Model-Based Diagnosis versus Error Explanation

Heinz Riener*
*Institute of Computer Science
University of Bremen, Germany
{hriener,fey}@informatik.uni-bremen.de

Görschwin Fey*†
†Institute of Space Systems
German Aerospace Center, Germany
goerschwin.fey@dlr.de

***Abstract*—Debugging techniques assist a developer in localizing and correcting faults in a system's description when the behavior of the system does not conform to its specification. Two fault localization techniques are model-based diagnosis and error explanation. Model-based diagnosis computes a subset of the system's components which when replaced correct the system. Error explanation determines potential causes of the system's misbehavior by comparing faulty and correct execution traces. In this paper we focus on fault localization for imperative, non-concurrent programs. We compare the two fault localization techniques in a unified setting presenting SAT-based algorithms for both. The algorithms serve as a vantage point for a fair comparison and allow for efficient implementations leveraging state-of-the-art decision procedures. Firstly, in our comparison we use constructed programs to study strengths and weaknesses of the two fault localization techniques. We show that in general none of the fault localization techniques is superior but that the computed fault candidates depend on the program structure. Secondly, we implement the SAT-based algorithms in a prototype tool utilizing a *Satisfiability Modulo Theories* (SMT) solver and evaluate them on mutants of the ANSI-C program TCAS from the *Software-Artifact Infrastructure Repository* (SIR).**

***Index Terms*—Debugging, fault localization, SAT**

I. Introduction

Functional verification and debugging activities account for more than half of the total development time of a system. Functional verification attempts to detect misbehavior of a system by checking conformance of the system's behavior to a specification. Debugging deals with localizing and correcting known misbehavior in the system's description.

In industrial applications, verification and debugging are still done manually. Developers or verification engineers extensively simulate and test the system guided by the specification to identify and correct misbehavior which is often complicated, frustrating, and time-consuming. Automated techniques have the potential to reduce the labor-intensive manual burden needed for verification and debugging and to decrease the overall time-to-market. Fault localization has been recognized as the most challenging and laborious task in the debugging process [1].

In this paper we focus on automated fault localization techniques for systems described as imperative, non-concurrent programs. Given the source code of a program which does not conform to its specification, a fault localization technique attempts to exactly pinpoint the misbehavior to the part of the source code that is faulty.

This work was supported in part by the German Research Foundation (DFG, grant no. FE 797/6-1).

Fault localization techniques divide into two broad categories. The first category [4], [5], [6], [7] is based on *Model-Based Diagnosis* (MBD) [2], [3]. MBD searches for components of the program that when replaced correct the program's misbehavior. Thus, fault localization is closely related to program repair, i.e., a component is potentially faulty *if and only if* (iff) replacing the component makes the program correct. The second category is formed by *explanation-based techniques* [8], [9], [10], [11], [12], [13] which "explain" a misbehavior by comparing the differences (and similarities) between faulty and correct execution traces.

Both technqiues have their merits. MBD-based techniques directly point to potentially faulty components. However, the number of potentially faulty components is usually large. Explanation-based techniques attempt to understand the symptoms of a system's misbehavior on a particular execution trace. The technique identifies parts of a faulty execution trace that can be changed to satisfy the specification.

There is only little work in comparing (and combining) MBD-based and explanation-based techniques [14], [15]. Köb and Wotawa [14] compare the reported results of an MBD-based and an explanation-based technique for a concurrent program which implements a simple locking protocol. They concluded that an automatic debugging environment can profit from both techniques. Fey et al. [15] describe a debugging approach for sequential circuits given as *Hardware Description Language* (HDL)-designs. They use a combination of an MBD-based and an explanation-based technique to improve the resolution of their approach.

In this paper we compare an MBD-based and an explanation-based fault localization technique. Both techniques allow for certain design choices. We adopt the techniques to provide a unified setting and to allow for a fair comparison. In particular, the MBD-based technique is similar to the approach described by Smith et al. [5] but focuses on fault localization in software programs rather than hardware circuits. The explanation-based technique is an adaption of *error explanation* [13].

We present algorithms to formalize the fault localization problem with respect to both techniques into logic formulae. The inputs are the source code of an imperative, non-concurrent program and a formal specification which is either given by local assertions annotated into the program's source or by a reference implementation of the program. The output is a set of potentially faulty program locations which we call *fault candidates*. The logic formulae are solved leveraging state-of-the-art decision procedures for the *Satisfiability* (SAT)

problem. A satisfying assignment corresponds to one fault candidate. We use incremental SAT to enumerate all fault candidates, i.e., we repeatedly check for satisfiability, extract the satisfying assignment, and block the assignment until the logic formulae become inconsistent. However, we do not need to enumerate all possible assignments of the logic formula but focus on a specific subset of the variables introduced for fault localization.

Our SAT-based algorithms are implemented in a prototype tool which uses *Quantified-Free Bit-Vector Logic* (QF_BV) and solves the logic formulae with the aid of a *Satisfiability Modulo Theories* (SMT) solver.

Our comparison identifies strengths and weaknesses of the fault localization techniques. The contribution of the paper is threefold:

1) We present algorithms, called MBD and BEST-EXPLAIN, for formalizing fault localization with respect to both techniques into instances of the satisfiability problem to form a unified framework for comparison. The algorithm MBD refers to classical model-based diagnosis leveraging multiple counterexamples. The algorithm BEST-EXPLAIN refers to a new generalization of error explanation which searches for the most similar pair of a faulty and a correct execution trace. Moreover, it is the first implementation of error explanation leveraging incremental SAT instead of Pseudo-Boolean (PB) constraint solving.
2) We study the fault localization techniques on constructed programs and show that depending on the program the fault candidates obtained with the two techniques can be very different, i.e., one of the techniques pinpoints the fault whereas the other reports fault candidates for almost all program locations.
3) We compare the fault localization techniques experimentally in a case study using mutants of the program TCAS from the *Software-Artifact Infrastructure Repository* (SIR) [16]. We implemented a prototype tool to compute fault candidates with MBD and BEST-EXPLAIN. We asses the quality of the fault candidates similar to Renieris and Reiss [11], i.e., we count their distance from the real faults with respect to a reference implementation of TCAS on the *Program Dependency Graph* (PDG). The distance refers to the length of the cause-effect chain the programmer has to examine in order to locate the real fault.

The remainder of the paper is structured as follows. In Section II, we present a simple program model and establish notation to encode a program into logic. In Section III, we introduce the two fault localization techniques, and describe algorithms for formalizing them into instances of the satisfiability problem. In Section IV, we identify strengths and weakness of the two techniques and study them in the context of constructed programs. In Section V, we present a case study using the ANSI-C program TCAS. Section VI concludes the paper.

II. Preliminaries

A. Program Model

1) Preprocessing: We use a bounded model checking approach for software programs similar to Kröning et al. [17] using loop unrolling and *Static Single Assignment* (SSA) form [18]: given an imperative, non-concurrent program P and an unrolling bound k, we unroll loops and recursive functions in the program with respect to k and transform the unrolled program into SSA form. Additionally, we assume that function calls have been inlined to simplify the following presentation. We denote this transformation by $\texttt{Preprocess}(P, k)$. The resulting program P^k contains no loops and no function calls and consists of statements over program variables, constants, and labels. The program is in SSA form, i.e., the value of each program variable is assigned on its first appearance in the program's source code and then remains constant.

2) Syntax and Semantics: In the following, we use the symbols s, l, v, and c (with or without indices) to denote statements, labels, program variables, and constants, respectively. Moreover, w (with or without index) is used to denote a program variable or constant. Each statement has a unique label. We write $\mathsf{s}(l)$ to denote the statement at label l. A statement is either an assignment statement, a branching statement, or a phi statement.

An *assignment statement* is of the form $v_r := e_u(w_1)$ or $v_r := e_b(w_1, w_2)$, where e_u and e_b are expressions and w_1 and w_2 are program variables or constants, respectively. We allow all usual arithmetic, relational, and logical expressions with the expected semantics, e.g., the logic negation $!w_1$, the arithmetic negation $-w_1$, or the arithmetic addition $w_1 + w_2$. The assignment statement evaluates the expression on the *Right-Hand Side* (RHS) by interpreting the semantics of the operators in the expression with the program variables or constants as operands and assigns the resulting value to the program variable v_r on the *Left-Hand Side* (LHS) when executed.

A *branching statement* is either of conditional form **if** v_c **then goto** l_1 **else goto** l_2 or of unconditional form **goto** l, where v_c is a program variable and l_1, l_2, and l are labels. The branching statement changes the flow of control in the program, i.e., it jumps to a statement at a particular label in the program when executed. The conditional branching statement jumps to $\mathsf{s}(l_2)$ if the value of v_c is equal to 0 and jumps to $\mathsf{s}(l_1)$ otherwise. The unconditional branching statement jumps under all conditions to $\mathsf{s}(l)$.

A *phi statement* $v_r := \phi([w_0, l_{i_0}], [w_1, l_{i_1}], \ldots, [w_n, l_{i_n}])$ denotes a special function which selects the program variable or constant w_j, $0 \leq j \leq n$, and assigns its value to v_r if the statement $\mathsf{s}(l_{i_j})$ was most recently executed before the phi statement with respect to the statements $\mathsf{s}(l_{i_k})$, $0 \leq k \leq n$. The phi statements establish the transformation into SSA form without the need for the introduction of identity assignments [19]. A ϕ-function is used when the value assigned to a program variable cannot be determined statically but depends on the control flow of the program when executed.

A program is syntactically described as a sequence of statements. The semantics of a program corresponds to a

semantic function over a set of program variables with a distinguished subset of input variables and a distinguished subset of output variables. We say a program is *deterministic* if an assignment to the input variables implies a unique assignment to all other program variables in order to satisfy the semantics of the statements. Otherwise, we say the program is *non-deterministic*. In the following, we focus on deterministic programs.

B. Encoding the Program

1) Formalization into a Logic Formula: The semantics of each statement can be formalized to obtain a logic formula. For each program variable, we introduce a logic variable which symbolically represents all possible values of the program variable. In the following, we use the symbols x and d (with or without indices) to denote logic variables and constant values, respectively.

We call the transformation from the program into a logic formula the *logic encoding*. The logic of choice may be *propositional logic* or propositional logic enriched with word-level data types and operations, e.g., QF_BV logic. Logic variables and word-level operations then need to be mapped into the respective primitives of the logic in use. Following an *eager approach*, we lower word-level operations when needed to semantically equivalent logic formulae using Boolean connectives similar to *Tseitin's encoding* [20]. We denote the logic formula which describes the semantics of the statement s by $\texttt{Encode}(s)$ and extend the definition of $\texttt{Encode}$ to programs. Let P be a program consisting of statements s_i, $0 \leq i \leq n$, then $\texttt{Encode}(P) := \bigwedge_{i=0}^{n} \texttt{Encode}(s_i)$.

In the following, we abstract from the logic encoding to simplify notation and describe the program P by a logic formula p over the logic variables x_i, $0 \leq i \leq r$, rather than the real logic primitives used to encode the logic variables.

Suppose P is a program encoded into the logic formula p. We use $\mathsf{Vars}(p)$ to refer to the set of all logic variables of p and separate $\mathsf{Vars}(p)$ into two disjoint subsets $\mathsf{In}(p)$ and $\mathsf{Aux}(p)$. We call $\mathsf{In}(p)$ the *input set* of p which is the set of logic variables corresponding to the input variables of P and we call $\mathsf{Aux}(p) := \mathsf{Vars}(p) \backslash \mathsf{In}(p)$ the *auxiliary set*.

Lastly, we use $(p_0, p_1, \ldots, p_k) := \texttt{Encode}(P, k)$ as shorthand for $p_i := \texttt{Encode}(P)$, $0 \leq i \leq k$. The sets $\mathsf{Vars}(p_i)$ of logic variables are pairwise disjoint, i.e., $\mathsf{Vars}(p_i) \cap \mathsf{Vars}(p_j) = \emptyset$ for all $i \neq j$ and $0 \leq i, j \leq k$.

2) Solving Logic Formulae: We leverage a decision procedure for the *satisfiability problem* with respect to the logic in use, i.e., a SAT solver or an SMT solver, to decide the satisfiability of the logic formula and assume that the solver returns a satisfying assignment in case the logic formula is satisfiable.

We denote a call to a SAT or SMT solver by $\texttt{SAT}(f)$, where f is a logic formula over logic variables to be checked for satisfiability. We assume that a SAT or SMT solver is a sound and complete decision procedure, i.e., the solver returns true iff f is satisfiable. We use $\texttt{Model}(f)$ to denote an arbitrary model of the logic formula f, i.e., a satisfying assignment for f in case it is satisfiable and undefined values otherwise. Thus, before $\texttt{Model}(f)$ is used, the logic formula f needs to be checked for satisfiability. This reflects the behavior of common SAT and SMT solvers. Additionally, we use $\texttt{Model}(f, V)$ with a second parameter V to denote the model restricted to a certain subset $V \subseteq \mathsf{Vars}(f)$. Omitting the second parameter in $\texttt{Model}(f)$ is equivalent to $\texttt{Model}(f, \mathsf{Vars}(f))$.

For the sake of simplicity, we assume that $\texttt{Model}(f)$ is a model in terms of the logic variables when f is a logic formula over logic variables x_i, $0 \leq i \leq r$, i.e., the real model with respect to the logic in use is implicitly mapped to the logic variables. The model can then be written as an *assignment* $t := (x_0 = d_0, x_1 = d_1, \ldots, x_r = d_r)$ of constant values $d_0, d_1, \ldots, d_r$ of the corresponding domains to the logic variables $x_0, x_1, \ldots, x_r$. We say that the assignment t for the logic formula f is satisfying iff $f(t)$ is satisfied when x_i is replaced by d_i for all $0 \leq i \leq r$. Moreover, an assignment t is *complete* (with respect to f) if t assigns a value to all logic variables $x \in \mathsf{Vars}(f)$ and *partial* otherwise.

A common operation when SAT or SMT solvers are used is restricting a particular set of possible assignments of a logic formula. We define a restriction operator $|$ in terms of the logic variables. Suppose f is a logic formula and $t := (x_0 = d_0, x_1 = d_1, \ldots, x_r = d_r)$ is a possibly partial assignment to the logic variables, we define the *positive restriction* $f|_t := f \wedge (\bigwedge_{i=0}^{r}(x_i = d_i))$ and the *negative restriction* $f|_{\neg t} := f \wedge (\bigvee_{i=0}^{r}(x_i \neq d_i))$. We extend the definition of the restriction operator $|$ to sets of assignments to logic variables. Let f be a logic formula over logic variables and $t_0, t_1, \ldots, t_r$ be assignments to logic variables, then we define $f|_{\{q_1, q_2, \ldots, q_r\}} := ((\ldots((f|_{q_0})|_{q_1})\ldots)|_{q_r})$ with $q_i \in \{t_i, \neg t_i\}$ for $0 \leq i \leq r$. We use $f|_{\neg T}$ as a shorthand for $f|_{\{\neg t_1, \neg t_2, \ldots, \neg t_r\}}$ where $T := \{t_1, t_2, \ldots, t_r\}$ is a set of assignments to logic variables.

C. Formal Specification

A formal specification is a set of logic properties which describe the correct behavior of a program. The formal specification is either provided by local assertions annotated into the program's source code or by a reference implementation of the program. In both cases the formal specification is encoded into a logic formula with an approach similar to the description above. When a reference implementation is used, the encoded program and the encoded specification share the same logic variables for inputs and outputs.

We use φ (without indices) to denote the logic formula obtained from encoding the formal specification. In our algorithms for fault localization, we often need multiple copies of the formal specification, each with its own set of logic variables. Thus, we use φ_i with index $0 \leq i \leq n$, to refer to logic formulae syntactically equivalent to φ where the set of logic variables $\mathsf{Vars}(\varphi_i)$ and $\mathsf{Vars}(\varphi_j)$ of each pair (φ_i, φ_j) of logic formulae are pairwise disjoint for $i \neq j$ and $0 \leq i, j \leq n$.

D. Fault Model

We focus on fault localization, i.e., in our setting the program's source code does not conform to its formal specification. We deal only with simple mistakes which have

one root cause but may manifest in the program's source code as multiple faulty program locations. For instance, a programmer erroneously assumes that array indices start at index 1 instead of index 0. The mistake has one root cause, the misunderstanding of array indexing in the programming language, but manifests in the program's source code as multiple faulty program locations corresponding to all array accesses in the program.

Consider a faulty program P with a simple mistake. We use the closest reference implementation P' to P to classify the mistake. The closest reference implementation is the program with the smallest number of changes with respect to P which conforms to the formal specification. We distinguish six types of *simple changes*: (1) An *operator mutation* which refers to a change of a relational or arithmetic operator. (2) A *logic mutation* which refers to a change of a logic operator. (3) A *variable mutation* which refers to the replacement of a program variable with a constant or another program variable in the scope. (4) A *constant mutation* which refers to the change of a constant value. (5) *Additional code* which refers to an additional statement in P with respect to P'. (6) *Missing code* which refers to a missing statement in P with respect to P'. Moreover, we say a *complex change* refers to any change which does not fit to one of the simple changes.

We define a fault as a simple change, i.e., a fault corresponds to a single statement. A complex change corresponds to multiple faults. However, for complex faults we are often unable to distinguish between replacements, additional code, and missing code. We refer to a fault in a statement s with label l either by saying that the statement s or the statement $\mathtt{s}(l)$ at label l is faulty.

Suppose P is a faulty program. A fault influences the program's behavior when the program is executed, i.e., the semantics of the program diverges from the expected semantics. When the developer (or user) is confronted with unexpected misbehavior of the program, i.e., the program does not conform to the specification, a symptom of the fault becomes observable. For instance, a test case fails unexpectedly or a logic property of a formal specification is refuted by the program. In the following, we say that the observable symptoms of a fault are *failures*.

The fault localization problem is to trace a failure to its root cause. An "ideal" fault localization technique would precisely report a fault for each program location which is faulty. However, in general there is an unlimited number of possible corrections for a particular fault and thus the "ideal" technique does not exist. When an exact localization is not possible, the fault localization technique determines a proper subset of the statements of the program that may be faulty, the so-called fault candidates. The major obstacle in fault localization is the large number of fault candidates that need to be manually examined in order to identify the real faults.

Figure 1 and 2 show the programs Q_0 and R_0, respectively, which we use as running examples. Each program has one injected fault. The program code complies to the program model introduced in Section II-A. However, we make two simplifying assumption: (1) we assume that all program variables and constants are of integer type and (2) we avoid dealing with overflow semantics but assume that the values of all program variables and constants are restricted such that no overflows are possible. Moreover, to improve readability we use special names i_0, i_1, and o_0 to distinguish the program's input and output variables from other program variables.

Fig. 1. Program Q_0 with a faulty output variable $\mathtt{s}(l_5)$.

$$
\begin{array}{ll}
l_0: & v_0 := i_0 \neq 0 \\
l_1: & v_1 := i_0 + c_0 \\
l_2: & \textbf{if } v_0 \textbf{ then goto } l_4 \textbf{ else goto } l_3 \\
l_3: & \textbf{goto } l_4 \\
l_4: & v_2 := \phi([v_1, l_2], [c_1, l_3]) \\
l_5: & o_0 := v_2 * \textbf{error}(c_2)
\end{array}
$$

Fig. 2. Program R_0 with a faulty phi statement $\mathtt{s}(l_4)$.

$$
\begin{array}{ll}
l_0: & v_0 := i_0 + c_0 \\
l_1: & v_1 := i_1 \neq 0 \\
l_2: & \textbf{if } v_1 \textbf{ then goto } l_4 \textbf{ else goto } l_3 \\
l_3: & \textbf{goto } l_4 \\
l_4: & v_2 := \phi([c_1, l_2], [\textbf{error}(c_2), l_3]) \\
l_5: & o_0 := v_0 + v_2
\end{array}
$$

The injected faults manifest themselves in the program code as erroneous constant values, i.e., the changes are constant mutations. We mark the faults in the program code symbolically with the special operator **error**. The operator **error** takes a constant value c as operand and evaluates to a constant value $c' \neq c$ when executed. The program becomes correct if $\textbf{error}(c)$ is replaced by c, where c is a constant value. We denote the correct programs with a superscript C. For instance, to obtain the correct program Q_0^C from Q_0 the expression $\textbf{error}(c_2)$ has to be replaced by c_2. The program Q_0 results in failure if executed on input $i_0 \neq c_0$ and the program R_0 results in failure if executed on input $i_1 = 0$ regardless of the value assigned to i_0. For all other inputs the two programs behave correctly, i.e., the programs and their reference implementations compute the same output when executed on the same input.

III. Fault Localization

In this section we introduce the two fault localization techniques. For each technique we describe the general idea and present an algorithm for formalizing the fault localization problem into instances of the satisfiability problem. The MBD-based technique, called MBD, is motivated by Reiter's fault diagnosis framework [2] but adapted for software programs. Our approach is similar to the technique described by Smith et al. [5] but focuses on software programs rather than hardware circuits. However, we do not need test cases as part of the input but use model checking to "extract" the test cases from a fixed number of counterexamples.

The explanation-based technique, called BEST-EXPLAIN, is an adaption of error explanation [13]. Given a counterex-

ample, error explanation searches with the aid of a model checker for the most similar execution trace that satisfies the formal specification. However, we do not use an initial counterexample, instead we search for the most similar pair of faulty and correct execution traces to explain a failure. The differences are then mapped to the program's source code to identify fault candidates.

Our setting is unified in the sense that both techniques, MBD and BEST-EXPLAIN, are based on a model checker and use the same input data, a program, a formal specification, and an unrolling bound, to calculate a set of fault candidates. Additionally, the accuracy of MBD can be tuned by specifying the number of counterexamples used for fault localization.

A. Model-Based Diagnosis

For the general description of MBD, we follow in terms and notation Reiter [2] and adapt the description to fault localization when needed: a *system's description*, a model of a system, consisting of components and a set of *observations* about the correct behavior of the system are given. Both the system's description and the observations are formalized as logic sentences SD and OBS, respectively. The components of the system are defined as COMP.

We focus on fault localization in the system's description, i.e., the system's description and the observations are inconsistent because the system's description contains one or more faults. The goal of MBD is to determine a subset $\mathsf{COMP}' \subseteq \mathsf{COMP}$ of the components that are responsible for the inconsistency and serve as fault candidates. In order to determine this subset, an *abnormal predicate* $\mathsf{ab}(c)$ is introduced for each component $c \in \mathsf{COMP}$. The component c behaves as usual if the abnormal predicate is false and behaves non-deterministically otherwise. For each component $c \in \mathsf{COMP}$, this is formally expressed as $\neg\mathsf{ab}(c) \rightarrow \mathsf{Behavior}(c)$, where $\mathsf{Behavior}(c)$ is a logic sentence which describes the normal behavior of c. A subset $\mathsf{COMP}' \subseteq \mathsf{COMP}$ is a set of fault candidates iff $\mathsf{SD} \cup \mathsf{OBS} \cup \{\neg\mathsf{ab}(c) \mid c \in \mathsf{COMP} \backslash \mathsf{COMP}'\}$ is consistent.

Reiter [2] proposed a hitting set algorithm to determine a minimal set $\mathsf{COMP}' \subseteq \mathsf{COMP}$ of fault candidates. A set of fault candidates is minimal iff no proper subset $\mathsf{COMP}'' \subset \mathsf{COMP}'$ is a set of fault candidates. The size $|\mathsf{COMP}'|$ of a set of fault candidates is the number of elements of the set.

Procedure 1 shows our MBD-based fault localization algorithm in pseudo code. Similar to Smith et al. [5] and Fey et al. [7], our algorithm relies on a SAT solver. The input of the algorithm is the source code of a program P, a formal specification φ, and an unrolling bound k. The number l of counterexamples used for fault localization can be provided to tune the algorithm's accuracy. Intuitively, the algorithm computes a non-deterministic repair with respect to the inputs of all blocked counterexamples, i.e., a subset of the program statements is selected which when replaced with non-deterministic behavior make the program conform to its formal specification. The counterexamples are obtained using SAT-based bounded model checking. The output of the algorithm is a set of fault candidates.

Procedure 1: MBD-Based Fault Candidate Computation

Input : the source code P of a program, a formal specification φ, an unrolling bound k, a number l of counterexamples
Output: a set of fault candidates FC

```
1  begin
2      P^k := Preprocess(P,k);
3      p̃_1 := Encode(P^k);
4      C := ∅, l' := 0;
5      ψ := p̃_1 ∧ ¬φ;
6      while SAT(ψ|_¬C) ∧ (l' < l) do
7          C := C ∪ Model(ψ|_¬C);
8          l' := l' + 1;
9      end
10     let (c_1, c_2, ..., c_l') := C;
11     (p̃_2, p̃_3, ..., p̃_l') := Encode(P^k, l' − 1);
12     (p̂_1, p̂_2, ..., p̂_l', AB) := AbnPd(P^k, (p̃_1, p̃_2, ..., p̃_l'));
13     ψ := ⋀_{i=1}^{l'} (p̂_i ∧ φ_i)|_In(c_i) ∧ Σ_{a∈AB} a = 1;
14     F = ∅;
15     while SAT(ψ|_¬F) do
16         m := Model(ψ|_¬F, AB);
17         F := F ∪ {m};
18     end
19     FC := MapToSource(F);
20     return FC;
21 end
```

First, we preprocess and encode P to obtain the logic formula $\tilde{p}_1$ as described in Section II-A1 (line 2-3). We generate l' different counterexamples with respect to the formal specification (line 4-10) by checking the satisfiability of the logic formula $\tilde{p}_1 \wedge \neg\varphi$ with the aid of a SAT solver. Initially, we start with an empty set of counterexamples C (line 4) and successively block them in the following iterations until either the number of counterexamples l has been reached or all counterexamples have been explored and the logic formula becomes unsatisfiable. We call the actual number of generated counterexamples $l' \leq l$. In practice l' is usually equal to l. The counterexamples are collected in a set and blocked using the restriction operator (line 7). To simplify notation, we refer to the set of counterexamples as an ordered sequence which is expressed using the **let** keyword in the pseudo code (line 10).

Then, we encode the program P again to obtain l' syntactically equivalent formulae $\tilde{p}_i$, $1 \leq i \leq l'$ (line 11) with $l' \leq l$. Also, we duplicate the formal specification to obtain l' syntactically equivalent logic formulae φ_i, $1 \leq i \leq l'$. Recall that each of these logic formulae uses its own set of logic variables.

We add abnormal predicates to the logic formulae $\tilde{p}_i$, $1 \leq i \leq l'$, denoted by `AbnPd` (line 12). In our setting a component is a statement. We introduce one abnormal predicate per statement and reuse the abnormal predicates for the same statements in different logic formulae $\tilde{p}_i$. Let P be a program consisting of statements s_i, $0 \leq i \leq n$, and let a_i, $0 \leq i \leq n$, be abnormal predicates, we define

$\hat{p} = \bigwedge_{i=0}^{n}(\neg a_i \rightarrow \texttt{Encode}(s_i))$. The abnormal predicates can be used to "disable" the behavior of particular statements of P: if an abnormal predicate a_i is assigned true, the corresponding clause $(\neg a_i \rightarrow \texttt{Encode}(s_i))$ is satisfied without the need for checking the satisfiability of $\texttt{Encode}(s_i)$. If an abnormal predicate is assigned false, the behavior of that statement is not changed, i.e., the corresponding clause reduces to $\texttt{Encode}(s_i)$. We use a mapping $\mathsf{stmt}_{ab}(a) = s$ to map an abnormal predicate a to the statement s when $(\neg a \rightarrow \texttt{Encode}(s))$ is a clause in $\hat{p}$. The abnormal predicates in all logic formulae $\hat{p}_i$, $1 \leq i \leq l'$ are the same, i.e., setting an abnormal predicate a to true disables the behavior of the statement $\mathsf{stmt}_{ab}(a)$ in all logic formulae $\hat{p}_i$, $1 \leq i \leq l'$. We use AB to denote the set of all abnormal predicates.

Lastly, we restrict the assignments to the inputs of the logic formulae $\hat{p}_i$ to the values of the counterexamples and enforce with a cardinality constraint that exactly one abnormal predicate is true (line 13), i.e., we search for an assignment to the abnormal predicates that satisfies the formal specification φ over the input values of the counterexamples. Assuming that there is only one faulty program statement, MBD can pinpoint the fault utilizing one abnormal predicate. However, our fault model introduced in Section II-D considers multiple faulty program locations, too. Thus, in our setting we cannot guarantee that MBD will compute a fault candidate which corresponds to the real fault. All assignments to the abnormal predicates are systematically blocked (line 14-18) and mapped to the corresponding statements in the program's source code (line 19) denoted by `MapToSource`, i.e., we collect a statement s as a fault candidate if there is an assignment $u \in F$ which assigns an abnormal predicate $a = 1$ with $\mathsf{stmt}_{ab}(a) = s$.

B. Error Explanation

Our explanation-based technique is an adaption of error explanation [9], [13]. The underlying motivation of error explanation arises from the *counterfactual approach* to causality [21]. The key idea is that an observed effect e is causally dependent on a cause c in an environment E iff for all environments E' similar to E it is more likely that the removal of the cause c also removes the effect e than removing the cause c without the effect e. The difference (and similarity) of the environments E and E' is expressed by means of a *distance metric* over possible environments. A distance metric over a set $\mathbb{S}$ is any binary function $d : \mathbb{S} \times \mathbb{S} \rightarrow \mathbb{R}$ that is positive definite, symmetric, and satisfies the triangle inequality.

In error explanation, the execution traces of a program P serve as the possible environments, the cause is a fault, and the effect is a failure. An execution trace corresponds to a terminating path with respect to the control flow of the program and is formalized as a complete assignment to all logic variables. Once all logic variables are assigned to a value, the sequence of statements to be executed is fixed. The difference of two execution traces is measured using the Hamming distance [22] defined on execution traces. Given two execution traces $t := (x_0^t = c_0^t, x_1^t = c_1^t, \ldots, x_r^t = c_r^t)$ and $t' := (x_0^{t'} = c_0^{t'}, x_1^{t'} = c_1^{t'}, \ldots, x_r^{t'} = c_r^{t'})$ of P of finite length r, where x_i^τ are logic variables and c_i^τ are constant values with $\tau \in \{t, t'\}$ and $0 \leq i \leq r$. The distance $d(t, t')$ of t and t' is defined as $d(t, t') := \sum_{i=0}^{r} \Delta(i)$ where $\Delta(i) = 1$ iff $c_i^t \neq c_i^{t'}$. An execution trace is *correct* if the assignment to the logic variables satisfies the formal specification, i.e., no assertion statement along that path fails or the values of the output variables are equal to the values of the output variables of a reference implementation when executed on the same inputs. Otherwise, the execution trace is *faulty*. In error explanation, a model checker is used to determine a counterexample which serves as a faulty execution trace. In order to obtain a set of fault candidates, a correct execution trace with minimal distance to the counterexample is computed. The differences between the two execution traces are then examined on the source code of the program.

Procedure 2: Explanation-based Fault Candidate Computation

Input : the source code P of a program, a formal specification φ, and an unrolling bound k
Output: a set of fault candidates FC

```
1  begin
2      P^k := Preprocess(P,k);
3      (p̃, p̃') := Encode(P^k, 2);
4      let (x_1, x_2, ..., x_m) := Vars(p̃);
5      let (x'_1, x'_2, ..., x'_m) := Vars(p̃');
6      d_i := (x_i ≠ x'_i) for 0 ≤ i ≤ m;
7      Δ := Σ_{i=0}^{m} d_i;
8      ψ := (p̃ ∧ ¬φ_1) ∧ (p̃' ∧ φ_2);
9      k' = MIN(ψ ∧ (Δ = k), k, m);
10     v := Model(ψ ∧ (Δ = k'), {d_i | 0 ≤ i ≤ m});
11     FC := MapToSource(v);
12     return FC;
13 end
```

We adapt error explanation for fault localization rather than explaining counterexamples. Procedure 2 shows our explanation-based fault localization algorithm, called BEST-EXPLAIN, in pseudo code. The input and output of the algorithm, the preprocessing, and the encoding (line 2,3) are similar to MBD. In BEST-EXPLAIN, however, we encode the program only twice into the logic formulae $\tilde{p}$ and $\tilde{p}'$ (line 3).

Our BEST-EXPLAIN algorithm searches for the most similar pair of a correct and a faulty execution trace, i.e., a faulty and a correct execution trace with minimal distance, and collects the statements corresponding to the differences of the two traces as fault candidates. We formalize this as an optimization problem similar to Groce et al. [13] which minimizes the distance between the logic variables of the two logic formulae $\tilde{p}$ and $\tilde{p}'$. Notice that BEST-EXPLAIN is different from the original error explanation algorithm because it searches for both the correct and the faulty execution trace by minimizing the distance between all program variables.

We introduce Boolean variables d_i, $0 \leq i \leq m$, for corresponding pairs of logic variables of $\tilde{p}$ and $\tilde{p}'$, where m is the

number of logic variables in $\tilde{p}$ and $\tilde{p}'$, respectively, (line 6,7). We then define $\Delta := \sum_{i=0}^{m} d_i$ as the arithmetic sum of the Boolean variables d_i, $0 \leq i \leq m$.

We use $(\Delta = k)$ as the optimization criterion and minimize it by iteratively calling a SAT solver. The optimization is denoted by $k' = \texttt{MIN}(\psi \wedge (\Delta = k), k, m)$ in the pseudo code (line 9), where $\psi \wedge (\Delta = k)$ is the logic formula, k is the parameter to be minimized, and m serves as a trivial upper bound for k. We start with checking whether $\psi \wedge (\Delta = m)$ is satisfiable and use binary search to determine the minimum k' such that $\psi \wedge (\Delta = k')$ is satisfiable but $\psi \wedge (\Delta = k' - l)$ is unsatisfiable for all $l > 0$. Then, we extract a model v of $\psi \wedge (\Delta = k')$ for the Boolean variables d_i, $0 \leq i \leq m$ (line 10). We map the model v to the source code of the program P and determine the fault candidates FC similar to MBD denoted by `MapToSource` (line 11). Different from MBD, `MapToSource` collects the statements with corresponding Boolean variables $d_i = 1$, $0 \leq i \leq m$, in v.

IV. Comparison

In this section we compare MBD and error explanation on constructed programs. We show that depending on the program the fault candidates computed by the two fault localization techniques can be very different. In particular, we construct programs for which one of the techniques computes less fault candidates than the other and show that by systematically adding statements the difference between the sets of fault candidates grows.

We use the two faulty programs Q_0 and R_0 from Section II-D. Suppose MBD and BEST-EXPLAIN define functions $\mathsf{FC_d}$ and $\mathsf{FC_e}$ which map a program to the number of computed fault candidates. We generalize the programs Q_0 and R_0 to sequences of faulty programs $(Q_k)_{k\in\mathbb{N}}$ and $(R_k)_{k\in\mathbb{N}}$ where Q_{i+1} and R_{i+1} are obtained from Q_i and R_i by adding one statement, respectively. We argue that for the sequences of faulty programs Q_k and R_k the number of fault candidates with respect to one technique remains constant whereas the number of fault candidates with respect to the other technique grows linearly with the size of the program:

$$\mathsf{FC_d}(Q_k) = \mathcal{O}(1) \text{ and } \mathsf{FC_e}(Q_k) = \mathcal{O}(k) \quad (1)$$

$$\mathsf{FC_d}(R_k) = \mathcal{O}(k) \text{ and } \mathsf{FC_e}(R_k) = \mathcal{O}(1) \quad (2)$$

Intuitively, we show that one of the fault localization techniques reports a fault candidate for each added statement and the other does not.

A. MBD can be superior to Error Explanation

Consider the faulty program Q_0 from Figure 1. There is only one correct execution trace when the program is executed on input $i_0 = -c_0$ which forces $o_0 = 0$ and eliminates the effect of the faulty statement at l_5. All other inputs $i_0 \neq -c_0$ correspond to faulty execution traces. MBD computes one fault candidate for the real faulty statement at label l_5 regardless of which faulty execution trace is used as counterexample. A program can always be corrected at its output variable. However, we can choose constant values c_0, c_1, c_2 such that no other fault candidates exist. For instance, assume we attempt to correct the program at $\mathsf{s}(l_1)$. MBD creates logic formulae q_0 and q_0^C from Q_0 and its reference implementation Q_0^C and substitutes the logic variable corresponding to v_1 in q_0 with an open variable R. If the output variables of q_0 and q_0^C become equal allowing arbitrary values for R then $\mathsf{s}(l_1)$ is a fault candidate. In particular, $\mathsf{s}(l_1)$ is a fault candidate iff $R * \mathbf{error}(c_2) = c_1 * c_2$. If we choose the constants c_1, c_2, and $\mathbf{error}(c_2)$ such that $\mathbf{error}(c_2)$ does not divide $c_1 * c_2$ without remainder then the value of R is not an integer and no correction for $\mathsf{s}(l_1)$ is possible. Similar arguments apply to all other statements. Thus, we can choose certain values for c_0, c_1, c_2, and $\mathbf{error}(c_2)$ such that $\mathsf{s}(l_5)$ remains the only fault candidate.

Our error explanation algorithm searches for a pair of a faulty and a correct execution trace with minimal distance. The correct execution trace t_+ forces $v_0 = 1$, $v_1 = 0$, $v_2 = 0$, and $o_0 = 0$. We distinguish two cases of faulty execution traces with respect to the control flow of the program: (1) $i_0 = 0$ and (2) $i_0 = d$ with $d \in \mathsf{Int}\backslash\{-c_0, 0\}$. In the first case, the faulty execution trace t_{-1} forces $v_0 = 0$, $v_1 = c_0$, $v_2 = c_1$, and $o_0 = c_1 * \mathbf{error}(c_2)$. In the second case, for all faulty execution traces t_{-2} the program variables evaluate to $v_0 = 1$, $v_1 = d'$, $v_2 = d'$, and $o_0 = d' * \mathbf{error}(c_2)$ with $d' = \mathsf{Int}\backslash\{0, c_0\}$. The distances of the faulty and the correct execution traces are $d(t_+, t_{-1}) = 4$ and $d(t_+, t_{-2}) = 3$ if $d' = c_1$, respectively. Thus, error explanation will generate three fault candidates for v_1, v_2, and o_0.

We construct a sequence Q_k, $k \in \mathbb{N}$, of faulty programs from Q_0. We systematically add statements of the form $w_i := w_{i-1} + g_i$, $1 \leq i \leq k$, to Q_0 between $\mathsf{s}(l_1)$ and $\mathsf{s}(l_2)$ to obtain Q_k where w_i are new program variables and g_i are constants. Additionally, we define $w_0 := v_1$ and replace v_1 by w_k in $\mathsf{s}(l_4)$ to connect the added statements to the rest of the program. For fault localization with MBD the same argument applies: we can choose constants c_0, c_1, c_2, and $\mathbf{error}(c_2)$ such that there is only one fault candidate which considers correction at the output variable. Error explanation reports by construction one fault candidate for each added statement. There is one correct execution trace for Q_k if the program is executed on input $i_0 = -\sum_{i=1}^{k} g_i - c_0$ which forces $w_j = \sum_{i=j}^{k} g_i$. We distinguish two cases of faulty execution traces: (1) $i_0 = 0$ and (2) $i_0 = d$ with $d \in \mathsf{Int}\backslash\{0, -\sum_{i=1}^{k} g_i - c_0\}$. In the first and the latter case, $w_j = c_0 + \sum_{i=1}^{j} g_j$ and $w_j = d + c_0 + \sum_{1}^{j} g_i$, respectively. Thus, the distance between the correct and the faulty execution traces are guaranteed to grow with k for non-trivial g_i, $1 \leq i \leq k$.

B. Error Explanation can be superior to MBD

Consider the faulty program R_0 from Figure 2. There are several correct and faulty execution traces depending on the value assigned to the input i_1. If $i_1 \neq 0$ the program results in failure and otherwise if $i_1 = 0$ the program's output is correct with respect to its reference implementation R_0^C because the erroneous constant value at l_4 does not propagate to the output variable. MBD computes one fault candidate for every statement. A program can always be correct at its output

variable and thus MBD computes one fault candidate for $\mathsf{s}(l_5)$. Additionally, MBD computes fault candidates for $\mathsf{s}(l_0)$, $\mathsf{s}(l_1)$, $\mathsf{s}(l_2)$, and $\mathsf{s}(l_4)$, i.e., MBD can replace v_1 to avoid executing the fault and replace v_0 or v_2 to correct the output value.

Error explanation selects a faulty and a correct execution trace with minimal distance and compares them to compute fault candidates. For all faulty execution traces t_-, the input variable $i_0 = y_0$ and $i_1 = 0$ force $v_0 = y_0 + c_0$, $v_1 = 0$, $v_2 = \mathbf{error}(c_2)$, and $o_0 = y_0 + c_0 + \mathbf{error}(c_2)$, where y_0 is an open variable. For all correct execution traces t_+, the input variables $i_0 = y_1$ and $i_1 = d$, $d \in \mathsf{Int}\backslash\{0\}$ which forces $v_0 = i_0 + c_0$, $v_1 = d$, $v_2 = c_1$ and $o_0 = i_0 + c_0 + c_1$, where y_1 is an open variable. The distance d between a faulty t_- and a correct execution trace t_+ is minimal if $y_0 = y_1$, i.e., $d(t_+, t_-) = 3$. Thus, error explanation generates the three fault candidates $\mathsf{s}(l_1)$, $\mathsf{s}(l_4)$, and $\mathsf{s}(l_5)$.

Again, we construct a sequence (R_k), $k \in \mathbb{N}$, of faulty programs from R_0. We systematically add statements of the form $w_i := w_{i-1} + g_i$, $1 \leq i \leq k$, to R_0 between $\mathsf{s}(l_0)$ and $\mathsf{s}(l_1)$ to obtain R_k where w_i are new program variables and g_i are constants. We connect the new statements to the rest of the program: we define $w_0 = v_0$ and replace v_0 in $\mathsf{s}(l_5)$ by w_k. MBD computes a fault candidate for each added statement, i.e., all w_i are added up to the output variable o_0 in $\mathsf{s}(l_5)$ and, thus, each w_i can be used to correct the value of o_0. However, when error explanation is used the values of w_i are equal for all faulty and correct execution traces when executed with the same input i_0. Thus, error explanation computes no new fault candidates for the added statements.

V. Case Study

In this section, we compare both fault localization techniques experimentally in a case study using the *Traffic Collision Avoidance System* (TCAS) from SIR [16]. TCAS is an imperative, non-concurrent ANSI-C program which implements a collision avoidance system for aircraft in 135 lines[1]. SIR provides 41 mutants of TCAS[2]. Each mutant corresponds to the correct program with injected faults. The mutant contains a simple mistake with respect to the correct program. These mistakes may refer to a single or multiple faulty statements.

For the comparison, we compute fault candidates for each mutant of TCAS with both fault localization techniques. All our experiments were conducted on a PC AMD Phenom™ II X4 Processor which has 4 cores with 3 GHz each and 8 GB RAM. We use the 64-bit version of Boolector 1.4.1 as SMT solver. Our prototype tool interacts with Boolector via API calls. In Section V-A we describe the implementation of the algorithms, MBD and BEST-EXPLAIN, and in Section V-B we list the results for the mutants of TCAS.

A. Implementation

We implemented the algorithms, MBD and BEST-EXPLAIN, into a C++ application using the libraries Boost 1.4.9, *Low Level Virtual Machine* (LLVM) 3.0 [23], and metaSMT pre-release 4 [24]. The preprocessing, previously denoted by `Preprocess`, leverages the LLVM compiler infrastructure. We use LLVM's C/C++ compiler front-end to translate the ANSI-C program first into LLVM *bitcode*, i.e., a RISC-like intermediate representation similar to the program model introduced in Section II-A. The LLVM compiler infrastructure provides transformations that can be used to establish SSA form.

Our logic encoding, previously denoted by `Encode`, is based on QF_BV logic rather than propositional logic. The QF_BV logic enriches the syntax and semantics of propositional logic with bit-vectors and, additionally, defines word-level operators on the bit-vectors. For instance, an LLVM instruction which adds two 32-bit integers can be formalized using two bit-vectors of length 32 and the addition operator defined on bit-vectors. The logic encoding is similar to Section II-B, i.e., we define an encoding for each LLVM instruction type.

Logic formulae are checked for satisfiability using an SMT solver which supports QF_BV logic. In practice SMT solvers are effective for solving logic formulae of moderate to large size. We use metaSMT as a generic interface to different SMT solvers which either allows for dumping a file in the SMT-LIB version 2 format or for interacting with a solver via API calls.

B. Experimental Results

Model checking the correct TCAS program with our prototype tool takes 3.36 seconds. Table I presents the fault localization results for all 41 mutants of TCAS with respect to both fault localization techniques. The table is built as follows: the first column names the benchmark. The second column classifies the change with respect to the fault model introduced in Section II-D. For simple changes, we use the first letter of the mutation as abbreviation, i.e., `O` denotes an operator mutation, `L` denotes a logic mutation, `V` denotes a variable mutation, `C` denotes a constant mutation, `A` denotes additional code, and `M` denotes missing code. Moreover, we use $\star$ to indicate a complex change. In order to assess the quality of the fault candidates we use an approach similar to Renieris and Reiss [11], i.e., we compute the distance of the fault candidates to the real faulty program locations on the PDG constructed from the mutant. The distance refers to the length of the cause-effect chain the programmer has to examine until the real fault is found. We built the PDG for all mutants. The third column lists the diameter of the PDG. The rest of the table is split into 4 parts each having 5 columns. The first three parts list the results for MBD using 1, 5, and 10 faulty execution traces and the last part lists the results for BEST-EXPLAIN. For each of those parts, we present the number FC of reported fault candidates, the time t in seconds for computing the fault candidates, and the minimal, maximal and average distance of the fault candidates to a real faulty program location. We marked the faulty program locations on the PDG manually. For simple changes, we marked the statement that differ between the mutant and the correct TCAS program. For complex changes, we marked the statements which in our opinion correspond to the root cause of all failures. The minimal and

[1] All source lines are counted using the Unix tool `sloccount`.

[2] The source code is publicly available on http://sir.unl.edu/portal/index.php.

TABLE I
FAULT LOCALIZATION FOR THE ANSI-C PROGRAM TCAS

Program			MBD(1)					MBD(5)					MBD(10)					BEST-EXPLAIN				
Id	Type	ϕ	FC	t	Min	Max	Avg	FC	t	Min	Max	Avg	FC	t	Min	Max	Avg	FC	t	Min	Max	Avg
tcas(01)	O	29	69	6.18	0	8	5	69	14.40	0	8	5	69	26.76	0	8	5	14	79.79	1	7	4
tcas(02)	O	29	75	4.39	0	7	4	75	11.73	0	7	4	68	23.60	0	7	4	3	27.52	5	5	5
tcas(03)	L	29	71	3.93	0	9	4	71	11.98	0	9	4	71	32.42	0	9	4	3	13.01	3	5	4
tcas(04)	L	29	71	4.36	0	8	4	64	12.27	0	8	4	64	18.95	0	8	4	17	23.68	2	7	5
tcas(05)	M	29	62	3.37	0	10	5	62	11.60	0	10	5	62	23.95	0	10	5	4	9.75	5	7	5
tcas(06)	O	29	66	4.99	0	6	4	59	11.50	0	6	3	59	27.94	0	6	3	6	19.47	1	4	2
tcas(07)	C	29	43	3.34	0	8	5	43	8.91	0	8	5	43	20.46	0	8	5	5	23.31	2	4	3
tcas(08)	C	29	70	5.14	0	8	4	63	11.30	0	8	4	63	23.92	0	8	4	5	26.95	2	4	3
tcas(09)	O	29	31	2.26	0	7	4	31	7.71	0	7	4	31	17.11	0	7	4	8	28.84	1	5	3
tcas(10)	O	29	83	5.99	0	6	3	76	14.43	0	6	3	86	26.42	0	6	3	6	21.95	1	4	2
tcas(11)	*	28	63	4.03	0	6	3	63	9.94	0	6	3	56	24.91	0	6	3	6	20.50	1	4	2
tcas(12)	L	29	67	3.67	0	10	5	67	11.62	0	10	5	59	27.92	0	10	5	3	14.66	4	4	4
tcas(13)	C	29	65	3.73	0	10	5	65	12.86	0	10	5	58	22.24	0	10	5	4	13.31	1	4	2
tcas(14)	C	29	12	1.27	0	5	2	12	04.03	0	5	2	12	8.60	0	5	2	3	23.73	5	6	5
tcas(15)	A	29	56	3.40	0	7	3	56	10.92	0	7	3	56	23.71	0	7	3	4	11.41	1	4	2
tcas(16)	C	29	43	3.37	0	8	5	43	8.56	0	8	5	43	21.34	0	8	5	5	19.27	2	4	3
tcas(17)	C	29	71	4.21	0	8	4	71	10.82	0	8	4	64	27.90	0	8	4	5	19.44	2	4	3
tcas(18)	C	29	43	3.45	0	8	5	43	9.41	0	8	5	43	17.01	0	8	5	5	27.27	2	4	3
tcas(19)	C	29	72	4.47	0	8	4	65	11.04	0	8	4	68	29.17	0	8	4	5	23.56	2	4	3
tcas(20)	O	29	75	5.55	3	8	5	68	11.57	3	8	5	68	26.12	3	8	5	19	39.41	3	7	5
tcas(21)	*	27	31	3.04	2	7	5	31	8.58	2	7	5	31	18.27	2	7	5	9	40.40	4	5	4
tcas(22)	*	27	25	2.79	0	6	3	25	7.67	0	6	3	25	16.86	0	6	3	9	62.59	3	5	4
tcas(23)	*	26	64	3.63	0	6	4	64	10.04	0	6	4	57	21.68	0	6	4	8	30.45	2	4	3
tcas(24)	*	26	31	3.03	2	8	5	31	8.35	2	8	5	31	18.90	2	8	5	8	34.72	4	5	4
tcas(25)	O	29	44	4.76	0	8	4	44	10.40	0	8	4	44	20.67	0	8	4	9	23.58	5	8	6
tcas(26)	*	29	67	3.75	0	9	5	67	11.97	0	9	5	60	25.50	0	9	5	3	10.59	4	5	4
tcas(27)	M	29	62	3.36	0	10	5	62	11.61	0	10	5	62	23.81	0	10	5	4	12.02	4	6	4
tcas(28)	A	29	74	4.39	0	7	4	74	11.65	0	7	4	67	25.60	0	7	4	3	27.99	5	5	5
tcas(29)	*	28	70	3.75	2	8	5	70	11.57	2	8	5	63	27.64	2	8	5	3	20.64	4	5	4
tcas(30)	*	28	78	4.27	2	8	5	78	13.66	2	8	5	71	25.17	2	8	5	3	22.75	4	5	4
tcas(31)	*	20	56	3.72	2	7	3	56	11.41	2	7	3	49	22.54	2	7	3	13	14.39	1	4	2
tcas(32)	*	28	58	4.19	0	6	3	51	9.82	0	6	2	51	24.43	0	6	2	15	17.52	1	5	3
tcas(33)	*	29	66	6.15	0	8	4	66	13.83	0	8	4	66	26.02	0	8	4	5	39.54	2	4	3
tcas(34)	A	30	58	3.79	0	9	5	58	10.24	0	9	5	58	28.92	0	9	5	9	15.20	1	6	4
tcas(35)	*	29	74	4.38	0	7	4	74	11.63	0	7	4	67	25.59	0	7	5	3	22.54	5	5	5
tcas(36)	C	29	15	1.28	0	6	3	15	4.60	0	6	3	15	10.55	0	6	3	18	50.16	0	7	3
tcas(37)	V	19	66	2.84	2	9	5	59	7.59	2	9	5	59	17.50	2	9	5	3	18.48	4	5	4
tcas(38)	*	27	43	2.56	2	9	5	43	6.19	2	9	5	43	12.41	2	9	5	5	19.70	5	5	5
tcas(39)	O	29	44	4.76	0	8	4	44	10.40	0	8	4	44	20.65	0	8	4	9	31.34	5	8	6
tcas(40)	M	20	39	2.37	2	8	5	39	7.21	2	8	5	39	15.84	2	8	5	9	15.55	2	4	3
tcas(41)	M	27	72	4.14	2	8	5	65	10.47	2	8	5	65	28.21	2	8	5	16	37.36	4	8	6

maximal distance refer to the shortest and longest shortest path of the PDG from a fault candidate to any marked faulty program location. The average distance is the arithmetic mean of all shortest paths of the PDG from a fault candidate to any marked faulty program location.

For all 41 mutants of TCAS, MBD(1) computes the fault candidates faster than BEST-EXPLAIN but the computed fault candidate sets are always larger. Using more counterexample hardly reduces the number of reported fault candidates, i.e., the counterexamples provided by the SMT solver are not guaranteed to activate different control paths.

The algorithm MBD localizes most faults exactly producing a fault candidate with minimal distance 0. However, our fault model considers multiple faults and thus MBD is not guaranteed to compute a fault candidate which pinpoints the fault when only one abnormal predicate is enabled. This applies to complex changes and simple changes when a single faulty location in ANSI-C transforms to multiple faults in the LLVM bitcode.

The explanation-based algorithm, BEST-EXPLAIN, tends to avoid the execution of the faulty program locations. For instance, the values of a condition evaluate to true in the correct execution trace and false in the faulty execution trace. Notice that the program is in SSA form. If a program variable is not assigned because a statement was not executed, the value of the program variable is non-deterministic. The logic variables corresponding to this program variable become trivially equal when searching for a pair of a faulty and a correct execution trace with minimal distance. Thus, BEST-EXPLAIN does not report fault candidates with minimal distance 0 if the correct execution trace does not execute the faulty statement. As a result BEST-EXPLAIN rarely computes fault candidates with minimal distance 0.

VI. CONCLUSIONS

In this paper, we compared MBD-based and explanation-based fault localization techniques. We adapted the fault localization techniques to provide a unified setting and presented algorithms, called MBD and BEST-EXPLAIN, for formalizing them into instances of the satisfiability problem. The inputs of the algorithms are the source code of an imperative, non-concurrent program written in a programming language like ANSI-C and an unrolling bound. The output is a set of

potentially faulty statements, called fault candidates, which need to be manually examined.

Firstly, we explored the strengths and weaknesses of the two fault localization techniques using constructed programs. We showed that the fault candidates computed with the two techniques can be very different with respect to the program. Secondly, we compared the two techniques experimentally in a case study using 41 mutants of the TCAS program. We implemented both fault localization techniques in a prototype tool leveraging the LLVM compiler infrastructure to encode the program into a logic formula and an SMT solver to decide satisfiability of the logic formula. We assessed the quality of the computed fault candidates with respect to both fault localization techniques by counting their distance to the real faulty program locations on the PDG.

REFERENCES

[1] I. Vessey, "Expertise in debugging computer programs: An analysis of the content of verbal protocols," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 16, no. 5, pp. 621–637, 1986.

[2] R. Reiter, "A theory of diagnosis from first principles," *Artificial Intelligence*, vol. 32, no. 1, pp. 57–95, 1987.

[3] J. de Kleer and B. C. Williams, "Diagnosing multiple faults," *Artificial Intelligence*, vol. 32, no. 1, pp. 97–130, 1987.

[4] W. Mayer, M. Stumptner, and F. Wotawa, "Model-based debugging or how to diagnose programs automatically," in *International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, 2002, pp. 746–757.

[5] A. Smith, A. Veneris, M. F. Ali, and A.Viglas, "Fault diagnosis and logic debugging using boolean satisfiability," *IEEE Transactions on CAD*, vol. 24, no. 10, pp. 1606–1621, 2005.

[6] B. Jobstmann, A. Griesmayer, and R. Bloem, "Program repair as a game," in *Conference on Computer Aided Verification*, 2005, pp. 226–238.

[7] G. Fey, S. Staber, R. Bloem, and R. Drechsler, "Automatic fault localization for property checking," *IEEE Transactions on CAD*, vol. 27, no. 6, pp. 1138–1149, 2008.

[8] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 193–200, 2002.

[9] A. Groce and W. Visser, "What went wrong: Explaining counterexamples," in *International Conference on Model Checking Software*, 2003, pp. 121–136.

[10] T. Ball, M. Naik, and S. K. Rajamani, "From symptom to cause: Localizing errors in counterexample traces," in *Symposium on Princples of Programming Languages*, 2003, pp. 97–105.

[11] M. Renieris and S. P. Reiss, "Fault localization with nearest neighbor queries," in *IEEE International Conference on Automated Software Engineering*, 2003, pp. 30–39.

[12] H. Jin, K. Ravi, and F. Somenzi, "Fate and free will in error traces," *International Journal on Software Tools for Technology Transfer*, vol. 6, no. 2, pp. 102–116, 2004.

[13] A. Groce, S. Chaki, D. Kröning, and O. Strichman, "Error explanation with distance metrics," *International Journal on Software Tools for Technology Transfer*, vol. 8, no. 3, pp. 229–247, 2006.

[14] D. Köb and F. Wotawa, "A comparison of fault explanation and localization," in *International Workshop on Principles of Diagnosis*, 2005, pp. 157–162.

[15] G. Fey, A. Sülflow, and R. Drechsler, "Towards unifying localization and explanation for automated debugging," in *International Workshop on Microprocessor Test and Verification*, 2010, pp. 3–8.

[16] H. Do, S. G. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Software Engineering: An International Journal*, vol. 10, no. 4, pp. 405–435, 2005.

[17] D. Kröning, "Software verification," in *Handbook of Satisfiability*, A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds. IOS Press, 2009, pp. 505–532.

[18] B. Alpern, M. N. Wegman, and F. K. Zadeck, "Detecting equality of variables in programs," in *Symposium on Princples of Programming Languages*, 1988, pp. 1–11.

[19] B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Global value numbers and redundant computations," in *Symposium on Princples of Programming Languages*, 1988, pp. 12–27.

[20] G. S. Tseitin, "On the complexity of derivation in propotional calculus," in *Automation and Reasoning: Classical Papers in Computational Logic 1967-1970*, 1983, originally published in 1970.

[21] D. Lewis, "Causation," *Journal of Philosophy*, vol. 70, pp. 556–567, 1973.

[22] R. W. Hamming, "Error detecting and error correcting codes," *Bell System Technical Journal*, vol. 29, no. 2, pp. 147–160, 1950.

[23] C. Lattner, "LLVM: An infrastructure for multi-stage optimization," Master's thesis, University of Illinois at Urbana-Champaign, 2002.

[24] F. Haedicke, S. Frehse, G. Fey, D. Große, and R. Drechsler, "metaSMT: Focus on your application not on solver integration," in *International Workshop on Design and Implementation of Formal Tools and Systems*, 2011, pp. 22–29.