, vol. 372, pp. 1811–1818. IOS Press, Poland (2023). https://doi.org/10.3233/FAIA230468, https://doi.org/10.3233/FAIA230468
33. pycparser: . https://github.com/eliben/pycparser (2024), [Online; accessed 18-April-2024]
34. Reiter, R.: A theory of diagnosis from first principles. Artif. Intell. **32**(1), 57–95 (1987). https://doi.org/10.1016/0004-3702(87)90062-2, https://doi.org/10.1016/0004-3702(87)90062-2
35. Rothenberg, B., Grumberg, O.: Must fault localization for program repair. In: Lahiri, S.K., Wang, C. (eds.) Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12225, pp. 658–680. Springer (2020). https://doi.org/10.1007/978-3-030-53291-8_33, https://doi.org/10.1007/978-3-030-53291-8_33
36. Safarpour, S., Mangassarian, H., Veneris, A.G., Liffiton, M.H., Sakallah, K.A.: Improved design debugging using maximum satisfiability. In: Formal Methods in Computer-Aided Design, 7th International Conference, FMCAD 2007, Austin, Texas, USA, November 11-14, 2007, Proceedings. pp. 13–19. IEEE Computer Society (2007). https://doi.org/10.1109/FAMCAD.2007.26, https://doi.org/10.1109/FAMCAD.2007.26
37. Soremekun, E.O., Kirschner, L., Böhme, M., Zeller, A.: Locating faults with program slicing: an empirical analysis. Empir. Softw. Eng. **26**(3), 51 (2021). https://doi.org/10.1007/S10664-020-09931-7, https://doi.org/10.1007/s10664-020-09931-7
38. Wong, W.E., Debroy, V., Choi, B.: A family of code coverage-based heuristics for effective fault localization. J. Syst. Softw. **83**(2), 188–208 (2010). https://doi.org/10.1016/J.JSS.2009.09.037, https://doi.org/10.1016/j.jss.2009.09.037
39. Wong, W.E., Debroy, V., Gao, R., Li, Y.: The dstar method for effective software fault localization. IEEE Trans. Reliab. **63**(1), 290–308 (2014). https://doi.org/10.1109/TR.2013.2285319, https://doi.org/10.1109/TR.2013.2285319
40. Wong, W.E., Gao, R., Li, Y., Abreu, R., Wotawa, F.: A survey on software fault localization. IEEE Trans. Software Eng. **42**(8), 707–740 (2016). https://doi.org/10.1109/TSE.2016.2521368, https://doi.org/10.1109/TSE.2016.2521368
41. Wotawa, F., Nica, M., Moraru, I.: Automated debugging based on a constraint model of the program and a test case. J. Log. Algebraic Methods Program. **81**(4), 390–407 (2012). https://doi.org/10.1016/J.JLAP.2012.03.002, https://doi.org/10.1016/j.jlap.2012.03.002
42. Xie, Y., Aiken, A.: Scalable error detection using boolean satisfiability. In: Palsberg, J., Abadi, M. (eds.) Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005. pp. 351–363. ACM (2005). https://doi.org/10.1145/1040305.1040334, https://doi.org/10.1145/1040305.1040334
43. Zeller, A.: Yesterday, my program worked. today, it does not. why? In: ESEC/FSE'99, 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering 1999. Lecture Notes in Computer Science, vol. 1687, pp. 253–267. Springer (1999)