Automated Error Localization and Correction for Imperative Programs

Robert Könighofer and Roderick Bloem
Institute for Applied Information Processing and Communications (IAIK),
Graz University of Technology

***Abstract*—In this paper, we present a novel debugging method for imperative software, featuring both automatic error localization and correction. The input of our method is an incorrect program and a corresponding specification, which can be given in form of assertions or as a reference implementation. We use symbolic execution for program analysis. This allows for a wide range of different trade-offs between resource requirements and accuracy of results. Our error localization method rests upon model-based diagnosis and SMT-solving. Error correction is done using a template-based approach which ensures that the computed repairs are readable. Our method can handle all sorts of incorrect expressions, not only under a single-fault assumption but also for multiple faults. Finally, we present experimental results, where an implementation for C programs is used to debug mutants of the TCAS case study of the Siemens suite.**

I. Introduction

A lot of research has been done in the past decades to automate detection of errors in programs, be it software or hardware. But once an error is detected, the hard work only begins: the error has to be located and corrected. This is usually done manually, which is time-consuming, costly, frustrating, and increases time-to-market. More and better automation in these steps is definitely needed.

Many existing approaches, especially for automatic error correction, are based on fully formal methods with limited scalability when it comes to larger state spaces. On the other hand, simulation-based methods suffer from limited accuracy. Trade-offs are usually not possible. Another often insufficiently addressed issue is that synthesized corrections have to be readable. A method which produces repairs as Boolean functions that cannot be understood is of limited use, because the repaired program cannot be maintained. Furthermore, repairs should affect only small parts of the program. This lowers the chances that unspecified properties of the program get lost.

We propose a novel method for automatic error localization and correction, explicitly addressing all these challenges. It is outlined in Fig. 1. An incorrect program and its specification are the inputs. The specification may be given via assertions in the code or as a reference implementation. First, we preprocess the program to express that components may be faulty. Our fault model can handle all kinds of incorrect expressions and is not restricted to single-faults. Next, symbolic execution is applied to transform the debugging problem into the domain of logic. Error localization rests upon model-based diagnosis [27], [24]. It computes sets of components of the program which can be replaced in such a way that the program becomes correct. In the correction step, such replacements are finally computed. This is done using a template-based approach, where repairs are iteratively refined. This refinement is guided by counterexamples. All reasoning is done with a Satisfiability Modulo Theories (SMT) solver.

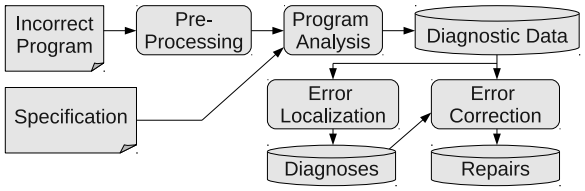

Fig. 1. The flow of our debugging method.

Symbolic execution is a semi-formal technique to analyze program behavior path-by-path. This makes our method degrade nicely: considering more paths means better accuracy but also higher resource requirements. Error localization and correction provide additional parameters for trade-offs. Fine-grained error localization leads to repairs that affect only small program parts. Readable repairs are obtained by restricting the search to templates for expressions. Multiple templates can be used successively: starting with a simple template, one can switch to a more expressive one if no repair is found. Within a certain template, we use heuristics to find simple instantiations first. Our debugging method is especially suited for debugging simple pieces of software, e.g., C programs modeling hardware designs at a high abstraction-level.

Model-based diagnosis [27], [24] has already been applied to locate errors in logic programs [7], functional programs [32], VHDL designs [14], Java programs [26], knowledge bases [11], ontologies [13], and temporal logic specifications [25]. We apply the technique to an abstraction of a program found by symbolic execution and combine it with error correction. In [18], errors are located using a model-checker. Our diagnosis approach is similar, but we require only one symbolic execution pass rather than several model-checker calls, and we compute diagnoses differently. Another related

This work was supported in part by the European Commission through project DIAMOND (FP7-2009-IST-4-248613).

978-0-9835678-1-3/11/$26.00 ©2011 IEEE

diagnosis method is presented in [21]. It computes a maximal set of statements that can remain unchanged for the program to become correct for a given input. The complement forms the diagnosis. This is encoded as a Maximum Satisfiability (MAX-SAT) problem. In contrast, our method allows to use several failure-inducing inputs simultaneously; a diagnosis must allow to fix the program for *all* these inputs.

One approach to program repair is to transform a finite-state program into a finite-state game and to compute a repair as a strategy in this game [19], [20]. This has also been extended to programs with virtually infinite state space (i.e., software) using predicate abstraction [17]. In contrast, we use symbolic execution, a technique especially suitable for software.

Our repair method is very related to program sketching [30], [29], a paradigm where the user provides a program with unknown parts ("holes") and a specification. A tool synthesizes the holes automatically. For complex unknowns, the user has to provide so-called "generators", which are functions containing only unknown integer values. These generators serve the same purpose as our repair templates: reducing the synthesis of components to the search for integer constants. The main difference is that our method works in a push-button manner, i.e., templates do not have to be (but can be) provided by the user. Moreover, in the repair setting, holes have to be computed first, they should be small, and their implementation readable. Templates for expressions are also used to synthesize loop invariants for program verification [6]. The differences to our approach for computing repairs lie in the constraints that have to be fulfilled (program correctness rather than inductiveness) and how instances are computed (iterative refinement rather than quantifier elimination). The idea of synthesizing parts of a program by iterative, counterexample-guided refinements has already been used in [3] and [30]. We extend the basic idea with a heuristic to speed up convergence. Program Synthesis is also addressed in [31], where imperative programs are synthesized from a given specification and flowgraph structure. This work uses program a verification tool performing a fixed-point computation to synthesize a solution.

A quite different repair approach is to repeatedly mutate an incorrect program and check if it becomes correct [8]. The problem is the huge search space for mutants. Our repair method is much more systematic. Finally, there are genetic programming methods, combining mutation with crossing and selection according to some notion of fitness [1], [12].

In summary, the contributions of this paper are as following.

- We present a new debugging approach which produces readable repairs at the source level and degrades nicely.
- We combine many existing techniques in a novel way: symbolic execution, model-based diagnosis, templates for unknown expressions, and iterative repair refinement.
- We show how model-based diagnosis can be applied to a program abstraction found by symbolic execution.
- We present a heuristic to speed up repair refinement.
- Finally, we present experimental results using an implementation of our debugging approach for C programs.

This paper is organized as follows. Section II explains techniques underlying our approach and establishes notation. Section III presents our debugging method as outlined in Fig. 1. Section IV discusses alternatives and trade-offs. Section V presents first experimental results, and Section VI concludes.

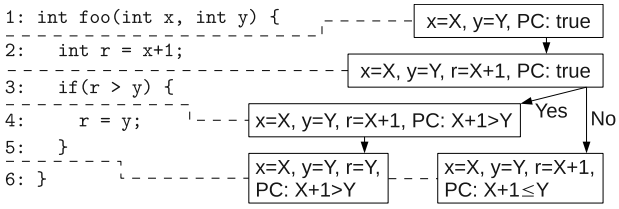

Fig. 2. Example: Symbolic execution.

II. Preliminaries

A. Symbolic Execution

Symbolic execution [5], [23] is a program analysis technique. It executes a program with symbols as inputs. Symbols are placeholders that can take on any value in some domain. Symbolic execution keeps track of the symbolic values (expressions involving symbols and constants) of all program variables. Whenever a branching point is reached, the execution forks. For every branch, a condition expressing when it is taken is computed. Along an execution path, the branch conditions are accumulated to a path condition. Thus, a path condition states when a certain execution path is activated. In practice, the maximum path length and the number of paths to analyze are limited to ensure termination.

Example 1. *Fig. 2 illustrates symbolic execution on an example. Two symbols X and Y are used for the unknown values of x and y. Boxes contain execution states, dashed lines link them to the program, and arrows indicate the execution flow. In Line 3, the execution forks since both branches are feasible. The condition which has to be fulfilled for the program to reach a certain state is denoted as PC. The path conditions can be read from the PC-fields in the leaves of the tree. This program has two paths with conditions $X + 1 > Y$ and $X + 1 \leq Y$.*

Concolic execution [15], [28] is a variant of symbolic execution where the program is executed using concrete and symbolic inputs simultaneously. The execution path is determined by the concrete values. Along this path, symbolic variable values are tracked and a path condition is computed. After one execution, the conjuncts of the path condition are used to compute concrete input values that trigger a different path. For our purposes, concolic execution produces the same outcomes as symbolic execution, namely path conditions.

B. Model-Based Diagnosis

Model-based diagnosis [24], [27] (MBD) is a method to locate errors in a system by explaining conflicts between a model of the system and an observation of some incorrect behavior. We follow the notation of [27] in this work.

Let SD be a model of a system, and let OBS be an observation of an erroneous behavior, both given as sets of logical

sentences. The system consists of a set of components CMP. A component $c \in \mathsf{CMP}$ can behave abnormally (denoted $\mathsf{AB}(c)$) or normally (written $\neg\,\mathsf{AB}(c)$). Every component c is described with a logical sentence of the form $\neg\,\mathsf{AB}(c) \Rightarrow N_c$, with N_c defining the normal behavior of c. That is, abnormal components can behave arbitrarily. The system description SD is composed of component descriptions and a set of logical sentences defining the interplay of components. The observation OBS contradicts SD in the sense that, if all components behaved normally, it would be impossible to observe OBS. That is, the set $\mathsf{SD} \cup \mathsf{OBS} \cup \{\neg\,\mathsf{AB}(c) \mid c \in \mathsf{CMP}\}$ of logical sentences is inconsistent, i.e., contains a logical contradiction.

MBD computes diagnoses, which are sets of components that may be responsible for observation OBS. Formally, a set $\Delta \subseteq \mathsf{CMP}$ is a *diagnosis* iff $\mathsf{SD} \cup \mathsf{OBS} \cup \{\neg\,\mathsf{AB}(c) \mid c \in \mathsf{CMP} \setminus \Delta\}$ is consistent. The components in Δ may be responsible for OBS because assuming that these components behave abnormally renders OBS possible. A diagnosis Δ is *minimal* if no subset $\Delta' \subset \Delta$ is a diagnosis. If Δ is a diagnosis, then clearly every $\Delta' \supseteq \Delta$ is a diagnosis as well. Hence, we are only interested in minimal diagnoses.

Diagnoses can be computed via conflicts. A *conflict* is a set $C \subseteq \mathsf{CMP}$ of components such that $\mathsf{SD} \cup \mathsf{OBS} \cup \{\neg\,\mathsf{AB}(c) \mid c \in C\}$ is inconsistent. I.e., a conflict is a set of components that cannot all have behaved normally. A conflict C is *minimal* if no subset $C' \subset C$ is a conflict. A *hitting set* for a collection $\mathcal{K}$ of sets is a set H such that $\forall K \in \mathcal{K} \,.\, H \cap K \neq \emptyset$ holds. A hitting set H is *minimal* if no subset $H' \subset H$ is a hitting set. A set $\Delta \subseteq \mathsf{CMP}$ is now a minimal diagnosis iff Δ is a minimal hitting set for all conflicts. The intuitive reason is that a diagnosis must explain all conflicts, so it must share at least one element with every conflict. Reiter [27] presents a hitting set computation algorithm which computes conflicts on-the-fly and produces diagnoses in order of increasing cardinality.

C. Notation of Vectors and Domains

We write $\mathcal{S}$ for the set of finite strings. Overlines are used to indicate vectors. For two vectors $\overline{a} = (a_1, \ldots, a_m)$ and $\overline{b} = (b_1, \ldots, b_n)$, we write $\overline{a}||\overline{b}$ for the concatenation $(a_1, \ldots, b_n)$. For symbolic execution, we assume that all symbols are taken from a sufficiently large set $\mathbb{S}$. To simplify notation, we also assume that all symbols range over some domain $\mathbb{D}$ of values. For instance, $\mathbb{D}$ may be $\mathbb{Z}$ or $\mathbb{B}^m$. An extension to different domains for different symbols is straightforward.

We denote with D_{ex} a domain of symbolic expressions and with D_{co} a domain of symbolic conditions. Let $e_1, e_2 \in D_{ex}$ and $c_1, c_2 \in D_{co}$. We assume that $e_1 = e_2$, $e_1 \leq e_2$, $e_1 \geq e_2$, $c_1 \vee c_2$, $c_1 \wedge c_2$, and $\neg c_1$ are in D_{co} as well, with the expected semantics ("$=$" means equality here, not an assignment). For $c \in D_{co}$ we write $c[\overline{a}]$ with $\overline{a} = (a_1, \ldots, a_m) \in \mathbb{S}^m$ to indicate that c may only depend on the symbols $a_1, \ldots, a_m$. Let $\overline{b} = (b_1, \ldots, b_m) \in (\mathbb{S} \cup \mathbb{D})^m$ be a vector of symbols and constants. Then $c[\overline{b}]$ denotes condition $c[\overline{a}]$ where all symbols a_i, with $1 \leq i \leq m$, have been replaced by b_i. Analogously for expressions. Finally, we assume that a sound and complete decision procedure (e.g., an SMT-solver) for the satisfiability of conditions $c \in D_{co}$ is available.

III. Debugging Method

This section introduces our debugging approach, as outlined in Fig. 1, in more detail. A discussion of practical aspects and some alternatives will be given in Section IV. The input of our debugging method is a program P and a specification S. The program may contain calls to a special function `input`, returning an unknown input value $v \in \mathbb{D}$. As a specification, assertions in the code are supported natively. This also allows using reference implementations: the reference implementation is executed with the same inputs and results are compared with assertions. The user can define the desired notion of equivalence with suitable assertions. The program is assumed to violate the specification for some input.

A. Pre-Processing

Our method needs to report components of the program P as possibly faulty, and to suggest replacements. This requires a notion of a component and a corresponding fault model.

The ideal fault model can explain all errors, is fine-grained, and enables efficient algorithms. Clearly, these properties cannot all be maximized at the same time. As a trade-off, we assume that only the right-hand side (RHS) of assignments may be erroneous. Alternatives will be discussed in Section IV. The reasons for our decision are as follows. Assignments are the fundamental operation in any imperative program. The fault model allows for efficient program analysis, since an unknown RHS can be handled symbolically, just like an input. (See Section III-B.) Moreover, it is fine-grained and can easily be extended to incorrect expressions by assigning every expression to a temporary variable. Consequently, we consider the RHS of every assignment as a replaceable component of the program. The rest of the program is deemed unmodifiable.

Example 2. *Consider the following program (in C syntax).*

```
1  int max(int x, int y) {
2    int r = x;
3    if(y > x)
4      r = x;
5    assert(r >= x && r >= y);
6    return r;
7  }
```

It will serve as a running example. It should compute the maximum of two numbers but contains an error in Line 4, which should read "`r = y`". With $\mathsf{CMP} = \{c_1, c_2\}$ *we identify two components,* c_1 *being the* `x` *in Line 2 and* c_2 *being the* `x` *in Line 4. By re-writing Line 3 to "`int tmp = y > x; if(tmp)`", the condition* `y > x` *can be handled as a third component* c_3*. However, for simplicity of explanations,* c_3 *is not considered as a component in the running example.*

Next, we express that components may be faulty. We create a modified program $\tilde{P}$, which is identical to P except that each assignment `LHS = RHS;` is textually replaced by

```
LHS = cmp(c, RHS, v1, v2, ..., vn);
```

Here, cmp is a special function indicating components, c is (a unique identifier of) the component $c \in \mathsf{CMP}$, and v1, ..., vn are the variables which are in scope when component c is executed. We can think of cmp as a shorthand for

```
assumeCorrect(c) ? RHS : rep_c(v1,...,vn).
```

If component c is assumed to be correct, there is no need to modify it. Otherwise it can be replaced by a new expression, which is embodied by the (yet unknown) function rep_c. The error localization step will find out which components are assumed to be incorrect. The error correction step will compute implementations of the functions rep_c for all incorrect components.

We define a function $\mathrm{Vars} : \mathsf{CMP} \to \mathcal{S}^*$ mapping each component $c \in \mathsf{CMP}$ to a vector of variable names v1, ..., vn. These are the variables in scope when c is executed. The mapping can be computed easily during pre-processing and will be required for applying and outputting repairs.

Example 3. *The program P from Example 2 gives $\tilde{P} =$*

```
1 int max(int x, int y) {
2   int r = cmp(1,x,x,y);
3   if(y > x)
4     r = cmp(2,x,x,y,r);
5   assert(r >= x && r >= y);
6   return r;
7 }
```

where the calls to cmp intuitively mean
assumeCorrect(c1) ? x : rep_c1(x,y) and
assumeCorrect(c2) ? x : rep_c2(x,y,r).

B. Program Analysis

We execute $\tilde{P}$ symbolically or concolically to get diagnostic information. Unlike standard symbolic execution, we maintain two sets of symbols: input symbols and repair symbols.

Input symbols represent unknown input values. Whenever the function input is called, a fresh input symbol is created. We denote the vector of created input symbols (in arbitrary order) as $\bar{i} = (i_1, \ldots, i_A) \in \mathbb{S}^A$. Similarly, repair symbols represent the unknown values returned by the function cmp. These symbols will be denoted as $\bar{r} = (r_1, \ldots, r_B) \in \mathbb{S}^B$.

With the following functions, every repair symbol r_b is associated with additional information. $\mathrm{CmpOf} : \{r_1, \ldots, r_B\} \to \mathsf{CMP}$ maps r_b to the component which produced it. $\mathrm{Org} : \{r_1, \ldots, r_B\} \to D_{ex}$ maps r_b to the value that would be produced by the unmodified component $\mathrm{CmpOf}(r_b)$. $\mathrm{Vals} : \{r_1, \ldots, r_B\} \to D_{ex}^*$ maps r_b to the values of all variables in scope when $\mathrm{CmpOf}(r_b)$ produced r_b. These functions are built up from the symbolic values of the first parameter, the second parameter, and the subsequent parameters of cmp, respectively.

We say that an execution path is a sequence of statements which starts with the initial statement and ends with the termination of the program. For every execution path p in $\tilde{P}$, a path condition $\mathrm{PC}^p[\bar{i}||\bar{r}] \in D_{co}$ is created during the symbolic execution. The paths are divided into the two sets PASS and FAIL. A path p is in FAIL if it violates the specification, i.e., ends in an abnormal program termination after an assertion violation. It is in PASS otherwise. We define a condition $\pi[\bar{i}||\bar{r}] \in D_{co}$ as

$$\pi[\bar{i}||\bar{r}] = \bigvee_{p \in \mathrm{PASS}} \mathrm{PC}^p[\bar{i}||\bar{r}]. \quad (1)$$

Lemma 1. *Let $\overline{v_i} \in \mathbb{D}^A$ and $\overline{v_r} \in \mathbb{D}^B$ be two vectors of concrete values. Assuming that all paths of $\tilde{P}$ have been analyzed, the condition $\pi[\overline{v_i}||\overline{v_r}]$ is* true *iff $\tilde{P}$ fulfills the specification S, given that $\overline{v_i}$ is used as input vector and $\overline{v_r}$ is the vector of values returned in calls to the function cmp.*

Proof: Let p_e be the path activated by $\overline{v_i}$ and $\overline{v_r}$ in $\tilde{P}$. Clearly, $\mathrm{PC}^p[\overline{v_i}||\overline{v_r}]$ is true iff p=p_e. Lemma 1 holds since $\mathrm{PC}^{p_e}[\overline{v_i}||\overline{v_r}]$ is a disjunct of $\pi[\overline{v_i}||\overline{v_r}]$ iff p_e satisfies S. ■

Proposition 1. *Let P be an incorrect program and let $\overline{v_i} \in \mathbb{D}^A$ be an input vector. Assuming that all paths of $\tilde{P}$ have been analyzed, the condition*

$$\exists \bar{r} \,.\, \pi[\overline{v_i}||\bar{r}] \wedge \bigwedge_{r_b \text{ in } \bar{r}} r_b = \mathrm{Org}(r_b)[\overline{v_i}||\bar{r}]$$

is true *iff P fulfills the specification S when executed with input $\overline{v_i}$.*

Proof: Let $\overline{v_r} \in \mathbb{D}^B$ be the concrete values returned by calls to cmp in $\tilde{P}$. According to Lemma 1, $\pi[\overline{v_i}||\overline{v_r}]$ evaluates to true iff $\tilde{P}$ fulfills S for input $\overline{v_i}$. The additional conjuncts in the formula require that all components in $\tilde{P}$ return the same value as the respective components in P. Hence, the formula valuates to true iff P fulfills S for input $\overline{v_i}$. ■

Lemma 1 states how the condition $\pi[\bar{i}||\bar{r}]$ can be used to make statements about the correctness of the instrumented program $\tilde{P}$, depending on the inputs and the components. Proposition 1 establishes the link to the correctness of the original program P, using the information in Org.

Definition 1. *The tuple* $\Gamma = (\mathsf{CMP}, \mathrm{Vars}, \bar{i}, \bar{r}, \mathrm{CmpOf}, \mathrm{Org}, \mathrm{Vals}, \pi[\bar{i}||\bar{r}])$ *is called* diagnostic data.

Example 4. *For P from Example 2 we have* $\mathsf{CMP}=\{c_1, c_2\}$, $\mathrm{Vars}(c_1)=(x, y)$, $\mathrm{Vars}(c_2)=(x, y, r)$, $\bar{i}=(X, Y)$, $\bar{r}=(R_1, R_2)$, $\mathrm{CmpOf}(R_1)=c_1$, $\mathrm{CmpOf}(R_2)=c_2$, $\mathrm{Org}(R_1)=X$, $\mathrm{Org}(R_2)=X$, $\mathrm{Vals}(R_1)=(X, Y)$, $\mathrm{Vals}(R_2)=(X, Y, R_1)$, *and* $\pi[\bar{i}||\bar{r}] = (Y > X \wedge R_2 \geq X \wedge R_2 \geq Y) \vee (Y \leq X \wedge R_1 \geq X \wedge R_1 \geq Y)$.

The diagnostic data Γ is the output of the program analysis step and will serve as input for error localization and error correction (recall again Fig. 1).

C. Error Localization

Our method for error localization rests upon MBD as introduced in Section II-B. This section explains how MBD can be applied in our setting. The next section will then discuss how diagnoses can actually be computed.

Standard MBD takes as input a model of a system together with a contradicting observation. The contradiction manifests itself in conflicts, which need to be explained. In our setting, a program conflicts with its specification, so we need a different

notion of a conflict. Deriving diagnoses from conflicts works in the standard way.

We define a function $\mathsf{repairable} : 2^{\mathsf{CMP}} \to \{\mathsf{true}, \mathsf{false}\}$. Intuitively, $\mathsf{repairable}(Q)$ maps a set $Q \subseteq \mathsf{CMP}$ to true iff program $\tilde{P}$ can be repaired for all inputs, assuming that all components $c \in Q$ are correct and need not be modified. Formally, we define

$$\mathsf{repairable}(Q) \Leftrightarrow \forall \bar{i} \,.\, \exists \bar{r} \,.\, \pi[\bar{i}||\bar{r}] \wedge \bigwedge_{r \in R} r = \mathrm{Org}(r)[\bar{i}||\bar{r}], \quad (2)$$

where R stands for $\{r \mid \mathrm{CmpOf}(r) \in Q\}$ and $\forall \bar{i}$ is a shorthand for $\forall i_1 \ldots \forall i_A$. Likewise for $\exists \bar{r}$. The definition says that a program is repairable iff for all inputs, there exist values that can be returned by the components (the function `cmp` in $\tilde{P}$) such that the specification is fulfilled. Components which are assumed to be correct can only return the value that would be returned by the original version of that component.

Lemma 2. *The function* repairable *is monotonic in that, for all* $Q' \subseteq Q \subseteq \mathsf{CMP}$, $\mathsf{repairable}(Q)$ *implies* $\mathsf{repairable}(Q')$.

Monotonicity is obvious since removing elements from Q only removes conjuncts in the definition of repairable.

Definition 2. *A set* $\Delta \subseteq \mathsf{CMP}$ *is a diagnosis for program* P *iff* $\mathsf{repairable}(\mathsf{CMP} \setminus \Delta) = \mathsf{true}$. *A set* $C \subseteq \mathsf{CMP}$ *is a conflict iff* $\mathsf{repairable}(C) = \mathsf{false}$.

A diagnosis is a set of components that can be modified such that P becomes correct. The reason is that, for every input, it is possible to find some value that can be returned by the components $c \in \Delta$ such that the specification is fulfilled. Hence, diagnoses represent fault candidates. A conflict is a set of components from which at least one component has to be modified in order to obtain a correct program.

Example 5. *For the program* P *in Example 2 we have:*

Case	*Set* Q	$\mathsf{repairable}(Q)$	*Diagnosis*	*Conflict*
1	$\emptyset$	true	$\{c_1, c_2\}$	
2	$\{c_1\}$	true	$\{c_2\}$	
3	$\{c_2\}$	false		$\{c_2\}$
4	$\{c_1, c_2\}$	false		$\{c_1, c_2\}$

We have that $\mathsf{repairable}(\{c_1\})$=true *because* c_2 *can be modified to render* P *correct. For every input* X, Y*, there is a value (namely* Y*) to return by* c_2 *such that the assertion holds. Hence,* c_2 *may be responsible for the incorrectness of* P *— it is a diagnosis. On the other hand,* $\mathsf{repairable}(\{c_2\})$=false *because for* X=0, Y=1 *the specification is violated no matter what is returned by* c_1*, simply because the value is overwritten by* c_2*. Hence,* c_1 *cannot be responsible for the incorrectness, i.e.,* $\{c_1\}$ *is not a diagnosis. The other two cases are trivial.*

D. Computation of Diagnoses

The following theorem, which is a slight adaptation of Theorem 4.4 from [27], states that minimal diagnoses can be computed as minimal hitting sets for the collection of conflicts.

Theorem 1. *A set* $\Delta \subseteq \mathsf{CMP}$ *of components is a minimal diagnosis for program* P *iff it is a minimal hitting set for the collection* $\mathcal{K}$ *of conflicts for* P.

Proof: Using Lemma 2, the proof in [27] applies. ∎

We use the hitting set tree algorithm of Reiter [27] (with the fix of [16]) to compute diagnoses. It requires a procedure to compute a conflict not containing a certain set N of elements, if such a conflict exists. Such a procedure can be implemented by returning $\mathsf{CMP} \setminus N$ if $\mathsf{repairable}(\mathsf{CMP} \setminus N) = \mathsf{false}$ and `None` otherwise. Deciding repairability according to Eq. 2 is computationally hard or, depending on $\mathbb{D}$, D_{ex} and D_{co}, even undecidable. The reason is the quantifier alternation. Therefore, we check repairability only for a given set $J \subseteq 2^{(\mathbb{D}^A)}$ of input vectors. That is, instead of $\mathsf{repairable}(Q)$ we compute

$$\mathsf{repairable}'(Q) \Leftrightarrow \bigwedge_{\overline{v_i} \in J} \exists \bar{r} \,.\, \pi[\overline{v_i}||\bar{r}] \wedge \bigwedge_{r \in R} r = \mathrm{Org}(r)[\overline{v_i}||\bar{r}]$$

with $R = \{r \mid \mathrm{CmpOf}(r) \in Q\}$. We use only inputs that make P violate S because for all other inputs P is trivially repairable. When applying concolic execution for program analysis, such concrete input values are computed anyway. Using symbolic execution, path conditions can be solved to obtain values for J.

The quantifier-free part of $\mathsf{repairable}'$ is in D_{co}. Therefore, a query $\mathsf{repairable}'(Q)$ can be solved using one satisfiability check per input vector. An alternative is to swap the conjunction over the inputs with the quantification, rename all repair symbols to fresh ones for every conjunct, and use only one satisfiability check. In more detail, this works as follows. Let $\overline{r_i}$ be the vector of fresh symbols corresponding to $\bar{r}$ for input $\overline{v_i}$, and let r_i be the fresh symbol corresponding to symbol r in $\bar{r}$. We now have that $\mathsf{repairable}'(Q)$ is true iff

$$\bigwedge_{\overline{v_i} \in J} \pi[\overline{v_i}||\overline{r_i}] \wedge \bigwedge_{r \in R} r_i = \mathrm{Org}(r)[\overline{v_i}||\overline{r_i}] \quad (3)$$

is satisfiable.

The performance of Reiter's algorithm increases if the computed conflicts are minimal. A minimal conflict not containing a certain set N of elements can be computed in different ways. One way is to use a failure-preserving minimization algorithm like Delta Debugging [33] or QuickExplain [22] to repeatedly invoke $\mathsf{repairable}'$ with different subsets of $\mathsf{CMP} \setminus N$ until a minimal subset for which $\mathsf{repairable}'$ evaluates to false is found. Another option is based on the observation that every minimal conflict corresponds to an unsatisfiable core in Eq. 3. By rearranging the conjuncts, Eq. 3 can be rewritten to

$$\left(\bigwedge_{\overline{v_i} \in J} \pi[\overline{v_i}||\overline{r_i}]\right) \wedge \bigwedge_{c \in Q} \bigwedge_{\{r \mid \mathrm{CmpOf}(r) = c\}} \bigwedge_{\overline{v_i} \in J} r_i = \mathrm{Org}(r)[\overline{v_i}||\overline{r_i}].$$

This illustrates that every component $c \in Q$ corresponds to a certain conjunct in the definition of $\mathsf{repairable}'$. A minimal conflict not containing a certain set N of components can therefore be computed as a minimal unsatisfiable core of a

constraint system with $\bigwedge_{\overline{v_i} \in J} \pi[\overline{v_i}||\overline{r_i}]$ as a fixed part and

$$\bigcup_{c \in (\mathsf{CMP} \setminus N)} \left(\bigwedge_{\{r|\mathrm{CmpOf}(r)=c\}} \bigwedge_{\overline{v_i} \in J} r_i = \mathrm{Org}(r)[\overline{v_i}||\overline{r_i}] \right)$$

as a set of retractable constraints. Hence, if the solver is able to compute unsatisfiable cores, this feature can be exploited to compute minimal conflicts more efficiently.

Theorem 2. *Every diagnosis Δ with respect to the definition of* repairable *is also a diagnosis with respect to* repairable$'$.

Proof: Clearly, we have that repairable(Q) implies repairable$'(Q)$ for all $Q \subseteq \mathsf{CMP}$. ■

Theorem 2 states that using repairable$'$ instead of repairable can only lead to false positives but not to missing diagnoses.

E. Error Correction

Our method for error correction takes as input an incorrect program P, the diagnostic data $\Gamma = (\mathsf{CMP}, \mathrm{Vars}, \overline{i}, \overline{r}, \mathrm{CmpOf}, \mathrm{Org}, \mathrm{Vals}, \pi[\overline{i}||\overline{r}])$, and a diagnosis $\Delta \subseteq \mathsf{CMP}$. If successful, it produces a repaired program P' which differs from P only in the components Δ. Assuming that program analysis was perfectly accurate, P' cannot violate its specification for any input. The focus of our algorithm is on efficiency and readability of repairs rather than completeness.

New expressions have to be synthesized for all components $c \in \Delta$. We reduce the search for expressions to the search for constants by creating templates for unknown expressions. Templates consists of program variables and template parameters. Concrete parameter values define a concrete expression.

Example 6. *The template k_0+k_1·`v1`+k_2·`v2`, where k_0, k_1, k_2 are parameters and `v1`, `v2` are program variables, can express any linear expression over the variables. The values k_0=−2, k_1=1, and k_2=0 represent expression `v1`−2.*

Templates also provide control over the expressions subjected to search. To get simple repairs, it makes sense to start with simple templates and switch to more expressive templates if no repair is found with the simple ones.

Formally, for every component $c \in \Delta$, we create a template $T_c[\overline{k_c}||\overline{\mathrm{pv}_c}] \in D_{ex}$ as an expression over two vectors of fresh symbols $\overline{k_c} \in \mathbb{S}^*$ and $\overline{\mathrm{pv}_c} \in \mathbb{S}^{|\mathrm{Vars}(c)|}$. The symbols $\overline{\mathrm{pv}_c}$ represent the values of the program variables in scope when component c is executed. Symbols in $\overline{k_c}$ represent unknown parameter values. We write $\overline{k}$ for the concatenation of all $\overline{k_c}$ with $c \in \Delta$. Moreover, we define $K_c = |\overline{k_c}|$ and $K = |\overline{k}|$.

For all components $c \in \Delta$, let $\overline{v_{k,c}} \in \mathbb{D}^{K_c}$ be concrete values for the template parameters $\overline{k_c}$, and let $\overline{v_k}$ be the concatenation of all $\overline{v_{k,c}}$. We write $P' = \mathsf{apply}(\overline{v_k}, P)$ to denote that program P is transformed to program P' by replacing all components $c \in \Delta$ with expression $T_c[\overline{v_{k,c}}||\,\mathrm{Vars}(c)]$. That is, in all templates, parameters are replaced by the values defined in $\overline{v_k}$, program variable symbols are replaced by the respective variable names, and components $c \in \Delta$ of P are replaced by the so instantiated templates.

In order to check if a certain template instantiation yields a correctly repaired program, we define a function correct : $\mathbb{D}^A \times \mathbb{D}^K \to \{\mathsf{true}, \mathsf{false}\}$ such that $\mathsf{correct}(\overline{i}, \overline{k})$ is true iff

$$\exists \overline{r}\,.\,\pi[\overline{i}||\overline{r}] \wedge \bigwedge_{r \notin R} r = \mathrm{Org}(r)[\overline{i}||\overline{r}] \wedge \bigwedge_{r \in R} \exists \overline{\mathrm{pv}_c}\,.\,r = T_c[\overline{k_c}||\overline{\mathrm{pv}_c}] \wedge \overline{\mathrm{pv}_c} = \mathrm{Vals}(r), \quad (4)$$

where c is short for $\mathrm{CmpOf}(r)$ and $R = \{r \mid \mathrm{CmpOf}(r) \in \Delta\}$. The intuition behind Eq. 4 is as follows. $\pi[\overline{i}||\overline{r}]$ expresses when $\tilde{P}$ behaves correctly, depending on the unknown inputs $\overline{i}$ and the unknown values $\overline{r}$ returned by the components. Every symbol r that has been produced by an incorrect component $c \in \Delta$ is bound to the value that would be produced by the corresponding template T_c. This value is obtained by binding the symbols $\overline{\mathrm{pv}_c}$ to the values $\mathrm{Vals}(r)$ the program variables had when c was executed to produce r (the equality is meant element-wise). Every symbol r produced by a correct component $c \notin \Delta$ is bound to the value $\mathrm{Org}(r)$ that would have been produced by the unmodified component c.

Lemma 3. *Let P be an incorrect program, $\overline{v_i} \in \mathbb{D}^A$ be an input vector, and $\overline{v_k} \in \mathbb{D}^K$ be template parameter values. Then,* $\mathsf{correct}(\overline{v_i}, \overline{v_k})$ *maps to* true *iff the program $P' = \mathsf{apply}(\overline{v_k}, P)$ fulfills the specification S when executed with input $\overline{v_i}$.*

Proof: Let $\overline{v_r} \in \mathbb{D}^B$ be the concrete values returned by calls to `cmp` in $\tilde{P}$. According to Lemma 1, $\pi[\overline{v_i}||\overline{v_r}]$ evaluates to true iff $\tilde{P}$ fulfills S for input $\overline{v_i}$. The additional conjuncts in Eq. 4 make correct map to true iff a special version $\tilde{P}'$ of $\tilde{P}$ satisfies S for input $\overline{v_i}$. In $\tilde{P}'$, all components $c \notin \Delta$ return the same value as the original implementation of c in P. All components $c \in \Delta$ return the values that would have been returned by template T_c, instantiated with parameters defined in $\overline{v_k}$. This program $\tilde{P}'$ is exactly $P' = \mathsf{apply}(\overline{v_k}, P)$. ■

Theorem 3. *Let $\overline{v_k}$ be a vector of concrete templates values such that* $\mathsf{correct}(\overline{v_i}, \overline{v_k})$ *holds for all input vectors $\overline{v_i}$. Then, $P' = \mathsf{apply}(\overline{v_k}, P)$ is a correct program.*

Proof: Lemma 3 implies that P' cannot violate its specification S for any input. Hence, P' is correct. ■

F. Computation of Repairs

This section explains how repairs can be computed following Theorem 3. Observe that all quantified variables are bound to a value in Eq. 4. Therefore, an equivalent condition $\mathsf{correct}'[\overline{i}||\overline{k}] \in D_{co}$ can be defined by replacing all quantified variables by their value. What remains is the implicit quantifier alternation $\left(\exists \overline{k}\,.\,\forall \overline{i}\,.\,\mathsf{correct}'[\overline{i}||\overline{k}]\right)$ in Theorem 3, which renders the problem intractable or even undecidable. For error localization, we handled this issue by requiring correctness for some inputs only. Here, we avoid false positives. We follow the idea of [30] and [3] to compute repairs through iterative refinements that are guided by counterexamples.

The process is illustrated in Fig. 3. There is a database I of input vectors $\overline{v_i} \in \mathbb{D}^A$, which is initially empty. In every iteration, a repair candidate is computed in form of template

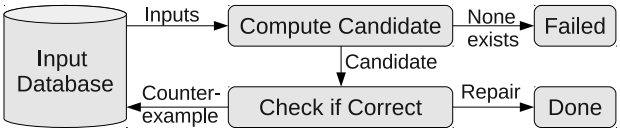

Fig. 3. Counterexample-guided repair refinement.

parameter values $\overline{v_k} \in \mathbb{D}^K$ such that $P' = \mathsf{apply}(\overline{v_k}, P)$ is correct for all inputs $\overline{v_i}$ in I. This is done by computing a satisfying assignment $\overline{v_k}$ for the symbols $\overline{k}$ in condition

$$\bigwedge_{\overline{v_i} \in I} \mathsf{correct}'[\overline{v_i}||\overline{k}]. \tag{5}$$

If Eq. 5 is unsatisfiable, the program cannot be repaired with the given templates and the procedure aborts. Otherwise, it is checked if $\overline{v_k}$ repairs the program for *all* inputs, i.e., if

$$\neg\, \mathsf{correct}'[\bar{i}||\overline{v_k}] \tag{6}$$

is unsatisfiable. If so, then $P' = \mathsf{apply}(\overline{v_k}, P)$ is a correct program and we are done. Otherwise, a satisfying assignment $\overline{v_i}$ for $\bar{i}$ in Eq. 6 is extracted. This $\overline{v_i}$ is a counterexample for the correctness of P'. It is added to I and another iteration is started, which produces a better candidate. This is repeated. We limit the number of iterations to ensure termination. If further repairs should be computed, we add conjuncts to Eq. 5 requiring that $\overline{k}$ is different to all previously computed repairs.

Example 7. *Let $\Delta = \{c_2\}$ be the diagnosis for the program P from Example 2, and let $k_0{+}k_1{\cdot}x{+}k_2{\cdot}y{+}k_3{\cdot}r$ be the template for c_2. We have that $correct((X,Y),(k_0,k_1,k_2,k_3)) =$*

$$\begin{aligned}\exists R_1, R_2 .((Y > X \wedge R_2 \geq X \wedge R_2 \geq Y) \vee (Y \leq X \wedge \\ R_1 \geq X \wedge R_1 \geq Y)) \wedge R_1 = X \wedge \\ R_2 = k_0 + k_1 \cdot X + k_2 \cdot Y + k_3 \cdot X,\end{aligned}$$

which can be simplified to $correct'[(X,Y,k_0,k_1,k_2,k_3)] =$

$$(Y \leq X) \vee (k_0 + k_1 \cdot X + k_2 \cdot Y + k_3 \cdot X \geq Y).$$

The computation of a repair could proceed as following. First, a satisfying assignment for $\overline{k} = (k_0,k_1,k_2,k_3)$ in Eq. 5 is computed with $I = \emptyset$. A possible solution is $\overline{k} = (0,0,0,0)$, which corresponds to the expression "`0`". Next, it is checked if replacing c_2 by `0` renders P correct. This is done by checking Eq. 6 for satisfiability, i.e., by searching for a counterexample. Eq. 6 is equal to $\neg(Y \leq X \vee 0 \geq Y)$, a satisfying assignment is $X = 2$, $Y = 4$. Hence, replacing c_2 by `0` does not repair P for all inputs. The database of inputs is extended to $I = \{(2,4)\}$, and an improved candidate expression is computed by solving Eq. 5, which is now equal to $(4 \leq 2) \vee (k_0 + k_1 \cdot 2 + k_2 \cdot 4 + k_3 \cdot 2 \geq 2)$. A solution is $\overline{k} = (1,0,1,0)$, which corresponds to the expression "`y+1`". Again, we verify if replacing c_2 by `y+1` renders P correct. Now, Eq. 6 is equal to $\neg(Y \leq X \vee 1 + Y \geq Y)$ and hence unsatisfiable. This means that no more counterexample exists. Replacing c_2 by `y+1` is a valid repair, the algorithm terminates.

G. Heuristics to Speed Up Convergence

Repair refinement can be seen as a game with two players. Player 1 comes up with candidates, Player 2 attempts to disprove them. In our experiments, we discovered two problems of this procedure. First, even if simple repairs exist, the play may end up computing and excluding more and more complex candidates. E.g., for one program, the sequence of candidates

Iteration	Candidate for a certain component
1	`0`
2	`-v0`
3	`250*v1 + 248*v2 - 2*v3 - v4`
4	and so on, becoming more and more complex

was observed, although the constant `500` was a repair for that component. Second, if both players do the least to fulfill their duty, progress may be insufficient. E.g., for the program in Example 2 with $\Delta = \{c_2\}$, the following may happen:

Repair candidate for c_2	Counterexample
`0`	x=0, y=1
`1`	x=1, y=2
`2`	and so on

We solve these two issues heuristically by improving the two players. Intuitively, we want "simple" candidates and "nasty" counterexamples. We say that a candidate is "simple" if many template parameters k_i are small or, even better, equal to some special value s_i, which makes terms in the template disappear (e.g., zero in case of a template for linear expressions). To implement this, we define a set $\rho_1 \subseteq 2^{D_{co}}$ of constraints as

$$\rho_1 = \bigcup_{k_i \in \overline{k}} \big(k_i = s_i\big) \cup \bigcup_{k_i \in \overline{k}} \big(k_i \leq M \wedge k_i \geq -M\big),$$

where M is a constant defining what "small" means. We compute template parameters by solving a Maximum Satisfiability (MAX-SAT) problem with Eq. 5 as fixed part and ρ_1 as the set of retractable constraints from which as many as possible should be fulfilled. Likewise, we say that a counterexample is "nasty" if it contains large, uncorrelated values. Again, we formulate a MAX-SAT problem with Eq. 6 as fixed part and

$$\rho_2 = \bigcup_{i_a \in \bar{i}} \big(i_a \geq N \vee i_a \leq -N\big)$$

as the set of retractable constraints, where N is a constant which is much larger than M. In order to break correlations between values in the counterexample we additionally randomize it: values are changed to large random values as long as the modified input vector is still a counterexample.

IV. Discussion and Alternatives

Our debugging method offers a lot of configuration parameters. This includes the number of execution paths to analyze, the number $|J|$ of inputs for diagnosis, the maximum number of repair refinements, and the templates to use. Moreover, the domains D_{ex} and D_{co} (i.e., the SMT-theories) determine which language constructs can be handled exactly, and which ones have to be approximated. As an advantage, our method

can by tailored to a broad range of programs. On the other hand, it may take some attempts to find a good configuration.

Our debugging method acts conservatively in that it targets a known good program termination for every input. The reason is the way $\pi[\bar{i}||\bar{r}]$ is defined in Eq. 1. An alternative is to use

$$\pi[\bar{i}||\bar{r}] = \neg \bigvee_{p \in \text{FAIL}} \text{PC}^p[\bar{i}||\bar{r}], \tag{7}$$

to avoid known specification violations. Using Eq. 1, diagnoses and repairs may be missed. Using Eq. 7, we may find false positives and allow endless loops. Both have their merits.

The more calls to `cmp` are introduced during pre-processing, the more execution paths become feasible. This observation can be exploited to refine the diagnostic data for error correction: A separate symbolic execution pass can be triggered for every diagnosis Δ before doing correction. In this pass, only the components in Δ are instrumented with calls to `cmp`. This gives higher path coverage for repair at the costs of having an additional program analysis step per diagnosis.

In principle, the fault model can be extended to include also faults in the LHS of assignments and even to missing or additional statements. A naive way is to apply case-splitting, but this is computationally expensive. More clever methods are subject to future work.

In first experiments, we observed that the quality of the produced repairs heavily depends on the quality of the given specification. This is neither surprising, nor is this problem specific to our method. It can happen that the computed correction simply prevents executions from ever reaching specific assertions. We plan to address this issue in the future by incorporating additional requirements such as the avoidance of unreachable code.

V. Experimental Results

In this section, we present first experimental results to demonstrate the feasibility of our approach. We implemented our debugging method for C programs. For program analysis we extended CREST [2], a concolic testing tool. Yices version 1.0.28 [10] is utilized with linear integer arithmetic as SMT-solver. Supporting other solvers and theories, especially bit-vectors and arrays, is planned. Thus, arrays and pointers are only handled approximatively at the moment. Currently, we use only templates for linear expressions. For expressions which occur as a condition in the program, we use templates of the form $k_0 + k_1 \cdot \texttt{v1} + k_2 \cdot \texttt{v2} + \ldots \ \texttt{OP} \ 0$, where $\texttt{v1}, \texttt{v2}, \ldots$ are program variables, $k_0, k_1, \ldots$ are template parameters, and $\texttt{OP} \in \{=, <, >, \leq, \geq\}$. The unknown comparison operator is encoded symbolically so that it can be handled like any other template parameter. Our implementation is part of a larger tool named FoREnSiC, which is under development and will feature also other formal, semi-formal, and dynamic debugging methods.

In our experiments, we set $|J| = 2$ for error localization, we limited the number of repair refinements to 10, the number of repairs to compute per diagnosis to 5, and set a time-out to all SMT-solver calls to 60 seconds. The experiments were performed on an Intel P7350 processor with 2×2.0 GHz and 3 GB RAM, running a 32-bit Linux. The implementation and scripts to reproduce the results are available for download[1].

TABLE I
Performance Results.

Column	1	2	3	4	5	6	7	8	9
	Diagnosis		Repair		No Heuristic		Sketch (8 bit)		
	Time	$\lvert\{\Delta\}\rvert$	Time	Found	Time	Found	Template Variables	Time	Memory
	[sec]	[-]	[sec]	[-]	[sec]	[-]	[-]	[s]	[MB]
`tcas2`	70	2	271	2	355	1	1	29	297
`tcas7`	123	2	42	5	444	0	4	20	1459
`tcas8`	122	2	37	5	465	0	1	6.3	42
`tcas16`	123	2	43	5	1027	0	2	22	764
`tcas17`	125	2	41	5	234	0	2	11	666
`tcas18`	124	2	38	5	35	0	2	9.3	744
`tcas19`	123	2	40	5	691	0	1	6.3	42
`tcas36`	3.0	0	-	-	-	-	4	7.0	796
total	813	14	512	32	3250	1		111	4810

Performance results

Table I summarizes performance results for some mutants of the TCAS program from the Siemens suite [9]. The program implements a traffic collision avoidance system for aircrafts in about 180 lines of code. We use the reference implementation as specification, which effectively doubles the size. For the TCAS examples, we do not consider conditions as components because they appear to challenge the solver, resulting in time-outs for many cases. We will try to overcome this issue by improving the encoding of the symbolic search for comparison operators, and by switching to a more recent solver. Consequently, Table I contains only mutants where the error is on the RHS of an assignment. We limited the number of paths to analyze with concolic execution to 400.

In every mutant of Table I, 44 components were identified. Program analysis took about 5 seconds. The time for diagnosis is listed in Column 1. Column 2 gives the number of diagnoses found. The Columns 3 and 4 show the error correction time and the number of found repairs with the heuristic of Section III-G enabled. The Columns 5 and 6 contain the same information for the heuristic being disabled. The Columns 7 to 9 summarize a comparison of our repair method with the program sketching tool Sketch [30][2]. We re-implemented the TCAS mutants in the input language of Sketch. Then, we manually replaced the faulty components with repair templates, using holes for the unknown template parameters. In this setting, Sketch ran out of memory for all cases. In order to have Sketch find a repair, we had to reduce the bit-width of an integer to 8 (for which we had to lower

[1] See http://www.iaik.tugraz.at/content/research/design_verification/others/. An official release of FoREnSiC will follow.

[2] We used Sketch version 1.3.0 with the solver ABC. When using MiniSat as a solver, the tool run out of memory.

constants in the program). Moreover, we had to reduce the number of program variables in the templates. Column 7 gives the maximum number of template variables so that Sketch can still find a repair. The last two columns list the time and memory requirements of Sketch, respectively.

In our experiments, we observed that a low number of inputs J (we use only two) is sufficient for our method to yield precise diagnoses. (See Column 2). Only for `tcas36`, no diagnosis could be found. The reason is that (by far) not all execution paths through the pre-processed program were analyzed. However, with other parameter configurations (e.g., using Eq. 7 instead of Eq. 1 and an extra program analysis pass per diagnosis; cf. Section IV) our method finds 5 repairs also for this mutant. The time for error localization is rather high compared to error correction (Column 1 vs. Column 3). This may be due to an inefficient implementation: we do not yet utilize unsatisfiable core functionality of the solver to compute minimal conflicts, as described in Section III-D. The program has many global variables, so each repair may depend on many variables. Nevertheless, error correction is surprisingly fast in our experiments. Furthermore, our heuristic to improve convergence in repair computation works well. It leads to more repairs being found in less time. Our tool is able to check a repaired program for correctness using the model checker CBMC [4]. This was successful in all cases.

At least for the analyzed TCAS examples, our repair method seems to perform better than Sketch. The repair templates used by our tool contain all variables which are in scope at the respective location in the program. This means at least 10 program variables and 11 template parameters for each template. For Sketch, we had to drastically reduce the number of program variables in repair templates in order to obtain a repair. (See Column 7.) Moreover, the memory requirements of our implementation are insignificant (below 80 MB in all cases). A plausible explanation is that Sketch breaks the synthesis problem down to Boolean satisfiability problems, while we use an SMT-solver. Furthermore, our tool did not analyze all execution paths of the TCAS mutants. Note, however, that the comparison with Sketch is not totally fair due to different input languages, solvers, and tool objectives.

Analysis of some Repairs

In this section, we take a closer look on the repair process for some programs. We start with our running example (cf. Example 2). For CMP=$\{c_1, c_2\}$, our tool identifies $\{c_2\}$ as the only diagnosis. The expressions `y`, `y+1`, `y+2`, etc., are computed as possible replacements of c_2. If the condition is considered as a third component c_3 (cf. Example 2) our tool finds the diagnoses $\{c_2\}$ and $\{c_1, c_3\}$. The former is repaired as before. For the latter, our tool computes the replacements

c_1	c_3
`y`	`x - y >= 0`,
`y+1`	`2*x - 2*y > 0`,
`y+2`	`3*x - 3*y > 0`,
`y+3`	`-x + y < 0`, and
`y+4`	`4*x - 4*y >= 0`.

In the mutant `tcas2` from Table I, the function

```
1 InhibitBiasedClimb() {
2   return (ClimbInhibit ? UpSep +
        NOZCROSS : UpSep);
3 }
```

has been modified: The constant `NOZCROSS` $=$ `100` has been replaced by the constant `MINSEP` $=$ `300`. The front-end of CREST simplifies the body of this function to:

```
1   if (ClimbInhibit) {
2     tmp = UpSep + 300;
3   } else {
4     tmp = UpSep;
5   }
6   return (tmp);
```

Our tool identifies the RHS of Line 2 as a diagnosis. For this diagnosis, the following repair candidates are computed.

Iteration	Candidate expression	Correct
1	`0`	no
2	`UpSep`	no
3	`UpSep + 100`	yes
4	`2*UpSep + 101`	no
5	`UpSep + 99`	no
6	`OtherTrAlt + UpSep + 99`	no
7	`-DwnSep + 2*UpSep + 199`	yes

Finally, the repair process aborts due to a time-out. The repair of Iteration 3 corresponds to the original program and is thus correct. The one found in Iteration 7 is correct because `InhibitBiasedClimb()` is only used in comparisons of the form `InhibitBiasedClimb() > DwnSep`. Since `UpSep` and `DwnSep` are integer variables, `-DwnSep + 2*UpSep + 199 > DwnSep` is true iff `UpSep + 100 > DwnSep` is true.

In the mutant `tcas18` from Table I, the statement

```
1 PosRAAltThresh[2] = 640;
```

has been modified by replacing the constant `640` with `640+50`. The RHS of this assignment is among the computed diagnoses. Our tool computes the following sequence of repair candidates for this diagnosis.

Iteration	Candidate	Correct
0	`0`	no
1	`400`	no
2	`500`	no
3	`640`	yes
4	`-OwnTrackedAltRate + 639`	no
5	`UpSep - 1`	no
6	`UpSep - 1000`	no
7	`-OwnTrackedAlt - 1`	no
8	`AltLayerValue + 638`	yes
9	`-AltLayerValue + 642`	yes
10	`2*AltLayerValue + 636`	yes
11	`-2*AltLayerValue + 644`	yes

The repair computed in Iteration 3 corresponds to the original program. The repairs found in the iterations 8 to 11 render the

program correct because the array `PosRAAltThresh` is only read at index `AltLayerValue`. Hence, the modification in `tcas18` affects the behavior only for `AltLayerValue` $= 2$. For this case, the expressions computed in the iterations 8 to 11 are equal to `640`. Thus they render the program correct.

These examples demonstrate that our method is able to find nontrivial corrections also for nontrivial programs.

VI. Conclusion

In this paper, we presented a novel method for automatic error localization and correction in imperative programs. It offers a wide range of different trade-offs between accuracy and resource requirements. Our method is based on symbolic execution, abstracting the debugging problem into the domain of logic. We showed how model-based diagnosis can be applied to locate errors using this abstraction. Our correction method is based on templates, a technique borrowed from the field of synthesizing loop invariants. This ensures that repairs are readable. We compute repairs with iterative refinements and presented a heuristic to speed this process up. This heuristic additionally prefers simple repairs. We implemented our debugging method for C programs. Although the implementation is still in a proof-of-concept state, experimental results demonstrate that the method works and can be used not just for toy examples.

In the future, we plan to investigate extensions of the fault model, develop methods to obtain more useful repairs for sketchy specifications, combine our method with other debugging approaches, and extend our tool to support more theories and solvers.

References

[1] A. Arcuri. On the automation of fixing software bugs. In *30th International Conference on Software Engineering (ICSE'08)*, pages 1003–1006. ACM, 2008.

[2] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *23rd International Conference on Automated Software Engineering (ASE'08)*, pages 443–446. IEEE, 2008.

[3] K.-H. Chang, I. L. Markov, and V. Bertacco. Fixing design error with counterexamples and resynthesis. In *Asia and South Pacific Design Automation Conference (ASP-DAC'07)*, pages 944–949, 2007.

[4] E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, pages 168–176, 2004.

[5] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Software Eng.*, 2(3):215–222, 1976.

[6] M. Colón, S. Sankaranarayanan, and H. Sipma. Linear invariant generation using non-linear constraint solving. In *Proc. Computer Aided Verification (CAV'03)*, pages 420–432. Springer, 2003. LNCS 2725.

[7] L. Console, G. Friedrich, and D. Theseider Dupré. Model-based diagnosis meets error diagnosis in logic programs. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'93)*, pages 1494–1499. Morgan-Kaufmann, 1993.

[8] V. Debroy and W. E. Wong. Using mutation to automatically suggest fixes for faulty programs. In *Third International Conference on Software Testing, Verification and Validation (ICST'10)*, pages 65–74. IEEE, 2010.

[9] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005.

[10] B. Dutertre and L. M. de Moura. A fast linear-arithmetic solver for DPLL(T). In *Proc. Computer Aided Verification (CAV'06)*, pages 81–94. Springer, 2006. LNCS 4144.

[11] A. Felfernig, G. Friedrich, D. Jannach, and M. Stumptner. Consistency-based diagnosis of configuration knowledge bases. *Artificial Intelligence*, 152:213–234, 2004.

[12] S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues. A genetic programming approach to automated software repair. In *Genetic and Evolutionary Computation Conference (GECCO'09)*, pages 947–954. ACM, 2009.

[13] G. Friedrich and K. M. Shchekotykhin. A general diagnosis method for ontologies. In *International Semantic Web Conference*, pages 232–246. Springer, 2005. LNCS 3729.

[14] G. Friedrich, M. Stumptner, and F. Wotawa. Model-based diagnosis of hardware designs. *Artificial Intelligence*, 111(1-2):3–39, 1999.

[15] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Conference on Programming Language Design and Implementation (PLDI'05)*, pages 213–223. ACM, 2005.

[16] R. Greiner, B. A. Smith, and R. W. Wilkerson. A correction to the algorithm in Reiter's theory of diagnosis. *Artificial Intelligence*, 41(1):79–88, 1989.

[17] A. Griesmayer, R. Bloem, and B. Cook. Repair of Boolean programs with an application to C. In *18th Conference on Computer Aided Verification (CAV'06)*, pages 358–371, 2006. LNCS 4144.

[18] A. Griesmayer, S. Staber, and R. Bloem. Fault localization using a model checker. *Software Testing, Verification and Reliability*, 20(2):149–173, 2010.

[19] B. Jobstmann, A. Griesmayer, and R. Bloem. Program repair as a game. In *17th Conference on Computer Aided Verification (CAV'05)*, pages 226–238. Springer, 2005. LNCS 3576.

[20] B. Jobstmann, S. Staber, A. Griesmayer, and R. Bloem. Finding and fixing faults. *Journal of Computer and System Sciences*, 2011. In Press.

[21] M. Jose and R. Majumdar. Cause clue clauses: error localization using maximum satisfiability. In *Conference on Programming Language Design and Implementation (PLDI'11)*, pages 437–446. ACM, 2011.

[22] U. Junker. QUICKXPLAIN: Preferred explanations and relaxations for over-constrained problems. In *Proc. National Conference on Artificial Intelligence (AAAI'04)*, pages 167–172. AAAI Press/MIT Press, 2004.

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

[24] J. de Kleer and B. C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32:97–130, 1987.

[25] R. Koenighofer, G. Hofferek, and R. Bloem. Debugging unrealizable specifications with model-based diagnosis. In *Proc. Haifa Verification Conference (HVC'10)*, pages 29–45. Springer, 2010. LNCS 6504.

[26] C. Mateis, M. Stumptner, D. Wieland, and F. Wotawa. Model-based debugging of Java programs. In *Proc. Fourth International Workshop on Automated Debugging (AADEBUG'00)*, 2000.

[27] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32:57–95, 1987.

[28] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proc. 10th European Software Engineering Conference and 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE'05)*, pages 263–272. ACM, 2005.

[29] A. Solar-Lezama. The sketching approach to program synthesis. In *Programming Languages and Systems, 7th Asian Symposium (APLAS'09)*, pages 4–13. Springer, 2009. LNCS 5904.

[30] A. Solar-Lezama, L. Tancau, R. Bodik, V. Saraswat, and S. A. Seshia. Combinatorial sketching for finite programs. In *Proc. Architectural Support for Programming Languages and Operating Systems (ASPLOS'06)*, pages 404–415. ACM, 2006.

[31] S. Srivastava, S. Gulwani, and J. S. Foster. From program verification to program synthesis. In *Symposium on Principles of Programming Languages (POPL'10)*, pages 313–326. ACM, 2010.

[32] M. Stumptner and F. Wotawa. Debugging functional programs. In *Proceedings on the 16th International Joint Conference on Artificial Intelligence (IJCAI'99)*. Morgan Kaufmann, 1999.

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