Locating Loop Errors in Programs: A Scalable and Expressive Approach using LocFaults

Mohammed Bekkouche[1,*]

[1]LabRI-SBA Laboratory, École Supérieure en Informatique, Sidi Bel Abbes 22000, Algeria

Abstract

A model checker can generate a lengthy and complicated trace of counterexamples for an erroneous program, with the loop instructions being the largest part of this trace. Consequently, the location of errors in loops is critical to analyzing the overall program. In this paper, we delve into the scalability potential of LocFaults, our error localization approach that utilizes Control Flow Graph (CFG) paths from counterexamples to calculate the Minimal Correction Deviations (MCDs) and Minimal Correction Subsets (MCSs) for each MCD found. The study presents the efficiency of LocFaults on programs with While-loops unfolded b times and deviated conditions ranging from 0 to n. Preliminary results show that LocFaults, constraint-based and flow-driven, is faster and provides more expressive information for the user compared to BugAssist, which is based on SAT and transforms the entire program into a Boolean formula.

Keywords

Error localization, LocFaults, BugAssist, Off-by-one bug, Minimal Correction Deviations, Minimal Correction Subsets

1. Introduction

Errors are inevitable in a program; they can disrupt proper operation and lead to serious financial consequences, posing a threat to human well-being [1]. Some software bug stories are cited in this link [2]. Consequently, the debugging process, which involves detecting, localizing, and correcting errors, is essential. Localizing errors is the most costly step, requiring the identification of the precise locations of suspicious instructions [3] to help the user understand why the program failed and facilitate error correction. When a program P does not conform to its specification (i.e., contains errors), a model checker can produce a trace of a counterexample, which is often long and challenging to comprehend, even for experienced programmers. To address this issue, we have proposed an approach named LocFaults [4], which is based on constraints that explore the paths of the program's Control Flow Graph (CFG) from the counterexample to calculate the minimal subsets necessary to restore the program's compliance with its postcondition. Ensuring that our method is highly scalable to meet the enormous complexity of software systems is a crucial criterion for its quality [5].

TACC 2023: Tunisian-Algerian Joint Conference on Applied Computing, November 06–08, 2023, Sousse, Tunisia

*Corresponding author.

m.bekkouche@esi-sba.dz (M. Bekkouche)

https://www.esi-sba.dz/fr/index.php/personnel/bekkouche-mohammed/ (M. Bekkouche)

0000-0002-8305-0542 (M. Bekkouche)

Several statistical approaches for error localization have been proposed, such as Tarantula [6, 7], Ochiai [8], AMPLE [8], Pinpoint [9], FLCN-S [10], FTFL [11], ConsilientSFL [12], and Poster [13]. Among them, Tarantula is the most famous and uses different metrics to calculate the degree of suspicion of each instruction in the program while running a series of tests. However, these approaches have a drawback in that they require a large number of test cases, while our approach only uses one counterexample. Another challenge with statistical approaches is the need for an oracle to determine if a test case's result is correct or not. To address this issue, we utilize the Bounded Model Checking (BMC) framework, which only requires a postcondition or assertion to verify.

Our approach aims to simplify the problem of error localization by reducing it to computing a minimal set that explains why a Constraint Satisfaction Problem (CSP) is infeasible. The CSP represents the constraints of the program, counterexample, and the assertion or postcondition violated. The calculated set can be either a Minimal Correction Subset (MCS) or a Minimal Unsatisfiable Subset (MUS). Generally, testing the feasibility of a CSP over a finite domain is an NP-complete problem, which is one of the most difficult NP problems. Thus, explaining the infeasibility in a CSP is equally challenging, if not harder, and can be classified as an NP-hard problem. While BugAssist [14, 15] is a BMC-based error localization method that employs a Max-SAT solver to compute the merger of MCSs of the Boolean formula of the entire program with the counterexample, it becomes inefficient for large programs. LocFaults also works from a counterexample to calculate MCSs.

In this paper, we investigate the scalability of LocFaults on programs with While-loops that are unfolded b times, and a number of deviated conditions ranging from 0 to 3. Our approach contributes in the following ways compared to BugAssist:

- We avoid transforming the entire program into a system of constraints. Instead, we use the CFG (Control Flow Graph) of the program to gather the constraints of the counterexample path and paths derived from it. We assume that at most k conditionals may contain errors, and we calculate MCSs only on the counterexample path and paths that correct the program.
- We do not convert program instructions into a SAT (Boolean satisfiability) formula. Instead, we use numerical constraints that will be handled by constraint solvers.
- We do not rely on MaxSAT solvers as black boxes. Instead, we use a generic algorithm that uses a constraint solver to calculate MCSs.
- We limit the size of the generated MCSs and the number of deviated conditions.
- We can work together multiple solvers during the localization process and choose the most efficient one according to the category of the CSP (Constraint Satisfaction Problem) constructed. For example, if the CSP of the detected path is linear over integers, we use a MIP (Mixed Integer Programming) solver. If it is nonlinear, we use a CP (Constraint Programming) and/or MINLP (Mixed Integer Nonlinear Programming) solver.

Based on our practical experience, as demonstrated in Section 5, we have found that the restrictions and distinctions employed by LocFaults make it faster and more expressive.

The paper is organized as follows. Section 2 introduces the definition of MUS and MCS. In Section 3, we define the $\leq$ k-MCD problem. In section 4, we describe our contribution to treating

erroneous loops, including the Off-by-one bug. The results of our experimental evaluation are presented in Section 5. Section 6 includes the conclusion and discussion of future work.

2. Definitions

In this section, we introduce the definition of an IIS/MUS and MCS.

CSP. A CSP (Constraint Satisfaction Problem) P is defined as a triple $< X, C, D >$, where:

- X : a set of n variables x_1, x_2, ..., x_n.
- $C = \{c_1, c_2, ..., c_m\}$ is the set of constraints.
- D : the tuple $< D_{x_1}, D_{x_2}, ..., D_{x_n} >$. The set D_{x_i} contains the values of the variable x_i.

A solution for P is an instantiation of the variables $I \in D$ that satisfies all the constraints in C. P is infeasible if it has no solutions. A sub-set of constraints C' in C is also said infeasible for the same reason except that it is limited to the constraints in C'.

We denote as:

- $Sol(< X, C', D >) = \emptyset$, to specify that C' has no solutions, so it is infeasible.
- $Sol(< X, C', D >) \neq \emptyset$, to specify that C' has at least one solution, so it is feasible.

A Linear Program, denoted as LP, is said to be linear if all the constraints in the set C are expressed as linear equations or inequalities. It is considered continuous if the domain of all variables is real. If at least one variable in the set X is an integer or a binary (which is a special case of an integer), and the constraints are linear, then P is referred to as a Mixed-Integer Linear Program (MIP). If the constraints are expressed as nonlinear equations or inequalities, then P is referred to as a Nonlinear Program (NLP).

Let $P =< X, C, D >$ an infeasible CSP, we define for P:

IS. An IS (Inconsistent Set) is an infeasible subset of constraints in the constraint set infeasible C. C' is an IS iff:

- $C' \subseteq C$.
- $Sol(< X, C', D >) = \emptyset$.

IIS or MUS. An IIS (Irreducible Inconsistent Set) or MUS (Minimal Unsatisfiable Subset) is an infeasible subset of constraints of C, and all its strict subsets are feasible. C' is an IIS iff :

- C' is an IS.
- $\forall C'' \subset C'.Sol(< X, C'', D >) \neq \emptyset$, (each of its parts contributes to the infeasibility), C' is called irreducible.

MCS. C' is a MCS (Minimal Correction Set) iff :

- $C' \subseteq C$.
- $Sol(< X, C \backslash C', D >) \neq \emptyset$.
- $\nexists\, C'' \subset C'$ such as $Sol(< X, C \backslash C'', D >) = \emptyset$.

3. The problem $\leq$ k-MCD

Given an erroneous program modeled in a CFG[1] $G = (C, A, E)$, where C is the set of conditional nodes, A is the set of assignment blocks, and E is the set of arcs, along with a counterexample, a Minimal Correction Deviation (MCD) is a set $D \subseteq C$ such that propagating the counterexample on all the instructions of G from the root, while having negated each condition[2] in D, allows the output to satisfy the postcondition. A MCD is called minimal (or irreducible) if no element can be removed from D without losing this property. In other words, D is a minimal program correctness in the set of conditions. The size of the minimal deviation is its cardinality. The problem of finding all MCDs of size smaller or equal to k is denoted as $\leq$ k-MCD.

As an illustration (refer to Fig. 1), consider the CFG of the program AbsMinus (refer to Fig. 1b). When provided with the counterexample $\{i = 0, j = 1\}$, this program has one minimal deviation of size 1. While the deviation $\{i_0 \leq j_0, k_1 = 1 \wedge i_0 \neq j_0\}$ does correct the program, it is not minimal. In fact, the only minimal correction deviation for this program is $\{k_1 = 1 \wedge i_0 \neq j_0\}$.

Table 1
The progress of LocFaults for the program AbsMinus

Deviated conditions	MCD	MCS	Figure
$\emptyset$	/	$\{r_1 = i_0 - j_0 : 13\}$	Fig. 1c
$\{i_0 \leq j_0 : 8\}$	No	/	Fig. 1d
$\{k_1 = 1 \wedge i_0! = j_0 : 10\}$	Yes	$\{k_0 = 0 : 7\}$, $\{k_1 = k_0 + 2 : 9\}$	Fig. 1e
$\{i_0 \leq j_0 : 8$, $k_1 = 1 \wedge i_0! = j_0 : 10\}$	No	/	Fig. 1f

Table 1 provides a summary of the progress of LocFaults for the program AbsMinus, with at most 2 conditions deviated from the counterexample $\{i = 0, j = 1\}$. The table displays the conditions deviated, indicating whether they are minimal or non-minimal deviations, and the MCSs (Minimal Correction Sets) calculated from the constructed constraint system : see columns 1, 2, and 3, respectively. Column 4 shows the figure that illustrates the path explored for each deviation. Additionally, the first and the third columns show the instruction and its corresponding line in the program.

For example, the first line in the table indicates that a single MCS ($\{r_1 = i_0 - j_0 : 13\}$) was found on the path of the counterexample.

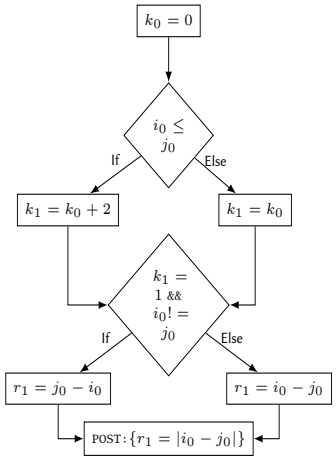
4. Error localization in loops

In the context of Bounded Model Checking (BMC) for programs, unfolding can be applied to the entire program or to loops separately [5]. Our algorithm, LocFaults [4], for error localization

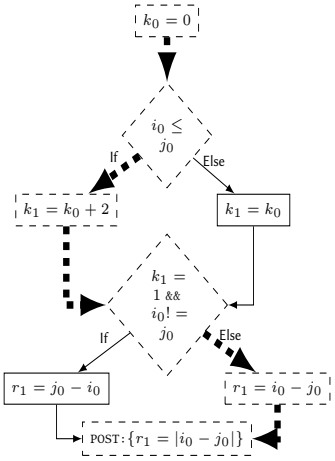
[1]We use Dynamic Single Assignment (DSA) form [16] transformation that ensures that each variable is assigned only once on each path of the CFG.
[2]To navigate to the intended branch, we negate the condition to take the opposite branch.

```
1 class AbsMinus {
2 /*@ ensures
3 @ ((i < j) ==> (\result == j-i
      )) &&
4 @ ((i >= j) ==> (\result == i-
      j)); */
5 int AbsMinus (int i, int j) {
6  int result;
7  int k = 0;
8  if (i <= j) {
9   k = k+2; } // error : k = k
       +2 instead of k=k+1
10  if (k == 1 && i != j) {
11   result = j-i; }
12  else {
13   result = i-j; }
14  return result; } }
```

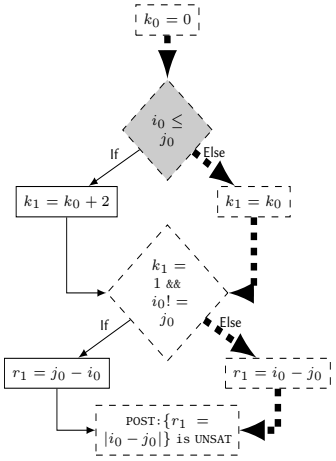
(a) The program AbsMinus



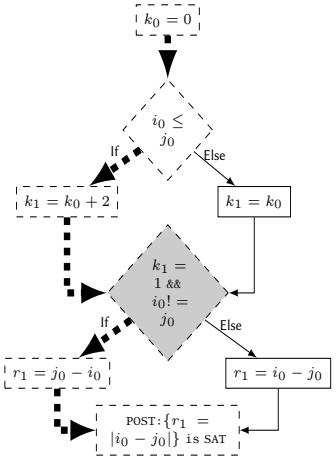
(b) The CFG in DSA form of AbsMinus



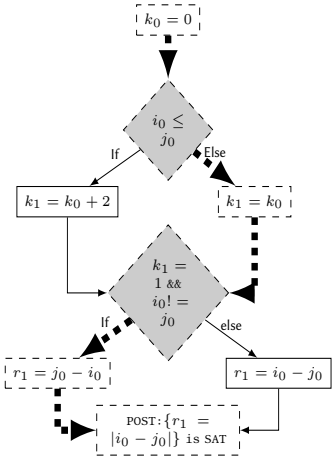
(c) The path of the counterexample



(d) The path obtained by deviating the condition $i_0 \leq j_0$



(e) The path by deviating the condition $k_1 = 1 \wedge i_0! = j_0$



(f) The path of a non-minimal deviation

Figure 1: The exploration of AbsMinus CFG performed by LocFaults

follows the latter approach, where we use a bound b to unfold loops by replacing them with nested conditional statements of depth b. For example, consider the program Minimum (refer to Fig. 2), which contains a single loop that calculates the minimum value in an array of integers. The effect on the control flow graph of the program Minimum before and after unfolding is illustrated in Figures 2 and 3, respectively. The while-loop is unfolded three times, since three iterations are required to calculate the minimum value for an array of size 4.

LocFaults takes the CFG of the erroneous program, CE (a counterexample), b_{mcd} (a bound on the number of deviated conditions), and b_{mcs} (a bound on the size of calculated MCSs) as input. It enables us to explore the CFG in depth by diverting a maximum of b_{mcd} conditions from the counterexample's path by performing the following steps:

- Propagating CE on the CFG until the postcondition is reached. Then, it calculates the

```
1 class Minimum {
2 /*The minimum in an array of n integers
      */
3 /*@ ensures
4   @ (\forall int k;(k>=0 && k<tab.
        length) ; tab[k]>=min);
5   @*/
6   int Minimum(int[] tab){
7     int min=tab[0];
8     int i=1;
9     while(i<tab.length-1){ /*error, the
          condition should be (i<tab.
          length)*/
10      if(tab[i]<min){
11        min=tab[i];
12      }
13      i=i+1;
14    }
15    return min;
16  }
17 }
```

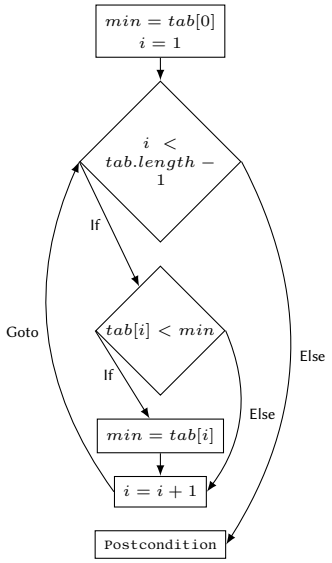


Figure 2: The program Minimum and its normal CFG (non-unfolded). The postcondition is $\{\forall$ int $k; (k \geq 0 \wedge k < tab.length); tab[k] \geq min\}$

MCSs on the CSP of the generated path to locate errors on the counterexample's path.

- Seeking to enumerate the sets $\leq b_{mcd} - MCD$. For each found MCD, it calculates the MCSs on the path that reaches the last deviated condition and allows for taking the path of the deviation.

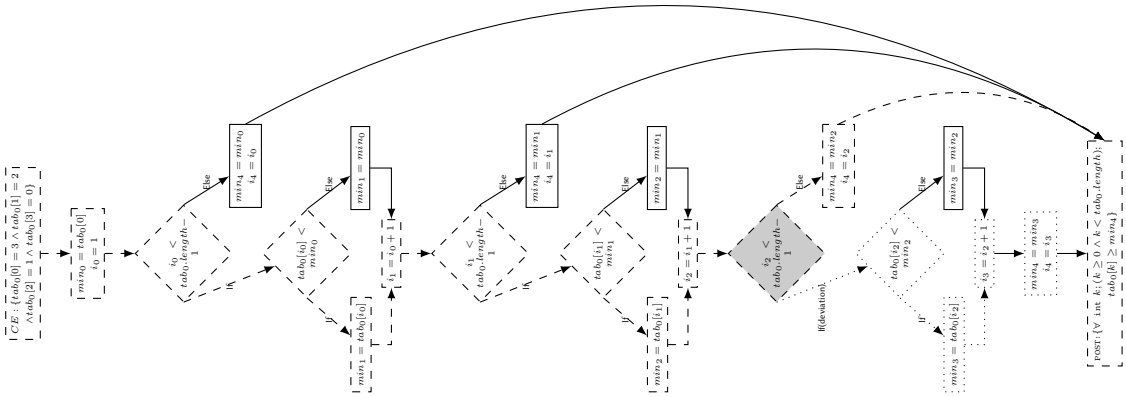


Figure 3: The CFG in DSA form of the program Minimum after unfolding its loop three times, with a highlighted path of a counterexample in dashed lines and a deviation that satisfies the postcondition in dotted lines.

Off-by-one bugs are among the most common errors associated with loops. These bugs cause loops to iterate one too many or one too few times. The cause of these bugs can be

traced back to improper initialization of loop control variables or an erroneous loop condition. The program Minimum provides an example of this type of error. The error occurs due to a falsified instruction in the loop condition (line 9) of the While loop. The correct condition should be (i < tab.length), where tab.length is the number of elements in the table tab. Using the counterexample $\{tab[0] = 3, tab[1] = 2, tab[2] = 1, tab[3] = 0\}$, we illustrated the initial faulty path in Figure 3 (shown in dashed lines) and the deviation for which the postcondition is satisfiable (the deviation and the path beyond the deviated condition are displayed in dotted lines).

Table 2 shows the erroneous paths generated (column PATH) and the corresponding MCSs calculated (column MCSs) for at most one deviated condition from the execution of the counterexample. The first row pertains to the path of the counterexample, while the second row corresponds to the path obtained by deviating the condition $\{i_2 \leq tab_0.length - 1\}$.

LocFaults identifies a single MCS on the path of the counterexample that includes the constraint $min_2 = tab_0[i_1]$. This constraint arises from the instruction on line 11 in the second iteration of the unfolded loop. When a condition is deviated, the algorithm suspects the third condition of the unfolded loop, i.e., $i_2 < tab_0.length - 1$. This deviation implies that we need to execute a new iteration to satisfy the postcondition.

Table 2
Paths and MCSs generated by LocFaults for the program Minimum

PATH	MCSs
$\{CE : [tab_0[0] = 3 \wedge tab_0[1] = 2 \wedge tab_0[2] = 1 \wedge tab_0[3] = 0],$ $min_0 = tab_0[0], i_0 = 1, min_1 = tab_0[i_0], i_1 = i_0 + 1, min_2 = tab_0[i_1],$ $i_2 = i_1 + 1, min_4 = min_2, i_4 = i_2, POST : [(tab_0[0] \geq min_4)$ $\wedge (tab_0[1] \geq min_4) \wedge (tab_0[2] \geq min_4) \wedge (tab_0[3] \geq min_4)]$	$\{min_2 = tab_0[i_1]\}$
$\{CE : [tab_0[0] = 3 \wedge tab_0[1] = 2 \wedge tab_0[2] = 1 \wedge tab_0[3] = 0],$ $min_0 = tab_0[0], i_0 = 1, min_1 = tab_0[i_0], i_1 = i_0 + 1,$ $min_2 = tab_0[i_1], i_2 = i_1 + 1, [\neg(i_2 \leq tab_0.length - 1)]$	$\{i_0 = 1\},$ $\{i_1 = i_0 + 1\},$ $\{i_2 = i_1 + 1\}$

This example illustrates a case of a program with an incorrect loop. The error lies in the stopping criterion, which prevents the program from iterating until the last element of the input array. LocFaults, with its deviation mechanism, is capable of accurately locating this type of error. It not only identifies suspicious instructions in the unfolded loop that were not present in the original program, but also provides information about the iterations in which they occur. This information could be extremely useful for programmers to identify errors in the loop.

5. Practical experience

To evaluate the scalability of our method, we compared its performance with that of BugAssist[3] using a benchmark set that we created[4]. The benchmark set comprises various implementations

[3]The tool BugAssist can be accessed at http://bugassist.mpi-sws.org/.
[4]The source code for all programs is available at http://capv.toile-libre.org/Benchs_Mohammed.html

of BubbleSort, Sum, and SquareRoot programs, consisting of 19, 48, and 91 variations, respectively. These programs incorporate loops, enabling us to study the scalability of our approach compared to BugAssist. To increase the program's complexity, we augmented the number of loop iterations in the execution of each tool. Both LocFaults and BugAssist were subjected to the same bound of unfolding loops.

To generate the CFG and counterexample, we employed the tool CPBPV [17], which stands for Constraint-Programming Framework for Bounded Program Verification. LocFaults and BugAssist are designed to operate on Java and C programs, respectively. To enable a fair comparison, we created two equivalent versions for each program:

- one version annotated with a JML specification in Java,
- another version annotated with the same specification in ACSL, using ANSI-C.

Both versions consist of an identical number of lines of instructions, including errors. The precondition defines the counterexample used for the program.

For computing the MCSs, we employed the CPLEX solvers of IBM ILOG, encompassing both MIP and CP solvers. We have implemented the algorithm proposed by Liffiton and Sakallah [18]. This implementation requires as input the infeasible set of constraints that correspond to the identified path and a bound on the size of calculated MCSs.

BugAssist leverages the CBMC tool [19] to generate erroneous traces and input data. We utilized MSUnCore2 [20] as the Max-SAT solver (used by BugAssist).

The experiments were conducted on an Intel Core i7-3720QM 2.60 GHz processor with 8 GB of RAM.

5.1. Benchmarks with loops

These benchmarks are utilized to evaluate the scalability of LocFaults in comparison to BugAssist for programs with loops, based on the increase of unfolding b. We selected three programs with loops: BubbleSort, Sum, and SquareRoot, and introduced an Off-by-one bug in each of them. The benchmark for each program is generated by increasing the number of unfolding b, where b represents the number of iterations through the loop in the worst case. Additionally, we vary the number of deviated conditions for LocFaults from 0 to 3.

We utilized the MIP solver of CPLEX for BubbleSort. For Sum and SquareRoot, we integrated the two solvers of CPLEX, CP and MIP, during the localization process. Specifically, during the collection of constraints, we employ a variable to store information on the type of CSP being constructed. When LocFaults identifies an erroneous path[5] and before calculating MCSs, it selects the appropriate solver based on the type of CSP associated with that path. If the CSP is non-linear, it uses the CP OPTIMIZER solver; otherwise, it uses the MIP solver.

For each benchmark, we provided an excerpt from the table containing the computation time[6]. Columns P and L represent the time of pretreatment and calculation of MCSs, respectively.

[5] An erroneous path is one on which we identify MCSs.

[6] The complete tables can be found at http://www.capv.toile-libre.org/Benchs_Mohammed.html#ravb, and the sources of these results are available at http://www.capv.toile-libre.org/Benchs_Mohammed.html#sr.

5.1.1. BubbleSort benchmark

BubbleSort is an implementation of the bubble sort algorithm. The erroneous statement in the program causes the program to sort the input array by considering only its $n - 1$ first elements, leading to incorrect results. The malfunction of BubbleSort is due to the insufficient number of iterations performed by the loop, which is caused by the faulty initialization of the variable i as $i = tab.length - 1$; the correct instruction should be $i = tab.length$.

The graph in Figure 4 depicts the variation in computation times for different versions of LocFaults and BugAssist, based on the number of unfoldings.

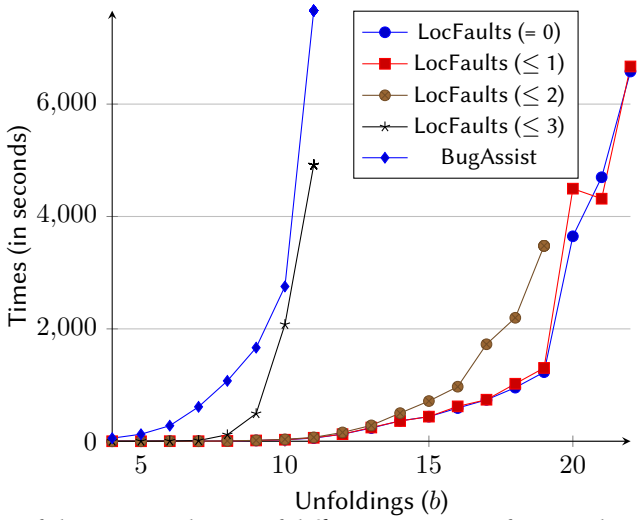


Figure 4: Comparison of the time evolution of different versions of LocFaults and BugAssist for the BubbleSort benchmark, with an increasing unwinding loop limit.

The runtime of LocFaults and BugAssist shows exponential growth with the number of unfoldings, with BugAssist consistently having the highest computation times. BugAssist may not be effective for this benchmark. Among the different versions of LocFaults (with up to 3, 2, 1, or 0 conditions deviated), all remain usable up to a certain unfolding threshold. The number of unfoldings at which the computation time of BugAssist becomes prohibitively high is lower compared to LocFaults. Additionally, the computation time of LocFaults with up to 3 conditions deviated is lower than that of LocFaults with up to 2 conditions deviated, which is also lower than that of LocFaults with up to 1 condition deviated. The computation times of LocFaults with up to 1 and 0 conditions deviated are nearly identical.

5.1.2. SquareRoot and Sum benchmarks

The SquareRoot program (refer to Figure 5) is designed to find the integer part of the square root of the integer 50. An error is injected at line 13, resulting in the incorrect return value of 8, whereas the correct value should be 7. This program has been previously used in the paper describing the BugAssist approach. It contains a linear numerical calculation in its loop and non-linear calculation in its postcondition.

```
1  public class SquareRoot {
2      /*@ ensures((res*res<=val) && (res+1)*(res+1)>val);*/
3      public static int squareRoot ()
4      {
5          int val = 50;
6          int i = 1;
7          int v = 0;
8          int res;
9          while (v < val){
10             v = v + 2*i + 1;
11             i = i + 1;
12         }
13         res = i; //error: should be res = i-1
14         return res;
15     }
16 }
```

Figure 5: The SquareRoot program.

With an unwinding limit of 50, BugAssist identifies the following suspicious instructions for this program: $\{9, 10, 11, 13\}$. The localization time is $36.16s$ and the pre-treatment time is $0.12s$.

LocFaults identifies suspicious instructions by providing their location in the program (instruction line), as well as the line of the condition and the iteration number of the loop leading to that instruction. For example, $\{9 : 2.11\}$ indicates that the suspicious instruction is on line 11 of the program, which is inside a loop with the stop condition at line 9 and the iteration number is 2. The sets of suspected instructions identified by LocFaults are listed in Table 3. The

Table 3
MCD and MCSs calculated by LocFaults for SquareRoot.

$\emptyset$	$\{5\}, \{6\}, \{9 : 1.11\}, \{9 : 2.11\}, \{9 : 3.11\},$ $\{9 : 4.11\}, \{9 : 5.11\}, \{9 : 6.11\}, \{9 : 7.11\}, \{13\}$
$\{9 : 7\}$	$\{5\}, \{6\}, \{7\}, \{9 : 1.10\}, \{9 : 2.10\}, \{9 : 3.10\},$ $\{9 : 4.10\}, \{9 : 5.10\}, \{9 : 6.10\}, \{9 : 1.11\},$ $\{9 : 2.11\}, \{9 : 3.11\}, \{9 : 4.11\}, \{9 : 5.11\}, \{9 : 6.11\}$

pretreatment time is 0.769s. The time for exploring the CFG and calculating MCSs is 1.299s.

We conducted a study of the times for LocFaults and BugAssist with values of "val" ranging from 10 to 100 (where the number of unfoldings "b" used is equal to "val"), in order to analyze the combinatorial behavior of each tool for this program.

The Sum program receives a positive integer n from the user and calculates the value of $\sum_{i=1}^{n} i$ as per the postcondition. The error in Sum lies in the condition of its loop, causing it to calculate the sum $\sum_{i=1}^{n-1} i$ instead of $\sum_{i=1}^{n} i$. The program contains linear numerical instructions within the core of the loop, along with a nonlinear postcondition.

The time results for the SquareRoot and Sum benchmarks are presented in Tables 4 and 5, respectively. It is observed that the execution time of BugAssist increases rapidly, while the times of LocFaults remain relatively constant. Furthermore, the times of LocFaults with at most 0, 1, and 2 conditions deviated are comparable to those of LocFaults with at most 3 conditions deviated.

Table 4
Computation time for SquareRoot Benchmark.

Program	b	LocFaults					BugAssist	
		P	L				P	L
			$= 0$	≤ 1	≤ 2	≤ 3		
V0	10	$1.096s$	$1.737s$	$2.098s$	$2.113s$	$2.066s$	$0.05s$	$3.51s$
V10	20	$0.724s$	$0.974s$	$1.131s$	$1.117s$	$1.099s$	$0.05s$	$6.54s$
V20	30	$0.771s$	$1.048s$	$1.16s$	$1.171s$	$1.223s$	$0.08s$	$12.32s$
V30	40	$0.765s$	$1.048s$	$1.248s$	$1.266s$	$1.28s$	$0.09s$	$23.35s$
V40	50	$0.769s$	$1.089s$	$1.271s$	$1.291s$	$1.299s$	$0.12s$	$36.16s$
V50	60	$0.741s$	$1.041s$	$1.251s$	$1.265s$	$1.281s$	$0.14s$	$38.22s$
V70	80	$0.769s$	$1.114s$	$1.407s$	$1.424s$	$1.386s$	$0.19s$	$57.09s$
V80	90	$0.744s$	$1.085s$	$1.454s$	$1.393s$	$1.505s$	$0.22s$	$64.94s$
V90	100	$0.791s$	$1.168s$	$1.605s$	$1.616s$	$1.613s$	$0.24s$	$80.81s$

Table 5
Computation time for Sum Benchmark.

Program	b	LocFaults					BugAssist	
		P	L				P	L
			$= 0$	≤ 1	≤ 2	≤ 3		
V0	6	$0.765s$	$0.427s$	$0.766s$	$0.547s$	$0.608s$	$0.04s$	$2.19s$
V10	16	$0.9s$	$0.785s$	$1.731s$	$1.845s$	$1.615s$	$0.08s$	$17.88s$
V20	26	$1.11s$	$1.449s$	$7.27s$	$7.264s$	$6.34s$	$0.12s$	$53.85s$
V30	36	$1.255s$	$0.389s$	$8.727s$	$4.89s$	$4.103s$	$0.13s$	$108.31s$
V40	46	$1.052s$	$0.129s$	$5.258s$	$5.746s$	$13.558s$	$0.23s$	$206.77s$
V50	56	$1.06s$	$0.163s$	$7.328s$	$6.891s$	$6.781s$	$0.22s$	$341.41s$
V60	66	$1.588s$	$0.235s$	$13.998s$	$13.343s$	$14.698s$	$0.36s$	$593.82s$
V70	76	$0.82s$	$0.141s$	$10.066s$	$9.453s$	$10.531s$	$0.24s$	$455.76s$
V80	86	$0.789s$	$0.141s$	$13.03s$	$12.643s$	$12.843s$	$0.24s$	$548.83s$
V90	96	$0.803s$	$0.157s$	$34.994s$	$28.939s$	$18.141s$	$0.31s$	$785.64s$

6. Conclusion

The LocFaults method detects suspicious subsets by analyzing the paths of the CFG to identify the MCDs and MCSs from each MCD, utilizing constraint solvers. On the other hand, the BugAssist method calculates the merger of MCSs by transforming the entire program into a Boolean formula and leveraging Max-SAT solvers. Both methods start from a counterexample to identify potential issues. In this paper, we have presented a scalability exploration of LocFaults, with a focus on handling loops with the Off-by-one bug. The initial results indicate that LocFaults is more effective than BugAssist for programs with loops. The execution times of BugAssist tend to rapidly increase with the number of loop unfoldings, while LocFaults shows better scalability in this aspect.

As part of our future work, we plan to validate our results on programs with more complex loops. We also intend to compare the performance of LocFaults with existing statistical methods. To further enhance our tool, we are developing an interactive version that presents suspect

subsets one by one, leveraging the user's knowledge to select the conditions that should be deviated. Additionally, we are considering ways to extend our method to handle numerical instructions involving calculations on floating-point

References

[1] R. N. Charette, Why software fails, IEEE spectrum 42 (2005) 36.

[2] M. Bekkouche, Bug stories, 2015. URL: http://www.capv.toile-libre.org/Bug_stories.html.

[3] W. E. Wong, R. Gao, Y. Li, R. Abreu, F. Wotawa, A survey on software fault localization, IEEE Transactions on Software Engineering 42 (2016) 707–740.

[4] M. Bekkouche, H. Collavizza, M. Rueher, Locfaults: A new flow-driven and constraint-based error localization approach, in: Proceedings of the 30th Annual ACM Symposium on Applied Computing, 2015, pp. 1773–1780.

[5] V. D'silva, D. Kroening, G. Weissenbacher, A survey of automated techniques for formal software verification, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 27 (2008) 1165–1178.

[6] J. A. Jones, M. J. Harrold, J. Stasko, Visualization of test information to assist fault localization, in: Proceedings of the 24th international conference on Software engineering, 2002, pp. 467–477.

[7] J. A. Jones, M. J. Harrold, Empirical evaluation of the tarantula automatic fault-localization technique, in: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, 2005, pp. 273–282.

[8] R. Abreu, P. Zoeteweij, A. J. Van Gemund, On the accuracy of spectrum-based fault localization, in: Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007), IEEE, 2007, pp. 89–98.

[9] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, E. Brewer, Pinpoint: Problem determination in large, dynamic internet services, in: Proceedings International Conference on Dependable Systems and Networks, IEEE, 2002, pp. 595–604.

[10] A. Zakari, S. P. Lee, I. A. T. Hashem, A single fault localization technique based on failed test input, Array 3 (2019) 100008.

[11] A. Dutta, K. Kunal, S. S. Srivastava, S. Shankar, R. Mall, Ftfl: A fisher's test-based approach for fault localization, Innovations in Systems and Software Engineering 17 (2021) 381–405.

[12] A. Majd, M. Vahidi-Asl, A. Khalilian, B. Bagheri, Consilientsfl: using preferential voting system to generate combinatorial ranking metrics for spectrum-based fault localization, Applied Intelligence 52 (2022) 11068–11088.

[13] Q. I. Sarhan, Á. Beszédes, Poster: Improving spectrum based fault localization for python programs using weighted code elements, in: 2023 IEEE Conference on Software Testing, Verification and Validation (ICST), IEEE, 2023, pp. 478–481.

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

[15] M. Jose, R. Majumdar, Bug-assist: Assisting fault localization in ansi-c programs, in: Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23, Springer, 2011, pp. 504–509.

[16] M. Barnett, K. R. M. Leino, Weakest-precondition of unstructured programs, in: Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, 2005, pp. 82–87.
[17] H. Collavizza, M. Rueher, P. Van Hentenryck, Cpbpv: a constraint-programming framework for bounded program verification, Constraints 15 (2010) 238–264.
[18] M. H. Liffiton, K. A. Sakallah, Algorithms for computing minimal unsatisfiable subsets of constraints, Journal of Automated Reasoning 40 (2008) 1.
[19] E. Clarke, D. Kroening, F. Lerda, A tool for checking ansi-c programs, in: 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 10, Springer, 2004, pp. 168–176.
[20] J. Marques-Silva, The msuncore maxsat solver, SAT (2009) 151.