Automatic Generation of Local Repairs for Boolean Programs

Roopsha Samanta, Jyotirmoy V. Deshmukh and E. Allen Emerson
Department of Electrical and Computer Engineering and Department of Computer Sciences,
The University of Texas at Austin, Austin TX 78712, USA
{roopsha,deshmukh,emerson}@cs.utexas.edu

***Abstract*—Automatic techniques for software verification focus on obtaining witnesses of program failure. Such counterexamples often fail to localize the precise cause of an error and usually do not suggest a repair strategy. We present an efficient algorithm to automatically generate a repair for an incorrect sequential Boolean program where program correctness is specified using a pre-condition and a post-condition. Our approach draws on standard techniques from predicate calculus to obtain annotations for the program statements. These annotations are then used to generate a synthesis query for each program statement, which if successful, yields a repair. Furthermore, we show that if a repair exists for a given program under specified conditions, our technique is always able to find it.**

I. Introduction

In spite of their significance in ensuring program correctness, techniques for automatic error localization and repair are relatively unexplored. Error traces generated by model checking and static analysis tools may be long, encumbered with unnecessary data, and may give little insight into the actual location of the error within the program. Generally, verification fails to suggest a concrete strategy for error localization or repair, and developers end up relying on their own intuition and experience to debug and fix faulty programs. On the other end of the spectrum, synthesis of programs from high level specifications has been well-studied [1]. While synthesis solves an important theoretical problem, synthesizing a large piece of code is a computationally intensive task, and mandates writing a very rich and detailed specification. Thus, it is potentially more cost-effective to debug and repair existing software than attempt to synthesize it from scratch. While one cannot hope to completely eliminate the human programmer from the debugging process, bugs that are amenable to automatic repair should be dealt with accordingly.

In this paper, we present a method for efficient and automatic repair of programs that tries to bridge the gap between manual repair and automatic synthesis. Our strategy is applicable to a large subset of sequential Boolean programs [2], and has complexity comparable to that of model checking. Popular verification tools such as SLAM [3], and BLAST [4] use Boolean programs as abstractions for programs written in higher level programming languages, as they are inherently simpler to analyze than concrete programs.

Using the imperative programming paradigm, we specify *total correctness* of a Boolean program $\mathcal{P}$ as a Hoare(-style) triple $\langle\varphi\rangle\mathcal{P}\langle\psi\rangle$, where φ is a pre-condition that specifies initial values, and ψ is a post-condition that expresses the relation between the desired final values and the initial values. Our repair algorithm has two main steps. In the first step, we annotate $\mathcal{P}$ by propagating φ and ψ through the program statements. In the second step, we choose either an automatically generated or a user-specified order to target statements for repair. For every chosen statement, we attempt to synthesize a local repair using the propagated φ and ψ. Local synthesis serves to minimize modifications made to the original program, a desirable property for any automatic repair engine. Once such a synthesis query succeeds, a repair is extracted and our algorithm halts. If all queries fail, the algorithm reports that the program is irreparable within the constraints imposed by the repair model. We prove that our algorithm is sound, and specify the conditions under which it is complete. The worst-case complexity of our algorithm is exponential in the number of program variables. In practice, the algorithm can be much more effficient due to symbolic implementation.

A. Related Work

Extant work in automatic repair of software programs is directed towards repair of both sequential and concurrent programs using different program models like Kripke structures, UNITY [5] programs, game graphs and Boolean programs. Repair is geared towards satisfying various correctness specifications like propositional assertions, invariants, UNITY properties and temporal logics. The most pertinent approach repairs Boolean programs that violate propositional assertions [6]. This work is an extension of the authors' earlier work on repairing programs with LTL specifications [7], [8], and reduces automatic repair to finding a winning strategy for a pushdown game graph (obtained from the original program). The size of such a game graph can be quite formidable, and the complexity of finding a winning strategy is doubly exponential in the number of program variables.

There has been some work in proof-directed error localization and repair [9], [10], which borrows a proof planning framework from Artificial Intelligence (AI) for proof automation. This work is still in a nascent phase, and its use is likely to be limited to functional programs. In another paper [11], which also draws on principles from AI, the authors propose a repair algorithm based on abductive reasoning and theory revision. This work focuses on reactive programs with specifications in CTL, and is not very efficient as it requires invocation of a

model checker to ascertain the validity of each repair.

Repair of Kripke structures with respect to CTL and other temporal logic formulas has received attention in both the AI [12] and formal verification [13] communities. Work done in automatic addition of UNITY properties to programs [14] is also relevant. It is worthwhile to mention *sketching* [15], [16], which is a software synthesis technique that utilizes a partial program or a *sketch* written by a programmer, and completes it to satisfy a specification.

There has been a lot of work on automatically correcting sequential and combinational circuits, and we refer the reader to a paper by Staber *et al.* [8] for a summary of such techniques. We also note a recent paper [17] in which the authors translate the problem of localizing and fixing faults in sequential circuits to solving Quantified Boolean Formulas (QBFs). While we use QBFs in our approach as well, our QBFs have only two alternating quantifiers unlike theirs, and are more tractable.

In addition to the above work, a multitude of algorithms [18], [19], [20] have been proposed for error localization based on analyzing error traces. Many of these techniques try to localize faults by finding correct program traces most similar to an incorrect counterexample trace. Some of these techniques could be exploited to obtain heuristics for the order in which we target program statements for repair.

The rest of the paper is organized as follows. In Section II we briefly look at the syntax and semantics of Boolean programs. We explain the propagation mechanism in Section III, and present our repair strategy in Section IV. We present an extension of our approach to programs with function calls in Section V, and conclude in Section VI.

II. Preliminaries

Boolean programs have a control-flow structure that is similar to their source code and closely resemble C programs. The key difference is that all variables in a Boolean program are Boolean variables. Moreover, Boolean programs support nondeterminism, an inherent by-product of the counterexample-guided abstraction refinement (CEGAR) paradigm used in tools that generate Boolean programs. In this section, we present a programming language with a syntax that greatly resembles that of Boolean programs as defined in [2], and is more amenable to our technique. For instance, we forego parallel assignments in favour of single assignments. We remove `assert` statements from the programming language and replace such statements with specifications based on pre- and post-conditions. Note that any sequential Boolean program with the syntax of [2] can be converted into an equivalent program with our syntax and correctness specification. Thus, we consider it unnecessary to distinguish between the two, and refer to our programs, with their marginally modified syntax, as Boolean programs as well.

A. Syntax and Semantics

A Boolean program $\mathcal{P}$ is defined as a tuple $(\mathcal{F}, \texttt{main}, \mathcal{V})$, where $\mathcal{F}$ is a set of functions, `main` is the initial function, and $\mathcal{V} = \{v_1, v_2, \ldots, v_t\}$ is a set of Boolean global variables. The `main` function is a tuple $(S, \mathcal{V})$, where $S = (s_1; s_2; \ldots; s_n)$ is a sequence of n statements. A function $f \in \mathcal{F}$ is a tuple $(S_f, \mathcal{V}_{f,l})$, where $S_f = (s_{f,1}; s_{f,2}; \ldots; s_{f,q})$ is a sequence of q statements and $\mathcal{V}_{f,l}$ is a set of Boolean local variables. The set of variables *visible* to function f is denoted by $\mathcal{V}_f$.

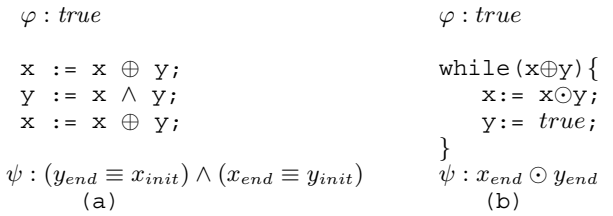

Fig. 1. Examples of Boolean programs

An expression in our programming language is a Boolean expression, and is allowed to contain a *fair* nondeterministic choice expression, $nd(0,1)$, which nondeterministically evaluates to *true* or *false*. For example, the expression $v_2 \wedge nd(0,1)$ evaluates to *false* when $v_2 = false$, and nondeterministically evaluates to *false* or *true* when $v_2 = true$. The choice is fair in the sense that it does not permanently evaluate to a single value. Program statements are either function calls or `return`, `skip`, assignment, conditional or loop statements. As in [2], every function call is required to be followed by a `skip` statement. Every function has a `return` statement. The syntax of the other statements is defined below.

Assignment statement: v_j `:=` E, where E is an expression over $2^{\mathcal{V}}$, or a function call. Each assignment statement is an assignment to a single program variable. For an assignment statement in function f, E is an expression over $2^{\mathcal{V}_f}$ or a function call.

Conditional statement: `if` (G) S_{if} `else` S_{else}, where G is an expression, and S_{if}, S_{else} are sequences of statements.

Loop statement: `while` (G) S_{body}, where G is an expression, and S_{body} is a sequence of statements.

The guard G in a conditional or loop statement is either a deterministic expression, or, the expression $nd(0,1)$. G is not allowed to be an arbitrary nondeterministic expression, which conforms with most reasonable abstraction frameworks. We show two example Boolean programs in Figure 1. Program (a) is meant to swap values of two variables without the use of a temporary variable, while Program (b) is meant to iterate till the values of x and y are the same. Both programs are incorrect.

We further associate a state transition graph with a Boolean program, with a state corresponding to a specific statement in the program and a valuation of the visible variables prior to the execution of the statement. The action of a statement in conjunction with the control-flow at that point of the program defines a transition from one state to another. In this paper, we will interchangeably use both the syntactic model and the state transition graph of a Boolean program without distinguishing them unless necessary. We now briefly describe our specification and repair model.

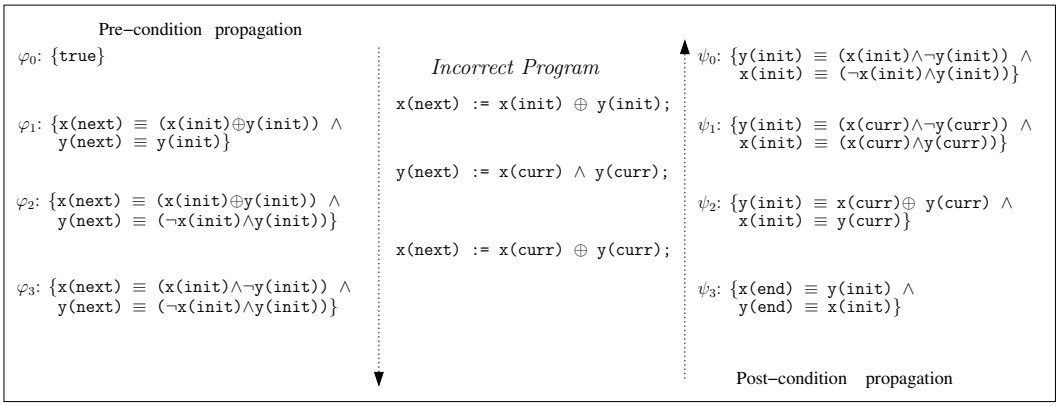

Fig. 2. Pre-condition and Post-condition Propagation

B. Specification and Repair Model

We assume that program correctness is specified as a pair of Boolean expressions known as the pre-condition φ, and the post-condition ψ. φ represents the initial states of a Boolean program and is a Boolean expression over the initial values of the global program variables $\mathcal{V}$. ψ represents the desired final states for initial states specified by φ, and is a Boolean expression relating the initial and final values of $\mathcal{V}$. φ and ψ do not contain any $nd(0,1)$ expressions. Formally, we specify total correctness of the program $\mathcal{P}$ using the Hoare triple: $\langle\varphi\rangle\mathcal{P}\langle\psi\rangle$. Further, we interpret the nondeterminism in an abstract Boolean program as Dijkstra's *demonic* nondeterminism [21]. Thus, $\mathcal{P}$ is correct if and only if the execution of $\mathcal{P}$, begun in any state satisfying φ, terminates in a state satisfying ψ, for *all* choices that $\mathcal{P}$ might make.

Assumptions made about the types of errors that could occur in a program are said to constitute an *error model*. Many repair techniques start with a set of preconceived targets for repair based on a programmer's notion of what the program should have been. Consequently, a lot of effort is expended on error localization. However, there may be a way to realize the desired specifications by "fixing" an entirely different statement than what the developer had conceived. We use this fact in our approach, and attempt to repair any statement that makes the Hoare-triple $\langle\varphi\rangle\mathcal{P}\langle\psi\rangle$ $true$. Thus, since our technique obviates the need for error localization, we focus on a *repair model*, as opposed to an error model.

The repair model captures the kinds of repairs that can be automatically generated by our technique. Our algorithm tries to repair individual program statements by either deleting them or modifying them without changing their type. A modified assignment statement may differ from the original one in both the left and right hand side of the assignment operator. For a conditional statement, a modification constitutes a change to either the test condition G or a statement within S_{if} or S_{else}. Similarly, in a loop statement, a modification changes either the loop guard G or a statement within S_{body}.

In our approach, the repair process has two main steps.

1) We annotate the program text by propagating φ and ψ through each program statement.
2) We use these annotations to inspect statements for repairability, and generate a repair if possible.

In the following sections, we treat each of these steps in detail. For ease of exposition, we first describe our algorithm for a program without any function calls, and extend our approach to programs with functions calls in Section V.

III. Step I: Program Annotation

Propagation of the initial pre-condition φ and the final post-condition ψ through program statements is based on the techniques used for Hoare logic [22], [23], with some modifications to the technique for propagating ψ. We denote the initial pre-condition φ by φ_0, and pre-conditions propagated forward through statements $s_1, \ldots, s_n$ by $\varphi_1, \ldots, \varphi_n$. Similarly, the final post-condition ψ is denoted by ψ_n, and post-conditions propagated back through statements $s_n, \ldots, s_1$ are denoted by $\psi_{n-1}, \ldots, \psi_0$.

To aid efficient computation and storage of propagated pre- and post-conditions, we add indices to identify certain valuations of the program variables. The initial set of variable values is denoted by $\mathcal{V}(init)$, and the final set of variable values obtained after execution of the n^{th} statement is denoted by $\mathcal{V}(end)$. Further, the sets of values of the variables *before* and *after* execution of any statement in the program are distinguished by representing them as $\mathcal{V}(curr)$ and $\mathcal{V}(next)$, respectively. We refer the reader to Figure 2 for the indexed and annotated version of Program (a) from Figure 1.

Henceforth, an expression E over $\mathcal{V}(curr)$ is sometimes denoted by $E(curr)$ for clarity. Moreover, if $E(curr)$ contains an $nd(0,1)$ expression, we use the notations $E(curr)|_0$ and $E(curr)|_1$ to denote the expressions obtained from $E(curr)$ when $nd(0,1)$ evaluates to $false$ and $true$, respectively. The need for maintaining these indices will become clear as we summarize the computations for propagating pre- and post-conditions in the following subsections. Note that our propagation techniques, combined with the absence of $nd(0,1)$ expressions in φ_0 and ψ_n, ensure the absence of $nd(0,1)$ expressions in all propagated pre- and post-conditions.

A. Post-condition Back Propagation

Propagation of a post-condition ψ_i through statement s_i corresponds to computing the weakest pre-condition $wp(s_i, \psi_i)$ that would make the Hoare triple $\langle \psi_{i-1} \rangle s_i \langle \psi_i \rangle$ $true$. In other words, $wp(s_i, \psi_i)$ represents the set of all *input* states such that execution of s_i is guaranteed to terminate in a state satisfying ψ_i, for all (nondeterministic) choices made by s_i. In the event that $wp(s_i, \psi_i)$ evaluates to $false$ for any statement s_i, our algorithm aborts propagation of post-conditions, and proceeds to the next phase, *i.e.*, pre-condition propagation (Section III-B) and repair generation (Section IV). Given a statement s_i and a post-condition ψ_i (an expression over $\mathcal{V}(next)$ and $\mathcal{V}(init)$), we obtain the weakest pre-condition $\psi_{i-1} = wp(s_i, \psi_i)$ using the following inductive rules:

1) Assignment statement: $v_j(next)$ `:=` $E(curr)$*:* The weakest pre-condition is computed from ψ_i by replacing the variable $v_j(next)$ in ψ_i with its assigned expression $E(curr)$, and swapping all other variables $v_m(next)$ in ψ_i with $v_m(curr)$. Thus, the resulting expression ψ_{i-1} is an expression over $\mathcal{V}(curr)$ and $\mathcal{V}(init)$ and is given by[1]:

$$\psi_{i-1} = \psi_i[v_j(next) \rightarrow E(curr), \\ \text{for each } m \neq j, v_m(next) \rightarrow v_m(curr)].$$

If $E(curr)$ contains an $nd(0,1)$ expression, the weakest pre-condition is computed as the *conjunction* of the weakest pre-conditions over the statements given by $v_j(next)$ `:=` $E(curr)|_0$ and $v_j(next)$ `:=` $E(curr)|_1$.

2) Sequential composition: To propagate the post-condition ψ_i back through a sequence of statements $(s_{i-1}; s_i)$, it is first propagated through s_i to obtain $wp(s_i, \psi_i)$; $wp(s_i, \psi_i)$ is then used as the post-condition to be propagated through s_{i-1} to obtain the required weakest pre-condition over $(s_{i-1}; s_i)$. In order to maintain the correct indices, all variables $v_m(curr)$ in the expression for $wp(s_i, \psi_i)$ are swapped with $v_m(next)$ before propagating through s_{i-1}. This rule is expressed as:
$wp((s_{i-1}; s_i), \psi_i) = wp(s_{i-1}, wp(s_i, \psi_i))$.

3) Conditional statement: `if` $(G(curr))$ S_{if} `else` S_{else}*:* The weakest pre-condition of a conditional statement is computed to be the same as the weakest pre-condition over S_{if} if the guard G is $true$, and as the weakest pre-condition over S_{else} if G is $false$. The weakest pre-condition over S_{if} and S_{else} is computed inductively using Rule 2. Thus we have:

$$\psi_{i-1} \;=\; (G(curr) \Rightarrow wp(S_{if}, \psi_i)) \;\wedge \\ (\neg G(curr) \Rightarrow wp(S_{else}, \psi_i)).$$

If $G(curr) = nd(0,1)$, the weakest pre-condition is given by the *conjunction* of the weakest pre-conditions over S_{if} and S_{else}: $wp(S_{if}, \psi_i) \wedge wp(S_{else}, \psi_i)$.

[1]We use the standard notation $\delta[y \rightarrow x]$ to represent the expression obtained by replacing all occurrences of y in δ by x.

4) Loop statement: `while` $(G(curr))$ S_{body}*:* The post-condition propagated through loop statements is computed as a fixpoint over the weakest pre-condition of each loop iteration. We define the weakest pre-condition of the k^{th} loop iteration, $wp(S^k_{body}, \psi_i)$, as the set of input states that terminate in a state satisfying ψ_i after executing the loop *at most* k times. These sets of states are given by the following inductive expressions:

$$wp(S^0_{body}, \psi_i) = Y_0 \wedge \neg G,$$
$$wp(S^k_{body}, \psi_i) = \bigvee_{l=0}^{k-1} (wp(S^l_{body}, \psi_i)) \vee (Y_k \wedge G),$$

where, $Y_0 = \psi_i$, $Y_k = wp(S_{body}, Y_{k-1} \wedge \neg G)$.

The last term in the expression for $wp(S^k_{body}, \psi_i)$ represents the set of input states that enter the loop, and exit it after exactly k loop iterations, terminating in a state satisfying ψ_i. The weakest pre-condition of the $(k-1)^{th}$ loop iteration is computed in terms of variables in $\mathcal{V}(curr)$, which are then swapped with their respective variables in $\mathcal{V}(next)$, and used for computing the weakest pre-condition of the k^{th} iteration. Since Boolean programs have a finite number of states, fixpoint computation over these monotonically increasing sets of weakest pre-conditions terminates after a bounded number of iterations, say L, by the Tarski-Knaster Theorem. The weakest pre-condition for the loop statement s_i, which corresponds to states that exit the loop eventually and satisfy ψ_i, is given by:

$$\psi_{i-1} \;=\; \bigvee_{l=0}^{L} (wp(S^l_{body}, \psi_i)), \text{ or equivalently,}$$
$$\psi_{i-1} \;=\; (\psi_i \wedge \neg G) \vee \bigvee_{l=1}^{L} wp(S_{body}, Y_{l-1} \wedge \neg G).$$

If $G(curr) = nd(0,1)$, then we define a different set of iterants Z_k, which correspond to input states that terminate in a state satisfying ψ_i in the k^{th} loop iteration when G evalues to $false$. We remind the reader that $nd(0,1)$ expresses fair choice between $true$ and $false$. Hence, the loop guard is guaranteed to evaluate to $false$ eventually. The weakest pre-condition, which corresponds to all input states that terminate in a state satisfying ψ_i for all values of the guard, can be computed as the fixpoint: $\psi_{i-1} \;=\; \bigwedge_{l=0}^{L'} Z_l$, where, $Z_0 = \psi_i$, $Z_k = wp(S_{body}, Z_{k-1})$.

B. Pre-condition Forward Propagation

Traditionally, the definition of the strongest post-condition (sp) is the dual of the weakest *liberal* pre-condition (wlp). These are used in the *partial correctness* framework and do not concern themselves with program termination [23]. Hence, the traditional sp should ideally be called the strongest *liberal* post-condition. We redefine the strongest post-condition to be the dual of the weakest pre-condition for total correctness. Thus, propagation of a pre-condition φ_{i-1} through a statement s_i corresponds to computing the strongest post-condition $sp(s_i, \varphi_{i-1})$ that would make the Hoare triple $\langle \varphi_{i-1} \rangle s_i \langle \varphi_i \rangle$ $true$. In other words, $sp(s_i, \varphi_{i-1})$ represents the smallest set of *output* states such that execution of s_i, begun in all states

satisfying φ_{i-1}, is guaranteed to terminate in one of them for all (nondeterministic) choices that s_i might make. It is important to note that the strongest post-condition is undefined if there exists *some* input state for which *some* execution of s_i may not terminate. If the strongest post-condition $sp(s_i, \varphi_{i-1})$ evaluates to *false* or is undefined for any statement s_i, our algorithm aborts propagation of pre-conditions, and proceeds to the next phase, *i.e.*, repair generation (Section IV). We refer the interested reader to Appendix A for a detailed treatment of strongest post-condition computation for each type of program statement.

IV. Step II: Synthesis of Repairs

In this section, we present an algorithm to generate repairs given an annotated program $\mathcal{P}$ that does not satisfy its specification, *i.e.*, the Hoare triple, $\langle\varphi\rangle\mathcal{P}\langle\psi\rangle$ is *false*. We also prove that our algorithm is sound and complete with respect to our repair model. Before proceeding, we establish some groundwork in order to motivate the algorithm.

A. Groundwork

We first characterize our notion of correctness in the following lemma, which is stated without proof. The first part of the lemma is standard [23], and the second part of the lemma follows from our definition of strongest post-conditions (Appendix A).

Lemma 1: Characterization of total correctness:
For any statement s in a given program $\mathcal{P}$,

$$\langle\varphi\rangle s\langle\psi\rangle \equiv \varphi \Rightarrow wp(s,\psi),$$

$$\langle\varphi\rangle s\langle\psi\rangle \equiv \begin{cases} sp(s,\varphi) \Rightarrow \psi, & \text{when } sp(s,\varphi) \text{ is defined,} \\ false, & \text{otherwise.} \end{cases}$$

Thus, when $sp(s, \varphi)$ is undefined, the Hoare triple $\langle\varphi\rangle s\langle\psi\rangle$ is *false* for all possible ψ.

As shown in Section III, we annotate each program statement s_i with a propagated pre-condition φ_{i-1} and a propagated post-condition ψ_i. This provides us with n *local* Hoare triples corresponding to the n program statements. In the following lemma, we present an interesting relation between the local Hoare triples and the Hoare triple for the entire program. This lemma is the basis for our repair algorithm. We claim that $\langle\varphi\rangle\mathcal{P}\langle\psi\rangle$ is *false* if and only if all the local Hoare triples are *false*. Further, all the local Hoare triples are *false* if and only if any one local Hoare triple is *false*.

Lemma 2: For a Boolean program $\mathcal{P}$, composed of a sequence of n statements, $s_1; s_2; \ldots; s_n$, the following expressions are equivalent when all strongest post-conditions $\psi_1, \psi_2, \ldots, \psi_n$ are defined:

$\langle\varphi\rangle\mathcal{P}\langle\psi\rangle$,
$\langle\varphi_{i-1}\rangle s_i\langle\psi_i\rangle$, for any given i in $\{1, 2, \ldots, n\}$,
$\langle\varphi_{i-1}\rangle s_i\langle\psi_i\rangle$, for all i in $\{1, 2, \ldots, n\}$.

Proof: We first prove an equivalence between the second and third expressions. Let $(s_{i-1}; s_i; s_{i+1})$ be a sequence of any three consecutive statements of program $\mathcal{P}$. Consider the Hoare triple, $\langle\varphi_{i-1}\rangle s_i\langle\psi_i\rangle$. We have:

$$\begin{aligned} \langle\varphi_{i-1}\rangle s_i\langle\psi_i\rangle &\equiv \varphi_{i-1} \Rightarrow wp(s_i,\psi_i) && \text{(Lemma 1)} \\ &\equiv sp(s_{i-1},\varphi_{i-2}) \Rightarrow \psi_{i-1} && \text{(Definition)} \\ &\equiv \langle\varphi_{i-2}\rangle s_{i-1}\langle\psi_{i-1}\rangle && \text{(Lemma 1).} \end{aligned}$$

We can show $\langle\varphi_{i-1}\rangle s_i\langle\psi_i\rangle \equiv \langle\varphi_i\rangle s_{i+1}\langle\psi_{i+1}\rangle$ using a similar argument. Extending this result to program statements preceding s_{i-1} and succeeding s_{i+1}, we arrive at an equivalence between the second and third expressions.

We prove an equivalence between the first and second expressions by noting that: $\psi_0 = wp(s_1,\psi_1) = wp((s_1; s_2),\psi_2) = \ldots = wp((s_1; s_2; \ldots; s_n),\psi) = wp(\mathcal{P},\psi)$. Then, we have:

$$\begin{aligned} \langle\varphi\rangle\mathcal{P}\langle\psi\rangle &\equiv \varphi \Rightarrow wp(\mathcal{P},\psi) && \text{(Lemma 1)} \\ &\equiv \varphi_0 \Rightarrow wp(s_1,\psi_1) \\ &\equiv \langle\varphi_0\rangle s_1\langle\psi_1\rangle && \text{(Lemma 1).} \end{aligned}$$

A similar equivalence can be obtained between $\langle\varphi\rangle\mathcal{P}\langle\psi\rangle$, and $\langle\varphi_{i-1}\rangle s_i\langle\psi_i\rangle$, for any i.

■

B. Repair Algorithm

A consequence of Lemma 2 is that if the i^{th} Hoare triple is made *true* by modifying s_i, then the program so obtained satisfies the specification. Hence, our repair strategy proceeds by examining program statements in some specific order to identify potential candidates for repair. If propagation of post-conditions is aborted due to an empty weakest pre-condition for statement s_i, we check for repairable statements among the statements $(s_i; s_{i+1}; \ldots; s_n)$. Similarly, if propagation of pre-conditions is aborted due to an undefined or empty strongest post-condition for statement s_i, we check for repairable statements among the statements $(s_0; s_1; \ldots; s_i)$.

Before checking if a statement can be repaired by modifying it, we do a simple check to see if the statement can be repaired by deleting it. We can do this by computing the QBF:

$\forall_{\mathcal{V}(init)}\ \varphi_{i-1} \Rightarrow \psi_i(curr)$.

Here, $\psi_i(curr)$ is the post-condition ψ_i with all variables in $\mathcal{V}(next)$ swapped with their corresponding variables in $\mathcal{V}(curr)$. Thus, this QBF checks if the local Hoare triple given by $\langle\varphi_{i-1}\rangle s_i\langle\psi_i\rangle$ can be made *true* if s_i is deleted. If this QBF is *true*, we return a new program $\widehat{\mathcal{P}}$ with s_i deleted as the repaired program. If this QBF is *false*, we proceed with the algorithm outlined in the rest of this section, and assume that deletion of s_i is not an option for repair.

For every statement s_i to be inspected, we pose a query to check if s_i can be repaired. If the query returns a `yes`, we proceed to synthesize a repair and declare success. If not, we proceed to the next statement in the given order. If none of the statements can be repaired, we report failure in repairing the program under current constraints. We now explain how to formulate `Query` and `Repair` for each type of program statement.

1) Assignment Statement: Suppose s_i is an assignment statement that we wish to repair. Let $\hat{s}_{i,j}$ denote a potential repair for s_i that assigns an expression $expr$ to the variable v_j, *i.e.*, $\hat{s}_{i,j}$ is v_j := $expr$. The Query function determines if there exists such an expression. To be able to do this, we use a variable z to denote $expr$, and pose the following question: does there exist some variable v_j such that for all initial values of the program variables, there exists an assignment z to v_j, which makes the local Hoare triple, $\langle\varphi_{i-1}\rangle v_j := z\langle\psi_i\rangle$, $true$. Formally, Query returns yes if, for any j, the following QBF is $true$, and no otherwise:

$\forall_{\mathcal{V}(init)}\exists z\ \varphi_{i-1} \Rightarrow \widehat{\psi}_{i-1,j}$.

Each $\widehat{\psi}_{i-1,j}$ is the weakest pre-condition corresponding to the proposed repair $\hat{s}_{i,j}$: v$_j$:= z, and is an expression over z, $\mathcal{V}(init)$ and $\mathcal{V}(curr)$.

Suppose the m^{th} QBF evaluates to $true$. The key challenge is in obtaining an expression for z in terms of variables in $\mathcal{V}(curr)$. We may use the QBF's certificate of validity to derive such a z. In this paper, we provide a more direct solution to obtain z. Let us define $T = \varphi_{i-1} \Rightarrow \hat{\psi}_{i-1,m}$. We denote the positive cofactor of T w.r.t. z as $T|_z$[2]. By properties of Boolean functions, we can show that $\exists_z\ T$ is equivalent to $T|_{[z\rightarrow T|_z]}$, where $T|_{[z\rightarrow T|_z]}$ represents the expression obtained by replacing all occurences of z in T by $T|_z$. Thus, $z = T|_z$ is a witness to the validity of the QBF and could yield $expr$.

However, $T|_z$ is an expression containing variables in $\mathcal{V}(init)$ and $\mathcal{V}(curr)$. Hence the repair algorithm tries to express variables in $\mathcal{V}(init)$ in terms of variables in $\mathcal{V}(curr)$. If this is feasible, $expr$ is obtained solely in terms of variables from $\mathcal{V}(curr)$. If this cannot be done, the repair algorithm suggests adding at most t new constants, $\{v_1(0), v_2(0), \ldots, v_t(0)\}$, to store the initial values of the program variables. Note that this is the only time the repair algorithm suggests addition of constants or insertion of statements. In the following lemma, we prove that the repair generated as above is sound.

Lemma 3: Replacing the statement s_i by the statement $\hat{s}_i$: v_m := $expr$, where v_m is identified from the Query function, and $expr$ is obtained as above, makes the Hoare triple $\langle\varphi_{i-1}\rangle\hat{s}_i\langle\psi_i\rangle$ $true$.

Proof: Recall that T is $\varphi_{i-1} \Rightarrow \widehat{\psi}_{i-1,m}$. Furthermore, by Lemma 1, we know that T represents correctness of the local Hoare triple $\langle\varphi_{i-1}\rangle\hat{s}_i\langle\psi_i\rangle$. Since $z = T|_z$ is a witness to the validity of T, $z = T|_z$ is a witness to the validity of the local Hoare triple. In other words, our approach for obtaining $expr$ by selecting an appropriate z ensures that the local Hoare triple is made $true$. ■

2) Conditional Statement: Let s_i be a conditional statement that we wish to repair. Query for s_i checks for the possibility of repairing either the guard, or a statement in S_{if}, or a statement in S_{else}.

Let $\widehat{G}(curr)$ represent the repair for the guard. We check for its existence by formulating a similar QBF as above and checking for its validity. If the QBF is valid, we can derive $\widehat{G}(curr)$ from the cofactor of T in a similar manner. If the QBF is not valid, then we make two separate sets of queries to check if any statement in S_{if} or in S_{else} can be repaired to yield $\widehat{S}_{if}$ or $\widehat{S}_{else}$, respectively.

Lemma 4: Replacing the conditional statement s_i by the statement $\hat{s}_i$: if ($\widehat{G}(curr)$) S_{if} else S_{else} or the statement $\hat{s}_i$: if ($G(curr)$) $\widehat{S}_{if}$ else S_{else} or the statement $\hat{s}_i$: if ($G(curr)$) S_{if} else $\widehat{S}_{else}$, where $\widehat{G}(curr)$, $\widehat{S}_{if}$ and $\widehat{S}_{else}$ are obtained as above, makes the Hoare triple $\langle\varphi_{i-1}\rangle\hat{s}_i\langle\psi_i\rangle$ $true$.

Proof: The proof is similar to the one for Lemma 3. ■

3) Loop Statement: Let s_i be a loop statement that we wish to repair. As in the repair for a conditional statement, Query for a loop statement checks for the possibility of repairing either the guard or a statement in S_{body}. Query and Repair for loop statements have an additional responsibility to ensure termination of the repaired loop, and differ from the versions we have seen so far. We explain these in detail to highlight these differences.

To check if the loop guard can be repaired, Query checks if there exist states which may enter the loop, execute it iteratively and never satisfy the post-condition ψ_i on completion of a loop execution. Such states are non-terminating, irrespective of the loop guard, and hence, their presence implies the absence of a repair for the loop guard. This set of states can be computed using a terminating fixpoint over the monotonically decreasing sets, $\bigwedge_k Y_k$, where,
$Y_0 = \varphi_{i-1} \wedge \neg\psi_i$, $Y_k = sp(S_{body}, Y_{k-1}) \wedge \neg\psi_i$.

If this fixpoint is non-empty, Query returns a no and the algorithm declares that the loop guard cannot be repaired. If this fixpoint is empty, Query returns yes, and the Repair function computes another fixpoint to yield the desired repair for the loop guard, $\widehat{G}(curr)$. The construction of this fixpoint ensures that any state which exits the loop satisfies ψ_i. The iterants of this fixpoint are the same as above, and the actual fixpoint representing $\widehat{G}(curr)$ is computed using the monotonically increasing sets, $\bigvee_k Y_k$. The rationale for our repair choice for the loop guard is provided by the following lemma.

Lemma 5: Given that the statement s_i: while ($G(curr)$) S_{body} cannot be repaired by deleting it, it can be repaired by modifying its loop guard if and only if the fixpoint $\bigwedge_k Y_k$, where the Y_k's are computed as above, evaluates to $false$, in which case, the fixpoint $\bigvee_k Y_k$, yields a valid repair for s_i.

Proof Sketch: See Appendix B.

If the loop guard cannot be repaired, we formulate a set of queries to check if any statement in S_{body} can be repaired. For simplicity of explanation, we show how to formulate a query for checking if an assignment statement in S_{body} can be repaired. The method extends inductively to inner conditional and loop statements. Let the p^{th} statement in S_{body} be an assignment statement, $s_{i,p}$. As before, let $\hat{s}_{i,p,j}$ denote a potential repair for $s_{i,p}$ that assigns an expression $expr$ to the variable v_j, *i.e.*, $\hat{s}_{i,p,j}$: v_j := $expr$. Using a variable z to

[2]This is standard terminology for the expression obtained by substituting all occurences of z in the Boolean expression T by $true$.

denote $expr$, `Query` computes the weakest pre-condition of the loop corresponding to this repair, $\widehat{\psi}_{i-1,p,j}$, and returns a `yes` if the following QBF is $true$, and `no` otherwise:

$\forall_{\mathcal{V}(init)} \exists_z \varphi_{i-1} \Rightarrow \widehat{\psi}_{i-1,p,j}$.

To compute the weakest pre-condition, we compute a fixpoint as outlined in Section III-A, noting that each iterant of the fixpoint, and hence, the final fixpoint would be an expression over z, $\mathcal{V}(init)$, and $\mathcal{V}(curr)$. If the QBF is valid, we can derive z from the positive cofactor of T or, $\varphi_{i-1} \Rightarrow \widehat{\psi}_{i-1,p,j}$ as before. Since the weakest pre-condition computation guarantees termination, the derived repair z ensures that the loop terminates. If the QBF is not valid, we attempt to repair another statement within the loop body.

Lemma 6: Replacing the loop statement s_i by the statement $\hat{s}_i$: `while` $(\widehat{G}(curr))$ S_{body} or the statement $\hat{s}_i$:`while` $(G(curr))$ $\widehat{S}_{body}$, where $\widehat{G}(curr)$ and $\widehat{S}_{body}$ are computed as above, makes the Hoare triple $\langle \varphi_{i-1} \rangle \hat{s}_i \langle \psi_i \rangle$ $true$.

Proof: The proof for soundness of $\hat{s}_i$: `while` $(\widehat{G}(curr))$ S_{body} follows from Lemma 5. The proof for soundness of $\hat{s}_i$: `while` $(G(curr))$ $\widehat{S}_{body}$ is similar to that of Lemma 3. ■

We now prove that our algorithm is sound and complete with respect to our repair model.

Theorem 1: Given a Hoare triple $\langle \varphi \rangle \mathcal{P} \langle \psi \rangle$ that is $false$ due to the erroneous Boolean program $\mathcal{P}$, if there exists another Boolean program $\widehat{\mathcal{P}}$ satisfying the conditions enumerated below, our algorithm finds one such $\widehat{\mathcal{P}}$. If there exists a unique $\widehat{\mathcal{P}}$, then the algorithm finds it. If our algorithm finds a $\widehat{\mathcal{P}}$, then $\langle \varphi \rangle \widehat{\mathcal{P}} \langle \psi \rangle$ evaluates to $true$.

1) $\widehat{\mathcal{P}}$ differs from $\mathcal{P}$ in exactly *one* statement, *i.e.*, there exists exactly one i such that the statement s_i is absent in $\widehat{\mathcal{P}}$, or is replaced by another statement $\hat{s}_i$ in $\widehat{\mathcal{P}}$,
2) If s_i is an assignment statement, $\hat{s}_i$ is an assignment statement which may differ both in the left and right hand sides of the assignment operator,
3) If s_i is a conditional statement, $\hat{s}_i$ is a conditional statement that differs from s_i in either the guard G or in a statement in S_{if} or S_{else} and
4) If s_i is a loop statement, $\hat{s}_i$ is a loop statement that differs from s_i in either the guard G or in S_{body}.

Proof: The soundness result follows from Lemmas 1, 2, 3, 4 and 6. The completeness result is a direct consequence of our query formulation, which checks for the existence of exactly those repairs that satisfy the above repair model. ■

Note that, by completeness, we refer to the completeness of our algorithm in the domain of Boolean programs. When such programs are used within the CEGAR framework, the algorithm can no longer be complete due to the nature of abstraction.

Example 1: In the annotated program from Figure 2, the QBF for the 2^{nd} statement evaluates to $true$, and generates the expected repair: `y(2) := x(1)` $\oplus$ `y(1);` for the program.

Example 2: Program (b) in Figure 1 does not satisfy its specification. In fact, it does not terminate for some input values. Since the program consists of one loop statement, it does not require any annotation. Let us replace the expression on the right-hand side of the 2^{nd} assignment statement within the loop with z, and invoke the `Query` operation. The weakest pre-condition of the loop can be computed as the expression: $\psi_{0,2,y} = (x \wedge y) \vee (\neg x \wedge \neg y) \vee (z \wedge x \wedge \neg y) \vee (z \wedge \neg x \wedge y)$, and the QBF given by $\forall_{x,y} \exists_z (true \rightarrow \psi_{0,2,y})$ evaluates to $true$. We then compute the cofactor of $\psi_{0,2,y}$ to obtain $z = true$, which is a valid repair.

C. Complexity Analysis and Implementation

All fixpoint computations in our algorithm can be done in time exponential in the number of program variables in the worst-case. Moreover, the `Query` function for assignment and conditional statements involves checking validity of a QBF with exactly two alternating quantifiers and lies in the second polynomial hierarchy, *i.e.*, it is $\Sigma_2 P-$complete in the number of program variables. All other operations like swapping of variables, subsititution, Boolean manipulation and cofactor computation can be done in either constant time or time exponential in the number of program variables in the worst-case.

Thus, the worst-case complexity of our algorithm is exponential in the number of program variables. In practice, most of these computations can be done very efficiently using BDDs. The initial results obtained from a preliminary implementation of our algorithm that uses a Java-based BDD library [24] have been promising. Use of BDDs allows symbolic storage and efficient manipulation of pre-conditions and post-conditions as well as efficient computation of fixpoints. The `forall` and `exist` methods for BDDs facilitate computing validity of a desired QBF and hence enable the `Query` check. Similarly, the `Repair` operation can be performed easily by computing the cofactor of a BDD.

V. Annotation and Repair in presence of Functions

A. Annotation

Our algorithm can be extended to programs containing non-recursive function calls, and function calls with a restricted form of recursion (*e.g.* tail recursion). The basic idea is to use *function summaries* to represent the action of function calls. Suppose the formal parameters of a function f are denoted by $arg_0, \ldots, arg_z$. A *forward summary* of the function f, denoted by $S_{\{f,+\}}$ can be obtained by strongest post-condition propagation through f, by assuming an initial pre-condition of the form $\bigwedge_y (arg_y \equiv x_y)$. Here the x_y's are symbolic values representing the actual parameters of f at its call-site. Similarly, a *backward summary* of f, denoted by $S_{\{f,-\}}$ can be obtained by weakest pre-condition propagation through f, assuming a final post-condition that represents the return value (ret).

$S_{\{f,+\}}$ is used at each call-site of f during pre-condition propagation through the function containing the call-site. To propagate φ_i across a call to f, the x_y's in $S_{\{f,+\}}$ are instantiated with the values of the actual parameters at the call-site, and the result is combined with additional conjunct φ_i

(appropriately renaming the $curr$ and $next$ copies at the call-site). $S_{\{f,-\}}$ is used at each call-site of f during post-condition propagation through the function containing the call-site. To propagate ψ_i across a call-site of the form $v_m := f(\ldots)$, we replace ret in $S_{\{f,-\}}$ by v_m, and conjoin the result with conjunct ψ_i (appropriately renaming the $curr$ and $next$ copies at the call-site).

Since we preclude recursive calls, the call-graph for our programs is acyclic, and hence can be sorted in reverse topological order. We compute function summaries in this order to ensure that the summaries are always available at the call-site of each function.

B. Repair

To repair a statement within the function, we proceed as in the case of assignment, loop and conditional statements, by making an expression within that statement unknown (by replacing it with z). However, a function could be called multiple times within a program. A repair to a statement within the function could impact the overall pre- and post-conditions. Hence, once a function is considered for repair, the program needs to be re-annotated, before we can solve for z.

VI. Conclusions and Future Work

In this paper, we present an efficient and automatic technique for the repair of a large subset of Boolean programs with respect to a total correctness specification denoted by the Hoare triple $\langle\varphi\rangle\mathcal{P}\langle\psi\rangle$. Our technique first annotates each program statement with propagated pre- and post-conditions. It then queries each statement to check if it can be deleted or modified locally to validate its corresponding local Hoare triple. Upon a successful query, the algorithm generates a suitable repair. We prove that our algorithm is sound and complete with respect to the repair model used. Our approach obviates the need for error localization. Moreover, our algorithm avoids performing an exhaustive search for possible repairs, and has tractable complexity.

We remark that it may be possible to extend our approach to repair multiple statements by using multiple variables $z_1, z_2, \ldots, z_h$ to denote desired repairs for suspect expressions, and formulating a QBF over all these variables; the repair can be extracted from the certificate of validity of the QBF. Further, we note that Boolean programs can model both sequential and combinational circuits, and hence, our techniques can be used for repairing such circuits.

As an extension to this work, it would be interesting to explore richer formalisms like pushdown systems to deal with Boolean programs with arbitrary recursive functions. Also, it would be useful to extend our current approach to programs with bounded integers. With appropriate usage of SMT solvers, we may be able to extend the current techniques to such programs. Reasoning over such programs may help us avoid the abstraction-refinement loop encountered during Boolean program repair, and also pave the way for adapting our technique for microcode repair.

References

[1] P. C. Attie, A. Arora, and E. A. Emerson, "Synthesis of Fault-tolerant Concurrent Programs," *ACM Trans. Program. Lang. Syst.*, vol. 26, no. 1, pp. 125–185, 2004.

[2] T. Ball and S. K. Rajamani, "Boolean Programs: A Model and Process for Software Analysis."

[3] ——, "Automatically Validating Temporal Safety Properties of Interfaces," in *Proceedings of the 8th International Workshop on Model Checking of Software (SPIN)*. Springer-Verlag, 2001, pp. 103–122.

[4] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Software Verification with BLAST," in *Proceedings of the 10th International Workshop on Model Checking of Software (SPIN)*, 2003, pp. 235–239.

[5] K. M. Chandy and J. Misra, *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[6] A. Griesmayer, R. Bloem, and B. Cook, "Repair of Boolean Programs with an Application to C," in *18th Conference on Computer Aided Verification (CAV)*, T. Ball and R. B. Jones, Eds., 2006, pp. 358–371.

[7] B. Jobstmann, A. Griesmayer, and R. Bloem, "Program Repair as a Game," in *17th Conference on Computer Aided Verification (CAV)*, K. Etessami and S. K. Rajamani, Eds. Springer-Verlag, 2005, pp. 226–238.

[8] S. Staber, B. Jobstmann, and R. Bloem, "Finding and Fixing Faults," in *13th Conference on Correct Hardware Design and Verification Methods (CHARME)*, D. Borrione and W. Paul, Eds. Springer-Verlag, 2005, pp. 35–49.

[9] L. A. Dennis, "Program Slicing and Middle-Out Reasoning for Error Location and Repair," in *Disproving: Non-Theorems, Non-Validity and Non-Provability*, 2006.

[10] L. A. Dennis, R. Monroy, and P. Nogueira, "Proof-directed Debugging and Repair," in *Seventh Symposium on Trends in Functional Programming*, H. Nilsson and M. van Eekelen, Eds., 2006, pp. 131–140.

[11] F. Buccafurri, T. Eiter, G. Gottlob, and N. Leone, "Enhancing Model Checking in Verification by AI Techniques," *Artif. Intell.*, vol. 112, no. 1-2, pp. 57–104, 1999.

[12] Y. Zhang and Y. Ding, "CTL Model Update for System Modifications," *J. of Artif. Intell. Res.*, vol. 31, pp. 113–155, 2008.

[13] P. C. Attie and J. Saklawi, "Model and Program Repair via SAT Solving," Tech. Rep., 2007.

[14] A. Ebnenasir, S. S. Kulkarni, and B. Bonakdarpour, "Revising UNITY Programs: Possibilities and Limitations," in *International Conference on Principles of Distributed Systems (OPODIS)*, 2005, pp. 275–290.

[15] A. Solar-Lezama, R. Rabbah, R. Bodik, and K. Ebcioglu, "Programming by Sketching for Bit-streaming Programs," in *ACM SIGPLAN conference on Programming language design and implementation (PLDI)*. ACM, 2005, pp. 281–294.

[16] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat, "Combinatorial Sketching for Finite Programs," in *12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2006, pp. 404–415.

[17] S. Staber and R. Bloem, "Fault Localization and Correction with QBF," in *Tenth International Conference on Theory and Applications of Satisfiability Testing (SAT)*, 2007.

[18] A. Groce, S. Chaki, D. Kroening, and O. Strichman, "Error Explanation with Distance Metrics," *Int. J. Softw. Tools Technol. Transf.*, vol. 8, no. 3, pp. 229–247, 2006.

[19] T. Ball, M. Naik, and S. K. Rajamani, "From Symptom to Cause: Localizing Errors in Counterexample Traces," in *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*. ACM, 2003, pp. 97–105.

[20] H. Jin, K. Ravi, and F. Somenzi, "Fate and Free Will in Error Traces," *Int. J. Softw. Tools Technol. Transf.*, vol. 6, no. 2, pp. 102–116, 2004.

[21] E. W. Dijkstra, *A Discipline of Programming*. Prentice Hall, 1976.

[22] M. Huth and M. Ryan, *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge Univeristy Press, 2004.

[23] E. W. Dijkstra and C. S. Scholten, *Predicate Calculus and Program Semantics*. Springer-Verlag New York, Inc., 1990.

[24] J. Whaley, "JavaBDD: An Efficient BDD Library for Java," http://javabdd.sourceforge.net/.

APPENDIX A
STRONGEST POST-CONDITION COMPUTATION

Pre-condition propagation involves recording the value of each program variable in terms of the initial values of the variables, conjoined with the conditions imposed by the program's control-flow. As will be evident below, the propagated pre-condition at any point in the program is a conjunction of α, which is some condition imposed by the program control-flow, and, $\bigwedge_m\ (v_m(next) \equiv \mathcal{F}_m(init))$, where $\mathcal{F}_m(init)$ is a an expression over $\mathcal{V}(init)$ and represents v_m in terms of the initial values of the program variables. For an expression E over $\mathcal{V}(curr)$, we use a special notation $E_{v(curr)\rightarrow\mathcal{F}(init)}$ to denote the expression obtained by substituting each variable $v_m(curr)$ in E by its corresponding expression $\mathcal{F}_m(init)$. Thus, $E_{v(curr)\rightarrow\mathcal{F}(init)}$ is an expression over $\mathcal{V}(init)$.

For the purpose of explaining pre-condition propagation, we assume that we are given a a statement s_i and a pre-condition $\varphi_{i-1} = \alpha\ \wedge \bigwedge_m (v_m(curr) \equiv \mathcal{F}_m(init))$, which is an expression over $\mathcal{V}(init)$ and $\mathcal{V}(curr)$. The strongest post-condition $\varphi_i = sp(s_i, \varphi_{i-1})$ can also be expressed in the same form: $\alpha\ \wedge \bigwedge_m (v_m(next) \equiv \mathcal{F}_m(init))$ and is obtained by the following rules for inductive computation:

1) Assignment statement: $v_j(next)$ `:=` $E(curr)$*:* The strongest post-condition of an assignment statement, $sp(x{:=}E, \delta)$, is given by $\exists_y(\delta[x \rightarrow y] \wedge x \equiv E[x \rightarrow y])$, [23]. In this setup, y represents the "previous" value of x, prior to the assignment, with y being unknown. This necessitates existential quantification over y to obtain the post-condition. In our program syntax, the use of indices enables us to preserve history information, and thus skip existential quantification. Further, since the single assignment statement, $v_j(next)$ `:=` $E(curr)$ leaves all but the j^{th} variable unchanged, we can express $sp(v_j(i)$ `:=` $E,\ \varphi_{i-1})$ as:

$$\varphi_i = \ \varphi_{i-1} \wedge \bigwedge\ (v_j(next) \equiv E)\ \wedge \bigwedge_{m\neq j} (v_m(next) \equiv v_m(curr)).$$

Equivalently, we replace φ_{i-1} by including α and substituting all variables $v_m(curr)$ with their corresponding $\mathcal{F}_m(init)$ expressions to obtain an expression for φ_i in terms of $\mathcal{V}(init)$ and $\mathcal{V}(next)$ as follows:

$$\varphi_i = \ \alpha \wedge \bigwedge (v_j(next) \equiv E_{v(curr)\rightarrow\mathcal{F}(init)})\ \wedge \bigwedge_{m\neq j} (v_m(next) \equiv \mathcal{F}_m(init)).$$

If $E(curr)$ contains an $nd(0,1)$ expression, the strongest post-condition is computed as the *disjunction* of the strongest post-conditions over the statements, $v_j(next) := E(curr)|_0$ and $v_j(next) := E(curr)|_1$.

2) Sequential composition: The rule for propagation of the pre-condition φ_{i-1} through a sequence of statements $(s_i; s_{i+1})$ is similar to the rule for propagation of a post-condition. As before, in order to maintain the correct indices, all variables, $v_m(next)$ in the expression for $sp(s_i, \varphi_{i-1})$ are swapped with $v_m(curr)$ before propagating through s_{i+1}. The rule can be expressed as:
$sp((s_i; s_{i+1}), \varphi_{i-1}) = sp(s_{i+1}; sp(s_i, \varphi_{i-1}))$.

3) Conditional statement: `if` $(G(curr))\ S_{if}$ `else` S_{else}*:* The strongest post-condition of a conditional statement is computed as the disjunction of the strongest post-conditions over S_{if} and S_{else}. Thus we have:

$$\varphi_i = sp(\varphi_{i-1} \wedge G_{v(curr)\rightarrow\mathcal{F}(init)}, S_{if})\ \vee\ sp(\varphi_{i-1} \wedge \neg G_{v(curr)\rightarrow\mathcal{F}(init)}, S_{else}).$$

Replacing φ_{i-1} and invoking our induction hypothesis about the form of a computed strongest post-condition, we get the following expression:

$$\varphi_i = (\alpha \wedge G_{v(curr)\rightarrow\mathcal{F}(init)} \wedge \bigwedge_m (v_m(next) \equiv \mathcal{F}_{m,if}(init)))\ \vee\ (\alpha \wedge \neg G_{v(curr)\rightarrow\mathcal{F}(init)} \wedge \bigwedge_m (v_m(next) \equiv \mathcal{F}_{m,else}(init))).$$

Note that we use different notations, $\mathcal{F}_{m,if}(init)$ and $\mathcal{F}_{m,else}(init)$, to denote the expressions over $\mathcal{V}(init)$ for an arbitrary variable v_m, arising from propagation through two different sequences of statements, S_{if} and S_{else}. Further simplifying, we have:

$$\varphi_i = \ \alpha \wedge \bigwedge_m (\ v_m(next) \equiv (\ (G_{v(curr)\rightarrow\mathcal{F}(init)} \wedge \mathcal{F}_{m,if}(init))\ \vee\ (\neg G_{v(curr)\rightarrow\mathcal{F}(init)} \wedge \mathcal{F}_{m,else}(init))\ )\ ).$$

If $G(curr) = nd(0,1)$, the strongest post-condition is computed as the *disjunction* of the strongest post-conditions over S_{if} and S_{else}, *i.e.*,

$$\varphi_i = \ \alpha \wedge \bigwedge_m (\ v_m(next) \equiv (\mathcal{F}_{m,if}(init) \vee \mathcal{F}_{m,else}(init))\ ).$$

4) Loop statement: `while` $(G)\ S_{body}$*:* As in post-condition propagation, pre-condition propagation of loop statements involves fixpoint computation over the sp operator. The propagated pre-condition of the k^{th} loop iteration is denoted as $sp(S^k_{body}, \varphi_{i-1})$. It is the set of output states, for which execution of the loop body terminates in *at most* k iterations, when begun in some input state satisfying φ_{i-1}. It is computed as follows:

$$sp(S^0_{body}, \varphi_{i-1}) = Y_0 \wedge \neg G,$$
$$sp(S^k_{body}, \varphi_{i-1}) = \bigvee_{l=0}^{k} (Y_l \wedge \neg G),$$

where, $Y_0 = \varphi_{i-1}, Y_k = sp(S_{body}, Y_{k-1} \wedge G)$.

The strongest post-condition of the $(k-1)^{th}$ loop iteration is computed in terms of variables in $\mathcal{V}(next)$ (and $\mathcal{V}(init)$), which are then swapped with their respective variables in $\mathcal{V}(curr)$, and used for computing the strongest post-condition of the k^{th} iteration. As before, let us suppose the fixpoint computation terminates after L iterations. The strongest post-condition for the loop statement, which corresponds to the smallest set of output states such that the loop, when begun in all states satisfying φ_{i-1} is guaranteed to terminate in one of them is thus given by:

$$\varphi_i = \ \neg G\ \wedge\ (\varphi_{i-1} \vee \bigvee_{l=1}^{L} sp(S_{body}, Y_{l-1} \wedge G)).$$

If each variable v_m has the value $\mathcal{F}_{m,k}(init)$ at the end of the l^{th} iteration, then the above expression can be rewritten as: $\varphi_i = \alpha' \wedge \bigwedge_m \Big(v_m(next) \equiv (\bigvee_{l=1}^{L} F_{m,l}(init))\Big)$.

If $G(curr) = nd(0,1)$, the strongest post-condition is expressed as the fixpoint: $\varphi_i = \bigvee_{l=0}^{L'} Z_l$, where $Z_0 = \varphi_{i-1}, Z_k = sp(S_{body}, Z_{k-1})$. Since the sets $\bigvee_l Z_l$ are monotonically increasing, the above fixpoint is guaranteed to terminate in, say, L' iterations.

The above computation for the strongest post-condition of a loop is accompanied by a check for termination for *all* input states satisfying φ_{i-1}. In the event that there exists even one input state which does not terminate for all choices made by the loop guard, the above computation for the strongest post-condition is invalid as the strongest post-condition is undefined.

The termination check can be done in multiple ways. One approach is to propagate the prospective strongest post-condition computed above, back through the loop, and check if the weakest pre-condition so obtained contains φ_{i-1}. If this check is $false$, it implies the existence of states in φ_{i-1} that do not terminate. This approach is based on a characterization of total correctness that is presented in Lemma 1 in Section IV.

Appendix B
Proof Sketch of Lemma 5

Since the loop statement cannot be repaired by deleting it, some states satisfying the pre-condition for the loop φ_{i-1} must enter the loop. Hence, any repair $\widehat{G}(curr)$ for the loop guard must contain some states satisfying φ_{i-1}. Consider the fixpoint over the sets $\bigwedge_k Y_k$, where

$$Y_0 = \varphi_{i-1} \wedge \neg\psi_i,$$
$$Y_k = sp(S_{body}, Y_{k-1}) \wedge \neg\psi_i.$$

This fixpoint represents the set of all states that satisfy the pre-condition φ_{i-1}, but not the post-condition ψ_i, enter the loop, execute it iteratively and never satisfy the post-condition ψ_i on completion of a loop execution. Suppose this fixpoint does not evaluate to $false$, *i.e.*, it contains some states. The loop guard that can repair the loop while including some states from φ_{i-1}, cannot exclude states from this set as such states do not satisfy ψ_i on exiting the loop. On the other hand, a loop guard cannot include states from this set as such states will never transition into states that can exit the loop suitably, satisfying ψ_i. Thus, the states from this set cannot belong to either $\widehat{G}(curr)$ or $\neg\widehat{G}(curr)$ leading to a contradiction.

Hence, if we cannot choose a loop-guard that is disjoint from φ_{i-1}, a choice for a loop guard that can help repair the loop statement exists if and only if the above fixpoint is empty.

Assuming that the above fixpoint is empty, consider the fixpoint over the sets $\bigwedge_k \widehat{G}_k$, where

$$\widehat{G}_0 = \varphi_{i-1} \wedge \neg\psi_i,$$
$$\widehat{G}_k = sp(S_{body}, \widehat{G}_{k-1}) \wedge \neg\psi_i.$$

By construction, this fixpoint accumulates all states that satisfy the pre-condition for any loop iteration, but not the post-condition ψ_i before entering the loop. If we select the loop guard to be this fixpoint, then any state that exits the loop is forced to satisfy ψ_i by construction. Thus each iterant of this fixpoint accumulates states that are chosen by the guard to execute the loop during that iteration. Moreoever, since the first fixpoint in this proof is empty, there are no states which do not terminate.

Note that the above fixpoint is non-empty as the first iterant $\varphi_{i-1} \wedge \neg\psi_i$ is required to be non-empty. If the first iterant were empty, no states from φ_{i-1} would enter the loop, thereby making the loop unnecessary and contradicting the assumption in the lemma.