Cost-Aware Automatic Program Repair

Roopsha Samanta[1,⋆], Oswaldo Olivo[2], and E. Allen Emerson[2]

[1] The University of Texas at Austin and IST Austria
rsamanta@ist.ac.at
[2] The University of Texas at Austin
{olivo,emerson}@cs.utexas.edu

Abstract. We present a formal framework for repairing infinite-state, imperative, sequential programs, with (possibly recursive) procedures and multiple assertions; the framework can generate repaired programs by modifying the original erroneous program in multiple program locations, and can ensure the readability of the repaired program using user-defined expression templates; the framework also generates a set of inductive assertions that serve as a proof of correctness of the repaired program. As a step toward integrating programmer intent and intuition in automated program repair, we present a *cost-aware* formulation — given a cost function associated with permissible statement modifications, the goal is to ensure that the total program modification cost does not exceed a given repair budget. As part of our predicate abstraction-based solution framework, we present a sound and complete algorithm for repair of Boolean programs. We have developed a prototype tool based on SMT solving and used it successfully to repair diverse errors in benchmark C programs.

1 Introduction

Program debugging — the process of fault localization and error elimination — is an integral part of ensuring correctness in existing or evolving software. Being essentially manual, program debugging is often a lengthy, expensive part of a program's development cycle. There is an evident need for improved formalization and mechanization of this process. However, program debugging is hard to formalize — there are multiple types of programming mistakes with diverse manifestations, and multiple ways of eliminating a detected error. Moreover, it is particularly challenging to assimilate and mechanize the expert human intuition involved in the choices made in manual program debugging.

In this paper, we present a *cost-aware* formulation of the automated program debugging problem that addresses the above concerns. Our formulation obviates the need for a separate fault localization phase by directly focusing on error elimination, i.e., program repair. We fix a set $\mathcal{U}$ of *update schemas* that may be applied to program statements for modifying them. An update schema is a

⋆ This author was supported in part by the European Research Council (ERC) under grant agreement 267989 (QUAREM) and by the Austrian Science Fund (FWF) NFN project S11402-N23 (RiSE)

compact description of a class of updates that may be applied to a program statement in order to repair it. For instance, the update schema `assign` $\mapsto$ `assign` permits replacement of the assignment statement $x := y$ with other assignment statements such as $x := x + y$ or $y := x + 1$, `assign` $\mapsto$ `skip` permits deletion of an assignment statement, etc. In this paper, $\mathcal{U}$ includes deletion of statements, replacement of assignment statements with other assignment statements, and replacement of the guards of conditional and loop statements with other guards. We assume we are given a *cost function* that assigns some user-defined cost to each application of an update schema to a program statement. Given an erroneous program $\mathcal{P}$, a cost function c and a repair budget δ, the goal of *cost-aware automatic program repair* is to compute a program $\widehat{\mathcal{P}}$ such that: $\widehat{\mathcal{P}}$ is correct, $\widehat{\mathcal{P}}$ is obtained by modifying $\mathcal{P}$ using a set of update schemas from $\mathcal{U}$ and the total modification cost does not exceed δ. We postulate that this *quantitative* formulation [6] is a flexible and convenient way of incorporating user intent and intuition in automatic program debugging. For instance, the user can define appropriate cost functions to search for $\widehat{\mathcal{P}}$ that differs from $\mathcal{P}$ in at most δ statements, or to penalize any modification within some *trusted* program fragment, or to favor the application of a particular update schema over another, and so on.

Our approach to cost-aware repair of imperative, sequential programs is based on predicate abstraction [13], which is routinely used by verification tools such as SLAM [5], SLAM2 [2], SATABS [8], etc. for analyzing infinite-state programs. These tools generate Boolean programs which are equivalent in expressive power to pushdown systems and enjoy desirable computational properties such as decidability of reachability [4]. Inevitably, Boolean programs have also been explored for use in automatic repair of sequential programs for partial correctness [14] and total correctness [22]. These papers, however, do not accommodate a quantitative formulation of the repair problem and can only compute repaired programs that differ from the original erroneous program in exactly one expression. Moreover, these papers do not attempt to improve the *readability* of the concrete program $\widehat{\mathcal{P}}$, obtained by concretizing a repaired Boolean program.

Our predicate abstraction-based approach to automatic program repair relaxes the above limitations. Besides erroneous $\mathcal{P}$, c, and δ, our framework requires a Boolean program $\mathcal{B}$, obtained from $\mathcal{P}$ through iterative predicate abstraction-refinement, such that $\mathcal{B}$ exhibits a non-spurious path to an error. We present an algorithm which casts the question of *repairability of* $\mathcal{B}$, given U, c, and δ, as an SMT query; if the query is satisfiable, the algorithm extracts a correct Boolean program $\widehat{\mathcal{B}}$ from the witness to its satisfiability. Along with $\widehat{\mathcal{B}}$, we also extract a set of inductive assertions from the witness, that constitute a proof of correctness of $\widehat{\mathcal{B}}$. This algorithm for Boolean program repair is sound and complete, relative to $\mathcal{U}$, c, and δ. A repaired Boolean program $\widehat{\mathcal{B}}$, along with its proof, is concretized to obtain a repaired concrete program $\widehat{\mathcal{P}}$, along with a proof of correctness. However, the concretized repairs may not be succinct or readable. Hence, our framework can also accept user-supplied templates specifying the desired syntax of the modified expressions in $\widehat{\mathcal{P}}$ to constrain the concretization.

Alternate approaches to automatic repair and synthesis of sequential programs [17,26–28] that do not rely on abstract interpretations of concrete programs, also often encode the repair/synthesis problem as a constraint-solving problem whose solution can be extracted using SAT or SMT solvers. Except for [28], these approaches, due to their bounded semantics, are imprecise and cannot handle total correctness[1]. The authors in [17] use SMT reasoning to search for repairs satisfying user-defined templates; the templates are needed not only for ensuring readability of the generated repairs, but also for ensuring tractability of their inherently undecidable repair generation query. They also include a notion of minimal diagnoses, which is subsumed by our more general cost-aware formulation. Given user-defined constraints specifying the space of desired programs and associated proof objects, the scaffold-based program synthesis approach of [28] attempts to synthesizes a program, along with a proof of total correctness consisting of program invariants and ranking functions for loops. In contrast to [28], our framework only interacts with a user for improving the readability of the generated repairs and for the cost function; all predicates involved in the generation of the repaired Boolean program and its proof are discovered automatically. Besides the above, there have been proposals for program repair based on computing repairs as winning strategies in games [15], abstraction interpretation [18], mutations [10], genetic algorithms [1,12], using contracts [29], and focusing on data structure manipulations [25,30]. There are also customized program repair engines for grading and feedback generation for programming assignments, cf. [24]. Finally, a multitude of algorithms [3, 7, 16, 31] have been proposed for fault localization, based on analyzing error traces. Some of these techniques can be used as a preprocessing step to improve the efficiency of our algorithm, at the cost of giving up on the completeness of the Boolean program repair module.

Summary of contributions: We define a new cost-aware formulation of automatic program repair that can incorporate programmer intuition and intent (Sec. 3). We present a formal solution framework (Sec. 4 and Sec. 5) that can repair infinite-state, imperative, sequential programs with (possibly recursive) procedures and multiple assertions. Our method can modify the original erroneous program in multiple program locations and can ensure the readability of the repaired program using user-defined expression templates. If our method succeeds in generating a repaired program $\widehat{\mathcal{P}}$, it generates a proof of $\widehat{\mathcal{P}}$'s correctness, consisting of inductive assertions, that guarantees satisfaction of *all* the assertions in the original program $\mathcal{P}$. As part of our predicate abstraction-based solution, we present a sound and complete algorithm for repair of Boolean programs. Finally, we present experimental results for repairing diverse errors in benchmark C programs using a prototype implementation (Sec. 6).

2 Background

Predicate abstraction [4, 13] is an effective technique for model checking infinite-state programs with respect to safety properties. It uses iterative

[1] Our framework can be extended to handle total correctness by synthesizing ranking functions along with inductive assertions.

counterexample-guided abstraction refinement to compute a finite-state, *conservative* abstraction of a concrete program $\mathcal{P}$ based on a finite set $\{\phi_1, \ldots, \phi_r\}$ of predicates. The resulting abstract program is termed a *Boolean program* $\mathcal{B}$ (see Fig. 1a and Fig. 1b): the control-flow of $\mathcal{B}$ is the same as that of $\mathcal{P}$ and the set $\{b_1, \ldots, b_r\}$ of variables of $\mathcal{B}$ are Boolean variables, with each b_i representing the predicate ϕ_i for $i \in [1, r]$. If $\mathcal{B}$ is found to be correct, the method concludes that $\mathcal{P}$ is correct. In our work, the interesting case is when the method terminates reporting an error. This happens when the method computes a Boolean program containing an abstract counterexample path which is found to be feasible in $\mathcal{P}$. Henceforth, we fix a concrete program $\mathcal{P}$, and a corresponding Boolean program $\mathcal{B}$ that exhibits such a non-spurious counterexample path. Let γ denote the mapping of the Boolean variables in $\mathcal{B}$ to their respective predicates: for each $i \in [1, r]$, $\gamma(b_i) = \phi_i$. The mapping γ can be extended in a standard way to expressions over the Boolean variables.

Program Syntax. For our technical presentation, we fix a common, simplified syntax for concrete and abstract programs (see [23] for a precise definition) — a program consists of a declaration of global variables, followed by a list of procedure definitions; a procedure definition consists of a declaration of local variables, followed by a sequence of (labeled) statements; a statement is a `skip`, (parallel) assignment, `assume`, `assert`, `goto`, (call-by-value) procedure `call` or `return` statement[2]. A Boolean expression is either a deterministic Boolean expression or the expression $*$, which nondeterministically evaluates to `true` or `false`.

We make the following assumptions: (a) there is a distinguished initial procedure `main`, (b) all variable and formal parameter names are globally unique, and (c) `goto` statements are used only to simulate the flow of control in structured programs. In addition, for Boolean programs, we assume: (a) all variables and formal parameters are Boolean, (b) all expressions are Boolean expressions and (c) the Boolean expressions in `assume` and `assert` statements are deterministic.
Notation. For program $\mathcal{P}$, let $\{F_0, \ldots, F_t\}$ be its set of procedures with F_0 being the `main` procedure, and let $GV(\mathcal{P})$ denote the set of global variables. For procedure F_i, let $\mathcal{L}_i$ denote the set of locations. Let $V(\mathcal{P})$ denote the set of all variables of $\mathcal{P}$, and $\mathcal{L}(\mathcal{P}) = \bigcup_{i=1}^{t} \mathcal{L}_i$ denote the set of locations of $\mathcal{P}$. For a location ℓ within a procedure F_i, let *inscope*(ℓ) denote the set of all variables in $\mathcal{P}$ whose scope includes l. We denote by *stmt*(ℓ), *formal*(ℓ) and *local*(ℓ) the statement at ℓ and the sets of formal parameters and local variables of the procedure containing ℓ, respectively. We denote by $entry_i \in \mathcal{L}_i$ the location of the first statement in F_i. For Boolean program $\mathcal{B}$, we use the same notation, replacing $\mathcal{P}$ with $\mathcal{B}$ as needed. When the context is clear, we simply use V, $\mathcal{L}$ instead of $V(\mathcal{P})$, $\mathcal{L}(\mathcal{B})$ etc.

Transition Graphs. In addition to a textual representation, we will often find it convenient to use a transition graph representation of programs (see Fig. 1c). The transition graph representation of $\mathcal{P}$, denoted $\mathcal{G}(\mathcal{P})$, comprises a set of labeled, rooted, directed graphs $\mathcal{G}_0, \ldots, \mathcal{G}_t$, with exactly one node, *err*, in common.

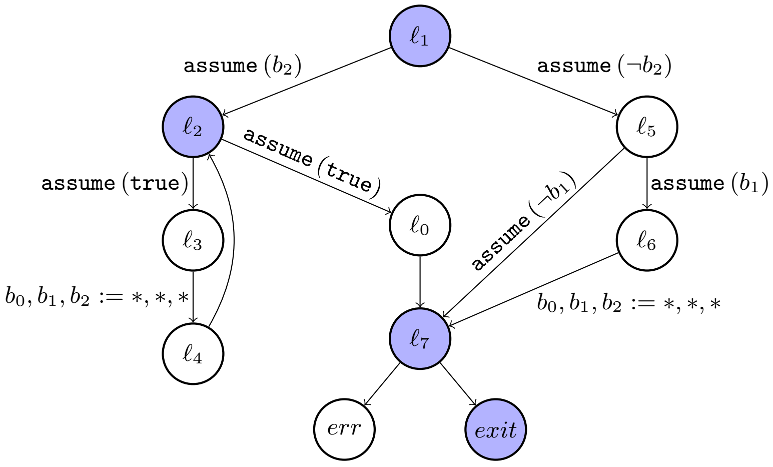
[2] We take the liberty of using `if` and `while` statements for our examples.

```
main() {
    int $x$;
    $\ell_1$ : if ($x \leq 0$)
    $\ell_2$ :     while ($x < 0$){
    $\ell_3$ :         $x := x + 2$;
    $\ell_4$ :         skip;
                   }
               else
    $\ell_5$ :     if ($x == 1$)
    $\ell_6$ :         $x := x - 1$;
    $\ell_7$ : assert ($x > 1$);
}
```

(a) $\mathcal{P}$

```
main() {
    /* $\gamma(b_0) = x \leq 1$, $\gamma(b_1) = x == 1$, $\gamma(b_2) = x \leq 0$ */
    Bool $b_0, b_1, b_2 := *, *, *$;
    $\ell_1$ : if ($\neg b_2$) then goto $\ell_5$;
    $\ell_2$ : if ($*$) then goto $\ell_0$;
    $\ell_3$ : $b_0, b_1, b_2 := *, *, *$;
    $\ell_4$ : goto $\ell_2$;
    $\ell_0$ : goto $\ell_7$;
    $\ell_5$ : if ($\neg b_1$) then goto $\ell_7$;
    $\ell_6$ : $b_0, b_1, b_2 := *, *, *$;
    $\ell_7$ : assert ($\neg b_0$);
}
```

(b) $\mathcal{B}$



(c) $\mathcal{G}(\mathcal{B})$

Fig. 1. An example concrete program $\mathcal{P}$, a corresponding Boolean program $\mathcal{B}$ and $\mathcal{B}$'s transition graph

Informally, the i^{th} graph $\mathcal{G}_i$ captures the flow of control in procedure F_i with its nodes and edges labeled by locations and corresponding statements of F_i, respectively (see [23] for a formal definition). The set N_i of nodes of $\mathcal{G}_i$, given by $\mathcal{L}_i \cup exit_i \cup err$, includes a unique entry node $entry_i$, a unique exit node $exit_i$ and the error node err (every node ℓ with $stmt(\ell)$ being an assert statement has two successors, one of which is err). A path π in $\mathcal{G}_i$ is a sequence of labeled connected edges; we denote the sequence of statements labeling the edges in π as $stmt(\pi)$.

Program Semantics and Correctness. An operational semantics can be defined for our programs in an obvious way, by formalizing the effect of each type of program statement on a program *configuration.* A configuration η of a program $\mathcal{P}$ is a tuple of the form (ℓ, Ω, ζ), where $\ell \in \mathcal{L}(\mathcal{P})$, Ω is a valuation of the variables in $\mathit{inscope}(\ell)$ and ζ is a stack of elements. Each element of ζ is of the form $(\widetilde{\ell}, \widetilde{\Omega})$, where $\widetilde{\ell}$ is a location and $\widetilde{\Omega}$ is a valuation of the variables in $\mathit{local}(\widetilde{\ell})$. A configuration (ℓ, Ω, ζ) of $\mathcal{P}$ is called an initial configuration if $\ell = \mathit{entry}_0$ and ζ is the empty stack. We use $\eta \rightsquigarrow \eta'$ to denote that $\mathcal{P}$ can transition from configuration η to η'; the transitions rules for each type of program statement at ℓ are standard (see [23] for details).

An *execution path* of program $\mathcal{P}$ is a sequence of configurations, $\eta \rightsquigarrow \eta' \rightsquigarrow \eta'' \rightsquigarrow \ldots$, obtained by repeated application of transition rules, starting from an initial configuration η. An execution path may be finite or infinite. The last configuration (ℓ, Ω, ζ) of a finite execution path may either be a *terminating configuration* with $\ell = \mathit{exit}_0$, or an *error configuration* with $\ell = \mathit{err}$ or a *stuck configuration.* An execution path ends in a stuck configuration η if no transition rule is applicable to η.

An assertion in program $\mathcal{P}$, is a statement of the form ℓ : `assert` (g), and represents the expected valuation of the program variables at location ℓ. We will use the term assertion to denote both the statement ℓ : `assert` (g) as well as the quantifier-free, first order expression g. We say a program configuration (ℓ, Ω, ζ) satisfies an assertion g, if the embedded variable valuation Ω satisfies g.

Given a program $\mathcal{P}$ (or, $\mathcal{B}$) annotated with a set of assertions, $\mathcal{P}$ (or, $\mathcal{B}$) is *partially correct* iff every *finite* execution path of $\mathcal{P}$ (or $\mathcal{B}$) ends in a terminating configuration (for all nondeterministic choices that $\mathcal{B}$ might make). $\mathcal{P}$ (or, $\mathcal{B}$) is *totally correct* iff every execution path is finite and ends in a terminating configuration (for all nondeterministic choices that $\mathcal{B}$ might make). Unless otherwise specified, an *incorrect* program is one that is not partially correct.

3 Cost-Aware Program Repair

3.1 The Problem

Let $\Sigma = \{\texttt{skip}, \texttt{assign}, \texttt{assume}, \texttt{assert}, \texttt{call}, \texttt{return}, \texttt{goto}\}$ denote the set of *statement types* in program $\mathcal{P}$. Given a statement s, let $\tau(s)$ be an element of Σ denoting the statement type of s. Let $\mathcal{U} = \{u_0, u_1, \ldots, u_d\}$ be a set of permissible, statement-level *update schemas*: $u_0 = \mathit{id}$ is the *identity* update schema that maps every statement to itself, and u_i, $i \in [1, d]$, is a function $\sigma \mapsto \widehat{\sigma}$, $\sigma, \widehat{\sigma} \in \Sigma \setminus \{\texttt{assert}\}$, that maps a statement type to a statement type. For each update schema u, given by $\sigma \mapsto \widehat{\sigma}$, we say u can be *applied* to statement s to get statement $\widehat{s}$ if $\tau(s) = \sigma$; we then have $\tau(\widehat{s}) = \widehat{\sigma}$. For example, u, given by $\texttt{assign} \mapsto \texttt{assign}$, can be applied to the assignment statement $\ell : x := y$ to get an assignment statement $\ell : y := x + 1$. Notice that update schemas in $\mathcal{U}$ do not affect the label of a statement, and that we do not permit modifying an `assert` statement. In this paper, we fix the following set of update schemas:

$$\begin{aligned}\mathcal{U} = \{\mathit{id}, &\texttt{assign} \mapsto \texttt{assign}, \texttt{assign} \mapsto \texttt{skip}, \texttt{assume} \mapsto \texttt{assume}, \\ &\texttt{call} \mapsto \texttt{call}, \texttt{call} \mapsto \texttt{skip}\}.\end{aligned} \tag{1}$$

We extend the notion of a statement-level update to a program-level update as follows. For programs $\mathcal{P}$, $\widehat{\mathcal{P}}$, let the respective sets of locations be $\mathcal{L}$, $\widehat{\mathcal{L}}$ and let

$stmt(\ell)$, $\widehat{stmt}(\ell)$ denote the respective statements at location ℓ. Let $\mathbb{R}_{\mathcal{U},\mathcal{L}} : \mathcal{L} \mapsto \mathcal{U}$ be an *update function* that maps each location of $\mathcal{P}$ to an update schema in $\mathcal{U}$. We say $\widehat{\mathcal{P}}$ is an $\mathbb{R}_{\mathcal{U},\mathcal{L}}$-update of $\mathcal{P}$ iff $\mathcal{L} = \widehat{\mathcal{L}}$ and for each $\ell \in \mathcal{L}$, $\widehat{stmt}(\ell)$ is obtained by applying $\mathbb{R}_{\mathcal{U},\mathcal{L}}(\ell)$ on $stmt(\ell)$.

Let $c_{\mathcal{U},\mathcal{L}} : \mathcal{U} \times \mathcal{L} \to \mathbb{N}$ be a cost function that maps a tuple, consisting of a statement-level update schema u and a location ℓ of $\mathcal{P}$, to a certain cost. Thus, $c_{\mathcal{U},\mathcal{L}}(u, \ell)$ is the cost of applying update schema u to the $stmt(\ell)$. We impose an obvious restriction on $c_{\mathcal{U},\mathcal{L}}$: $\forall \ell \in \mathcal{L} : c_{\mathcal{U},\mathcal{L}}(id, \ell) = 0$. Since we have already fixed the set $\mathcal{U}$ and the program $\mathcal{P}$, we henceforth use c, $\mathbb{R}$ instead of $c_{\mathcal{U},\mathcal{L}}$, $\mathbb{R}_{\mathcal{U},\mathcal{L}}$, respectively, The total cost, $Cost_c(\mathbb{R})$, of performing an $\mathbb{R}$-update of $\mathcal{P}$ is given by $\sum_{\ell \in \mathcal{L}} c(\mathbb{R}(\ell), \ell)$.

Given an incorrect concrete program $\mathcal{P}$ annotated with assertions, a cost function c, and a repair budget δ, the goal of cost-aware program repair is to compute $\widehat{\mathcal{P}}$ such that:

1. $\widehat{\mathcal{P}}$ is partially correct, and,
2. there exists $\mathbb{R}$:
   (a) $\widehat{\mathcal{P}}$ is some $\mathbb{R}$-update of $\mathcal{P}$, and
   (b) $Cost_c(\mathbb{R}) \leq \delta$.

If there exists such a $\widehat{\mathcal{P}}$, we say $\widehat{\mathcal{P}}$ is a $(\mathcal{U}, c, \delta)$-*repair of* $\mathcal{P}$.

Notice that without $\mathcal{U}$, c and δ, there would be no restriction on the *relation* of the repaired program $\widehat{\mathcal{P}}$ to the incorrect program $\mathcal{P}$; in particular, $\widehat{\mathcal{P}}$ could be any correct program constructed from scratch, without using $\mathcal{P}$ at all. Insightful choices for these can help prune the search space for repaired programs and help generate a repaired program *similar* to what the programmer may have in mind.

3.2 Solution Overview

We present a predicate abstraction-based framework for cost-aware program repair. Thus, in addition to $\mathcal{P}$, c, δ, our framework requires (a) a Boolean program $\mathcal{B}$ such that $\mathcal{B}$ is obtained from $\mathcal{P}$ via iterative predicate abstraction-refinement and $\mathcal{B}$ exhibits a non-spurious counterexample path, and (b) the corresponding function γ that maps Boolean variables to their respective predicates. The computation of a suitable repaired program $\widehat{\mathcal{P}}$ involves two main steps:

1. Cost-aware repair of $\mathcal{B}$ to obtain $\widehat{\mathcal{B}}$, and
2. Concretization of $\widehat{\mathcal{B}}$ to obtain $\widehat{\mathcal{P}}$.

In the following sections, we describe these two steps in detail.

4 Cost-Aware Repair of Boolean Programs

Our solution to cost-aware repair of a Boolean program $\mathcal{B}$ relies on automatically computing *inductive assertions*, along with a suitable $\widehat{\mathcal{B}}$, that together certify the partial correctness of $\widehat{\mathcal{B}}$. In what follows, we explain our adaptation of the method of inductive assertions [11, 19] for cost-aware program repair.

Cut-set. Let N be the set of nodes in $\mathcal{G}(\mathcal{B})$, the transition graph representation of $\mathcal{B}$. We define a cut-set $\Lambda \subseteq N$ as a set of nodes, called *cut-points*, such that for every $i \in [0, t]$: (a) $entry_i, exit_i \in \Lambda$, (b) for every edge (ℓ, ς, ℓ') in $\mathcal{G}_i$ where $stmt(\ell)$ is a procedure `call` or an `assert` statement, $\ell, \ell' \in \Lambda$, and (c) every cycle in $\mathcal{G}$ contains at least one node in Λ. A pair of cut-points ℓ, ℓ' is said to be *adjacent* if every path from ℓ to ℓ' contains no other cut-point. A *verification path* is any path from a cut-point to an adjacent cut-point.
Example: The set $\{\ell_1, \ell_2, \ell_7, exit\}$ (shaded blue) in Fig. 1c is a valid cut-set for Boolean program $\mathcal{B}$ in Fig. 1b. The verification paths in $\mathcal{G}(\mathcal{B})$ corresponding to this cut-set are: (1) $\ell_1 \xrightarrow{\texttt{assume}\,(b_2)} \ell_2$, (2) $\ell_2 \xrightarrow{\texttt{assume}\,(\texttt{T})} \ell_3 \xrightarrow{b_0, b_1, b_2 := *, *, *} \ell_4 \rightarrow \ell_2$, (3) $\ell_2 \xrightarrow{\texttt{assume}\,(\texttt{T})} \ell_0 \rightarrow \ell_7$, (4) $\ell_1 \xrightarrow{\texttt{assume}\,(\neg b_2)} \ell_5 \xrightarrow{\texttt{assume}\,(\neg b_1)} \ell_7$, (5) $\ell_1 \xrightarrow{\texttt{assume}\,(\neg b_2)} \ell_5 \xrightarrow{\texttt{assume}\,(b_1)} \ell_6 \xrightarrow{b_1, b_1, b_2 := *, *, *} \ell_7$ and (6) $\ell_7 \rightarrow exit$.

Inductive assertions. We denote an inductive assertion associated with cut-point ℓ in Λ by $\mathcal{I}_\ell$. Informally, an inductive assertion $\mathcal{I}_\ell$ has the property that whenever control reaches ℓ in any program execution, $\mathcal{I}_\ell$ must be `true` for the current values of the variables in scope. For Boolean program $\mathcal{B}$, $\mathcal{I}_\ell$ is a Boolean formula over $V_s[\ell]$, where $V_s[\ell]$ denotes an ℓ^{th} copy of the subset V_s of the program variables, with $V_s = GV \cup formal(\ell)$ if $\ell \in \{exit_1, \ldots, exit_t\}$, and $V_s = inscope(\ell)$ otherwise. Thus, except for the `main` procedure, the inductive assertions at the exit nodes of all procedures exclude the local variables declared in the procedure. Let $\mathcal{I}_\Lambda$ denote the set of inductive assertions associated with all the cut-points in Λ.

Verification Conditions. A popular approach to verification of sequential, imperative programs is to compute $\mathcal{I}_\Lambda$ such that $\mathcal{I}_\Lambda$ satisfies a set of constraints called *verification conditions.* Let π be a verification path in $\mathcal{G}_i$, from cut-point ℓ to adjacent cut-point ℓ'. The verification condition corresponding to π, denoted $VC(\pi)$, is essentially the Hoare triple $\langle \mathcal{I}_\ell \rangle\, stmt(\pi)\, \langle \mathcal{I}_{\ell'} \rangle$, where $stmt(\pi)$ is the sequence of statements labeling π. When $\mathcal{I}_\ell$, $\mathcal{I}_{\ell'}$ are *unknown*, $VC(\pi)$ can be seen as a constraint encoding all possible solutions for $\mathcal{I}_\ell$, $\mathcal{I}_{\ell'}$ such that: every program execution along path π, starting from a set of variable valuations satisfying $\mathcal{I}_\ell$, terminates in a set of variable valuations satisfying $\mathcal{I}_{\ell'}$.

Program Verification Using the Inductive Assertions Method. Given $\mathcal{B}$ annotated with assertions, and a set Λ of cut-points, $\mathcal{B}$ is partially correct if one can compute a set $\mathcal{I}_\Lambda$ of inductive assertions such that: for every verification path π between every pair ℓ, ℓ' of adjacent cut-points in $\mathcal{G}(\mathcal{B})$, $VC(\pi)$ is valid.
Example: It is not possible to compute such a set of inductive assertions for the Boolean program in Fig. 1b as the program is incorrect.

Cost-aware Repairability Conditions. Let $\mathcal{C} : N \rightarrow \mathbb{N}$ be a function mapping nodes in $\mathcal{G}$ to costs. We find it convenient to use $\mathcal{C}_\ell$ to denote the value $\mathcal{C}(\ell)$ at node/location ℓ. We set $\mathcal{I}_{entry_0} = \texttt{true}$ and $\mathcal{C}_\ell = 0$ if $\ell \in \{entry_0, \ldots, entry_t\}$.

Informally, $\mathcal{C}_\ell$ with $\ell \in N_i$ can be seen as recording the cumulative cost of applying a sequence of update schemas to the statements in procedure F_i from $entry_i$ to ℓ. Thus, for a specific update function $\mathbb{R}$ with cost function c, $\mathcal{C}_{exit_0}$ records the total cost $Cost_c(\mathbb{R})$ of performing an $\mathbb{R}$-update of the program. Given a verification path π in $\mathcal{G}_i$, from cut-point ℓ to adjacent cut-point ℓ', we extend the definition of $VC(\pi)$ to define the cost-aware repairability condition corresponding to π, denoted $CRC(\pi)$. $CRC(\pi)$ can be seen as a constraint encoding all possible solutions for inductive assertions $\mathcal{I}_\ell$, $\mathcal{I}_{\ell'}$ and update functions $\mathbb{R}$, along with associated functions $\mathcal{C}$, such that: every program execution that proceeds along path π via statements modified by applying the update schemas in $\mathbb{R}$, starting from a set of variable valuations satisfying $\mathcal{I}_\ell$, terminates in a set of variable valuations satisfying $\mathcal{I}_{\ell'}$, for all nondeterministic choices that the program might make along π.

Before we proceed, note that $\mathcal{I}_\ell$ is a Boolean formula over $V_s[\ell]$, where for all locations $\lambda \neq \ell'$ in verification path π from ℓ to ℓ', $V_s = inscope(\lambda)$. In what follows, the notation $[\![u]\!](stmt(\lambda))$ represents the class of statements that may be obtained by applying update schema u on $stmt(\lambda)$, and is defined for our permissible update schemas in Fig. 2. Here, f, f_1, f_2 etc. denote *unknown* Boolean expressions[3], over the variables in $inscope(\lambda)$. Note that the update schema `assign` $\mapsto$ `assign`, modifies *any* assignment statement, to one that assigns unknown Boolean expressions to *all* variables in V_s.

u	$[\![u]\!](stmt(\lambda))$
id	$stmt(\lambda)$
`assign` $\mapsto$ `skip`	`skip`
`assume` $\mapsto$ `skip`	`skip`
`call` $\mapsto$ `skip`	`skip`
`assign` $\mapsto$ `assign`	$b_1, \ldots, b_{\lvert V_s \rvert} := f_1, \ldots, f_{\lvert V_s \rvert}$
`assume` $\mapsto$ `assume`	`assume` f
`call` $\mapsto$ `call`	`call` $F_j(f_1, \ldots, f_k)$, where $stmt(\lambda)$: `call` $F_j(e_1, \ldots, e_k)$

Fig. 2. Definition of $[\![u]\!](stmt(\lambda))$

We now define $CRC(\pi)$. While there are three cases to consider, due to lack of space, we only define $CRC(\pi)$ when $stmt(\pi)$ does not contain a procedure `call` or `assert` statement. We refer the reader to [23] for the definitions of $CRC(\pi)$ when $stmt(\pi)$ contains a procedure `call` and when $stmt(\pi)$ contains an `assert` statement.

Let $\mathcal{A}_\lambda$ denote a Boolean formula/assertion associated with location λ in π. $CRC(\pi)$ is given by the (conjunction of the) following set of constraints:

[3] To keep our exposition simple, we assume that these unknown Boolean expressions are deterministic. However, in our prototype tool (see Sec. 6), we have the ability to compute modified statements with nondeterministic expressions such as $*$ or `choose`(f_1, f_2).

$$\begin{aligned}
&\mathcal{A}_\ell = \mathcal{I}_\ell \\
&\mathcal{A}_{\ell'} \Rightarrow \mathcal{I}_{\ell'} \\
\bigwedge_{\ell \preceq \lambda \prec \ell'} \bigwedge_{u \in \mathcal{U}_{stmt(\lambda)}} \mathbb{R}(\lambda) = u \;&\Rightarrow\; \mathcal{C}_{\lambda'} = \mathcal{C}_\lambda + c(u,\lambda) \wedge \\
&\qquad \mathcal{A}_{\lambda'} = sp([\![u]\!](stmt(\lambda)), \mathcal{A}_\lambda).
\end{aligned} \tag{2}$$

In the above, $\prec$ denotes the natural ordering over the sequence of locations in π with λ, λ' being consecutive locations. The notation $\mathcal{U}_{stmt(\lambda)} \subseteq \mathcal{U}$ denotes the set of all update schemas in $\mathcal{U}$ which may be applied to $stmt(\lambda)$. The notation $sp([\![u]\!](stmt(\lambda)), \mathcal{A}_\lambda)$ denotes the strongest postcondition of the assertion $\mathcal{A}_\lambda$ over the class of statements $[\![u]\!](stmt(\lambda))$. We define this strongest postcondition using multiple variable copies - a copy $V_s[\lambda]$ for each location λ in π. Let us assume that $\mathcal{A}_\lambda$ is a Boolean formula of the form[4]:

$$\mathcal{A}_\lambda = \rho[\ell, \grave{\lambda}] \wedge \bigwedge_{b \in V_s} b[\lambda] = \xi[\grave{\lambda}], \tag{3}$$

where $\grave{\lambda}$, λ are consecutive locations in π, $\rho[\ell, \grave{\lambda}]$ is a Boolean expression over all copies $V_s[\mu]$, $\ell \preceq \mu \preceq \grave{\lambda}$, representing the path condition imposed by the program control-flow, and $\xi[\grave{\lambda}]$ is a Boolean expression over $V_s[\grave{\lambda}]$ representing the λ^{th} copy of each variable b in terms of the $\grave{\lambda}^{th}$ copy of the program variables. Note that $\mathcal{A}_\ell = \mathcal{I}_\ell$ is of the form $\rho[\ell]$.

$[\![u]\!](stmt(\lambda))$	$sp([\![u]\!](stmt(\lambda)), \mathcal{A}_\lambda)$
`skip` `goto`	$\rho[\ell, \grave{\lambda}] \wedge \bigwedge_{b \in V_s} b[\lambda'] = b[\lambda]$
`assume` g	$g[\lambda] \wedge \rho[\ell, \grave{\lambda}] \wedge \bigwedge_{b \in V_s} b[\lambda'] = b[\lambda]$
`assume` f	$f[\lambda] \wedge \rho[\ell, \grave{\lambda}] \wedge \bigwedge_{b \in V_s} b[\lambda'] = b[\lambda]$
$b_1, \ldots, b_m := e_1, \ldots, e_m$	$\rho[\ell, \grave{\lambda}] \wedge \bigwedge_{b_i \in V_s, i \in [1,m]} b_i[\lambda'] = e_i[\lambda] \wedge \bigwedge_{b_i \in V_s, i \notin [1,m]} b_i[\lambda'] = b_i[\lambda]$
$b_1, \ldots, b_{\lvert V_s \rvert} := f_1, \ldots, f_{\lvert V_s \rvert}$	$\rho[\ell, \grave{\lambda}] \wedge \bigwedge_{b_i \in V_s} b_i[\lambda'] = f_i[\lambda]$

Fig. 3. Definition of $sp([\![u]\!](stmt(\lambda)), \mathcal{A}_\lambda)$

Given the above $\mathcal{A}_\lambda$, $sp([\![u]\!](stmt(\lambda)), \mathcal{A}_\lambda)$ is defined in Fig. 3. Observe that $sp([\![u]\!](stmt(\lambda)), \mathcal{A}_\lambda)$ is a Boolean formula of the same form as (3), over variable

[4] In general, $\mathcal{A}_\lambda$ is a disjunction over Boolean formulas of this form; $sp([\![u]\!](stmt(\lambda)), \mathcal{A}_\lambda)$ can then be obtained by computing a disjunction over the strongest postconditions obtained by propagating each such Boolean formula through $[\![u]\!](stmt(\lambda))$ using the rules in Fig. 3.

copies from $V_s[\ell]$ to $V_s[\lambda']$. For the entries `assume` g and $b_1, \ldots, b_m := e_1, \ldots, e_m$, the expressions $g, e_1, \ldots, e_m$ are *known* beforehand (these entries correspond to $u = id$). For the entries `assume` f and $b_1, \ldots, b_{|V_s|} := f_1, \ldots, f_{|V_s|}$, the expressions $f, f_1, \ldots, f_{|V_s|}$ are *unknown* (these entries correspond to $u =$ `assume` $\mapsto$ `assume` and $u =$ `assign` $\mapsto$ `assign`, respectively). Notation such as $f[\lambda]$ denotes that f is an unknown Boolean expression over $V_s[\lambda]$. For nondeterministic expressions in the RHS of an assignment statement $b_1, \ldots, b_m := e_1, \ldots, e_m$, the strongest postcondition is computed as the disjunction of the strongest postconditions over all possible assignment statements obtained by substituting each $*$ expression with either `false` or `true`.

Thus, to summarize, the set of constraints in (2) encodes all $\mathcal{I}_\ell$, $\mathcal{C}_\ell$, $\mathcal{I}_{\ell'}$, $\mathcal{C}_{\ell'}$ and $\mathbb{R}$ such that: if $\mathbb{R}$ is applied to the sequence of statements $stmt(\pi)$ to get some modified sequence of statements, say $\widehat{stmt}(\pi)$, and program execution proceeds along $\widehat{stmt}(\pi)$, then $sp(\widehat{stmt}(\pi), \mathcal{I}_\ell) \Rightarrow \mathcal{I}_{\ell'}$, and $\mathcal{C}_{\ell'}$ equals the cumulative modification cost, counting up from $\mathcal{C}_\ell$.

Cost-aware Boolean Program Repair. Given a cut-set Λ of $\mathcal{G}(\mathcal{B})$, let Π_Λ be the set of all verification paths between every pair of adjacent cut-points in Λ. Consider the following formula:

$$\exists \mathit{Unknown}\, \forall \mathit{Var} : \ \mathcal{C}_{exit_0} \leq \delta \ \wedge \bigwedge_{\pi \in \Pi_\Lambda} \mathit{CRC}(\pi) \ \wedge \ \mathit{AssumeConstraints} \qquad (4)$$

where *Unknown* is the set of all unknowns and *Var* is the set of all Boolean program variables and their copies used in encoding each $CRC(\pi)$. The set of unknowns includes the set $\mathcal{I}_\Lambda$ of inductive assertions, update function $\mathbb{R}$, unknown expressions f, f_1 etc. associated with applications of update schemas in $\mathbb{R}$ and valuations at each program location of the cumulative-cost-recording function $\mathcal{C}$. Finally, *AssumeConstraints* ensures that any modifications to the guards of `assume` statements corresponding to the same conditional statement are consistent. Thus, for every pair of *updated* `assume` (f_1), `assume` (f_2) statements labeling edges starting from the same node in the transition graph, the unknown functions f_1, f_2 are constrained to satisfy $f_1 = \neg f_2$.

If the above formula is `true`, then we can extract models for all the unknowns from the witness to the satisfiability of the formula: $\forall \mathit{Var} : \mathcal{C}_{exit_0} \leq \delta \wedge \bigwedge_{\pi \in \Pi_\Lambda} \mathit{CRC}(\pi) \wedge \mathit{AssumeConstraints}$. In particular, we can extract an $\mathbb{R}$ and the corresponding modified statements to yield a correct Boolean program $\widehat{\mathcal{B}}$. The following theorem states the soundness and completeness of the above algorithm for repairing Boolean programs for partial correctness (see [23] for the proof).

Theorem 41. *Given the set $\mathcal{U}$ specified in (1), and given an incorrect Boolean program $\mathcal{B}$ annotated with assertions, cost function c and repair budget δ,*

1. *if there exists a $(\mathcal{U}, c, \delta)$-repair of $\mathcal{B}$, the above method finds a $(\mathcal{U}, c, \delta)$-repair of $\mathcal{B}$,*
2. *if the above method finds a $\widehat{\mathcal{B}}$, then $\widehat{\mathcal{B}}$ is a $(\mathcal{U}, c, \delta)$-repair of $\mathcal{B}$.*

Example: For the Boolean program in Fig. 1b, our tool modifies two statements: (1) the guard for $stmt(\ell_1)$ is changed from $b2$ to $b0 \vee b1 \vee \neg b2$ and (2) the guard for $stmt(\ell_2)$ is changed from $*$ to $b0 \vee b1 \vee b2$.

5 Concretization

We now present the second step in our framework for computing a concrete repaired program $\widehat{\mathcal{P}}$. In what follows, we assume that we have already extracted models for $\widehat{\mathcal{B}}$ and $\mathcal{I}_\Lambda$.

Concretization of $\widehat{\mathcal{B}}$. This involves computing a mapping, denoted Γ, from each modified statement of $\widehat{\mathcal{B}}$ into a corresponding modified statement in the concrete program. We define Γ for each type of modified statement in $\widehat{\mathcal{B}}$. Let us fix our attention on a statement at location ℓ, with $V_s(\mathcal{B})$, $V_s(\mathcal{P})$ denoting the set of abstract, concrete program variables, respectively, whose scope includes ℓ. Let $r = |V_s(\mathcal{B})|$ and $q = |V_s(\mathcal{P})|$.

1. $\Gamma(\texttt{skip}) = \texttt{skip}$
2. $\Gamma(\texttt{assume}\,(g)) = \texttt{assume}\,(\gamma(g))$
3. $\Gamma(\texttt{call}\ F_i(e_1, \ldots, e_k)) = \texttt{call}\ F_i(\gamma(e_1), \ldots, \gamma(e_k))$
4. The definition of Γ for an assignment statement is non-trivial. In fact, in this case, Γ may be the empty set, or may contain multiple concrete assignment statements.
   We say that an assignment statement $b_1, \ldots, b_r := e_1, \ldots, e_r$ in $\mathcal{B}$ is *concretizable* if one can compute expressions $f_1, \ldots, f_q$ over $V_s(\mathcal{P})$, of the same type as the concrete program variables $v_1, \ldots, v_q$ in $V_s(\mathcal{P})$, respectively, such that a certain constraint is valid. To be precise, $b_1, \ldots, b_r := e_1, \ldots, e_r$ in $\mathcal{B}$ is concretizable if the following formula is `true`:

$$\exists f_1, \ldots, f_q\, \forall v_1, \ldots, v_q : \bigwedge_{j=1}^{r} \gamma(b_j)[v_1/f_1, \ldots, v_q/f_q] \ = \ \gamma(e_j) \qquad (5)$$

   Each quantifier-free constraint $\gamma(b_j)[v_1/f_1, \ldots, v_q/f_q] \ = \ \gamma(e_j)$ above essentially expresses the concretization of the abstract assignment $b_j = e_j$. The substitutions $v_1/f_1, \ldots, v_q/f_q$ reflect the *new* values of the concrete program variables after the concrete assignment $v_1, \ldots, v_q := f_1, \ldots, f_q$. If the above formula is `true`, we can extract models $expr_1, \ldots, expr_q$ for $f_1, \ldots, f_q$, respectively, from the witness to the satisfiability of the inner $\forall$-formula. We then say:

$$v_1, \ldots, v_q := expr_1, \ldots, expr_q \ \in \ \Gamma(b_1, \ldots, b_r := e_1, \ldots, e_r).$$

Example: For our example in Fig. 1, the modified guards, $b0 \vee b1 \vee \neg b2$ and $b0 \vee b1 \vee b2$, in $stmt(\ell_1)$ and $stmt(\ell_2)$ of $\widehat{\mathcal{B}}$, respectively are concretized into `true` and $x \leq 1$, respectively using γ.

Template-based concretization of $\widehat{\mathcal{B}}$. Our framework/tool can also accept user-supplied templates, specifying the desired syntax of the expressions in concrete modified statements. The concretization of $\widehat{\mathcal{B}}$ is then guided by the given templates. This is another avenue for incorporating programmer expertise and intent into automatic program repair. Due to lack of space, we skip a detailed description and refer the interested reader to [23].

Concretization of inductive assertions. The concretization of each inductive assertion $\mathcal{I}_\ell \in \mathcal{I}_\Lambda$ is simply $\gamma(\mathcal{I}_\ell)$.

6 Experiments with a Prototype Tool

We have built a prototype tool for repairing Boolean programs. The tool accepts Boolean programs generated by the predicate abstraction tool SATABS (version 3.2) [8] from sequential C programs. In our experience, we found that for C programs with multiple procedures, SATABS generates (single procedure) Boolean programs with all procedure calls inlined within the calling procedure. Hence, we only perform intraprocedural analysis in this version of our tool. The set of update schemas handled currently is $\{id, \texttt{assign} \rightarrow \texttt{assign}, \texttt{assume} \rightarrow \texttt{assume}\}$; we do not handle statement deletions. We set the costs $c(\texttt{assign} \rightarrow \texttt{assign}, \ell)$ and $c(\texttt{assume} \rightarrow \texttt{assume}, \ell)$ to some large number for every location ℓ where we wish to disallow statement modifications, and to 1 for all other locations — we essentially search for a repaired program with at most δ modifications amongst candidate locations. We initialize the tool with $\delta = 1$. We also provide the tool with a cut-set of locations for its Boolean program input.

The tool automatically generates an SMT query corresponding to the inner $\forall$-formula in (4). When generating this repairability query, for update schemas involving expression modifications, we stipulate every deterministic Boolean expression g be modified into a (unknown) deterministic Boolean expression f (as described in Fig. 2), and every nondeterministic Boolean expression be modified into a (unknown) nondeterministic expression of the form $\texttt{choose}(f_1, f_2)$. The SMT query is then fed to the SMT-solver Z3 (version 4.3.1) [20]. The solver either declares the formula to be satisfiable, and provides models for all the unknowns, or declares the formula to be unsatisfiable. In the latter case, we can choose to increase the repair budget by 1, and repeat the process.

Once the solver provides models for all the unknowns, we can extract a repaired Boolean program automatically. Currently, the next step, concretization, is automated in part. For assignment statements, we manually formulate SMT queries corresponding to the inner $\forall$-formula in (5), and feed these queries to Z3. If the relevant queries are satisfiable, we can obtain a repaired C program. If the queries are unsatisfiable, we attempt template-based concretization using linear-arithmetic templates. In some experiments, we allowed ourselves a degree of flexibility in guiding the solver to choose the right template parameters.

In Fig. 4, we present some of the details of repairing a C program drawn from the NEC Laboratories Static Analysis Benchmarks [21]. After our tool automatically generated a repaired Boolean program for this example, we manually wrote

```
int main() {
   int $x, y$;
   int $a[10]$;
   $\ell_1$ : $x := 1U$;
   $\ell_2$ : while ($x \leq 10U$) {
   $\ell_3$ :     $y := 11 - x$;
   $\ell_4$ :     assert ($y \geq 0 \wedge y < 10$);
   $\ell_5$ :     $a[y] := -1$;
   $\ell_6$ :     $x := x + 1$;
          }
}
```

Boolean program vars/predicates:
$\gamma(b_0) = y < 0$, $\gamma(b_1) = y < 10$

Boolean program repair:
Change $stmt(\ell_3)$ from
$b_0, b_1 := *, *$ to $b_0, b_1 :=$ F, T

Concrete program repair:
Change $stmt(\ell_3)$ to $y := 10 - x$

Inductive Assertions:
They were all *true*

Fig. 4. Repairing program `necex14`

an SMT query corresponding to (5) to concretize the assignment statement at location ℓ_3, and obtained $y := 0$ as the repair for the concrete program. Unsatisfied by this repair, we formulated a template-based SMT query, restricting the RHS of $stmt(\ell_3)$ to the template $-x + c$, where c is unknown. The query was found to be satisfiable, and yielded $c = 10$. As shown in Fig. 4, all inductive assertions generated for this example were `true`.

In Table 1, we present the results of repairing some handmade examples (`handmade2` is the same example as in Fig. 1), and some benchmark programs from NEC Labs [21] and the 2014 Competition on Software Verification [9]. The complexity of the programs from [9] stems from nondeterministic assignments and function invocations within loops. All experiments were run on the same machine, an Intel Dual Core 2.13GHz Unix desktop with 4 GB of RAM.

We enumerate the time taken for each individual step involved in generating a repaired Boolean program. The columns labeled LoC($\mathcal{P}$) and LoC($\mathcal{B}$) enumerate the number of lines of code in the original C program and the Boolean program generated by SATABS, respectively. The column labeled $V(\mathcal{B})$ enumerates the number of variables in each Boolean program. The column $\mathcal{B}$-time enumerates the time taken by SATABS to generate each Boolean program, the column Que-time enumerates the time taken by our tool to generate each repairability query and the column Sol-time enumerates the time taken by Z3 to solve the query. The columns # `Asg` and # `Asm` count the number of `assign` → `assign` and `assume` → `assume` update schemas applied, respectively, to obtain the final correct program.

Notice that our implementation either produces a repaired program very quickly, or fails to do so in reasonable time whenever there is a significant increase in the number of Boolean variables, e.g. for `veris.c_NetBSD-libc__loop_true`. This is because the SMT solver might need to search over simultaneous nondeterministic assignments to all the Boolean variables for every assignment statement in $\mathcal{B}$ in order to solve the repairability query. For the last two programs, SATABS was the main bottleneck, with SATABS failing to generate a Boolean program with a non-spurious counterexample after 10 minutes; we experienced issues while using SATABS on programs with a lot of character manipulation.

We emphasize that when successful, our tool can repair a diverse set of errors in programs containing loops, multiple procedures and pointer and array variables. In our benchmarks, we were able to repair operators (e.g., an incorrect

Table 1. Experimental results

Name	LoC($\mathcal{P}$)	LoC($\mathcal{B}$)	$V(\mathcal{B})$	$\mathcal{B}$-time	Que-time	Sol-time	# Asg	# Asm
`handmade1`	6	58	1	0.180s	0.009s	0.012s	0	1
`handmade2`	16	53	3	0.304s	0.040s	0.076s	0	2
`necex6`	24	66	3	0.288s	0.004s	0.148s	1	0
`necex14`	13	60	2	0.212s	0.004s	0.032s	1	0
`while_infinite_loop_1_true`	5	33	1	0.196s	0.002s	0.008s	0	1
`array_true`	23	57	4	0.384s	0.004s	0.116s	1	1
`n.c11_true`	27	50	2	0.204s	0.002s	0.024s	1	0
`terminator_03_true`	22	38	2	0.224s	0.004s	0.036s	1	1
`trex03_true`	23	58	3	0.224s	0.036s	0.540s	1	1
`trex04_true`	29	36	1	0.200s	0.004s	0.004s	2	0
`veris.c_NetBSD − libc__loop_true`	30	144	23	3.856s	-	-	-	-
`vogal_true`	41	-	-	$> 10m$	-	-	-	-
`count_up_down_true`	18	-	-	$> 10m$	-	-	-	-

conditional statement $x < 0$ was repaired to $x > 0$) and array indices (e.g., an incorrect assignment $x{:=}a[0]$ was repaired to $x{:=}a[j]$), and modify constants into program variables (e.g. an incorrect assignment $x{:=}0$ was repaired to $x{:=}d$, where d was a program variable). Also, note that for many benchmarks, the repaired programs required multiple statement modifications.

7 Discussion

The framework described in this paper computes a repaired concrete program in two separate steps: computation of a repaired Boolean program $\widehat{\mathcal{B}}$, followed by its concretization. The separation of these two steps is not necessary and is potentially sub-optimal. It may not be possible to concretize a repaired Boolean program computed in the first step, while there may exist some other concretizable $\widehat{\mathcal{B}}$. The solution is to directly search for $\widehat{\mathcal{B}}$ such that all modified statements of $\widehat{\mathcal{B}}$ are concretizable. This can be done by combining the constraints presented in Sec. 5 with the one in (4). As noted in Sec. 1, we can target total correctness of the repaired programs by associating ranking functions along with inductive assertions with each cut-point in Λ, and including termination conditions as part of the constraints. Finally, we wish to explore ways to ensure that the repaired program does not unnecessarily restrict correct behaviors of the original program. We conjecture that this can be done by computing the weakest possible set of inductive assertions and a least restrictive $\widehat{\mathcal{B}}$.

Acknowledgements. The authors would like to thank Gérard Basler, Daniel Kröning and Georg Weissenbacher for their help with SATABS.

References

1. Arcuri, A.: On the Automation of Fixing Software Bugs. In: International Conference on Software Engineering (ICSE), pp. 1003–1006. ACM (2008)
2. Ball, T., Bounimova, E., Kumar, R., Levin, V.: SLAM2: Static Driver Verification with under 4% False Alarms. In: Formal Methods in Computer Aided Design (FMCAD), pp. 35–42 (2010)

3. Ball, T., Naik, M., Rajamani, S.K.: From Symptom to Cause: Localizing Errors in Counterexample Traces. In: Principles of Programming Languages (POPL), pp. 97–105. ACM (2003)
4. Ball, T., Rajamani, S.K.: Boolean Programs: A Model and Process for Software Analysis. Tech. Rep. 2000-14, MSR (2000)
5. Ball, T., Rajamani, S.K.: Automatically Validating Temporal Safety Properties of Interfaces. In: Dwyer, M.B. (ed.) SPIN 2001. LNCS, vol. 2057, pp. 103–122. Springer, Heidelberg (2001)
6. Bloem, R., Chatterjee, K., Henzinger, T.A., Jobstmann, B.: Better Quality in Synthesis through Quantitative Objectives. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 140–156. Springer, Heidelberg (2009)
7. Chandra, S., Torlak, E., Barman, S., Bodik, R.: Angelic Debugging. In: International Conference on Software Engineering (ICSE), pp. 121–130. ACM (2011)
8. Clarke, E., Kroning, D., Sharygina, N., Yorav, K.: SATABS: SAT-based Predicate Abstraction for ANSI-C. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 570–574. Springer, Heidelberg (2005)
9. Competition on Software Verification (SV-COMP): Loops Benchmarks (2014), `http://sv-comp.sosy-lab.org/2014/benchmarks.php`
10. Debroy, V., Wong, W.E.: Using Mutation to Automatically Suggest Fixes for Faulty Programs. In: Software Testing, Verification and Validation (ICST), pp. 65–74 (2010)
11. Floyd, R.W.: Assigning Meanings to Programs. In: Mathematical Aspects of Computer Science, pp. 19–32. American Mathematical Society (1967)
12. Goues, C.L., Dewey-Vogt, M., Forrest, S., Weimer, W.: A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 Each. In: International Conference on Software Engineering (ICSE), pp. 3–13. IEEE Press (2012)
13. Graf, S., Saïdi, H.: Construction of Abstract State Graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
14. Griesmayer, A., Bloem, R., Cook, B.: Repair of Boolean Programs with an Application to C. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 358–371. Springer, Heidelberg (2006)
15. Jobstmann, B., Griesmayer, A., Bloem, R.: Program Repair as a Game. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 226–238. Springer, Heidelberg (2005)
16. Jose, M., Majumdar, R.: Cause Clue Clauses: Error Localization using Maximum Satisfiability. In: Programming Language Design and Implementation (PLDI), pp. 437–446. ACM (2011)
17. Könighofer, R., Bloem, R.: Automated Error Localization and Correction for Imperative Programs. In: Formal Methods in Computer Aided Design (FMCAD), pp. 91–100 (2011)
18. Logozzo, F., Ball, T.: Modular and Verified Automatic Program Repair. In: Object Oriented Programming Systems Languages and Applications (OOPSLA), pp. 133–146. ACM (2012)
19. Manna, Z.: Introduction to Mathematical Theory of Computation. McGraw-Hill, Inc. (1974)
20. de Moura, L., Bjørner, N.S.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
21. NEC: NECLA Static Analysis Benchmarks, `http://www.nec-labs.com/research/system/systems_SAV-website/benchmarks.php`

22. Samanta, R., Deshmukh, J.V., Emerson, E.A.: Automatic Generation of Local Repairs for Boolean Programs. In: Formal Methods in Computer Aided Design (FMCAD), pp. 1–10 (2008)
23. Samanta, R., Olivo, O., Emerson, E.A.: Cost-Aware Automatic Program Repair. CoRR abs/1307.7281 (2013)
24. Singh, R., Gulwani, S., Solar-Lezama, A.: Automatic Feedback Generation for Introductory Programming Assignments. In: Programming Language Design and Implementation, PLDI (2013)
25. Singh, R., Solar-Lezma, A.: Synthesizing Data-Structure Manipulations from Storyboards. In: Foundations of Software Engineering (FSE), pp. 289–299 (2011)
26. Solar-Lezama, A., Rabbah, R., Bodik, R., Ebcioglu, K.: Programming by Sketching for Bit-streaming Programs. In: Programming Language Design and Implementation (PLDI), pp. 281–294. ACM (2005)
27. Solar-Lezama, A., Tancau, L., Bodik, R., Seshia, S., Saraswat, V.: Combinatorial Sketching for Finite Programs. In: Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 404–415. ACM (2006)
28. Srivastava, S., Gulwani, S., Foster, J.S.: From Program Verification to Program Synthesis. In: Principles of Programming Languages (POPL), pp. 313–326. ACM (2010)
29. Wei, Y., Pei, Y., Furia, C.A., Silva, L.S., Buchholz, S., Meyer, B., Zeller, A.: Automated Fixing of Programs with Contracts. In: International Symposium on Software Testing and Analysis (ISSTA), pp. 61–72. ACM (2010)
30. Nokhbeh Zaeem, R., Gopinath, D., Khurshid, S., McKinley, K.S.: History-Aware Data Structure Repair using SAT. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 2–17. Springer, Heidelberg (2012)
31. Zeller, A., Hilebrandt, R.: Simplifying and Isolating Failure-Inducing Input. IEEE Trans. Softw. Eng. 28(2), 183–200 (2002)