Path-Based Program Repair

Heinz Riener[1]
[1]Institute of Computer Science
University of Bremen, Germany
{hriener,rehlers,fey}@cs.uni-bremen.de

Rüdiger Ehlers[1,2]
[2]DFKI GmbH
Bremen, Germany

Görschwin Fey[1,3]
[3]Institute of Space Systems,
German Aerospace Center, Germany
goerschwin.fey@dlr.de

We propose a path-based approach to program repair for imperative programs. Our repair framework takes as input a faulty program, a logic specification that is refuted, and a hint where the fault may be located. An iterative abstraction refinement loop is then used to repair the program: in each iteration, the faulty program part is re-synthesized considering a symbolic counterexample, where the control-flow is kept concrete but the data-flow is symbolic. The appeal of the idea is two-fold: 1) the approach lazily considers candidate repairs and 2) the repairs are directly derived from the logic specification. In contrast to prior work, our approach is complete for programs with finitely many control-flow paths, i.e., the program is repaired if and only if it can be repaired at the specified fault location. Initial results for small programs indicate that the approach is useful for debugging programs in practice.

1 Introduction

Debugging is one of the most frequent and challenging activities in software development. In order to fix a faulty program without introducing new bugs or subtle corner cases, an in-depth understanding of the source code is required. Error hints and counterexamples produced by static analysis or model checking tools are only of little help: they typically report a symptom of a failure but do not point to the actual cause or provide a repair for a program. Manually correcting a faulty program based on this information is hard and often becomes an iterative trial-and-error process driven by a developer's intuition.

Automatic program repair techniques aim at reducing this manual burden by utilizing a refuted logic specification to automatically compute a repair for a faulty program. Existing approaches mainly suffer from two problems: 1) they do not scale well to large programs or 2) they rely on structural restrictions of the considered repairs. The two problems are orthogonal. Scalability issues originate from the fact that correctness of the entire program is modeled. Structural restrictions allow for the enumeration of potential repairs and make the repairs more readable [8]. The choice of the "right" structure for a repair, however, is left to the user or guided by a brute-force search. Bad choices cause an exclusion of suitable repairs.

In this paper, we take program repair one step further and address these two problems with a novel approach, called *path-based program repair*. The approach combines symbolic path reasoning and software synthesis. We present a repair framework to automatically correct a faulty imperative program based on an iterative abstraction refinement loop, assuming that the location of the fault is known. For instance, the location may be guessed by a user or computed using a fault localization approach, e.g., [7, 12].

In each iteration of the loop, a model checker computes a symbolic counterexample. Symbolic counterexamples keep the data-flow symbolic but the control-flow concrete. Symbolic path reasoning then infers verification conditions in the local context of the faulty program part. The inferred verification conditions are used to formulate a synthesis problem. The faulty program part is then re-synthesized in order to exclude all symbolic counterexamples found so far. Consequently, the synthesized candidate repair corrects the program with respect to the considered symbolic counterexamples. The loop terminates

B. Buhnova, L. Happe, J. Kofroň (Eds.):
Formal Engineering approaches to Software
Components and Architectures (FESCA'15).
EPTCS ??, 2015, pp. 24–34, doi:10.4204/EPTCS.??.5

when no further symbolic counterexamples are found and is guaranteed to converge for programs with finitely many control-flow paths (provided that the model checker and the synthesis procedure do not diverge).

The appeal of the idea is two-fold: 1) our approach lazily repairs a faulty program by inferring verification conditions from counterexamples utilizing symbolic path reasoning and 2) the repairs are derived from the refuted logic specification using software synthesis. Symbolic path reasoning, on the one hand, has been effectively used to cope with scalability. Software synthesis, on the other hand, does not restrict candidate repairs to have a predefined structure [8, 2], such as linear expressions over program variables.

Path-based program repair establishes a framework for fixing faulty programs, building on existing ideas, but tunes them to specifically address program repair. Naïvely combining prior work is problematic: applying predicate abstraction [4] leads to coarse abstractions that are just good enough to either verify the program or refute its correctness along one of its executions. Thus, the abstraction is insufficient to subsequently find a "good" repair for the program, which should work for most, if not all, cases. Recent software synthesis approaches, e.g., those for the *Syntax-Guided Synthesis* (SyGuS) [1] problem, cannot decide realizability, which is crucial for termination in case the initially given fault locations cannot be repaired.

Contribution. The contribution of the paper can be summarized as follows:

1. an iterative abstraction refinement approach for program repair combining symbolic path reasoning and software synthesis, called path-based program repair;
2. a prototype implementation of the repair framework utilizing domain finitizing and synthesis based on *Binary Decision Diagrams* (BDD), which allows deciding realizability;
3. initial experimental results for our prototype on a small ANSI-C program.

The remainder of the paper is structured as follows: in Sec. 2, we describe the necessary background and in Sec. 3, we present path-based program repair. Sec. 4 is dedicated to our prototype implementation utilizing domain finitizing and BDD-based synthesis. We also give experimental results for a small ANSI-C program. Sec. 5 concludes the paper.

2 Background

2.1 Program and Specification

We focus on sequential, finite-state systems described in a high-level programming language like ANSI-C. Let Stmts be the set of all statements, a program corresponds to a finite-state automaton over Stmts, called *program automaton*, with control locations as nodes and program operations as edges. Without loss of generality, we assume that each program automaton has a distinguished entry node and a distinguished exit node denoting the program's entry and the program's exit.

Our approach to program repair is dedicated to static analysis of individual *control-flow paths* of a program. A control-flow path is a consecutive sequence of nodes starting at the entry node and ending at the exit node, i.e., a word in the language of the program automaton, which corresponds to a terminating execution of the program and respects the semantics of the statements. Loops are unrolled and decisions at branching points in the program are modeled, e.g., `if` or `while` statements are replaced with respective assumption statements. More formally, a control-flow path $\pi = s_1 s_2 \cdots s_n$ is a sequence of side-effect free statements given in *Static Single Assignment* (SSA) form, where each s_i is either an *assumption statement*

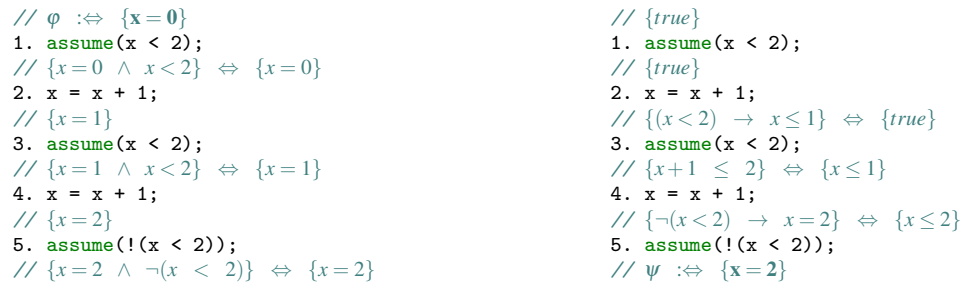

Figure 1: Predicate propagation along a control-flow path.

or an *assignment statement*. Along a fixed control-flow path, computing SSA form is straightforward because the costly placement of ϕ-nodes to assemble the control-flow from different control-flow paths is not necessary. An assignment statement is of form `v := e`, where v is a program variable and e is an expression over program variables and constants. An assumption statement is of form `assume(c)`, where c is a condition over the program variables and constants. We assume that the concatenation operation $\cdot$ is defined for a control-flow path $\pi = s_1 s_2 \dots s_n$ in the usual form such that the control-flow path can be represented as $\pi = \pi_A \cdot s_n$, $\pi = s_1 \cdot \pi_A$, or $\pi = \pi_A \cdot \pi_B$.

A (program) state σ is a valuation of all program variables, i.e., a mapping from the program variables to values within the respective domains. The specification is given as a precondition and a postcondition. The precondition defines the set of program states initially possible at the program's entry. The postcondition defines the set of program states allowed at the program's exit. We say that a program P is *correct if and only if* (iff) all executions starting in a state σ specified by the precondition reach a state σ' on termination that fulfills the postcondition. Otherwise we say that P is *faulty*.

In the following, sets of states are symbolically represented utilizing *First-Order Logic* (FOL) formulæ. Let $\phi \in \mathsf{FOL}$, we write $Vars(\phi)$ and $FV(\phi)$ to denote the variables and the free variables of ϕ, respectively. In several parts of the descriptions, we explicitly show that a FOL formula ϕ depends on a specific variable v by writing $\phi[v]$ and generalize this notation to show the dependence on a list $\overline{v}$ of variables by writing $\phi[\overline{v}]$. As usual, we call a FOL formula ϕ *satisfiable* iff an assignment to $Vars(\phi)$ exists that makes the formula **true** and *unsatisfiable* otherwise. Moreover, we say a FOL formula ϕ is *valid* iff ϕ is equivalent to **true**, i.e., ϕ is **true** for all assignments to $Vars(\phi)$. Lastly, $\phi[x/y]$ denotes ϕ where each free occurrence of variable x is replaced by variable y.

2.2 Symbolic Path Reasoning

We use Floyd/Hoare style computation to propagate predicates along a control-flow path. This is done by applying standard predicate transformers like weakest precondition and strongest postcondition to control-flow paths. A predicate transformer is a function $pt : \mathsf{FOL} \times \mathsf{Stmts} \to \mathsf{FOL}$ that maps a formula $\phi \in \mathsf{FOL}$ and a statement $s \in \mathsf{Stmts}$ to a formula $pt(\phi, s) \in \mathsf{FOL}$.

Definition 2.1. *Given a statement $s \in \mathsf{Stmts}$ executed on state σ to produce state σ' and a predicate $\phi \in \mathsf{FOL}$, the weakest precondition transformer wp and the strongest postcondition transformer sp compute*

- *the weakest predicate $wp(\phi, s) \in \mathsf{FOL}$, called* weakest precondition, *that guarantees $\sigma \models wp(\phi, s)$ if $\sigma' \in \phi$ and*
- *the strongest predicate $sp(\phi, s) \in \mathsf{FOL}$, called* strongest postcondition, *that guarantees $\sigma' \models sp(\phi, s)$ if $\sigma \in \phi$, respectively.*

Let $\phi \in \mathsf{FOL}$ and $s \in \mathsf{Stmts}$, we define the usual mechanic rules to compute the weakest precondition and the strongest postcondition for the two types of statements that occur in a control-flow path, respectively,

$$\begin{aligned} wp(\phi, \text{assume(c)}) &\Leftrightarrow c \rightarrow \phi \\ wp(\phi, \text{v := e}) &\Leftrightarrow \forall v'.(v' = e \rightarrow \phi[v/v']) \Leftrightarrow \phi[v/e] \\ sp(\phi, \text{assume(c)}) &\Leftrightarrow c \wedge \phi \\ sp(\phi, \text{v := e}) &\Leftrightarrow \exists v'.(v = e[v/v'] \wedge \phi[v/v']), \end{aligned}$$

and naturally generalize them to sequences $s_1 s_2 \cdots s_n$ of statements,

$$\begin{aligned} wp(\phi, s_1 s_2 \ldots s_n) &\Leftrightarrow wp(wp(\phi, s_n), s_1 s_2 \cdots s_{n-1}) \\ sp(\phi, s_1 s_2 \ldots s_n) &\Leftrightarrow sp(sp(\phi, s_1), s_2 s_3 \cdots s_n). \end{aligned}$$

Since loops are already unrolled, no loop invariants have to be found, and thus computing strongest postconditions and weakest preconditions along control-flow paths is decidable if the logic in use admits quantifier elimination. Moreover, note that computing the weakest precondition of an assignment statement amounts to replacing a variable by an expression. Thus, the application of costly quantifier elimination procedures is not necessary in this case.

We refer to the application of *wp* and *sp* to a given control-flow path $\pi = s_1 s_2 \ldots s_n$, i.e., a sequence of statements that respect the semantics of a program, and to a FOL formula as *backward propagation* and *forward propagation*, respectively.

Example 2.1. *In Fig. 1, we apply forward propagation and backward propagation, respectively, to a control-flow path $s_1 s_2 \ldots s_5$. Forward propagation (on the left) uses the precondition $\varphi \Leftrightarrow \{x = 0\}$ and the backward propagation (on the right) uses the postcondition $\psi \Leftrightarrow \{x = 2\}$.*

Definition 2.2. *Let π be a control-flow path with precondition φ and postcondition ψ. We say that a Hoare triple $\{\varphi\}\pi\{\psi\}$ holds iff the two equal conditions $\varphi \rightarrow wp(\psi, \pi)$ and $sp(\varphi, \pi) \rightarrow \psi$ are valid. Otherwise, we call π a (symbolic)* counterexample *for φ and ψ.*

This definition differs from the standard connotation of a counterexample given by a concrete input assignment returned by a model checking procedure. However, a symbolic counterexample in our sense can directly be determined after model checking by interpreting the program on the concrete input assignment and logging the corresponding control-flow path; thus, control-flow is concrete but data-flow symbolic.

2.3 Software Synthesis

We use synthesis to repair faulty programs and treat a synthesis procedure as a black box that derives program terms from a logic specification. The logic specification is a predicate $\phi[\bar{x}, \bar{y}] \in \mathsf{FOL}$, where $\bar{x}$ is a set of uncontrollable variables, $\bar{y}$ is a set of controllable variables, and $FV(\phi) \subseteq \bar{x} \cup \bar{y}$. A synthesis procedure computes terms over $\bar{x}$ such that replacing all occurrences of $\bar{y}$ in ϕ by their respective term yields a valid formula.

Definition 2.3. *A (software) synthesis procedure computes terms $\overline{T}$ from a given predicate $\phi \in \mathsf{FOL}$ and variables $\bar{x} \subseteq FV(\phi)$ such that*

$$\forall \bar{x}.\phi[\bar{y}/\overline{T}] \tag{1}$$

is valid, where $\overline{y} = FV(\phi) \backslash \overline{x}$.

A synthesis procedure may choose not to compute a term or may not terminate. If this can only happen if such a term replacement does not exist, i.e., if

$$\forall \overline{x} \exists \overline{y}.\phi$$

is valid, we call a synthesis procedure complete. *We also say that a complete synthesis procedure which always terminates is able to* detect unrealizability.

Note that this formulation defines a *software synthesis* problem [9, 5], where we search for a piece of terminating code that satisfies a given logic specification. This is in contrast to works on *reactive synthesis*, where a finite-state machine is to be synthesized that executes for an unbounded duration of time and satisfies a specification in some temporal logic, i.e., that reasons about the behavior of the finite-state machine over time.

Synthesis corresponds to quantifier elimination and synthesis procedures have been provided for relations expressed in different decidable logics, e.g., Boolean logic, linear arithmetic and sets [10], unbounded bit-vectors [5], term algebras and the theory of integer-indexed arrays with symbolic bounds on index ranges [6].

3 Path-Based Program Repair

3.1 The PBRepair Framework

The overall PBRepair approach is described as pseudo code shown in Alg. 1. Additionally, Tab. 1 gives a description of the main components used in the algorithm, and Fig. 2 describes the interaction between them.

The input to PBRepair is a faulty program P, a logic specification given as a pair of a precondition and a postcondition (φ, ψ), and an fault region e to be repaired, where e contains assignment statements to a set of variables $\overline{v}$. The algorithm returns a repaired program P' which is a copy of P but in which the code within the fault region e has been replaced by assignment statements to the variables $\overline{v}$ such that P' is correct with respect to φ and ψ.

The algorithm can be seen as an iterative abstraction refinement loop guided by the counterexamples provided by a model checker. The algorithm maintains a set Π of counterexamples and modifies a copy P' of the program (line 1). In each iteration, three steps are performed: firstly, the program is model checked with respect to its logic specification (line 2). If verification succeeds, then the algorithm terminates with the currently considered program P' as output. Otherwise, a (symbolic) counterexample π is provided by the model checker and added to Π (line 3).

Secondly, a synthesis procedure is invoked with the predicate $\Phi \in \mathsf{FOL}$ and a set of variables $\overline{y} \subseteq Var(\phi)$ to be synthesized (line 5). The predicate $\Phi \in \mathsf{FOL}$ accumulates the verification conditions by propagating the precondition φ and the postcondition ψ to the local context of the fault region e for all counterexamples $\pi \in \Pi$ (line 4). For a counterexample $\pi = \pi_A \cdot e \cdot \pi_B$ we use the strongest postcondition transformer sp to propagate φ forward and the weakest precondition transformer wp to propagate ψ backward until reaching the fault region. The variables $\overline{y}$ are fresh variables replacing $\overline{v}$ in $wp(\psi, \pi_B)$ and otherwise do not occur in Φ. The terms $\overline{T}$ produced as a result of synthesis are a repair for the fault region e in program P considering all counterexamples in Π. If Φ is unrealizable (line 6-8), then the algorithm terminates with an error indicating that the program cannot be repaired in the considered fault region e. For instance, this may happen if the initial fault region e has been provided by a user or an

Algorithm 1: PBRepair

input : faulty program P, precondition φ, postcondition ψ, fault region e
output: repaired program P', such that P' is correct

1 $\Pi := \emptyset$, $P' := P$
2 **while** $\pi := \mathsf{ModelCheck}(P', \varphi, \psi)$ **do**
3 $\quad \Pi := \Pi \cup \{\pi\}$
4 $\quad$ **let** $\Phi := \bigwedge_{\pi_A \cdot e \cdot \pi_B \in \Pi} (\mathsf{sp}(\varphi, \pi_A) \Rightarrow (\mathsf{wp}(\psi, \pi_B)[\overline{v}/\overline{y}]))$
5 $\quad\quad \overline{T} := \mathsf{Synthesize}(\Phi, \overline{y})$
6 $\quad\quad$ **if** Unrealizable **then**
7 $\quad\quad\quad$ **throw** *Unable to repair in fault region e.*
8 $\quad\quad$ **end**
9 $\quad P' := \mathsf{ApplyRepair}(P', e, \overline{T})$
10 **end**
11 **return** P'

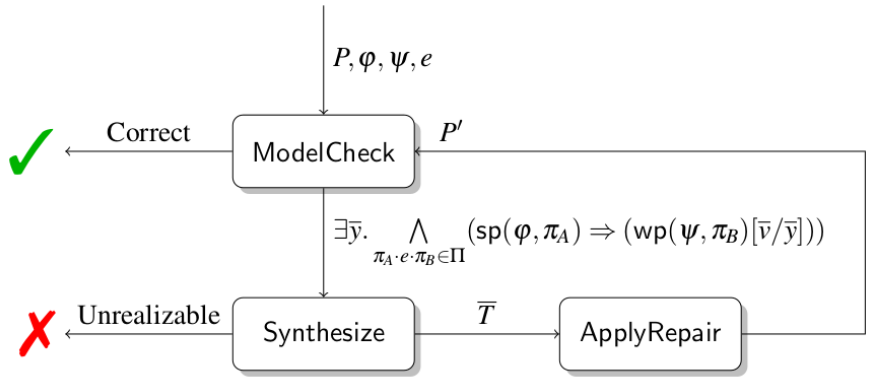

Figure 2: PBRepair: high-level overview of path-based program repair.

unsound fault localization algorithm. Otherwise, synthesis yields a list of terms $\overline{T}$ that are used to repair the program.

Thirdly, if the terms $\overline{T}$ could be computed (line 5), assignment statements to the variables in $\overline{v}$ are generated as a repair to the program in the syntax of the programming language in use. In P', the assignment statements replace the statements within the fault region (line 9). The program P' is correct (by construction) with respect to φ, ψ, and all counterexamples $\pi \in \Pi$. The algorithm loops until a correctly repaired program is found or the non-existence of a repair is detected.

3.2 Correctness and Termination

Lemma 3.1. *Let $\pi = \pi_A \cdot e \cdot \pi_B$ be a counterexample to precondition φ and postcondition ψ, where e is a list of assignment statements $\mathtt{v_i} := \mathtt{e_i}$ that is known to be faulty and π_A and π_B are known to be correct, then $\{\varphi\}\pi_A \cdot r \cdot \pi_B\{\psi\}$ holds for the assignment statements r of form $r : \mathtt{v_1} := \mathtt{T_1} \ldots \mathtt{v_n} := \mathtt{T_n}$, where $\overline{v} = [v_1, \ldots, v_n]$ and $\overline{T} = [T_1, \ldots, T_n]$ is the result of performing synthesis on the specification $sp(\varphi, \pi_A) \rightarrow wp(\psi, \pi_B)[\overline{v}/\overline{y}]$ for the output variables $\overline{y}$.*

Name	ModelCheck
Input	Source code of a program P, a precondition φ, and a postcondition ψ.
Output	A counterexample if P is faulty and **false** otherwise.
Description	Model checks the source code of P assuming φ and asserting ψ and returns a (symbolic) counterexample π.
Name	Synthesize
Input	A predicate ϕ and a vector $\bar{x} \subseteq FV(\phi)$ of controllable variables.
Output	A vector $\overline{T}$ of terms if ϕ is realizable with respect to ϕ or throws an exception.
Description	Computes a vector $\overline{T}$ of program terms on termination, such that $\forall\bar{x}\exists\bar{y}.\phi \rightarrow \forall\bar{x}.\phi[\bar{y}/\overline{T}]$ becomes valid. If $\forall\bar{x}\exists\bar{y}.\phi$ is not valid, i.e., is unsatisfiable, an exception is thrown which indicates that the synthesis problem is unrealizable.
Name	ApplyRepair
Input	Source code of program P, a fault region e, and program terms $\overline{T}$.
Output	A repaired copy P' of program P, where all changes are applied to fault region e.
Description	Transforms the program terms $\overline{T}$ into source code in the programming language in use and replaces them for the right-hand side of the assignment statements in fault region e in P'.

Table 1: A description of the main components used in Alg. 1.

Proof. The control-flow path $\pi = \pi_A \cdot r \cdot \pi_B$ is correct with respect to precondition φ and postcondition ψ iff $\{\varphi\}\pi_A \cdot r \cdot \pi_B\{\psi\}$ holds. Let $\Gamma_1, \Gamma_2 \in \mathsf{FOL}$ be the verification conditions that hold immediately before and immediately after the execution of the statements r, respectively, such that $\{\varphi\}\pi_A \cdot r \cdot \pi_B\{\psi\}$ decomposes to the three Hoare triples $H_1 : \{\varphi\}\pi_A\{\Gamma_1\}$, $H_2 : \{\Gamma_2\}\pi_B\{\psi\}$, and $H_3 : \{\Gamma_1\}r\{\Gamma_2\}$. Since π_A and π_B are correct, H_1 and H_2 hold, and only H_3 has to be proven. The Hoare triple H_3 holds iff (i) $\Gamma_1 \rightarrow wp(\Gamma_2, r : \mathtt{v_1} := \mathtt{T_1} \ldots \mathtt{v_n} := \mathtt{T_n})$ is valid (Def 2.2), which further simplifies to $\Gamma_1 \rightarrow \Gamma_2[\bar{v}/\overline{T}]$. The Hoare triples H_1 and H_2 hold iff (ii) $sp(\varphi, \pi_A) \rightarrow \Gamma_1$ and (iii) $\Gamma_2 \rightarrow wp(\psi, \pi_B)$ hold (Def 2.2), respectively. We use (ii) to weaken the left-hand side of (i) and (iii) to strengthen the right-hand side of (i), and obtain (iv) $sp(\varphi, \pi_A) \rightarrow wp(\psi, \pi_B)[\bar{v}/\overline{T}]$. The term replacements $\overline{T}$ guarantee the validity of implication (iv) (Def 2.3), and thus H_3 and $\{\varphi\}\pi\{\psi\}$ hold. □

Theorem 3.2. *Let P be a program with finitely many control-flow paths, (φ, ψ) be a pair of a precondition and a postcondition, and $e : \mathtt{v_1} = \mathtt{w_1}; \ldots \mathtt{v_n} := \mathtt{w_n}$ be a fault region in P, then algorithm PBRepair(P, φ, ψ, e) returns on termination either a program P' correct with respect to φ and ψ if P can be repaired in e or otherwise throws an error.*

Proof. Since P is faulty with respect to the specification, model checking in the first step produces a counterexample π on termination. If the model checking procedure does not terminate, then PBRepair does not terminate. If the fault region e is not contained in π, then PBRepair terminates with an error indicating that P cannot be repaired within the fault region e. This may happen when multiple faults are considered. Otherwise, a synthesis procedure is invoked to repair the program P at e. If the procedure does not terminate, PBRepair does not terminate. If the synthesis procedure reports unrealizability of the specification, no repair exists to make P correct in e and PBRepair throws an error. Otherwise, according

to Lemma 3.1 program terms $\overline{T}$ are synthesized such that the counterexample is removed. In each iteration, at least one counterexample is removed. Since the verification conditions of the counterexamples are accumulated, monotonicity is enforced, i.e., previously corrected counterexamples cannot become faulty again. Since P has only finitely many control-flow paths, PBRepair terminates after finitely many iterations if for all iterations, model checking and synthesis terminate. The finally produced program has no counterexamples and thus is correct with respect to φ and ψ. □

4 Implementation and Experimental Results

The repair framework presented in the previous section is generic in the sense that the framework can be instantiated with different model checkers and synthesis procedures abstracting from programming and specification languages. In this section, we present a prototype implementation of the repair framework for ANSI-C utilizing domain finitizing and give some initial experimental results indicating that path-based program repair can be useful for repairing real programs.

The specification, i.e., the precondition and postcondition, and all other logic formulæ are expressed in the SMT-LIB2 logic QF_BV, i.e., the quantifier-free fragment of first-order logic modulo bit-vector arithmetic. The manipulation of logic formulæ for computing weakest preconditions and strongest post-conditions has been implemented using the API of the theorem prover Z3 [11].

Symbolic counterexamples are computed by leveraging CBMC [3] in combination with a self-implemented execution tracer. CBMC model checks the program with respect to the given specification. When verification fails, an input assignment is extracted from CBMC's logfile. The execution tracer then re-simulates the program with this input assignment and dumps the statements executed in a textual representation similar to Fig. 1.

For synthesis, logic formulæ are bit-blasted to Boolean functions, more particularly *And-Inverter Graphs* (AIGs), by replacing each word-level variable by individual bit-level variables and each bit-vector operator by a corresponding Boolean circuit. After bit-blasting, a BDD-based synthesis procedure is applied to obtain a gate-level repair for the program. The synthesis procedure is complete, guarantees termination, and thus detects unrealizability. Basing the main synthesis work on BDDs has many advantages — they can perform the quantifier elimination step needed for synthesis in a natural and efficient way. Also, the question of how to compute an implementation from an input/output relation that is represented as a Boolean function is well-researched, so that we can apply this work.

In the last step, the gate-level repair is transformed to ANSI-C code in a straight-forward way: for each circuit gate $o = AND(a,b)$, a fresh variable o is introduced and assigned to the ANSI-C expression $(a \,\&\, b)$, where a and b are either other variables introduced by this conversion or input bits extracted from existing word-level program variables.

Fig. 3 shows `minmax`, a simple fragment of an ANSI-C program that determines the largest and the smallest value of three given inputs. All variables in the program fragment are of integer type. The logic specification φ and ψ annotated to the source code is complete, so that all possible faults are observable during model checking and can be repaired by our approach assuming the "right" fault region is provided as input. To improve scalability of synthesis, the bit-widths of integer variables are reduced to 2 bit.

In order to allow repairing conditional statements of form `if(c){...}`, where the guard condition c may be a complex or compound expression, in a preprocessing step, the conditional statement is replaced by `t = c; if(t){...}`, where t is a new temporary program variable.

Table 2 lists some initial experiments, where faults have been seeded into `minmax` and PBRepair is applied to repair them. The table is built as follows: each line corresponds to one seeded fault. The

```
// φ :⇔ {true}
1. most = input1;
2. least = input1;
3. if (most < input2)
4.   most = input2;
5. if (most < input3)
6.   most = input3;
7. if (input2 < least)
8.   least = input2;
9. if (input3 < least)
10.  least = input3;
// ψ :⇔ {
//   (most = input1 ∨ most = input2 ∨ most = input3) ∧
//   (most ≥ input1 ∧ most ≥ input2 ∧ most ≥ input3) ∧
//   (least = input1 ∨ least = input2 ∨ least = input3) ∧
//   (least ≤ input1 ∧ least ≤ input2 ∧ least ≤ input3) }
```

Figure 3: `minmax` program

Line	Type	Iterations	Control-Flow Path [AND gates]	Time [s]
1	Assignment	7	0000[2] 0100[2] 0011[16] 0110[17] 1010[12] 0010[18] 0001[7]	2
2	Assignment	4	1001[2] 0100[3] 0000[2] 1000[2]	3
3	Condition	4	1110[1] 0000[3] 1010[6] 0001[8]	4
4	Assignment	2	1000[3] 1001[6]	3
5	Condition	2	0101[1] 0000[5]	6
6	Assignment	2	0110[3] 0100[6]	4
7	Condition	5	0001[1] 1010[3] 0100[9] 0000[11] 1110[8]	4
8	Assignment	3	0111[2] 0010[5] 0110[4]	4
9	Condition	6	0111[1] 0000[1] 1001[4] 0011[8] 1000[18] 0010[20]	5
10	Assignment	2	0011[2] 0001[14]	4

Table 2: Path-Based Program Repair applied to `minmax`

first column shows the line number in which a fault was seeded, the second column lists the type of the erroneous statement, the third column gives the number of iterations needed by PBRepair to terminate and the fourth column lists the examined control-flow paths as bit strings for all iterations and the size of the corresponding candidate repair counted in AND gates in squared brackets. Each bit string $g_1g_2g_3g_4$ denotes the evaluation of the guard conditions, where g_1, g_2, g_3, g_4 correspond to the code lines 3, 5, 7, 9, respectively. The value 0 and 1 indicate that the respective guard condition evaluated to **false** and **true**, respectively, when executed. The last column gives the run-time in seconds. All experiments have been conducted on Intel(R) Core(TM) i5-2520M CPU @ 2.50GHz with 8GB RAM. The run-time was mainly spend in synthesizing the repair and the time required for model checking was negligible.

The repair framework proposed is fully automated and does not need any human intervention. Our initial experiments indicate that PBRepair can be used for repairing simple ANSI-C programs; i.e., the prototype implementation of PBRepair proposed was able to determine a repair for each of our seeded faults in only a few seconds. However, before applying our repair procedure to `minmax`, the bit-width of integers was manually reduced. Otherwise, quantification on up to 160 BDD variables is necessary which is challenging for today's BDD-based procedures. We claim that automated bit-width abstraction refinement for synthesis is in reach such that a synthesize-and-generalize approach is possible: first bit-widths are abstracted to a small number of BDD variables, a repair is synthesized from the abstraction, generalized to the full bit-widths, and subsequently verified considering the context of the program.

5 Conclusion

In this paper, we presented a path-based abstraction refinement approach to program repair which combines symbolic path reasoning and software synthesis. A prototype implementation of the repair framework has been presented utilizing domain finitizing and BDD-based synthesis. In contrast to other synthesis approaches, this allows for deciding realizability. Initial experimental results for our prototype on a small ANSI-C program have been presented.

The repair framework uses off-the-shelf model checking and synthesis tools, and thus inherits their scalability strength and barriers. In case of the BDD-based synthesis the limiting factor is the number of input and output variables after bit-blasting, which were manually reduced for our experiments. We conjecture that a customized bit-width abstraction refinement approach will substantially improve scalability, while allowing to keep the completeness and unrealizability detecting capabilities of BDD-based synthesis. A challenge that remains in this context is to foster readability of the computed implementation parts. We leave these improvements to future work.

Acknowledgements: This work was supported by the German Research Foundation (DFG, grant no. FE 797/6-1) and the Institutional Strategy of the University of Bremen, funded by the German Excellence Initiative.

References

[1] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak & Abhishek Udupa (2013): *Syntax-Guided Synthesis*. In: *Formal Methods in Computer-Aided Design*, pp. 1–17, doi:10.1109/FMCAD.2013.6679385.

[2] Roderick Paul Bloem, Rolf Drechsler, Görschwin Fey, Alexander Finder, Georg Hofferek, Robert Könighofer, Jaan Raik, Urmas Repinski & Andre Sülflow (2012): *FoREnSiC — An Automatic Debugging Environment for C Programs*. In: *Haifa Verification Conference*, pp. 260–265, doi:10.1007/978-3-642-39611-3_24.

[3] Edmund Clarke, Daniel Kroening & Flavio Lerda (2004): *A Tool for Checking ANSI-C Programs*. In: *Tools and Algorithms for Construction and Analysis of Systems*, pp. 168–176, doi:10.1007/978-3-540-24730-2_15.

[4] Cormac Flanagan & Shaz Qadeer (2002): *Predicate Abstraction for Software Verification*. In: *Symposium on Principles of Programming Languages*, pp. 191–202, doi:10.1145/503272.503291.

[5] Jad Hamza, Barbara Jobstmann & Viktor Kuncak (2010): *Synthesis for Regular Specifications over Unbounded Domains*. In: *Formal Methods in Computer-Aided Design*, pp. 101–109.

[6] Swen Jacobs, Viktor Kuncak & Philippe Suter (2013): *Reductions for Synthesis Procedures*. In: *Verification, Model Checking and Abstract Interpretation*, pp. 88–107, doi:10.1007/978-3-642-35873-9_8.

[7] Manu Jose & Rupak Majumdar (2011): *Cause Clue Clauses: Error Localization using Maximum Satisfiability*. In: *Programming Language Design and Implementation*, pp. 437–446, doi:10.1145/1993316.1993550.

[8] Robert Könighofer & Roderick Bloem (2011): *Automated Error Localization and Correction for Imperative Programs*. In: *Formal Methods in Computer-Aided Design*, pp. 91–100.

[9] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac & Philippe Suter (2010): *Complete Functional Synthesis*. In: *Programming Language Design and Implementation*, pp. 316–329, doi:10.1145/1806596.1806632.

[10] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac & Philippe Suter (2013): *Functional Synthesis for Linear Arithmetic and Sets*. *International Journal on Software Tools for Technology Transfer* 15(5–6), pp. 455–474, doi:10.1007/978-3-662-44202-9_15.

[11] Leonardo de Moura & Nikolaj Bjørner (2008): *Z3: An efficient SMT solver*. In: *Tools and Algorithms for Construction and Analysis of Systems*, pp. 337–340, doi:10.1007/978-3-540-78800-3_24.

[12] Heinz Riener & Görschwin Fey (2012): *Model-Based Diagnosis versus Error Explanation*. In: *Formal Methods and Models for Co-Design*, pp. 43–52, doi:10.1109/MEMCOD.2012.6292299.