The Journal of Systems and Software 209 (2024) 111897

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Localizing faults using verification technique☆

Sudakshina Dutta[1]

Indian Institute of Technology Goa, India

ARTICLE INFO

Keywords:
Fault
Localization
Verification
Region, Static

ABSTRACT

In model-based fault diagnosis, it is assumed that a correct model of each program being diagnosed is available. In general, these techniques require test cases and user-specified assertions to localize the fault. This paper aims to localize faults in a faulty program without user-specified assertions and without executing the programs, and therefore, without using test cases. Given the faulty and the correct versions of a program, a product code is automatically constructed and assertions are automatically generated. The proposed method is a full y automatic, model-based static approach to fault localization. The proposed method reduces the fault search space by removing equivalent regions from the product code using verification techniques. To identify these components, the bounded model checker CBMC is used. The invoked functions from the correct and the faulty programs are considered uninterpreted functions and MiniSat is used as a backend solver. The proposed method can also be applied on static slices of the correct and faulty programs. Also, the identified fault search space can be analyzed along with the generated counterexample trace to pinpoint the fault. The experimental data is presented that supports the applicability of our approach. We demonstrate the effectiveness of the proposed method using the Siemens TCAS and NTS benchmark suite. It is observed that the method can also successfully localize the wrong safety check bug produced by the LLVM compiler.

1. Introduction

The usage of software is fundamental to our lives today and its influence is constantly encountered with its increasing usage. Software fault localization or the act of identifying the locations of faults in a program, is widely recognized to be one of the most tedious, time-consuming, and expensive — yet equally critical activities in program debugging. Due to the increasing scale and complexity of software today, manually locating faults when failures occur is becoming infeasible, and consequently, there is an increasing demand for techniques that can guide software developers to locate faults in a program with minimal human intervention. As a result, a broad spectrum of fault localization techniques is proposed, each of which aims to streamline the fault localization process and make it more effective by attacking the problem in a unique way.

In this paper, we propose a fully automatic, model-based static approach to fault localization. The proposed method takes the correct and the faulty programs as input. It then identifies a region in the faulty program which subsumes the erroneous statement(s) with respect to the corresponding non-erroneous region of the correct program and it is termed as the *region of the fault*. The method for localizing faults continues in two passes. Separate product codes of the programs are constructed for the two passes. A product code (Barthe et al., 2016), Zaks and Pnueli (2008) is a composition of two input programs which reduces the equivalence checking problem to analysis of a cross-product program of the two input programs to leverage existing program analysis techniques. This method works well for structurally similar programs. In the forward pass, the proposed method identifies the *starting point* of the region of the fault and in the backward pass, the proposed method identifies the *ending point* of the region of the fault.

The proposed method reduces the search space of the fault by removing equivalent regions from the product codes using verification techniques. For reducing the search space, the concepts of a *nested block*, outermost nested block and section as explained in Dutta (2022) are used and these concepts are also useful to construct product code. A *nested block* corresponds to an entire if-then-else or loop segment of a code. The *outermost nested block* is a nested block that is not nested in any nested block.

In the *forward pass*, the product code is constructed in a lockstep fashion by taking a section of code from the beginning of the correct program and a section of code from the beginning of the faulty program. In the next step, assertions for proving equivalence of the added part are included and it is subjected to the equivalence checker. If validation succeeds, the next sections from both the programs and assertions to prove equivalence are appended in the product code, and

☆ Editor: Prof Raffaela Mirandola.

E-mail address: sudakshina@iitgoa.ac.in.

[1] The work has been done solely by myself.

https://doi.org/10.1016/j.jss.2023.111897
Received 22 August 2022; Received in revised form 5 September 2023; Accepted 3 November 2023
Available online 14 November 2023

this process continues until the validation process fails in some step and this results in identifying the starting point of the region of the fault. In the *backward pass*, a separate product code is constructed in a lockstep fashion by taking a section of code from the end of the correct program and a section of code from the end of the faulty program. In the next step, assertions for proving equivalence of the added part are included and it is validated using the equivalence checker. If validation succeeds, the previous sections from both programs are prepended in the product code, and this process continues until the validation process fails in some step and this results in identifying the ending point of the region of the fault. The starting and the ending points are pairs of states where the first and the second states are taken from the flowchart (Manna, 1971) representations of the correct and the faulty programs, respectively.

The proposed method assumes that a correct program and the corresponding faulty programs are structured, and written in C programming language and they are given as inputs to the proposed method. The techniques for checking the equivalence of two programs are used to identify the region of the code segment of a correct program against the faulty program where the fault is present. The bounded model checker for C and C++ programs, CBMC (Clarke et al., 2004), is used for identifying the region of the fault by validating the product code. The verification is performed by unwinding the loops in the program and passing the resulting program to a decision procedure. CBMC is able to determine automatically an upper bound on the number of loop iterations. This may even work when the number of loop unwindings is not constant. Initially, the product program for every function is constructed and validated using CBMC assuming equality of the input variables to derive the equality of output variables. If CBMC cannot determine the loop bound in any step, the unwinding depth is assumed to be 1 for all further steps of fault localization. Otherwise, CBMC determines the loop unwinding depths for all steps of forward or backward passes. The outputs of all the functions present in the benchmark programs depend on only on inputs. They are abstracted as uninterpreted functions and MiniSAT is used as backend solver.

A static program slice (Weiser, 1984), Tip (1995) consists of all statements in a program that may affect the values of a variable v in statement x and the method is widely used for localizing faults. The proposed method can be applied on static slices for output variables extracted from the correct and faulty programs instead of the entire programs to reduce the size of the region of fault. Since optimization of code leaves most code fragments unchanged, the proposed method removes the functionally equivalent portions from the product code of the static slices and constructs the region of the fault. If the bounded model checker CBMC fails to prove a property, it prints a program trace that begins with the function where the property occurs and ends in a state which violates the property. We can consider both the region of fault and the counterexample trace to pinpoint the source of the fault. The counterexample trace must contain the identified region of fault for a single fault. If we analyze the code segment of region of fault in the code corresponding to the counterexample trace, then the trace to analyze for fault localization becomes even shorter and locating the fault becomes easier.

Most of the existing approaches for fault localization (Ball et al., 2003; Chaki et al., 2004; Griesmayer et al., 2007; Grismayer et al., 2010; Groce and Visser, 2003; Groce et al., 2006, 2004; Griesmayer et al., 2007; Jose and Majumdar: Cause Clue Clauses, 2011) describe algorithms where a program, a correctness specification (either a post-condition, an assertion, or a "golden output"), and corresponding test case that demonstrates the violation of the specification are taken as inputs. In contrast, our method proceeds statically *without* user-specified assertions and *without* executing the programs. Our approach is a revision and extension of our earlier work (Dutta, 2022). In particular, the fault localization method now uses the widely used bounded model checker CBMC instead of the programming language Dafny which is used mostly for teaching purposes and does not scale for large programs. Also the proofs of termination and correctness of the method are also provided. The procedure has also been successful in localizing faults for a larger set of examples which include NTS (Dutta et al., 2019), https://github.com/ArpitaDutta/NTS_Repository benchmark suite as CBMC provides features e.g., handling global variables, etc.

This method has potential applications in the localization of faults in transformations applied by compilers. Recollect that an optimizing compiler applies a sequence of transformations on an input program to obtain a semantically equivalent output program that uses fewer resources. But transformations are not always bug-free. Hence, verification techniques are applied to ensure functional equivalence of the source and the transformed programs. We observed that the method proposed in the present paper can successfully localize a bug that is introduced in an input program by the LLVM compiler (Lattner and Adve, 2004). The proposed method is also validated by localizing the region of the fault for all the 41 erroneous versions of the programs against the correct TCAS benchmark program and all the erroneous mutants of NTS benchmark program. We have also applied the method to detect manually-seeded faults in the compiler-transformed code reported in the literature (Karfa et al., 2008).

The contributions of the paper are summarized below:

1. An approach to automatically identify the location of the fault is proposed which proceeds statically i.e., without executing the programs. To the best of our knowledge, the proposed method is the most effective, static method where the error is localized without requiring test cases and assertions and it localizes faults for a larger set of examples than previous methods (Dutta, 2022). The correctness proofs are provided in Section 11.
2. The wrong safety check bug of LLVM is localized. Also the proposed method detects all seeded errors for Siemens TCAS benchmark suite, all mutants of NTS benchmark suite and also detected manually-seeded faults in the compiler-transformed code available in the literature.
3. The proposed method can be applied to the static slices of the correct and the faulty programs for any output variable to further narrow down the search space of the fault. Also, both the counterexample trace and identified region of fault can be used to pinpoint the fault.

The paper is organized as follows. Section 2 aims to motivate the reader with an example, Section 3 describes the model used for applying verification techniques, Section 4 introduces the method of equivalence checking used for localization of fault, Section 5 represents the algorithms, Section 6 revisits the example in a more detailed manner, Section 7 presents experimental results, Section 8 presents an example to explain how the loops are handled, Section 9 presents related work, Section 10 represents threats to validity, Section 11 contains correctness proofs and Section 12 concludes the paper.

2. Illustrative examples

Consider the example shown in Fig. 1 where a correct program and the corresponding faulty program are shown. The programs are taken from TCAS benchmark suite. The faulty program corresponds to version $v1$ of the benchmark suite. The faulty program (Fig. 1(b)) is functionally almost similar to the source program; the only difference between the source and the faulty programs is that the output variable in the faulty program has been differently computed in the if-block. For brevity, the code segments shown in Fig. 1 mention only one function e.g., $Non\_Crossing\_Biased\_Climb$ where an error is present in the faulty program. All other functions are equivalent and hence are omitted from the presentation. Note that to localize faults of a pair of functions, all the invoked functions are considered as uninterpreted functions and their names begin with the prefix __CPROVER_uninterpreted_ in CBMC encoding to avoid spurious errors.

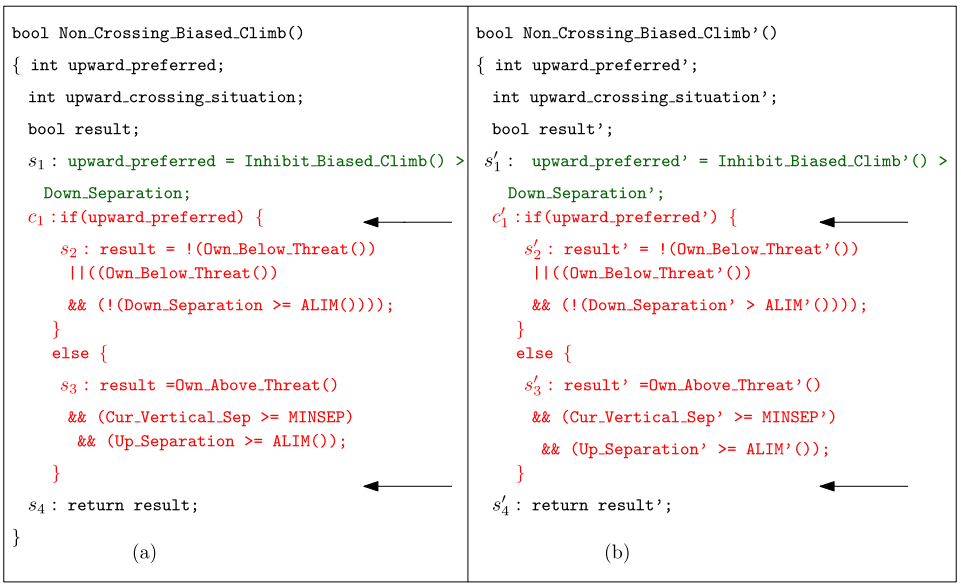

Fig. 1. TCAS benchmark programs *v1space*: (a) Correct program, (b) Erroneous program.

The faulty program has undergone modulo variable renaming Baader and Snyder (2001) which refers to an equational theory which is defined by the set of identities for renaming. In this context, every variable name of the correct program is replaced by its dashed version in the faulty program for ease of fault localization. In the process of fault localization, the values of input variables that have undergone modulo renaming are considered to be equal. In the correct program, the global variables *Down_Separation*, *Cur_Vertical_Sep*, $MINSEP$, *Up_Separation* are present and corresponding modulo renamed variables are present as global variables in the faulty program. The initial value of the global variables are assumed to be 0 by CBMC and hence the values of a variable and corresponding modulo renamed variable are equal. In the correct program of Fig. 1(a), the output variable *result* is computed in terms of the global variable *Down_Separation* and the functions *Own_Below_Threat*() and $ALIM()$ in the if-block. However, it is differently computed in the faulty program which results in a difference in the values of the modulo renamed output variable. The code segment that is proved to be equivalent in the forward pass is shown in green color and the segment of code which cannot be proved to be equivalent in forward pass is shown in red color. The same pair of nested blocks cannot be proved equivalent also in the backward pass. As shown in Fig. 1, the top arrows indicate the starting point and the bottom arrows indicate the ending point of the region of the fault. The region between the starting and the ending point is considered as region of fault. Although the arrows indicate statements in the programs, they refer to a pair of states taken from the flowchart representations of the programs. The values of the output variables *result* and *result*$'$ are considered for validation. Note that the proposed method removes the equivalent code segments (which refer to the code segment outside the `if-else` block) from the region of the fault using both the passes (detailed in Section 6).

The proposed method can also be applied to the static slices for the output variables instead of the entire input programs. Note that the static slicing method for the variable *result* considers every statement in the correct and the faulty programs. The proposed fault localization method reduces the search space of the fault even more by confining the search space to only the `if-else` block without the assignments to the variables *upward_preferred* and *upward_preferred*$'$. Also, we can consider both the counterexample trace and the region of fault to pinpoint the source of the fault. In the counterexample trace, the value *result* in the statement s_2 is computed as $TRUE$ whereas the value of *result*$'$ in the statement s'_2 is computed as $FALSE$. All the

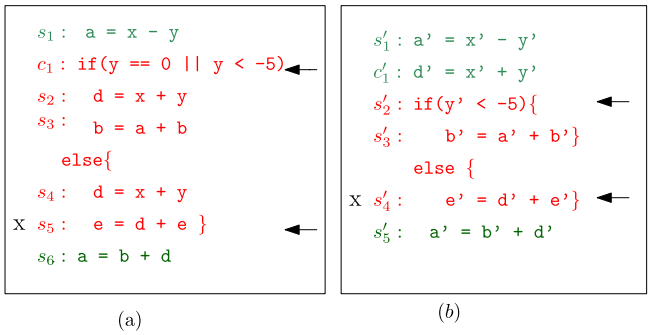

Fig. 2. (a) Source program, (b) transformed program after the forward pass, (c) Source program and (d) transformed program after the backward pass. The region of code which is proved to be equivalent in a pass are shown with green color and the region of code which cannot be proved to be equivalent in a pass are shown with red color. Also, the cross marks are put beside the portions of code that will be absent from the static slices.

other variable of the correct program and the corresponding variables of the erroneous program have the same values in the counterexample trace. The proposed method indicates that only the nested blocks are present in the region of fault. If we analyze the code segment of region of fault (i.e., the only pair of nested blocks), we can infer that either the segment of code corresponding to if−block or the segment of code corresponding to the $else$−block or code for both are the causes of error. A code segment of a faulty program can be referred to as cause of error if the code segment introduces the error which is manifested if the program is executed. However, if we only analyze the code segment of region of fault in the counterexample trace, the conclusion would be only the statements if−blocks i.e., s_2 and s'_2 to be the cause of error.

Consider the example shown in Fig. 2 where a correct program and the corresponding faulty program are shown in Figs. 2(a) and 2(b), respectively. The faulty program is the transformed program that is generated from the source program by applying code motion transformation (Gupta et al., 2004) on the correct program. Also, the *wrong safety check* bug of the LLVM compiler as reported in Yang et al. (2011) is seeded in the transformed program and the resulting program is shown in Fig. 2(b).

The transformed program has undergone modulo variable renaming. In the correct program, the input variables are x, y, the initial

values of the variables b and e and the output variable is a. In the transformed program, the input variables are x' x', y', the initial values of the variables b' and e' and the output variable is a' a'. In the correct program of Fig. 2(a), the output variable a is computed in terms of the variables b and d and they are differently assigned in the `if-else` branches. The variable d is assigned $x + y$ in both the `if-else` branches in the source program. In the transformed program, due to the applied code motion optimization technique, the common operations on the variable d are moved out to the preceding basic blocks. The transformed program (Fig. 2(b)) is functionally almost similar to the source program; the only difference between the source and the transformed program is that the if-condition in the transformed program has been wrongly simplified to $(y' < -5)$. The difference results in functional inequivalence of the programs (specifically when the values of y and y' are 0) and is referred to as a bug due to *wrong safety check*. In the process of fault localization, the values of input variables that have undergone modulo renaming are considered to be equal. Like the previous example, the region of code which is proved to be equivalent in forward pass are shown with green color (statements s_1 and s'_1). Also, the forward pass identifies the starting point of the region of fault which is indicated by the top arrows. The region of code which is proved to be equivalent in the backward pass is shown with green color (statements s_6 and s'_5). The backward pass identifies the ending point of the region of fault which is indicated by bottom arrows. The region of fault is shown with red color. Note that for each pass, the equivalence of two sections of codes are checked only for the variables of the correct program which are modulo renamed in the faulty program (e.g., a and a' for the sections which begin the programs). For ease of understanding, the arrows indicate statements in Fig. 2 although the said points refer to a pair of states of flowchart representation of the programs. The values of the output variables a and a' are considered for validation. Note that the proposed method removes the equivalent code segments (which refer to the assignments to the variables a and a') from the region of the fault using both the passes.

3. Flowcharts

For localizing faults, the source and the transformed programs are modeled using flowcharts (Manna, 1971). A flowchart $F = \langle Q, q_0, q_{n-1}, V, f, h, U\rangle$, where

1. $Q = \{q_0, q_1, \ldots, q_{n-1}\}$ is finite set consisting of control (C), join (J) and data states (D) partitioned into a set of control states C, join states J and data states D. In other words, $Q = C \cup J \cup D$,
2. $q_0 \in Q$ is the start state which is a data state,
3. $q_{n-1} \in Q$ is the end state which is also a data state,
4. V is the set of storage variables. It is the union of input (I), output (O) and temporary (T) variables i.e., $V = I \cup O \cup T$,
5. $f : C \times \{True, False\} \rightarrow Q$ is the state transition function from control states,
6. $h : D \cup J \rightarrow Q$ is the state transition function from data or join state,
7. U : It is the set of assignments belonging to the set D of the form of $\{x \Leftarrow e | x \in O \cup T\}$ and e is an expression over $I \cup T \cup M$, where M represents calls to a set of uninterpreted functions . The output of the referred uninterpreted functions depends only on input.

The flowchart contains a set of control states C and join states J. A control state corresponds to a boolean condition in a program. A join state corresponds to a specific control state present in the flowchart. It is not associated to any statement and generally depicts the closing of a block of statements of the program which starts with the conditions of the corresponding control state (in `if-else` or loop statements). The number of join states is the same as the number of control states. All other states are data states.

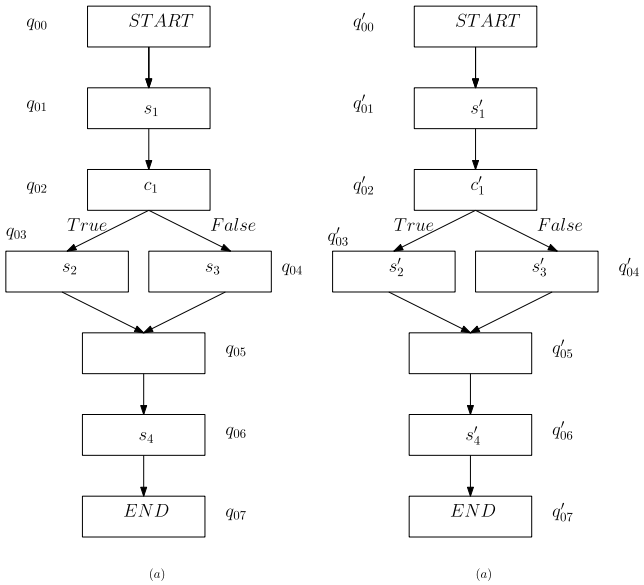

Fig. 3. The flowcharts of (a) the correct program (b) and the faulty program.

The flowcharts corresponding to the correct and the faulty programs of Fig. 1 are presented in Fig. 3. The flowchart corresponding to the correct program is shown in Fig. 3(a) and it is represented by $F_s = \langle Q_s, q_{0_s}, q_{n_s}, V_s, f_s, h_s, U_s\rangle$, say, where $Q_s = \{q_{00}, q_{01}, q_{02}, q_{03}, q_{04}, q_{05}, q_{06}, q_{07}\}$, $q_{0_s} = q_{00}$, $q_{n_s} = q_{07}$, $V_s = I_s \cup O_s \cup T_s$, where $I_s = \{Down\_Separation, Cur\_Vertical\_Sep, MINSEP, Up\_Separation\}$, $O_s = \{result\}$, $T_s = \{upward\_preferred\}$. Also note that $C = \{q_{02}\}$, $J = \{q_{05}\}$ and the rest of the states belong to the data set D. Some typical examples of f_s, h_s and u_s are $f(q_{02}, True) = q_{03}$, $f(q_{02}, False) = q_{04}$, $h(q_{01}) = q_{02}$, $U_s = \{s_2, s_3\}$. The flowchart corresponding to the transformed program is $F_t = \langle Q_t, q_{0_t}, q_{n_t}, V_t, f_t, h_t, U_t\rangle$, say, and all the components are similarly computed. In the source program, there is one nested block B which starts with the starting state q_{02} and ends with the ending state q_{05}.

Note that a (finite) elementary path segment in a flowchart is a sequence of states which starts with a start state or with a join state and ends if a control or a join or an end state is encountered. It is of the form of $\langle q_i, q_{i+1}, \ldots, q_{j-1}\rangle$ where $q_i, q_j \in Q$ are states and none of the states is repeated. Every elementary path segment α corresponds to a data transformation d_α. The data transformation d_α is a set of assignment statements in the path segment α. In this paper, the elementary path segments are referred to as path segments. Note that a path segment corresponds to the section which does not correspond to any nested block. In other words, it corresponds to a straight-line code segment which is not present in any loop or if-then-else block. In Fig. 3, there are two path segments that do not belong to any nested block which are $\alpha_1 space$: $\langle q_{00}, q_{01}\rangle$, $\alpha_2 space$: $\langle q_{06}\rangle$ in the correct program. Also, there are two path segments which do not belong to any nested block — $\alpha'_1 space$: $\langle q'_{00}, q'_{01}\rangle$, $\alpha'_2 space$: $\langle q'_{06}\rangle$ in the faulty program. A nested block B with the starting state q_{02} and the ending state q_{05} is present in the correct program. Similarly, in the faulty program, a nested block — B' is present which starts with the starting state q'_{02} and ends with the ending state q'_{05}.

An input program can be any sequence of nested blocks and path segments. A *nested block* is a set of states that typically starts with a designated control state and ends with a designated join state. The control state of a nested block is called the *starting state* and the join state of a nested block is called the *ending state*. There is only one single nested blocks present in both the programs given in Figs. 1(a) and 1(b) corresponding to the if-then-else statements present in both the programs. The nested blocks are outermost nested blocks as they are not enclosed in any other nested blocks.

4. Equivalence of flowcharts

Let the correct program be represented by the flowchart $F_0 = \langle Q_0, q_{0_0}, q_{0_{m-1}}, V_0, f_0, h_0, U_0\rangle$ and the faulty program be represented by $F_1 = \langle Q_1, q_{1_0}, q_{1_{n-1}}, V_1, f_1, h_1, U_1\rangle$. The flowchart F_0 is computationally equivalent to the flowchart F_1 if the values of input variables of the correct program and the corresponding modulo renamed variables in the faulty program are equal and the validation process terminates, then the values of output variables of the correct program and the corresponding modulo renamed variables in the faulty program are equal. In general, the process of formal verification is undecidable and it is not always possible for any algorithm to generate verification results. The proposed method uses CBMC for validation and it cannot generate verification results in case it cannot derive the number of iterations in a loop.

4.1. Corresponding states

It is a pair of states $\langle q_{0,i}, q_{1,j}\rangle$ where $q_{0,i} \in Q_0$ and $q_{1,j} \in Q_1$. In the forward pass of the proposed method, there is a pair of corresponding states which get updated as the algorithm for finding the starting point of the region of the fault proceeds. In the backward pass of the proposed method, there is a pair of corresponding states which gets updated as the algorithm for finding the ending point of the region of the fault proceeds.

4.1.1. Correponding states in the forward pass

Let $q_{0,i}, q_{0,k} \in Q_0$ and $q_{1,j}, q_{1,l} \in Q_1$. Let the pair of corresponding states in the forward pass be denoted as cf_s, say.

1. Initially, the pair of start states of the flowcharts are cf_s.
2. Let $cf_s = \langle q_{0,i}, q_{1,j}\rangle$ and there exists path segments α from $q_{0,i}$ to $q_{0,k}$ and β from $q_{1,j}$ to $q_{1,l}$. Assume that with the addition of code segments corresponding to α and β in the product code and assuming the equivalence of the values of input variables, the equivalence of the output variables can be proved. Now $\langle q_{0,k}, q_{1,l}\rangle$ becomes the new cf_s.
3. Let $cf_s = \langle q_{0,i}, q_{1,j}\rangle$ and two nested blocks B with starting state $q_{0,i}$ and ending state $q_{0,k}$ and B' with starting state $q_{1,j}$ and ending state $q_{1,l}$. On addition of the code segments corresponding to B and B' to the product code and assuming equality of the values of the input variables, the equivalence of the output variables can be proved. Now $\langle q_{0,k}, q_{1,l}\rangle$ becomes the new cf_s.

4.1.2. Correponding states in the backward pass

Let $q'_{0,i}, q'_{0,k} \in Q_0$ and $q'_{1,j}, q'_{1,l} \in Q_1$. Let the pair of corresponding states in the backward pass is denoted as cb_s, say.

1. Initially, the pair of end states of the flowcharts are cb_s.
2. Let $cb_s = \langle q'_{0,i}, q'_{1,j}\rangle$ and there exists path segments α' from $q'_{0,k}$ to $q'_{0,i}$ and β' from $q'_{1,l}$ to $q'_{1,j}$. Assume that with the addition of code segments corresponding to α' and β' in the product code and assuming the equality of the input variables, the equivalence of the output variables can be proved. Now $\langle q'_{0,k}, q'_{1,l}\rangle$ becomes the new cb_s.
3. Let $cb_s = \langle q'_{0,i}, q'_{1,j}\rangle$ and two nested blocks B with starting state $q'_{0,k}$ and ending state $q'_{0,i}$ and B' with starting state $q'_{1,l}$ and ending state $q'_{0,i}$. On addition of the code segments corresponding to B and B' to the product code and assuming equality of the input variables, the equivalence of the output variables can be proved. Now $\langle q'_{0,k}, q'_{1,l}\rangle$ becomes the new cb_s.

Successor path segment/nested block*space*: A path segment/nested block $\langle PB_{s_f}, PB_{t_f}\rangle$ for a pair $\langle s_f, t_f\rangle$ of states, which are the start states of the flowcharts or the ending states of some path segments/nested blocks in the forward pass, is called successor path segment/nested block if the predecessor states of the starting states of PB_{s_f} and PB_{t_f} are s_f and t_f, respectively.

Predecessor path segment/nested block*space*: A path segment/nested block $\langle PB_{s_b}, PB_{t_b}\rangle$ for a pair $\langle s_b, t_b\rangle$ of states, which are end states of the flowcharts or starting states of some path segments/nested blocks in the backward pass, is called predecessor path segment/nested block if the successor states of the ending states of PB_{s_b} and PB_{t_b} are s_b and t_b, respectively.

If two path segments or two nested blocks are proved to be not equivalent by CBMC, then they are extended in the same flowchart. The meaning of extending a path segment or nested block is considering the subsequent path segment or nested block in the same flowchart along with the path segment or the nested block which has been found to be non-equivalent. In the forward pass, the non-equivalent path segments or nested blocks are extended in the forward direction to identify the starting point of fault localization. In the backward pass, the non-equivalent path segments or nested blocks are extended in the backward direction to identify the ending point of fault localization. If after path segment or nested block extension equivalence cannot be established, then the pass ends there declaring the starting point (for the forward pass) or the ending point (for the backward pass) of the region of fault. Note that the successor state of the ending state of a path segment or block not having any loop is a single state. The successor state of a nested block corresponding to a loop is not the control state of that loop, but the successor state of the joining state of the nested block. A path segment or nested block which emanates from the successor state is called a successor path segment or successor block, respectively. It can also be called the successor section. Similarly, the predecessor state of the starting state of a path segment or block not having any loop is a single state. The predecessor state of the nested block corresponding to a loop is not the joining state of the loop, but the predecessor state of the starting state of the loop. A path segment or nested block which ends at the predecessor state is called a predecessor path segment or predecessor block, respectively. It can also be called predecessor section. Here the terms successor and predecessor have the usual meaning from graph theory.

4.2. Adding assertions

After adding every pair of sections in the product code, an assertion is also added. Note that the input or output variables for a section can be different from the input and output variables of the entire program. Variables that already have some non-garbage values before executing the sections are assumed as input variables for the section and the variables which are assigned some values in the section are assumed as output variables. As a pair of sections is added, the assertion is needed to prove their equivalence. Initially, the equality of the values of the input variables of the programs is assumed. After the addition of every pair of sections or extended sections, the assertion aims to prove equality of the output variables of the sections.

5. Algorithm for localizing faults

There are two passes in the fault localization algorithm. The forward pass algorithm, $findStartingPoint$, finds the starting point of the region of the fault (Algorithm 1) and the backward pass algorithm, $findEndingPoint$, finds the ending point of the region of the fault (Algorithm 3). In both the passes, the algorithms generate CBMC intermediate files and analyze the results to identify starting and ending points of the region of the fault, respectively.

In every pass, the product program is constructed incrementally in a function and assertions are added to check equivalence of the added code segments of the two programs. Some variables are mentioned as input and output variables of the programs. The proposed method aims

to identify region of fault responsible for inequivalence of the output variables of the correct and faulty programs. Also, in every step, only the variables which have undergone modulo variable renaming are considered for equivalence.

In forward pass, the product code is constructed in lockstep fashion by taking a section of code from the beginning of the correct program and a section of code from the beginning of the faulty programs. Also assertions to prove equivalence of the output variables of product program are added. This process starts from the beginning of the flowcharts. If equivalence can be established, the successor sections from both the programs and corresponding assertions are also added. If equivalence cannot be established, then the last added assertions are removed and successor sections are added with the assertions to prove equivalence of the output variables of product program (Algorithm 2). If equivalence can be established, the process continues. If the equivalence cannot be established, then the starting states of just the previously added sections are considered as the starting states of the region of fault.

In backward pass, a section of the correct program and a section of the faulty program are added in the product program and assertions to prove equivalence of the output variables of the product program are added. This process starts from the end states of the flowcharts. If equivalence can be established, the predecessor sections are added and corresponding assertions to prove equivalence of the output variables are also added (Algorithm 3). If equivalence cannot be established, then the last added assertions are removed and predecessor sections are added with the assertions to prove equivalence of the output variables of the product program. If equivalence can be established, the process continues. If the equivalence cannot be established, then the ending states of just the previously added sections are considered as the ending states of the region of fault. Note that path segments are referred to as paths in the algorithms for brevity.

6. The example — revisited

In this section, the key idea of localizing faults is explained with the example given in Fig. 1. The example considers the original programs instead of the static slices with respect to the output variables. Note that before proving equivalence of the presented functions $Non\_Crossing\_Biased\_Climb()$ and $Non\_Crossing\_Biased\_Climb'()$, all the other functions on which they depend e.g., $Inhibit\_Biased\_Climb()$, $Own\_Below\_Threat()$, $Own\_Above\_Threat()$, $ALIM()$ and their modulo renamed versions are validated using CBMC model checker. Once these functions are proved to be equivalent with their modulo renamed versions, the fault can be localized for the present function.

Finally, the forward pass and the backward passes are conducted to identify the starting and ending point of the region where the fault is present. Initially, the starting point of the region of the fault is set to $\langle q_{00}, q'_{00}\rangle$ which are the start states of the flowcharts and the first pair of corresponding states of the forward pass. A new method *product* is created in a new C file to validate the nested block or path segment pairs that emanate from the corresponding states. Hence the code segments corresponding to the path segments α_1 and α'_1 and the assert statement for validating the path segments are added there. As discussed in Section 4.2, while validating the path segments α and α', $Down\_Separation'$, $Down\_Separation'$ are considered as input variables. Also, the functions $Inhibit\_Biased\_Climb()$ and $Inhibit\_Biased\_Climb'()$ are considered uninterpreted functions and they are assumed to be equivalent; and $upward\_preferred$ and $upward\_preferred'$ are considered as output variables. The call to the functions which are already proven to be equivalent e.g., $Inhibit\_Biased\_Climb()$ and $Inhibit\_Biased\_Climb'()$, etc are assumed as different instances of the same function in the product code.

As verification of α and α' succeeds, the code segments and the assertions corresponding to the successor nested blocks B and B' are appended to the method *product* and the corresponding states of the forward pass is updated to $\langle q_{02}, q'_{02}\rangle$. Also, the starting point of the region of the fault is updated to the successor states ($\langle q_{02}, q'_{02}\rangle$) of the last added corresponding states as the path segments α_1 and α'_1 are proved to be equivalent in the forward pass and the fault cannot be present there. Next the code segments corresponding to the nested blocks B and B' and the assert statement for validating them are appended in the method *product*. Now the verification process does not succeed. Hence the nested blocks B and B' present in both the programs are extended with path segments α_2 and α'_2, respectively. Subsequently, the last added assert statements for validating B and B' are removed and the code segments corresponding to the path segment α_2 and α'_2 are to be added. As in path segments α_2 and α'_2 no expression is evaluated, nothing is added. However, the assert statement to prove equality of

Algorithm 1 *findStartingPoint*

Input The flowcharts F_0 and F_1, with the number of states n_s and n_t, respectively

Output A pair of corresponding states which is the $starting\_point = \langle s_0, s_1\rangle$ of the forward pass from the flowcharts F_0 and F_1, respectively

1: Let the pair $\langle cf_0, cf_1\rangle$ denote the corresponding states in the forward pass and $\langle s_0, s_1\rangle$ denote the successor states of the corresponding states. Initialize cf_0, cf_1, s_0 and s_1 with 0 and the variable buf to empty string
2: If $(s_0 == n_s - 1)||(s_1 == n_t - 1)$, then return $\langle s_0, s_1\rangle$. Else, go to Step 3
3: Identify the nested block pair $\langle B_0, B_1\rangle$ in the source and the transformed programs which start with the state s_0 and s_1, respectively. Set the ordered pair $PB = \langle B_0, B_1\rangle$. If no such block is found, go to Step 4. Else go to Step 5.
4: Identify the paths $\langle p_0, p_1\rangle$ in the source and the transformed programs which start with the state s_0 and s_1, respectively. Set the ordered pair $PB = \langle p_0, p_1\rangle$.
5: Denote the two components of PB to be PB_0 and PB_1. Construct the string representation of the function definition line based on the variables modified in PB_0 and PB_1 and the paths/nested blocks the string representation of which are already present in the C file
6: Write string representation of the members PB_0 and PB_1 sequentially in the C file. Also store the string representation in the variable buf i.e., $buf \leftarrow buf \circ PB_0 \circ PB_1$.
7: Write the assert statements V_f for validating the members of the PB in the file.
8: Check equivalence using the C file. If equivalence can be established, go to Step 9. Else, go to Step 10.
9: Add the assert statements in the variable buf i.e., $buf \leftarrow buf \circ V_f$. Also update the corresponding states and the successor states i.e., $cf_0 \leftarrow endState(PB_0)$, $cf_1 \leftarrow endState(PB_1)$, $s_0 \leftarrow nextState(endState(PB_0))$, $s_1 \leftarrow nextState(endState(PB_1))$ and go to Step 2.
10: Call the function for path extension $\langle n, \langle l_0, l_1\rangle, buf\ \rangle \leftarrow forwardPathExtension(F_0, F_1, buf, PB)$. If $n == verification\_failure$ or $extension\_failure$, then go to Step 11. Else, go to Step 12.
11: Declare the state pair $\langle s_0, s_1\rangle$ as the starting point of the region of fault and return
12: Update the corresponding states and the successor states i.e., $cf_0 \leftarrow l_0$, $cf_1 \leftarrow l_1$, $s_0 \leftarrow successorState(cf_0)$, $s_1 \leftarrow successorState(cf_1)$ and go to Step 2

Algorithm 2 *forwardPathExtension*

Input The flowcharts F_0, F_1 with the number of states n_s and n_t, respectively, the variable buf which contains the string representation and the assertions of all the paths/nested blocks which are proved to be equivalent, the pair of path or nested block $\langle pb_0, pb_1\rangle$ to be extended

Output A pair $\langle n, \langle l_0, l_1\rangle, buf\ \rangle$, where n is an enumerator which takes values from the set {verification_success, verification_failure, extension_success, extension_failure} and the pair $\langle l_0, l_1\rangle$ is the end points of the resulting extended path/nested blocks of the flowcharts F_0 and F_1, respectively. The updated variable buf which contains all path/nested block pairs and the assertions after path extension.

1: Let sp_0 and sp_1 be the starting point of the path/nested block with which pb_0 and pb_1, respectively are to be extended. Set $sp_0 \leftarrow successortState(endState(pb_0))$ $sp_1 \leftarrow successorState(endState(pb_1))$. If $sp_0 == n_s - 1$ or $sp_1 == n_t - 1$, return $\langle extension\_failure, \langle -1, -1\rangle\rangle$. Else, go to Step 2
2: Write the content of the variable buf in a C file
3: Identify the nested block pairs $\langle B_0, B_1\rangle$ which starts with sp_0 and sp_1, respectively and set $PB = \langle B_0, B_1\rangle$. If no such nested blocks are found, go to Step 4. Else, go to Step 5.
4: Identify the path pairs $\langle p_0, p_1\rangle$ which starts with sp_0 and sp_1, respectively. Set $PB = \langle p_0, p_1\rangle$
5: Assume the two components of PB as PB_0 and PB_1. Add the string representation of PB_0 and PB_1 to the variable buf and also in the file $buf \leftarrow buf \circ PB_0 \circ PB_1$
6: Extend pb_0 with PB_0 and pb_1 with PB_1
7: Write the assertions V_e in the file
8: Check equivalence using the C file. If equivalence can be established, go to Step 9. Else, go to Step 10
9: Update the variable buf with V_e i.e., $buf \leftarrow buf \circ V_e$. Return $\langle verification\_success, \langle endState(PB_0), endState(PB_1)\rangle, buf\ \rangle$
10: Return $\langle verification\_failure, \langle -1, -1\rangle, buf\ \rangle$

output variables *result* and *result′* is added. The verification process also does not succeed now. The resulting file is shown below.

```
int product()
{
   // variable and function declarations are omitted for brevity
   __CPROVER_assume(Up_Separation == Up_Separation1);
   __CPROVER_assume(Down_Separation == Down_Separation1);
   __CPROVER_assume(Cur_Vertical_Sep == Cur_Vertical_Sep1);
   __CPROVER_assume(MINSEP == MINSEP1);


   upward_preferred = __CPROVER_uninterpreted_Inhibit_Biased_
     Climb() > Down_Separation;
   upward_preferred1 = __CPROVER_uninterpreted_Inhibit_Biased_
     Climb1() > Down_Separation1;
   __CPROVER_assert(upward_preferred == upward_preferred1,
    "equality of upward_preferred");
   if (upward_preferred)
   {
      result = !(__CPROVER_uninterpreted_Own_Below_Threat()) ||
      ((__CPROVER_uninterpreted_Own_Below_Threat()) &&
      (!(Down_Separation >= __CPROVER_uninterpreted_ALIM())));
   }
   else
   {
      result = __CPROVER_uninterpreted_Own_Above_Threat() &&
      (Cur_Vertical_Sep >= MINSEP) &&
      (Up_Separation >= __CPROVER_uninterpreted_ALIM());
   }

   if (upward_preferred1)
   {
      result1 = !(__CPROVER_uninterpreted_Own_Below_Threat()) ||
      ((__CPROVER_uninterpreted_Own_Below_Threat()) &&
      (!(Down_Separation1 > __CPROVER_uninterpreted_ALIM())));
   }
   else
   {
      result1 = __CPROVER_uninterpreted_Own_Above_Threat() &&
      (Cur_Vertical_Sep1 >= MINSEP1) &&
      (Up_Separation1 >= __CPROVER_uninterpreted_ALIM());
   }

   __CPROVER_assert(result == result1, "equality");
}
```

As the process of verification does not succeed even after the extension process, the starting point of the region of the fault is not updated anymore and $\langle q_{02}, q'_{02}\rangle$ is denoted as the starting point of the region of the fault.

Now the backward pass is started to identify the ending point of the region of the fault. As the first step, the end states of the flowcharts $\langle q_{07}, q'_{07}\rangle$ are set as corresponding states of the backward pass. A new C method *product′* is created in a new C file to validate the nested block or path segment pairs that end at the corresponding states. As no code segment corresponding to the path segment pairs $\langle \alpha_2, \alpha'_2\rangle$ is present, the corresponding state is updated to $\langle q_{06}, q'_{06}\rangle$. Next the code segments corresponding to the nested blocks are added along with the assert statement to the method *product′*. Note that the variables *upward_preferred*, *Cur_Vertical_Sep*, *MINSEP*, *Up_Separation* and their modulo renamed versions are considered as the input vari-

Algorithm 3 *findEndingPoint*

Input: The flowcharts F_0 and F_1 with the number of states n_s and n_t, respectively

Output: A pair of corresponding states $ending\_point = \langle e_0, e_1\rangle$ from the flowcharts F_0 and F_1, respectively

1: Let the pair $\langle cb_0, cb_1\rangle$ denote the corresponding states in the backward pass and $\langle e_0, e_1\rangle$ denote the predecessor states of $\langle cb_0, cb_1\rangle$. Initialize cb_0 and e_0 with $n_s - 1$ and cb_1 and e_1 with $n_t - 1$ and initialize the variable buf to empty string
2: If $e_0 = 0$ or $e_1 = 0$, then declare $\langle e_0, e_1\rangle$ to be the ending point and return. Else go to Step 3
3: Identify the nested block pair $\langle B_0, B_1\rangle$ which ends with e_0 and e_1, respectively. and set $PB = \langle B_0, B_1\rangle$. If no such nested blocks are found go to Step 4. Else, go to Step 5
4: Identify the path pair $\langle p_0, p_1\rangle$ which ends with e_0 and e_1, respectively and set $PB = \langle p_0, p_1\rangle$
5: Denote the two components of PB to be PB_0 and PB_1. Construct the string representation of the function definition line based on the variable modified in PB_0 and PB_1 and the paths/nested blocks for which string representations are already present in the variable buf
6: Write the string representations of PB_0 followed by PB_1 in the C file
7: Write the assert statement V_b for validating PB_0 and PB_1 in the output C file
8: Write the content of the variable buf in the C file
9: Check equivalence using the C file. If equivalence can be established, go to Step 10. Else, go to Step 11
10: Prepend the string representation of PB_0 and PB_1 followed by the assertion V_b to the variable buf i.e., $buf \leftarrow PB_0 \circ PB_1 \circ V_b \circ buf$. Go to Step 13.
11: Call the function for backward path extension i.e., $\langle n, \langle l_0, l_1\rangle\rangle \leftarrow backwardPathExtension(F_0, F_1, buf, PB)$. If $n == verification\_failure$ or $extension\_failure$, then go to Step 12. Else, go to Step 13
12: Declare the state pair $\langle e_0, e_1\rangle$ as the ending point of fault localization and return
13: Update corresponding states and predecessor states. Store cb_0 to l_0 and cb_1 to l_1. Set $e_0 \leftarrow predecessorState(cb_0)$ and $e_1 \leftarrow predecessorState(cb_1)$. Go to Step 2

Algorithm 4 *backwardPathExtension*

Input: The flowcharts F_0, F_1, the variable buf, the pair of not equivalent path or nested block $\langle PB_0, PB_1\rangle$ which have to be backward extended

Output: A pair $\langle n, \langle l_0, l_1\rangle\rangle$, where n is an enumerator which takes values from the set {verification_success, verification_failure, extension_success, extension_failure} and the pair $\langle l_0, l_1\rangle$ is the start points of the resulting extended path/blocks of the flowcharts F_0 and F_1, respectively.

1: Let sp_0 and sp_1 be the starting point of PB_0 and PB_1, respectively. Set $sp_0 \leftarrow startState\ (PB_0)$ and $sp_1 \leftarrow startState\ (PB_1)$. Go to Step 2
2: Identify predecessor nested blocks $\langle B_0, B_1\rangle$ in the source and the transformed programs of the pair $\langle sp_0, sp_1\rangle$ and store $\langle B_0, B_1\rangle$ to $prev\_PB$ i.e., $prev\_PB \leftarrow \langle B_0, B_1\rangle$. If no such nested blocks are found, go to Step 3
3: Identify predecessor paths $\langle p_0, p_1\rangle$ in the source and the transformed programs of $\langle sp_0, sp_1\rangle$ and store $\langle p_0, p_1\rangle$ to $prev\_PB$ i.e., $prev\_PB \leftarrow \langle p_0, p_1\rangle$
4: Consider variables modified in $prev\_PB_0$, $prev\_PB_1$, PB_0, PB_1 and the path/nested blocks the string representation of which are present in the buffer and write the function definition line in a C file
5: Write the string representation of $prev\_PB_0$, PB_0, $prev\_PB_1$, PB_1 in the C file
6: Forward extend $prev\_PB_0$ with PB_0 and $prev\_PB_1$ with PB_1
7: Construct the assertion V_e for the extended path $prev\_PB_0$ with PB_0 against $prev\_PB_1$ with PB_1. Write the assertion in the file
8: Write the content of the variable buf in the C file
9: Check equivalence using CBMC. If equivalence can be established, go to Step 10. Else, go to Step 11
10: Add the string representation of $prev\_PB_0$, PB_0, $prev\_PB_1$, PB_1, V_e in the variable buf i.e., $buf \leftarrow prev\_PB_0 \circ PB_0 \circ prev\_PB_1 \circ PB_1 \circ V_e \circ buf$. Return $\langle verification\_success, \langle startState(prev\_PB_0), startState(prev\_PB_1)\rangle\rangle$
11: Add the string representation of $prev\_PB_0$, PB_0, $prev\_PB_1$, PB_1 in the variable buf i.e., $buf \leftarrow prev\_PB_0 \circ PB_0 \circ prev\_PB_1 \circ PB_1 \circ buf$. Return $\langle verification\_failure, \langle\ startState\ (prev\_PB_0), startState(prev\_PB_1)\rangle\rangle$.

ables and *result* and its modulo renamed version are considered as the output variables.

```
int product'()
{
   // variable and function declarations are omitted for brevity
    __CPROVER_assume(upward_preferred == upward_preferred1);
   __CPROVER_assume(Cur_Vertical_Sep == Cur_Vertical_Sep1);
   __CPROVER_assume(MINSEP == MINSEP1);
   __CPROVER_assume(Up_Separation == Up_Separation1);

   if (upward_preferred)
   {
      result = !(__CPROVER_uninterpreted_Own_Below_Threat()) ||
      ((__CPROVER_uninterpreted_Own_Below_Threat()) &&
      (!(Down_Separation > __CPROVER_uninterpreted_ALIM())));
   }
   else
   {
      result = __CPROVER_uninterpreted_Own_Above_Threat() &&
      (Cur_Vertical_Sep >= MINSEP) &&
      (Up_Separation >= __CPROVER_uninterpreted_ALIM());
   }

   if (upward_preferred1)
   {
      result1 = !(__CPROVER_uninterpreted_Own_Below_Threat()) ||
      ((__CPROVER_uninterpreted_Own_Below_Threat()) &&
      (!(Down_Separation1 >= __CPROVER_uninterpreted_ALIM())));
   }
   else
   {
      result1 = __CPROVER_uninterpreted_Own_Above_Threat() &&
      (Cur_Vertical_Sep1 >= MINSEP1) &&
      (Up_Separation1 >= __CPROVER_uninterpreted_ALIM());
   }

   __CPROVER_assert(result == result1, "equality");
}
```

As verification does not succeed, the corresponding states of the backward pass are not updated and the assert statement for validating the nested blocks B and B' is removed from the method $product'$. The nested blocks B and B' are backward extended with the path segments α_1 and α'_1, respectively. The code segments for α_1 and α'_1 are prepended in the method $product'$ before the code segments of the nested blocks; also, the assert statement for validating α_1 and B with α'_1 and B' are added after the code segments of the nested blocks. As the verification process again does not succeed, the last added corresponding states for the backward pass $\langle q_{06}, q'_{06}\rangle$ is referred to as the ending point for the faulty region. Hence the fault is localized in the region starting with the state pair $\langle q_{02}, q'_{02}\rangle$ and with the state pair $\langle q_{06}, q'_{06}\rangle$.

7. Experimental set up

The implementation has been checked in a machine with Intel $Core^{TM}$ i5 with 1.60 GHz × 8 CPU, 8 GB RAM and 64-bit operating system.

7.1. Research questions

In this study, the following research question is investigated.

- RQ: Formal verification problem is undecidable. What advantage does the proposed method have although it uses verification techniques?

7.2. Benchmark

The experiments are done on TCAS benchmarks of Siemens suite (Hutchins et al., 1994), NTS benchmark suite and 7 other test cases taken from the literature. The TCAS task of the Siemens test suite constitutes an aircraft collision avoidance system. To check the effectiveness of the fault localization tool, the authors of the suite created 41 versions of the program. The test cases are referred to as the versions by "v1" to "v41" and each was created by adding one or more faults, usually a change in a single line. In NTS, there are ten different programs. `adpcm` is an adaptive differential pulse code modulation program and `cfg_test` is a simple program taken from benchmarks of the CREST tool. The `elevator` program simulates elevator stop optimization via dynamic programming and `Merge2BST` combines two binary search trees with limited extra space. `nextDate` computes the day after adding a given number of days to a given date and `nsichneu` simulates an extended petri net. `Problem1`, `Problem2` and `Problem3` are taken from RERS-2018 challenge. `quick_sort` is a quick sort program for sorting integer values. The implementation has also been checked against the 7 test cases referred to in Karfa et al. (2008). The correct program and the corresponding compiler-transformed programs are taken from the literature and errors are manually seeded in the transformed programs.

Table 1
TCAS testcases and the comparison in terms of time, score with `cfault` (Griesmayer et al., 2007) and `BugAssist` (Jose and Majumdar: Cause Clue Clauses, 2011) tools.

TCAS testcase	T (s)	S	T_{cfault}	S_{cfault}	$T_{BugAssist}$	$S_{BugAssist}$	Advantage over static slicing
v1	0.017	0.910	2953	0.906	0.016	0.086	Yes
v2	0.238	0.946	836	0.975	0.068	0.046	No
v3	0.255	0.357	423	0.956	0.068	0.098	Yes
v4	0.235	0.910	423	0.956	0.096	0.092	Yes
v5	0.230	0.357	159	0.956	0.120	0.086	Yes
v6	0.018	0.982	253	0.919	0.108	0.086	No
v7	0.017	0.929	743	0.975	0.072	0.092	No
v8	0.242	0.929	26	0.886	0.112	0.086	No
v9	0.235	0.893	114	0.944	0.092	0.052	Yes
v10	0.012	0.982	269	0.925	0.136	0.092	No
v11	0.010	0.982	162	0.969	0.080	0.063	Yes
v12	0.012	0.357	1664	0.956	0.164	0.092	Yes
v13	0.013	0.357	149		0.080	0.092	Yes
v14	0.013	0.357	594		0.028	0.081	Yes
v15	0.013	0.929	283	0.956	0.104	0.075	Yes
v16	0.012	0.929	1263	0.906	0.104	0.092	No
v17	0.011	0.929	1300	0.975	0.096	0.092	No
v18	0.012	0.929	499	0.975	0.124	0.069	No
v19	0.011	0.929	691	0.975	0.112	0.092	No
v20	0.251	0.893	748	0.906	0.120	0.092	Yes
v21	0.250	0.893	585	0.906	0.108	0.086	Yes
v22	0.255	0.893	223	0.950	0.056	0.057	Yes
v23	0.250	0.893	885	0.944	0.100	0.063	Yes
v24	0.251	0.893	254	0.906	0.092	0.086	Yes
v25	0.252	0.893	68	0.950	0.068	0.069	Yes
v26	0.252	0.357	311	0.950	0.108	0.092	Yes
v27	0.251	0.357	153	0.956	0.108	0.109	Yes
v28	0.250	0.946	648	0.988	0.080	0.057	No
v29	0.252	0.946	224	0.981	0.092	0.057	No
v30	0.251	0.946	939	0.975	0.064	0.057	No
v31	0.250	0.911	449	0.913	0.008	0.109	Yes
v32	0.015	0.893	39	0.925	0.004	0.109	Yes
v33	0.015	0.929	892	0.369	–	–	No
v34	0.012	0.357	1906	0.956	0.100	0.086	Yes
v35	0.236	0.946	1069	0.988	0.060	0.057	No
v36	0.011	0.357	877	–	0.024	0.029	Yes
v37	0.012	0.982	822	0.969	0.040	0.086	No
v38	0.013	0.878	–	–	–	–	No
v39	0.236	0.893	66	0.950	0.088	0.069	Yes
v40	0.235	0.911	3017	0.944	0.088	0.063	Yes
v41	0.234	0.911	956	0.906	0.120	0.086	Yes

7.3. Results

Each test case has undergone the following changes before subjecting it to the proposed method. Every test case corresponds to two versions — one is the original test case, *orig*, say, and the other is seeded with faults, *seeded*, say. Also, the variables and the methods undergo minor name changes in the seeded programs e.g., the function `alt_sep_test` in v1 is changed to `alt_sep_test'` and `Climb_Inhibit` is changed to `Climb_Inhibit'`, etc. Some of the methods in the test cases directly return expressions. For these methods, an output variable is declared which is assigned the return expression. This is required to check their values in the assert statements.

In the next step, the methods are validated pairwise to pinpoint the fault in the method level. For example, the method *Non_Crossing_Biased_Descend* is validated with a similar method in the *seeded* program. Hence, to validate a test case from TCAS or a test case from NTS, first the call graph is constructed. Note that the call graph is required for finding the sequence in which the functions present in each of the test cases are validated. First, the method which does not call any other method is validated. If the validation is successful, then the methods which call the former methods are validated. The call to the former method in the latter method is replaced with uninterpreted functions in both the programs. If the validation is unsuccessful, the fault localization method outputs the region of the fault of the former method and terminates. If multiple faults are present for a benchmark program, then the fault is localized only for the pair of procedures

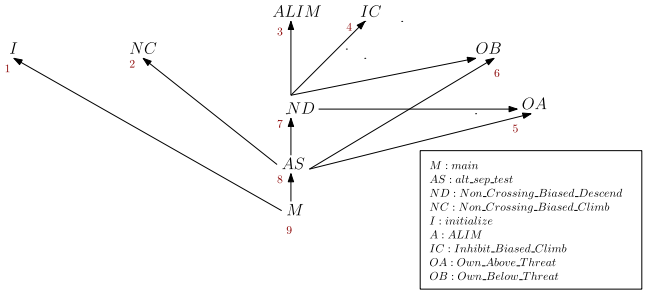

Fig. 4. The call graph and the sequence in which the functions are validated for "v1" of TCAS benchmarks.

where the fault is encountered for the first time and then the method exits. In this way, all the methods for a test case are validated.

The TCAS test cases can be categorized into 3 groups after doing the above changes. The first group contains 28 test cases where each of the transformed program is the result of mutation in a logical or relational operator. For all these test cases, the proposed method can indicate the location of the fault. The second group contains all the test cases for which the error is seeded in the *initialize* method. It is to be noted that the method assigns constant values in the elements of the global *Positive_RA_Alt_Thresh* array. Unlike in Dutta (2022), the proposed method can to localize fault for the transformed program for the test case v38 as CBMC can to handle arrays efficiently. The third

group has 4 test cases wherein all the cases, the initial values assigned to the global variables are modified. For all the test cases in this group, the proposed method can localize respective faults.

In Table 1, the running time (T) and the score (S) due to Renieres and Reiss (2003) of the implementation are compared with that of two well-cited methods of Griesmayer et al. (2007) and Jose and Majumdar: Cause Clue Clauses (2011) of fault-localization. The tool of Jose and Majumdar: Cause Clue Clauses (2011) is referred to as `BugAssist` tool in the literature and the tool of Griesmayer et al. (2007) is referred to as `cfault` tool for ease of reference. The first, second, third, fourth, fifth, sixth, seventh columns of Table 1 respectively represents the benchmark programs, time and score for the proposed method, time and score for the method of the (Griesmayer et al., 2007) and time and score for the method of Jose and Majumdar: Cause Clue Clauses (2011), respectively. The last column indicates whether the proposed method really reduces the search space of fault over static slicing for a specific test case. The time is considered by executing the "time" command of gcc and taking note of the system time in seconds. The score (Renieres and Reiss, 2003) is based on a program dependence graph (PDG) and gives the estimate of the code that can be ignored when looking for the actual fault. More precisely, if a method reports the fault search space R of length $|R|$ and if the size of the program dependence graph PDG is $|PDG|$, then the score is computed as $1 - \frac{|R|}{|PDG|}$. A higher score indicates the better performance of the method. The scores are good for all the benchmark programs except for those where the faults are present in the *alt_sep_test* function. The reason is the presence of faults in a nested block with majority of the code contents of the function. In the `cfault` tool, the authors have measured the score for TCAS program using the method of Renieres and Reiss (2003). The authors of BugAssist tool of Jose and Majumdar: Cause Clue Clauses (2011) have measured the score using "SizeReduc%" metric which is the percentage reduction in the code size given by their tool to locate the bug. We have calculated the score of Renieres and Reiss (2003) from the "SizeReduc%" metric. Fig. 4 represents the call graph of functions present in the TCAS benchmark program. A directed edge from the function A to B represents that the function A calls the function B. As the function *alt_sep_test* is dependent on many functions (directed edges are present from the *alt_sep_test* to many other functions which are also dependent on many other functions), the proposed method has to check the codes for all the functions on which it depends for locating the fault before validating the function. There are few benchmark programs e.g., V10, V11, V15, V31, V32, V40 where multiple errors are present. The proposed method identifies the first error and exits for these benchmark programs. Hence the score is high although some errors are present in the functions which are dependent on many other functions.

The table also provides the time and score information of the `cfault` tool and `the BugAssist` tool. Note that the time taken by the method of the `cfault` tool is much more compared to the other two methods in all the cases. Griesmayer et al. (2007) Also, the scores of the proposed method are more than in 9 situations. The proposed method takes less time than the method of the `BugAssist` tool in 15 situations and the score is much more than the `BugAssist` tool for all the situations which indicate that the proposed method really reduces the fault search space as compared to the `BugAssist` tool. Also, the proposed method could indicate the region of the fault for all the benchmark programs whereas the other two methods failed for some of the benchmark programs.

The implementation is also tested on NTS benchmark suite. The result of this experiment is shown in Table 2. In the first and second columns, the NTS benchmark and the number of faulty versions of the benchmark are listed. In the third and fourth columns, lines of code (LOC) and the number of functions present in the benchmark program are listed. In the fifth and sixth columns, the average time required to localize faults and the average score due to Renieres and Reiss (2003) are listed. In the seventh column, the average time due to a reported method HieDeep (Dutta et al., 2020) based on a deep neural network is reported. As the average amount of code executed per benchmark is presented graphically which lacks precise information, the exact score due to Renieres and Reiss (2003) could not be computed for HieDeep method. The result shows that the proposed method identifies the fault in less time for all the benchmark programs with large number of lines of codes are present. This indicates that the proposed method is scalable. For 75% of faulty versions, HieDeep requires at least 20.33% of code examination, whereas the proposed method requires 55% code examination. The score is 0 for the benchmark program `nsichneu` as the entire program of the correct and the faulty codes are considered for localizing the fault. Also, the scores are less for the benchmarks `merge2BSTree` and `Problem2` as almost the entire code segments of the correct and the faulty codes are considered for localizing the fault. Note that for one mutant of the red `quicksort`, two variants of `Problem2`, ten variants of `cfg_test` benchmark programs, the proposed method has failed to indicate the region of faults as the variants are not actually inequivalent to the correct programs. These variants are generated by changing some status messages. Some variants of `Problem2` and `nsichneu` benchmark programs have more than one error. The proposed method has only identified the first encountered error and the time and the score have been calculated accordingly. The time and the score of Table 2 correspond to only the mutants for which errors are localized. We have encoded the called functions from a function as uninterpreted functions and considered the called functions from the correct and faulty programs to be two instances of the same functions once they are proved equivalent. However, in the example of Section 6, the invocations from the correct and the faulty programs are considered different functions.

The implementation has also been checked against the 7 test cases referred to in Karfa et al. (2008). The result of running the test cases is shown in Table 3. The first 2 test cases of Table 3 are similar to the one shown in Fig. 1. The last 5 test cases correspond to the test cases given in Figs. 4, 5, 7, 8, 9, respectively of the Karfa et al. (2008). For test cases 3 and 4, common sub-expression elimination transformation is applied on the source programs and errors are seeded manually by changing a single line in the nested block. For test cases 5, 6 and 7, duplicating-down, duplicating-up, boosting-up code motions are applied, respectively. All these transformations preserved the structure of the source codes. The proposed method successfully identified the region of the fault for all the test cases. The test cases of Table 3 are very small (LOCs around 10 lines) without any function call. As the test cases are very small, the reported region of fault is significant with respect to the total lines of code and hence the scores are small. The proposed method shows a reduction of fault search space for 25 TCAS test cases and for all test cases shown in Table 3 if the method is applied over static slices of the output variables for every function.

7.4. RQ: Advantage of the method

The proposed method uses verification techniques repeatedly to locate the region of the fault. Although formal verification is undecidable, the method is applicable for static model-based fault diagnosis when assertions for behavioral specifications are not present. The experimental result shows that the method is scalable when the size of the code is large. For NTS benchmark suite, the competing method HieDeep examines less code to localize the fault. However, the method requires test cases to train the deep neural network. Also, the proposed method takes less time to localize the region of fault. The principal advantage of the proposed method lies in the fact that it is static method that requires no test case.

Table 2
Comparative result of NTS testcases for proposed method versus HieDeep method.

NTS testcase	Number of faulty versions	LOC	Number of function	Time (s)	S	HieDeep time (s)
adpcm	8	916	17	0.136	0.650	0.033
cfg_test	25	93	5	0.425	0.763	0.043
elevator	8	156	6	0.136	0.915	0.024
merge2BSTree	8	226	7	0.136	0.186	0.035
nextDate1	14	204	6	0.238	0.606	0.013
nsichneu	12	4266	2	0.204	0	7.220
Problem1	24	431	22	0.408	0.543	12.460
Problem2	10	680	31	0.170	0.341	17.170
Problem3	5	3878	84	0.085	0.594	36
quicksort	6	99	7	0.094	0.670	12.250

Table 3
Testcases and the time taken by the proposed method.

Serial number	Name	Time taken (seconds)	Score
1	wrong_safety_check1	0.017	0.250
2	wrong_safety_check2	0.017	0.250
3	common_subex_elim1	0.019	0.250
4	common_subex_elim2	0.017	0.220
5	dup_down_code_motion	0.015	0.272
6	dup_up_code_motion	0.013	0.250
7	boosting_up_code_motion	0.013	0.200

```
int my_sin{                          int my_sin'{
  int diff;                            int diff';
  int app = 0;                         int app' = 0;
  int inc = 1;                         int inc' = 1;
  while rad > 2*PI                     while rad' > 2*PI'
    rad -= 2*PI;                         rad' -= 2*PI';
  while rad < -2*PI                    while rad' < 2*PI'
    rad += 2*PI;                         rad' += 2*PI';
  diff = rad;                          diff' = rad';
  app = diff;                          app' = diff';
  diff = (diff *(-(rad*rad)))/         diff' = (diff' *(-(rad'*rad')))/
         ((2*inc)*(2*inc+1));                 ((2*inc')*(2*inc'+1));
  app = app + diff;                    app' = app' + diff';
  inc++;                               inc'++;
  while( my_fabs( diff) >= 1{          while( my_fabs'(diff') >= 1{
    diff = (diff *(-(rad*rad)))/         diff' = (diff' *(-(rad'*rad')))/
           ((2*inc)*(2*inc+1));                 ((2*inc')*(2*inc'+1));
     app = app + diff;                    app' = app' + diff';
     inc++;                               inc'++;
   }                                    }
   return app;                          return app';
}                                    }
     (a)                                  (b)
```

Fig. 5. The correct (a) and the faulty (b) programs corresponding to the v_1 of adpcm benchmark of NTS benchmark suite.

8. Handling loops

Consider the correct and faulty program segments shown in Fig. 5 which correspond to the *adpcm* programs (v_1) of NTS benchmark suite. The source program (5(a)) and the program corresponding to v_1 (5(b)) are almost equivalent; only the function *my_sin* and *my_sin'* are not equivalent due to the difference in the conditions of the second *while*-loop. In the forward pass, the sections corresponding to the code segments before the first loop pairs are checked for equivalence. As the sections are found equivalent, the first *while*-loop pairs for both the programs are checked for equivalence. However, CBMC cannot prove equivalence for the loops as the bounds cannot be derived. Hence, the unwinding bound is specified as 1 and the product programs after the addition of the corresponding sections is proved equivalent. After proving the equivalence of the first *while*-loop pairs, the second *while*-loop pairs for both the programs are checked for equivalence. The unwinding bound also applied to them. However, these sections cannot be proved equivalent as conditions of the loops are different; they could not be proved equivalent even after extension. Hence, the top arrows indicate the starting points of the region of fault.

Similarly, for the backward pass, the last loop pairs from both programs are checked for equivalence. Note that the methods my_fabs and my_fabs' are assumed as uninterpreted functions. After establishing the equivalence of these two methods, we replace the call to the method my_fabs' in the product code with the call to the method my_fab with the argument $diff$ as discussed in Section 6. The bounded model-checker CBMC proves the equivalence of these two calls using the theory of Equality Logic and Uninterpreted functions. However, as CBMC is unable to derive the bounds of the loops, the unwinding bound is specified as 1 and the product programs after the addition of the corresponding sections are proved equivalent. The immediately preceding pair of sections are proved equivalent and they are added to the product code. Next, the *while*-loop pairs (marked in red color) from both the programs are added and checked for the equivalence in the product code. These loops cannot be proved equivalent as the conditions of the loops are different. They could not be proved equivalent even after extension of nested blocks in the backward direction. Hence, the bottom arrows indicate the ending points of the region of fault.

9. Related work

Automated fault diagnosis has always been an important research area. There are many research works on the domain of Model-based diagnosis (Baah et al., 2010; Friedrich et al., 1999; Cristinel et al., 2000; Mayer et al., 2008; Mayer and Stumptner, 2007, 2003; Mayer et al., 2002). Some of the research works mentioned above contributes to fault localization by considering test cases (Baah et al., 2010; Friedrich et al., 1999; Mayer et al., 2008; Mayer and Stumptner, 2007, 2003). Differences between the observed program executions and the expected results (provided by programmers or testers) are used to identify model elements that are responsible for such observed abnormal behaviors. Some of these research works (Cristinel et al., 2000; Mayer and Stumptner, 2003; Mayer et al., 2002) concentrate on localizing faults for object-oriented language e.g., Java.

There are model checking-based fault localization techniques that rely on the use of model checkers to locate faults e.g., Ball et al. (2003), Chaki et al. (2004), Griesmayer et al. (2007), Grismayer et al. (2010), Groce and Visser (2003), Groce et al. (2006), Groce et al. (2004), Griesmayer et al. (2007) and Jose and Majumdar: Cause Clue Clauses (2011). All the above works are focused to analyze the error traces or the counter-examples to identify the source of the fault. In Ball et al. (2003), the authors use a model checker to explore all path segments present in a program except that of the counter-example. They identify successful execution path segments which do not cause a failure. An algorithm is proposed to identify the traces that appear in the execution path segment of the counter-example but not in any successful execution path segments. Program components related to these traces are those that are likely to contain the causes of bugs. However, the enumeration of all the traces of a program can be very expensive. In Chaki et al. (2004), the authors use the counter-example produced by the model-checker to identify the closest traces using some distance metric. The authors of Griesmayer et al. (2007), Grismayer et al. (2010) argue that a successful execution path segment can be different from

the path segment of the counter-example. Instead of searching for successful execution path segment with small changes from that of the original counter-example, they make minimal changes to the program so that the counter-example will not fail in the revised program. The component of the program that is changed for the counter-example is the potential source of the fault. The approach is automatic and it is implemented for C programs. The authors of Groce and Visser (2003) use an automated method for finding multiple versions of an error and similar executions that do not produce an error. They analyze these executions to identify portions of the source code crucial to distinguishing failing and succeeding runs. All the above-mentioned works generate counter-examples using a model checker if the source program violates the specification. The important steps that follow the discovery of a counter-example are generally not automated. The user must first decide if the counter-example shows genuinely erroneous behavior or is a manifestation of improper specification or abstraction. If the error is real, there remains the difficult task of understanding the error well enough to isolate and modify the faulty aspects of the system. The authors of Groce et al. (2006), Groce et al. (2004) describe an automated approach for assisting users in understanding and isolating errors in ANSI C programs. The approach is based on distance metrics for program executions. The authors of Griesmayer et al. (2007) present an automatic approach for fault localization in C programs. The method is based on model checking and reports only components that can be changed such that the difference between the actual and the intended behavior of the example is removed. To identify these components, they use the bounded model checker CBMC on an instrumented version of the program. The authors of Jose and Majumdar: Cause Clue Clauses (2011) present an algorithm for error cause localization based on a reduction to the maximal satisfiability problem (MAX-SAT).

All the above-mentioned approaches are either semi-automatic or execution-based and the model behavior is represented as specifications. To the best of our knowledge, the proposed method is the first method where the error is localized without requiring test cases and assertions. In the present paper, a fully automated static approach for localizing faults present in the transformed program is provided. The model program is given as the source program. It uses the bounded model checker CBMC which uses the MiniSAT automated theorem prover for discharging proof obligations. Verification techniques are used to identify the region of the fault.

10. Threats to validity

Threats to validity mainly lie in the benchmark suites used in the experiment. We have used the 41 buggy programs available in the widely used TCAS benchmark (which is an aircraft traffic collision avoidance system) and 120 mutants available for NTS benchmark programs. We also used 7 additional test cases referred to in the related literature. However, further experiments may be done to check the effectiveness of the proposed method in other kinds of programs. As mentioned in Section 1, the proposed method is applicable to structured non-recursive programs, programs and may require modifications to apply to other types of programs.

The proposed method works well if the correct and the faulty programs are similar in structure. If the correct program and the faulty programs are dissimilar in structures e.g., two loops of the correct programs are fused to generate a single loop of the faulty program, then the proposed method includes the entire source code segment involving two loops and the entire code segment involving the fused loop of the faulty program to be in the region of fault even in case of presence of no faults in the said loops. This is due to the fact that the proposed method proceeds to prove the equivalence of a pair of sections at a time and if they are proven not equivalent, the proposed method does not consider the next section if the present sections under consideration have loops.

11. Proofs

The following proofs except for termination proof hold if CBMC can derive the loop bounds automatically. The correct program is referred to as the first program and the faulty program is referred to as the second program.

Foward Analysis

Lemma 1. *If the control of the method $findStartingPoint$ is in an iteration i, $i \geq 1$, then the path segment/nested block pairs added in the variable buf in all the iterations $j < i$ are equivalent.*

Proof. In every iteration of the function $findStartingPoint$, a pair of path segments/nested block is added to the C file. The added path segment/nested block is the successor path segment/nested block pair of the sections added in iteration before the present iteration. When iteration i is invoked, all the $i-1$ pairs of path segment/blocks are already added. The above lemma will be proved by induction.

Base case ($i = 1$): In iteration $i = 0$, the variable buf only contains function definition line (Step 1). Now the variable buf gets appended with the path segment/block pair $\langle PB_0, PB_1 \rangle$ (Step 6). The variable buf is appended with the assertion (Step 9) VC if the pair of path segments/blocks are proved to be equivalent (Step 8). In the next step (Step 9), the assertions are added to the buffer and the control reaches iteration $i = 1$. If they are not proven to be equivalent, then the function $forwardPathExtension$ is called. Note that the function has to return $verification\_success$ for the control to reach to iteration $i = 1$.

The function $forwardPathExtension$ returns $verification\_success$ only from Step 9 where the variable buf is appended with the assertion V_e which is preceded by appending the string representation of the path segment/block pair with which the non-equivalent path segments/block pair is extended (Step 5). As the function $forwardPathExtension$ returns, iteration $i = 1$ is invoked through Step 12. Hence if the control of the method $findStartingPoint$ is in iteration $i = 1$, then the path segment/block pairs added in the variable buf in the 0th iteration are equivalent.

Induction step ($i = k$): Assume that the path segments/nested block pairs added in the variable buf in all the iterations $i < k$ are equivalent and the control has reached iteration $i = k$ from iteration $i = k-1$. We shall prove that the path segments/nested block pairs added in the variable buf in all iterations $i < k+1$ are also equivalent if the control reaches iteration $i = k+1$ from iteration $i = k$. Now in each iteration, buf is appended with some path segments/block pair and the corresponding assertion. If the path segments/nested block pairs are equivalent, then only control goes to the next iteration $i = k$ from Step 8 (by the arguments given above). If equivalence cannot be established, then the control reaches Step 10 and $forwardPathExtension$ is called. The function adds some path segments/block pairs and the corresponding assertion to the variable buf. As the function $forwardPathExtension$ returns, the control reaches Step 2 if the added path segments/block pairs are equivalent (Step 12). Otherwise, control returns from the function $findStartingPoint$ (Step 11) and it never comes back to Step 2 of iteration $i = k$. As all the path segment/nested block pairs till iteration $i = k-1$ are equivalent (as per hypothesis) and the control reaches iteration $i = k$, the path segment/nested block pairs added in the variable buf in all iterations $i = k$ are equivalent.

Hence, by induction principle, if the control of the method $findStartingPoint$ is in iteration i, $i \geq 1$ then the path segment/nested block pairs added in the variable buf in all the iterations $j < i$ are equivalent. □

Soundness

Theorem 1. *If the function corresponding to the algorithm 1 returns from Step 2, then there is no error present in the second program.*

Proof. As the function $findStartingPoint$ returns from the Step 2, the value of s_0 is $n_s - 1$ and the value of s_1 is $n_t - 1$. For flowchart representation as depicted in Section 3, the $n_s - 1$, $n_t - 1$ are the end states of the flowcharts F_0 and F_1. Suppose, the function returns from Step 2 in the kth iteration. By the lemma, it has been proved that all the path segment/nested blocks added in the variable buf before the kth iteration are equivalent. As the end states are reached for both the flowcharts, all the path segment/nested blocks of both the programs are already added and are proved to be equivalent. Hence no error is present in the second program. □

Completeness

Theorem 2. *If there is an error present in the second program, the algorithm visits Step 11 of the algorithm 1.*

Proof. Suppose, the second program is erroneous and the error is present in the nth path segment/nested block pair $\langle PB_n, PB_{n+1}\rangle$. There is no error present in any of the pairs $\langle PB_i, PB_{i+1}\rangle$, $i < n$. For each pair, the members of the pairs are written sequentially in the C file (Step 6), equivalence is established (Step 8) and the next iteration is invoked to write the next pair using the same steps. This process will continue until the pair $\langle PB_n, PB_{n+1}\rangle$ is written and the corresponding verification conditions are written in the C file (Steps 6, 7). Now, equivalence cannot be established in Step 8 and Step 10 is reached. Now the function $forwardPathExtension$ is called with the pair $\langle PB_n, PB_{n+1}\rangle$.

The function $forwardPathExtension$ returns from Steps 1, 9 and 10. It may happen that the start states of the successor path segment/nested blocks of the pair $\langle PB_n, PB_{n+1}\rangle$ mark ending states of the flow chart — in this case, the function returns from Step 1 declaring $extension\_failure$. Otherwise, the function returns from Steps 9 and 10. The Step 9 of the function $forwardPathExtension$ can only be reached from Step 8 which cannot happen in the present situation when $\langle PB_n, PB_{n+1}\rangle$ are not equivalent. Otherwise, Step 10 of the function $forwardPathExtension$ is executed marking $verification\_failure$. Hence the program can reach Step 11 from Step 10 of the function $findStartingPoint$.

As the function $forwardPathExtension$ returns $extension\_failure/verification\_failure$, the function $findStartingPoint$ goes to Step 11 and returns declaring $\langle s_{PB_n}, s_{PB_{n+1}}\rangle$, where s_{PB_n}, $s_{PB_{n+1}}$ are the starting states of the PB_n and PB_{n+1}, respectively, as the starting point of the region of fault. □

Backward Analysis

Lemma 2. *If the control of the method $findEndingPoint$ reaches the iteration i from the iteration $i-1$, $i \geq 1$, then the variable buf contains all equivalent the path segment/nested block pairs with their verification conditions which are added in all the iterations $j \leq i$.*

Proof. This will be proved with the help of induction.
Base case ($i = 1$): The variable buf is initialized to the empty string (Step 1). The variable buf is updated in Step 10 of the function $findEndingPoint$, in Step 10, and in Step 11 of the function $backwardPathExtension$. In the iteration $i = 0$ in the function $findEndingPoint$, the path segment/nested block pair PB_0, PB_1 and the assertion V_b are added in the C file (Steps 6, 7) and checked for equivalence (Step 9). If they are proved to be equivalent, then Step 10 of the function $findEndingPoint$ is executed and the variable buf is prepended with the string representation of the path segment/nested blocks PB_0, PB_1 and the corresponding assertion V_b. If the path segment/nested block pair PB_0 and PB_1 are not proved to be equivalent, then function $backwardPathExtension$ is called (Step 11). Although the variable buf is updated in both Steps 10 and 11 of the function backwardPathExtension, the updation of the variable buf in Step 11 is not considered as it returns $verification\_failure$ which causes the function $findEndingPoint$ to not to reach to the next iteration $i = 1$. In function $backwardPathExtension$, the path segments/nested block pair PB_0 and PB_1 are backward extended with the predecessor path segments/nested block pair $prev\_PB_0$ and $prev\_PB_1$, respectively. If the extended path segment/nested block pairs ($prev\_PB_0$ extended with PB_0 and $prev\_PB_1$ extended with PB_1) are proved equivalent, Step 10 is executed. In Step 10, the variable buf is updated with the path segment/nested block $prev\_PB_0$, PB_0, $prev\_PB_1$, PB_1, the corresponding assertion of backward path extension V_e.

Now iteration $i = 1$ of $findEndingPoint$ is reached through Step 13 which is reached either by Step 10 or by Step 11. More precisely, Step 13 is reached from Step 10 which is reached from Step 9 if PB_0 is proved equivalent to PB_1; otherwise, Step 13 is reached from Step 11 if the backward extended path segment of PB_0 is equivalent to the backward extended path of PB_1 (i.e., the return value of the $backwardPathExtension$ through Step 10 is a $verification\_success$). Hence, as the function $findEndingPoint$ reaches the iteration $i = 1$ from the iteration $i = 0$, then the variable buf contains all equivalent path segment/nested block pairs which are added in iteration $i = 0$ with the corresponding assertions.

Induction step ($i = k$): Let us assume that the variable buf contains all equivalent path segment/nested block pairs with their assertions which are added in all the iterations $i < k$ and the control has reached from iteration $i = k - 1$ to iteration $i = k$. We shall prove that the path segments/nested block pairs added in the variable buf in all the iterations $i < k + 1$ are also equivalent if the control reaches iteration $i = k + 1$ from the iteration $i = k$ Now from iteration, $i = k - 1$, iteration $i = k$ is reached from Step 13 which can be reached from Step 10 or Step 11. Now from Step 10, Step 13 can be reached if the path segment/nested block pairs added in the C file are equivalent. Also, Step 13 can be reached from Step 11 which happens if the function $backwardPathExtension$ returns $verification\_success$. More precisely, the control can only be returned from Step 10 of the function $backwardPathExtension$ which can be reached from the Step 9 if the backward extended path segment/nested block pairs (Steps 5, 6) followed by the assertion (Step 7) followed by the content of buf (Step 8) are proved to be equivalent. By induction hypothesis, all the path segment/nested blocks that are added to the variable buf are equivalent. Hence the extended path segment/nested block pairs are also equivalent and they are prepended to the variable buf with corresponding assertion V_e. Hence, as the function $findEndingPoint$ reaches the iteration k from the iteration $k - 1$, then the variable buf contains all equivalent path segment/nested block pairs with the corresponding assertions.

By induction principle, it is proved that if the control of the method $findEndingPoint$ reaches the iteration i from $i - 1$, $i \geq 1$, then the variable buf contains all equivalent the path segment/nested block pairs with the corresponding assertions which are added in all the iterations $j \leq i$. □

Soundness

Theorem 3. *If the control visits Step 2 of the function $findEndingPoint$, then there is no error present in the second program.*

Proof. If the control reaches Step 2, then $e_0 == 0$ and $e_1 == 0$. Suppose, it happened in iteration $i = n$. From the Lemma 2, we know that the path segment/nested block pairs added in the variable buf till iteration $i - 1$ are all equivalent.

In the function $findEndingPoint$, the path segment/nested block pairs are added in the following fashion — in the first iteration, the path segment/block pairs $\langle PB_0, PB_1\rangle$ are added such that the ending states of PB_0 and PB_1 are the end states of the flowcharts F_0 and F_1, respectively. Next the predecessors of PB_0 and PB_1 are added and the process continues till there is no predecessor path segment/nested blocks of the last added path segment/nested block pair or the last

added path segment/nested block are proved to be not equivalent. As the control reaches iteration $i = n$ when there is no predecessor of most recently added path segment/nested block is present, all the path segment/nested blocks are added to the C file and to the buffer. By Lemma 2, all path segment/nested block pairs are proved to be equivalent. □

Completeness

Theorem 4. *If there is an error present in the transformed program, the algorithm visits Step 12 of the algorithm* $findEndingPoint$.

Proof. Suppose, the transformed program is erroneous and the error is present in the nth path segment/nested block pair $\langle PB_n, PB_{n+1}\rangle$. There is no error present in any of the pairs $\langle PB_i, PB_{i+1}\rangle$, $i > n$. Now as there is no error present in any of the path segment/block pair $\langle PB_i, PB_{i+1}\rangle$, $i > n$, the path segment/nested block pairs are added to the C file and to the variable buf and the next iteration is invoked.

As the pair $\langle PB_n, PB_{n+1}\rangle$ and the corresponding assertions are written to the C file (Step 6) along with the content of the variable buf (Step 7) and it is subjected to validation, the equivalence of path segments/nested block pair cannot be established (Step 9) as an error is present in the pair $\langle PB_n, PB_{n+1}\rangle$. Now the $backwardPathExtension$ function is applied on the path segments/nested block pair $\langle PB_n, PB_{n+1}\rangle$.

As the function $backwardPathExtension$ is called, the pair of path segment/nested block $\langle PB_n, PB_{n+1}\rangle$ is backward extended with the predecessor path segment/nested block pair (Step 6) and backward extended path segment/nested block pair along with the assertion V_e and the variable buf is added to a new C file and it is subjected to validation process. The validation process will not succeed as the pair $\langle PB_n, PB_{n+1}\rangle$ are not equivalent and the function returns from Step 11 conveying $verification\_failure$.

As the function $backwardPathExtension$ returns $verification\_failure$, control reaches Step 12 and the ending states of the pair $\langle PB_n, PB_{n+1}\rangle$ are returned as the ending point of fault localization process. □

Termination

As both the correct and faulty programs have a finite number of sections and in every loop iteration of $findStartingPoint$ and $findEndingPoint$, at most two sections (one by the method itself and the other by extending the section) are added, the loops are going to terminate after a finite number of iterations.

12. Conclusion and future work

In this paper, a fully automatic approach for localizing faults is presented. The method is static in nature and it uses CBMC bounded model checker. The applicability of the method has been shown against TCAS and NTS benchmark programs and 7 other programs presented in other literature and it has been shown that the method can reduce the fault search space even more if applied over the static slices for the output variables. A limitation of the method is that it can only identify the first fault it encounters and then it exits. Future work includes localization of multiple faults using the above-mentioned solvers. Also, a method for over-approximating unwinding depth (Darke et al., 2015) can be used to devise a sound fault localization method if the loop bounds are unknown.

Data availability

Data will be made available on request.

References

Baader, F., Snyder, W., 2001. Unification theory. In: HandBook of Automated Reasoning, vol. 2. Elsevier and MIT Press, pp. 445–532.

Baah, G.K., Podgurski, A., Harrold, M.J., 2010. The probabilistic program dependence graph and its application to fault diagnosis. IEEE Trans. Softw. Eng. 36 (4), 528–545.

Ball, T., Naik, M., Rajamani, S., 2003. From symptom to cause: Localizing errors in counterexample traces. In: Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. New Orleans, Louisisana, USA, pp. 97—105.

Barthe, G., Crespo, J.M., Kunz, C., 2016. Product programs and relational program logics. J. Log. Algebraic Methods Program. 85 (5), 847–859.

Chaki, S., Groce, A., Strichman, O., 2004. Explaining abstract counterexamples. ACM SIGSOFT Softw. Eng. Notes 29 (6), 73–82.

Clarke, E.M., Kroening, D., Lerda, F., 2004. A tool for checking ANSI-C programs. In: Proceedings of Tools and Algorithms for the Construction and Analysis of Systems, TACAS. pp. 168—176.

Cristinel, M., Stumptner, M., Wotawa, F., 2000. Modeling java programs for diagnosis. In: Proceedings of the 14th European Conference on Artificial Intelligence, ECAI. Berlin, Germany, pp. 171–175.

Darke, P., Chimdyalwar, B., Venkatesh, R., Shrotri, U., Metta, R., 2015. Over-approximating loops to prove properties using bounded model checking. In: Proceedings of Design, Automation and Test in Europe Conference and Exhibition, DATE. Grenoble, France, pp. 1407–1412.

Dutta, S., 2022. Localizing faults using verification technique. In: Proceedings of Innovations in Software Engineering Conference, ISEC. pp. 1–11.

Dutta, A., Manral, R., Mitra, P., Mall, R., 2019. Hierarchically localizing software faults using DNN. IEEE Trans. Reliab. 69 (4), 1267–1292.

Dutta, A., Manral, R., Mitra, P., Mall, R., 2020. Hierarchically localizing software faults using DNN. IEEE Trans. Reliab. 69 (4), 1267–1292.

Friedrich, G., Stumptner, M., F. Wotawa, F, 1999. Model-based diagnosis of hardware designs. Artificial Intelligence 111 (1), 3–39.

Griesmayer, A., Staber, S., Bloem, R., 2007. Automated fault localization for C programs. Electron. Notes Theor. Comput. Sci. 174 (4), 95–111.

Grismayer, A., Staber, S., Bloem, R., 2010. Fault localization using a model checker. Softw. Test. Verif. Reliab. 20 (2), 149–173.

Groce, A., Chaki, S., Kroening, D., 2006. Error explanation with distance metrics. Int. J. Softw. Tools Technol. Transf. 8, 229–247.

Groce, A., Kroening, D., Lerda, F., 2004. Understanding counterexamples with explain. In: Computer Aided Verification (CAV). Berlin, Heidelberg, pp. 453–456.

Groce, A., Visser, W., 2003. What went wrong: Explaining counterexamples. In: Proceedings of International SPIN Workshop on Model Checking of Software. Berlin, Heidelberg, pp. 121–136.

Gupta, S., Savoiu, N., Dutt, N., Gupta, R., Nicolau, A., 2004. Using global code motions to improve the quality of results for high-level synthesis. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 23 (2), 302–312.

Hutchins, M., Foster, H., Goradia, T., Ostrand, T., 1994. Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria. In: Proceedings of 16th International Conference on Software Engineering, ICSE. Sorrento, Italy, pp. 191–200.

Jose, M., Majumdar: Cause Clue Clauses, R., 2011. Error localization using maximum satisfiability. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI. San Jose, California, USA, pp. 437–446.

Karfa, C., Sarkar, D., Mandal, C.R., Kumar, P., 2008. An equivalence-checking method for scheduling verification in high-level synthesis. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 27 (3), 556–569.

Lattner, C., Adve, V., 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In: Proceedings of the International Symposium on Code Generation and Optimization, CGO. San Jose, CA, USA.

Manna, Z., 1971. Mathematical theory of partial correctness. J. Comput. System Sci. 5 (3), 239–253.

Mayer, W., Abreu, R., Stumptner, M., Gemund, A.J.C., 2008. Prioritizing model-based debugging diagnostic reports. In: Procedings of the 19th International Workshop on Principles of Diagnosis. pp. 127–134.

Mayer, W., Stumptner, M., 2003. Model-based debugging using multiple abstract models. CoRR, http://arxiv.org/abs/cs/0309030.

Mayer, W., Stumptner, M., 2007. Abstract interpretation of programs for model-based debugging. In: Proceedings of the 20th International Joint Conference on Artificial Intelligence, IJCAI. Hyderabad, India, pp. 471–476.

Mayer, W., Stumptner, M., Wieland, D., Wotawa, F., 2002. Towards an integrated debugging environment. In: Proceedings of the 15th Eureopean Conference on Artificial Intelligence, ECAI. Lyon, France, pp. 422–426.

Renieres, M., Reiss, S.P., 2003. Fault localization with nearest neighbor queries. In: International Conference on Automated Software Engineering, Montreal. Canada, pp. 30–39.

Tip, F., 1995. A survey of program slicing techniques. J. Program. Lang. 3, 121–189.

Weiser, M., 1984. Program slicing. IEEE Trans. Softw. Eng. 352–357.

Yang, X., Chen, Y., Eide, E., Regehr, J., 2011. Finding and understanding bugs in c compilers. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI. San Jose, CA, USA, pp. 283–284.
Zaks, A., Pnueli, A., 2008. CoVaC: Compiler validation by program analysis of the cross-product. In: Proceedings of International Symposium on Formal Methods, FM. Turku, Finland.

Sudakshina Dutta The author has done her B.E. from Jadavpur University. She has completed her M.Tech and Ph.D. from Indian Institute of Technology Kharagpur. Presently, she is an Assistant Professor in Indian Institute of Technology Goa. Her research includes areas e.g., formal verification, static analysis, debugging, etc.