# Boolean Programs: A Model and Process For Software Analysis

Thomas Ball  Sriram K. Rajamani
{tball,sriram}@microsoft.com

February 29, 2000 (updated March 28, 2000)

Technical Report
MSR-TR-2000-14

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

# Boolean Programs: A Model and Process For Software Analysis

Thomas Ball  Sriram K. Rajamani
{tball,sriram}@microsoft.com

Software Productivity Tools
Microsoft Research

**Abstract.** A fundamental issue in model checking of software is the choice of a model for software. We present a model called *boolean programs* that is expressive enough to represent features in common programming languages and is amenable to model checking. We present a model checking algorithm for boolean programs using context-free-language reachability. The model checking algorithm allows procedure calls with unbounded recursion, exploits locality of variable scopes, and gives short error traces. Furthermore, we give a process for incrementally refining an initial skeletal boolean program $B$ (representing a source program $P$) with respect to a particular reachability query in $P$. The presence of infeasible paths in $P$ may lead to the model checker reporting false positive errors in $B$. We show how to refine $B$ by introducing boolean variables to rule out the infeasible paths. The process uses ideas from model checking, symbolic execution, and program slicing.

## 1 Introduction

What would model checking of software look like if it were a well established practice today? If it had the attributes that model checking of hardware circuits does, it might look like this:

- There would be a representation **R** for modeling software, analogous to the finite state machines (FSMs) used for modeling hardware circuits, and efficient algorithms to model check **R**.
- The model checking algorithms over **R** would report a shortest trace to an error when they find errors, as model checkers for FSMs do.
- Programming languages such as C, C++ and Java would have translations into **R**, just as hardware description languages such as VHDL and Verilog can be compiled into FSMs.
- An instance $r$ in **R** could be refined into an instance $r'$ in **R** and proved correct (either by construction, or by verification), just as FSMs can be refined and proved correct using the semantics of trace inclusion.
- The model checking algorithms on **R** would be able to exploit the inherent modularity and abstraction boundaries present in the source programs for efficiency.

We investigate how to model check temporal properties of sequential programs with an eye on these five desiderata. We consider programs written in a source language containing scalar variables over finite and infinite domains, the usual structured control-flow constructs, global variables, and procedural abstraction with call-by-value parameter passing, local variables, and recursion.

We define a target representation **R** called *boolean programs.* Boolean programs are a subset of the programs defined by the source language, namely those in which all variables and parameters have boolean type. What distinguishes boolean programs from FSMs is that boolean

| | | | |
|---|---|---|---|
| `numUnits: int;` | | `nU0: bool;` | `nU0: bool;` |
| `level: int;` | | | |
| `void getUnit() {` | `void getUnit() {` | `void getUnit() {` | `void getUnit() {` |
| [1] `canEnter: bool := F;` | [1] `...;` | [1] `...;` | [1] `cE: bool :=F;` |
| [2] `if (numUnits = 0) {` | [2] `if (?) {` | [2] `if (nU0) {` | [2] `if (nU0) {` |
| [3] `if (level > 10){` | [3] `if (?){` | [3] `if (?){` | [3] `if (?){` |
| [4] `NewUnit();` | [4] `...;` | [4] `...;` | [4] `...;` |
| [5] `numUnits := 1;` | [5] `...;` | [5] `nU0:=F;` | [5] `nU0:=F;` |
| [6] `canEnter := T;` | [6] `...;` | [6] `...;` | [6] `cE:=T;` |
| `}` | `}` | `}` | `}` |
| `} else` | `} else` | `} else` | `} else` |
| [7] `canEnter := T;` | [7] `...;` | [7] `...;` | [7] `cE:=T;` |
| | | | |
| [8] `if (canEnter)` | [8] `if (?)` | [8] `if (?){` | [8] `if (cE)` |
| [9] `if (numUnits = 0)` | [9] `if (?)` | [9] `if (nU0)` | [9] `if (nU0)` |
| [10] `assert(F);` | [10] `...;` | [10] `...;` | [10] `...;` |
| `else` | `else` | `else` | `else` |
| [11] `gotUnit();` | [11] `...;` | [11] `...;` | [11] `...;` |
| `}` | `}` | `}` | `}` |
| $P$ | $B_1$ | $B_2$ | $B_3$ |

**Fig. 1.** An example program $P$ and three boolean programs ($B_1$, $B_2$ and $B_3$) that abstract $P$ with an increasing level of precision.

programs contain procedures with recursion. As procedural abstraction and recursion are key components of all modern programming languages, we believe that boolean programs provide a good starting point for investigating model checking of software.

Our goal is to check whether a sequential program $P$ obeys a temporal property $\Phi$. It is well-known that this problem can be reduced to the problem of invariant checking (assertion violation) in an instrumented version of $P$ (a version of $P$ containing extra code that simulates the FSM corresponding to the complement of the property $\Phi$). Therefore, we will focus on the problem of checking whether or not a statement is reachable in $P$ (which might correspond to the reject state of the FSM). One of our motivating problems is to show that device drivers for operating systems obey certain temporal properties. Device drivers form the interface between an operating system and the devices that the operating system administers and controls. Most device drivers are created by companies other than operating systems vendors. Because drivers run in kernel space, a misbehaving driver can wreak havoc. Device driver programmers generally use the C language and must obey many rules about the order in which the driver can access operating system functions and resources. Although a device driver operates in a multi-threaded environment, the majority of properties it must obey concern its sequential behavior.

To make our goal concrete, we consider the program $P$ in Figure 1. In this program, "T" denotes the boolean value **true** and "F" denotes **false**. We are interested in knowing if the statement "`assert(F)`" at line 10 is reachable, as this would indicate that a "Unit" was acquired improperly. We will start with a boolean program that coarsely abstracts $P$ and incrementally refine it, driven by the goal of answering our reachability query:

- We first generate the "skeletal" boolean program $B_1$ from $P$. Program $B_1$ retains the control-flow structure of $P$. However, every variable declaration of $P$ has been removed, every assignment statement has been replaced by **skip** (denoted by "..." for readability), and every boolean expression has been replaced by "?" (non-deterministic boolean choice). We say that that $B_1$ abstracts $P$ as the set of feasible (executable) paths in $B_1$ is a superset of the set of feasible paths in $P$.
- We now ask the question: is line 10 reachable in $B_1$? A model checker over $B_1$ might be used to answer this query. The answer is "yes" and one such path leading to line 10 is $[1,2,7-10]$. However, this path is infeasible in $P$ because it constrains the variable $numUnits$ to be both equal to 0 (by the control transition from line 9 to 10) and not equal to 0 (line 2 to line 7).
- We want to refine program $B_1$ to eliminate this infeasible path, while ensuring that all feasible paths in $P$ are feasible in the new boolean program. What program state should we choose to model in the boolean program? The condition $(numUnits = 0)$ is an obvious choice. We create a boolean variable $nU0$ to model this condition and a new boolean program $B_2$. Program $B_2$ is constructed by examining the source statements of $P$, determining which statements affect the condition $(numUnits = 0)$ and updating the boolean variable $nU0$ so that it will conservatively track the condition $(numUnits = 0)$. That is, if $nU0$ is true at line $l$ in path $p$ in $B_2$ then $(numUnits = 0)$ is true at line $l$ in path $p$ in $P$. Informally, $B_2$ can be thought of as an abstract interpretation of program $P$ with respect to the condition $(numUnits = 0)$.
- In program $B_2$, the path $[1, 2, 7 - 10]$ is infeasible, since the boolean variable $nU0$ is **false** due to the control transition from line 2 to line 7, which rules out the transition from 9 to 10. Notice that $B_2$ eliminates another infeasible path from $P$, namely $[1 - 6, 8 - 10]$.
- We again ask the question: is line 10 reachable in $B_2$? The answer is still "yes" due to the path $[1 - 3, 8 - 10]$. This path is infeasible in $P$, as the variable $canEnter$ is initially set to **false** and not updated subsequently, while the control transition from line 8 to 9 is taken, indicating that $canEnter$ is **true**, a contradiction. This suggests that we need to add a boolean variable to model the $canEnter$ condition, namely $cE$.
- The boolean program $B_3$ is the result of transforming the source program to correctly update both variables $nU0$ and $cE$. Line 10 is not reachable in $B_3$, so it is not reachable in $P$. Note that $B_3$ contains no mention of the variable $level$, as its value does not impact the reachability of line 10.

In the above example, we have sketched a process for refining a program $P$ into a boolean program $B$ based on the goal of invariant detection ("line 10 is not reachable"). The process starts with a skeletal boolean program $B$ with no boolean variables. The boolean program is incrementally refined using the following iterative procedure:

1. We know that $B$ abstracts $P$. Is there a path $p$ in $B$ that witnesses a violation of the invariant? If not, then the invariant holds in $P$. This can be determined via model checking of the boolean program $B$.
2. If there is such a path $p$, is $p$ is feasible in $P$? There are three possible outcomes to this question, which generally is answered using an automated decision procedure:
   - Path $p$ is feasible in $P$. In this case, an error in $P$ has been found and the process terminates. Note that there is a one-to-one correspondence between the paths in $P$ and $B$, and every feasible path in $P$ is a feasible path in $B$.

- Path $p$ is infeasible in $P$. In this case, the process continues at step (3).
- The decision procedure used to determine the feasibility/infeasibility returns the answer "don't know". In this case, the process terminates with a "don't know" answer.

3. What are the conditions that imply the infeasibility of path $p$ in $P$?
4. Using these conditions, create boolean variables to model the conditions and transform the program $P$ into a new boolean program $B'$ such that $p$ is infeasible in $B'$, $B$ abstracts $B'$, and $B'$ abstracts $P$.
5. Replace $B$ with $B'$ and go back to 1.

Due to the small size of program $P$ in Figure 1, a model checker applied directly to program $P$ (or an encoding of $P$ in a language such as Promela) would determine that line 10 is not reachable in $P$. Of course, in general, it is undecidable to check if a program can reach some statement. Even if $P$ has only variables over finite domains then model checking over $P$ may be impractical if the number of variables is large. In such situations, it is useful to construct abstractions of $P$, which is what our process does.

The results of this paper, in addition to this refinement process, are solutions to steps 1, 2 and 4. For step 3, we rely on existing techniques for symbolically executing a single path (such as in [CL96] or [DE82]) and determining its feasibility, referred to here as *path simulation*. Our results are the following:

– *Model checking of a boolean program with procedures and recursion.* Using context-free languages, we give a trace-based and *stackless* semantics for our source language. We transform the reachability problem for boolean programs to context-free language(CFL) reachability [RHS95] and obtain a model checking algorithm for boolean programs. The model checking algorithm is cubic in the size of the control flow graph and exponential in the maximum number of *local* variables in the program. Further, it provides short error traces when errors are found.

– *Identification of a small set of conditions that imply the infeasibility of a path.* Strongest post-conditions, suitably modified for our application, yield a set of conditions that the boolean program must model in order to eliminate the infeasible path. The strongest postcondition formalizes what interface a path simulator must present to be of use in our refinement process. We also present an algorithm for reducing the set of conditions that imply the infeasibility of a path, which reduces the number of boolean variables needed in the boolean program. This algorithm is based on a path abstraction called *consistent path projections*, a new form of program slicing.

– *Derivation of a boolean program from the infeasibility conditions.* Given the information from the above step, we show how weakest preconditions (the dual of strongest postconditions) give a simple way to locally transform the statements of the source program $P$ into statements in the boolean program $B$. We present an algorithm to construct a boolean program $B$ from $P$ that abstracts $P$ and in which a particular path $p$ (that is infeasible in $P$) becomes infeasible. In general, several infeasible paths in $P$ could become infeasible in $B$ through this construction, especially if few boolean variables are needed to explain the infeasibility.

Our results should be of interest to the following communities:

```
Program      :=  Declaration*  Procedure*              |
Declaration  ::= Identifier : TypeIdentifer ;          | Comm        +::= assert ( BoolExpr )                |
Procedure    ::= ProcHeader {ProcBody}                 |                 Comm ; LabeledComm                  |
ProcHeader   ::= Identifier( Declaration* )            |                 skip                                |
ProcBody     ::= Declaration*  LabeledComm             |                 Identifier ( Expr* );
                                                       |                    '['Label']'skip /* call */       |
LabeledComm ::= '['Label']' Comm                       |                 return /* return */                 |
Comm         ::= Assignment                          | |
                 if ( BoolExpr ) {LabeledComm}         | Assignment ::=  (Variable)+:=(Expr)+
                     else {LabeledComm}              | |
                 while ( BoolExpr ) {LabeledComm}    | |
```

**Fig. 2.** BNF for the syntax of the **X** language.

- *Model Checking*: Boolean programs could form an intermediate language on which model checkers for software (based on BDDs or other representations) could be built to perform efficient reachability analysis.
- *Program Analysis*: Dataflow analyses typically assume that every control flow path is potentially executable and conservatively approximate at every point in which control flow from separate paths merges. Boolean programs could be a mechanism by which infeasible paths could be ruled out by maintaining correlations between variables, thereby improving the accuracy of dataflow analysis.
- *Path Simulation*: When using a path simulator to uncover program errors, selecting which paths to simulate is crucial to obtain good coverage. In general, path selection is a very hard problem. Our process does away with path selection, and uses path simulators on only one path at a time. One could view the model checker and the refinement process as implicit property-driven path selection.

The rest of this paper gives the background and techniques to implement the Steps 1–4 in the above iterative procedure. Section 2 defines the syntax and semantics of a simple programming language called **X**. Section 3 defines boolean programs and presents a model checking algorithm for boolean programs based on CFL-reachability (Step 1). Section 4 formalizes path simulation using strongest postconditions and shows how to check if a given path $p$ is feasible in $P$ (Step 2). If $p$ is infeasible, we produce a set of boolean expressions $E$ and annotations $A$ that "explain" the infeasibility (Step 3). Using $A$ and $E$, we construct a boolean program $\hat{\mathcal{B}}(P, E, A)$ which abstracts $P$, while maintaining the infeasibility of $p$ (Step 4). Section 5 details how to find a smaller set of conditions $E'$ and annotations $A'$ that explains the infeasibility of $p$, thus reducing the number of variables needed in the boolean program abstraction. Section 6 discusses related work. Section 7 concludes and points to future work.

## 2 The programming language X

We give a simple programming language called **X** that defines the source programs as well as the boolean programs. Figure 2 presents the concrete syntax for **X**. We use labels to give a trace semantics for **X** programs. *TypeIdentifer* ranges over integers and finite enumerations. We lift the traditional boolean type to a three-valued type: $\{\mathbf{true}, \mathbf{false}, ?\}$, where ? denotes the truth

| $\wedge$ | **true** | **false** | ? |
|---|---|---|---|
| **true** | **true** | **false** | ? |
| **false** | **false** | **false** | **false** |
| ? | ? | **false** | ? |

| $\vee$ | **true** | **false** | ? |
|---|---|---|---|
| **true** | **true** | **true** | **true** |
| **false** | **true** | **false** | ? |
| ? | **true** | ? | ? |

| ! | |
|---|---|
| **true** | **false** |
| **false** | **true** |
| ? | ? |

**Fig. 3.** Kleene's three-valued interpretation of $\wedge$, $\vee$ and ! (negation).

value "unknown". *Expr* and *BoolExpr* are side-effect free, contain the usual operators and may refer to constants of the appropriate types. Following [SRW99], we use Kleene's three-valued logic to interpret boolean expressions (see Figure 3). This three-valued logic is useful when defining boolean programs. A procedure call is required to be followed by a **skip** statement so that the return point of the call is explicit and labeled. The parallel assignment operator is useful in abbreviating a sequence of assignments such as $tmp{:=}x; x{:=}y; y{:=}tmp$ to $x, y{:=}y, x$. We use the parallel assignment operator in the construction of $\mathcal{B}(P, E)$ in Section 4.1.

The term *statement* denotes the portion of a command that is identified by a label. Intuitively, statements are entities that execute when single stepping an **X** program in a source-level debugger. Let $P$ be an **X** program. $L(P)$ denotes the set of labels in $P$ and $V(P)$ denotes the set of variables in $P$. If $P$ has **assert** statements, $L(P)$ contains a special label **err** (for error label). Let $l \in L(P)$ be a label. $S_P(l)$ denotes the statement at label $l$ in $P$. Labels and variable names are assumed to be globally unique.

**Variables and Scope.** Let $Globals(P)$ be the set of global variables of $P$. Let $Formals_P(l)$ be the set of formal parameters of the procedure whose body contains label $l$. Let $Locals_P(l)$ be the set of local variables and formal parameters at label $l$. For all $l \in L(P)$, $Formals_P(l) \subseteq Locals_P(l)$. Let $InScope_P(l)$ denote the set of all variables of $P$ whose scope includes $l$. For all $l \in L(P)$, $InScope_P(l) = Locals_P(l) \cup Globals(P)$. For a set $V \subseteq V(P)$, a *valuation* $\Omega$ to $V$ is a function that associates every variable in $V$ with a value of the appropriate type. $\Omega$ can be extended to expressions over $V$ in the usual way. For any function $f : D \to R$, $d \in D$, $r \in R$, $f[d/r] : D \to R$ is defined as $f[d/r](d') = r$ if $d = d'$, and $f(d')$ otherwise. For example, if $x, y$ are integer variables, $V = \{x, y\}$, and $\Omega = \{(x, 1), (y, 2)\}$, then $\Omega(x + y) = 3$, and $\Omega[x/5] = \{(x, 5), (y, 2)\}$.

**Successors.** Let $Succ_P : L(P) \to 2^{L(P)}$ be a function that maps each label to its control-flow successors. The value of $Succ_p(l)$ depends on the statement $S_P(l)$. If $S_P(l)$ is an assignment or **skip**, then $Succ_p(l)$ is a singleton set containing the label of the next sequential statement. If $S_P(l)$ is an **if**, **while** or **assert** statement, then $Succ_p(l) = \{Tsucc_P(l), Fsucc_P(l)\}$, where $Tsucc_P(l)$ denotes the successor of $l$ when the boolean expression evaluates to **true**, and $Fsucc_P(l)$ denote the successor of $l$ when the boolean expression evaluates to **false**. If $S_P(l)$ is an **assert** statement, $Fsucc_P(l) = \textbf{err}$. If the boolean expression evaluates to ?, it is treated as if the expression nondeterministically evaluated to **true** or **false**, independent of the state of $P$. Thus, **if**, **while** and **assert** statements in which the boolean expression evaluates to ? can behave nondeterministically.

For any procedure **pr** in $P$, let $First_P(\textbf{pr})$ be the label of the first statement in procedure **pr**. A procedure call at label $l$ to procedure **pr** has $First_P(\textbf{pr})$ as its unique successor. If $S_P(l)$ is a procedure call then $RetPt_P(l)$ is the label of the **skip** statement immediately following the

| | |
|---|---|
| $S \to MS$ | $\forall\langle\mathbf{call}, l, \Delta\rangle, \langle\mathbf{ret}, l, \Delta\rangle \in \Sigma(P)$ : |
| $\forall\langle\mathbf{call}, l, \Delta\rangle \in \Sigma(P)$ : | $M \to \langle\mathbf{call}, l, \Delta\rangle\ M\ \langle\mathbf{ret}, l, \Delta\rangle$ |
| $S \to \langle\mathbf{call}, l, \Delta\rangle\ S$ | $M \to MM$ |
| $S \to \epsilon$ | $M \to \epsilon$ |
| | $M \to \sigma$ |

**Table 1.** Production rules $\mathit{Rules}(P)$ for grammar $\mathcal{G}(P)$.

procedure call. If $S_P(l)$ is a return statement in procedure **pr**, then $\mathit{Succ}_P(l) = \{n \mid m, n \in L(P)$ and $S_P(m)$ is a call to **pr** and $n = \mathit{RetPt}_P(m)\}$.

**States and Transitions.** A *state* $\eta$ of $P$ is a pair $\langle l, \Omega\rangle$, where $l \in L(P)$ and $\Omega$ is a valuation to the variables in $\mathit{InScope}_P(l)$. $\mathit{States}(P)$ is the set of all states of P. Intuitively, a state contains the program counter and values to all the variables visible at that point. Note that our definition of state is different from the conventional notion of a program state, which includes the call stack (with activations records for all active procedures). The projection operator $\Gamma$ maps a state to its label: $\Gamma(\langle l, \Omega\rangle) = l$. We can extend $\Gamma$ to operate on sequences of states in the usual way.

We define a set $\Sigma(P)$ of terminals:

$$\Sigma(P) = \{\sigma\} \ \cup\ \{\ \langle\mathbf{call}, l, \Delta\rangle, \langle\mathbf{ret}, l, \Delta\rangle \mid \exists l_1 \in L(P), S_P(l_1) \text{ is a procedure call, and } l = \mathit{RetPt}(l_1), \text{ and } \Delta \text{ is a valuation to } \mathit{Locals}_P(l_1)\}$$

$\Sigma(P)$ is infinite if there are integer variables in $P$ and finite if there are no integer variables in $P$. In particular, $\Sigma(P)$ is finite if all the variables in $P$ are boolean variables. Intuitively, terminals are either $\sigma$, which is a place holder, or triples that are introduced whenever there is a procedure call in $P$. The first component of the triple is either **call** or **ret**, corresponding to the actions of a call to and return from that procedure, the second is the label of the return point of the call, and the third component keeps track of values of local variables of the calling procedure at the time of the call. We will use these terminals in a context-free grammar $\mathcal{G}(P)$ that specifies the legal sequences of calls and returns that a program $P$ may make. A context-free grammar $\mathcal{G}$ is a 4-tuple $\langle N, T, R, S\rangle$, where $N$ is a set of nonterminals, $T$ is a set of terminals, $R$ is a set of production rules and $S \in N$ is a start symbol. For each program $P$, we define a grammar $\mathcal{G}(P) = \langle\{S, M\}, \Sigma(P), \mathit{Rules}(P), S\rangle$, where $\mathit{Rules}(P)$ is defined in Table 1.

If we view terminals $\langle\mathbf{call}, l, \Delta\rangle$ and $\langle\mathbf{ret}, l, \Delta\rangle$ as matching left and right parentheses, the language $\mathcal{L}(\mathcal{G}(P))$ is the set of all strings over $\Sigma(P)$ that are sequences of partially-balanced parentheses. That is, every right parenthesis $\langle\mathbf{ret}, l, \Delta\rangle$ is balanced by a preceding $\langle\mathbf{call}, l, \Delta\rangle$ but the converse need not hold. The $\Delta$ component insures that the values of local variables at the time of a return are the same as they were at the time of the corresponding call (this must be the case because the **X** language has a call-by-value semantics). The nonterminal $M$ generates all sequences of balanced calls and returns, and $S$ generates all sequences of partially balanced calls and returns. This allows us to reason about non-terminating or abortive executions. Note again that the number of productions is infinite if program $P$ contains integer variables, but finite if $P$ contains only boolean variables.

We use $\eta_1 \xrightarrow{\alpha}_P \eta_2$, to denote that $P$ can make an $\alpha$-labeled transition from state $\eta_1$ to state $\eta_2$. Formally, $\eta_1 \xrightarrow{\alpha}_P \eta_2$ holds if $\eta_1 = \langle l_1, \Omega_1\rangle \in \mathit{States}(P)$, $\eta_2 = \langle l_2, \Omega_2\rangle \in \mathit{States}(P)$, $\alpha \in \Sigma(P)$,

| $S_P(l_1)$ | $\alpha$ | $l_2$ | $\Omega_2$ |
|---|---|---|---|
| **skip** | $\alpha = \sigma$ | $l_2 = \mathit{Succ}_P(l_1)$ | $\Omega_2 = \Omega_1$ |
| $x_1, x_2, \ldots, x_k :=$ $e_1, e_2, \ldots, e_k$ | $\alpha = \sigma$ | $l_2 = \mathit{Succ}_P(l_1)$ | $\Omega_2 = \Omega_1[x_1/\Omega_1(e_1))] \cdots [x_k/\Omega_1(e_k)]$ |
| **if**$(e)$ **while**$(e)$ **assert**$(e)$ | $\alpha = \sigma$ | $l_2 = \mathit{Tsucc}(l_1)$if $\Omega_1(e) = \mathbf{true}$ $l_2 = \mathit{Fsucc}(l_1)$if $\Omega_1(e) = \mathbf{false}$ $l_2 \in \mathit{Succ}(l_1)$if $\Omega_1(e) = ?$ | $\Omega_2 = \Omega_1$ if $\Omega_1(e) \neq ?$ $\Omega_2 \in \mathcal{S}(\{\Omega_1\}, e, l_2 = \mathit{Tsucc}(l_1))$ if $\Omega_1(e) = ?$ |
| **pr**$(e_1, e_2, \ldots, e_k)$ | $\alpha = \langle \mathbf{call}, \mathit{RetPt}_P(l_1), \Delta \rangle$, $\Delta(x) = \Omega_1(x), \quad x \in \mathit{Locals}_P(l_1)$ | $l_2 = \mathit{First}_P(\mathbf{pr})$ | $\Omega_2(x_i) = \Omega_1(e_i), \quad x_i \in \mathit{Formals}_P(l_2)$ $\Omega_2(g) = \Omega_1(g), \quad g \in \mathit{Globals}(P)$ |
| **return** | $\alpha = \langle \mathbf{ret}, l_2, \Delta \rangle$ | $l_2 \in \mathit{Succ}_P(l_1)$ | $\Omega_2(g) = \Omega_1(g), \quad g \in \mathit{Globals}(P)$ $\Omega_2(x) = \Delta(x), \quad x \in \mathit{Locals}_P(l_2)$ |

**Table 2.** Conditions on the state transitions $\langle l_1, \Omega_1 \rangle \xrightarrow{\alpha}_P \langle l_2, \Omega_2 \rangle$, for each statement construct in **X**. See Figure 4 for the definition of the function $\mathcal{S}$.

$$\begin{array}{lcl}
\mathcal{S}(O, x, b \in \{\mathbf{true}, \mathbf{false}\}) & = & \{\Omega[x/b] \mid \Omega \in O, \Omega(x) = ?\} \\
\mathcal{S}(O, !e, b \in \{\mathbf{true}, \mathbf{false}\}) & = & \mathcal{S}(O, e, !b) \\
\mathcal{S}(O, e_1 \wedge e_2, \mathbf{true}) & = & \mathcal{S}(\mathcal{S}(O, e_1, \mathbf{true}), e_2, \mathbf{true}) \\
\mathcal{S}(O, e_1 \wedge e_2, \mathbf{false}) & = & \mathcal{S}(O, e_1, \mathbf{false}) \cup \mathcal{S}(O, e_2, \mathbf{false}) \\
\mathcal{S}(O, e_1 \vee e_2, \mathbf{true}) & = & \mathcal{S}(O, e_1, \mathbf{true}) \cup \mathcal{S}(O, e_2, \mathbf{true}) \\
\mathcal{S}(O, e_1 \vee e_2, \mathbf{false}) & = & \mathcal{S}(\mathcal{S}(O, e_1, \mathbf{false}), e_2, \mathbf{false})
\end{array}$$

**Fig. 4.** Definition of the function $\mathcal{S}$.

where the conditions on $\eta_1$, $\eta_2$ and $\alpha$ for each statement construct are shown in Table 2. The transitions for **skip**, assignment statements and procedure call/return are straightforward.

The transition for a conditional expression $e$ (in **if** (e), **while** (e) or **assert** (e)) is complicated by the use of three-valued logic. Consider the evaluation of a boolean expression $e$ in a conditional statement in state $\Omega_1$. If $\Omega_1(e) = \mathbf{true}$ or $\Omega_1(e) = \mathbf{false}$ then the control flow successor is uniquely determined and $\Omega_2 = \Omega_1$. However, if $\Omega_1(e) = ?$ then the control flow successor $l_2$ is nondeterministically chosen from $\mathit{Succ}(l_1)$. Furthermore, the state $\Omega_2$ must be consistent with this choice. That is $\Omega_2$ must be drawn from a set of states in which $e$ evaluates to **true** (**false**) if $l_2 = \mathit{Tsucc}(l_1)$ ($l_2 = \mathit{Fsucc}(l_1)$). This set of states is defined by the function $\mathcal{S}$, defined in Figure 4. To illustrate this function, consider the boolean expression $e = x \wedge y$ in a state $\Omega_1$ where $\Omega_1(x) = \Omega_1(y) = ?$. Suppose that $l_2 = \mathit{Fsucc}(l_1)$ is chosen nondeterministically from $\mathit{Succ}(l_1)$. Then $\Omega_2$ must be drawn from the set:

$$\mathcal{S}(\{\Omega_1\}, x \wedge y, \mathbf{false}) = \{\Omega_1[x/\mathbf{false}], \Omega_1[y/\mathbf{false}]\}$$

That is, there are two ways in which $x \wedge y$ could evaluate to false given that $\Omega_1(x) = \Omega_1(y) = ?$. Note that we do not need to include the state $\Omega_1[x/\mathbf{false}][y/\mathbf{false}]$, as both the states $\Omega_2 = \Omega_1[x/\mathbf{false}]$ and $\Omega_3 = \Omega_1[y/\mathbf{false}]$ are more general than this state (the former having $\Omega_2(y) = ?$ and the latter having $\Omega_3(x) = ?$).

We assume that there is a distinguished procedure named **main**, where $P$ starts executing. A state $\eta = \langle l, \Omega \rangle$ is *initial* if $l = \mathit{First}_P(\mathbf{main})$ (all variables can take on arbitrary initial values). A finite sequence $\overline{\eta} = \eta_0 \xrightarrow{\alpha_0}_P \eta_1 \xrightarrow{\alpha_1}_P \cdots \eta_{n_1} \xrightarrow{\alpha_{n-1}}_P \eta_n$ is a *trajectory* of $P$ if (1) $\eta_0$ is an initial state of $P$, (2) for all $0 \leq i < n$, $\eta_i \xrightarrow{\alpha_i}_P \eta_{i+1}$, and (3) $\alpha_0 \alpha_1 \ldots \alpha_{n-1} \in \mathcal{L}(\mathcal{G}(P))$. If $\overline{\eta}$ is a trajectory, then its projection to labels $\Gamma(\eta_0), \Gamma(\eta_1), \ldots, \Gamma(\eta_n)$ is called a *trace* of $P$. The semantics of an **X** program is its set of traces. A state $\eta$ of $P$ is *reachable* if there exists a trajectory of $P$ that ends in $\eta$. A label $l \in L(P)$ is *reachable* if there exists a trace of $P$ that ends in $l$.

## 3 Model Checking of Boolean Programs

A *boolean program* is an **X** program in which all variables and procedure parameters have boolean type. An instance $\langle B, l\rangle$ of the *boolean program reachability problem* consists of a boolean program $B$, and $l \in L(B)$. The answer to the problem is "yes" if $l$ is reachable in $B$ and "no" if $l$ is not reachable in $B$. We give a decision procedure to answer the boolean-program-reachability problem that has the characteristics of model checking. In particular, if $l$ is reachable, the procedure yields a trajectory that proves the reachability of $l$.

**Context-free-language reachability.** Let $T$ be a set of terminals. A labeled graph over $T$ is a tuple $G = \langle N, A\rangle$ where $N$ is a set of nodes, and $A \subseteq N \times T \times N$ is a set of labeled arcs.

An instance $\langle G, \mathcal{G}, N_1, N_2\rangle$ of the *context-free-language (CFL) reachability* problem is a graph $G = \langle N, A\rangle$ over a set of terminals $T$, a context-free grammar $\mathcal{G} = \langle N', T, R, S\rangle$, and a pair of sets $N_1, N_2$, where $N_1 \subseteq N$, and $N_2 \subseteq N$. The answer to the problem is "yes" if there exist $n_1 \in N_1$, $n_2 \in N_2$, and $s \in \mathcal{L}(\mathcal{G})$, such that there is a path from $n_1$ to $n_2$ in $G$ with the string of labels $s$, and "no" otherwise. It is known that CFL-reachability can be solved in $O((|N| \times |\mathcal{G}|)^3)$.

**Boolean program reachability as CFL-reachability.** We give a reduction of boolean program reachability to CFL-reachability. Let $\langle B, l\rangle$ be an instance of the boolean program reachability problem. Construct a labeled graph, the *exploded graph*, $G_B = \langle N_B, A_B\rangle$ with labels over $\Sigma(B)$, where $N_B = \mathit{States}(B)$, and $A_B = \{\langle s_1, \alpha, s_2\rangle | s_1 \xrightarrow{\alpha}_B s_2\}$. Let $S_1$ be the set of initial states of $B$, and $S_2$ be the set of states of $B$ with label $l$.

The boolean program reachability problem $\langle B, l\rangle$ can be reduced to the CFL-reachability problem $\langle G_B, \mathcal{G}(B), S_1, S_2\rangle$. In fact, we can use the CFL-reachability algorithm of Reps et al. [RHS95], which uses a form of memoization at procedure calls and works in a directed manner from the **main** procedure. If $l$ is reachable in $B$, the CFL-reachability procedure or Reps et al.'s algorithm can be used to give a trajectory that ends in a state labeled $l$.

**Example.** Figure 5 presents a program with four procedures (*main*, $A$, $B$, and $C$) and a part of its exploded graph. In this program, there is one global variable $g$ and procedures $A$, $B$ and $C$ each have one formal parameter. In the exploded graph, we have only considered the values **false** or **true** (and not ?) for the boolean variables. Thus, there are two possible valuations for $g$ in procedure *main* at each of the four labels in *main* and four possible valuations for $g$ and the formal parameter in each label of the procedures $A$, $B$, and $C$. The trajectory in Figure 5 shows that the label [12] is reachable (some transitions in the exploded graph are not shown in the figure).
Initially, the global variable $g$ is **false**. The call to procedure $A$ results in a state in which both $g$ and the formal parameter $a$ have the value **false**. Note that the transitions associated with procedure calls and returns are labeled with terminals. For example, the transition from [5] to [11] is labeled with $\langle \textbf{call}, [6], \{a = \textbf{false}\}\rangle$, recording that the

```
     g: bool := false;
     main() {
[1]    A(g);
[2]    skip;
[3]    B(!g);
[4]    skip;
     }

     A(a: bool) {
[5]    C(a);
[6]    skip;
[7]    return;
     }

     B(b: bool) {
[8]    C(!b);
[9]    skip;
[10]   return;
     }

     C(c: bool) {
[11]   if (c) {
[12]     assert(false);
       } else {
[13]     g := !g;
       }
[14]   return;
     }
```

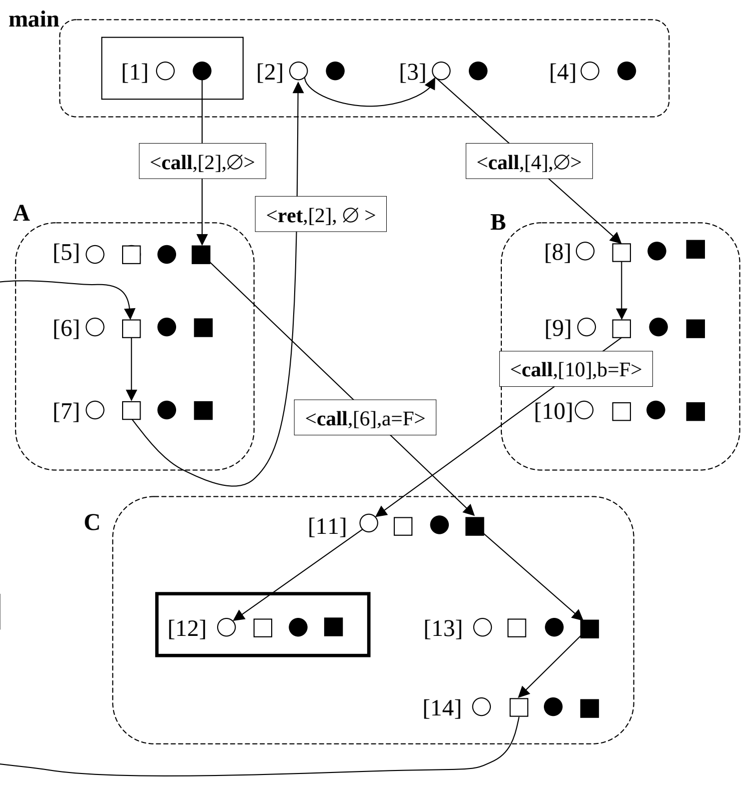

**Fig. 5.** A program with four procedures and a trajectory in its exploded graph that shows that label [12] is reachable. White nodes denote states in which $g = \mathbf{true}$ while black nodes denote states in which $g = \mathbf{false}$. In procedures with a formal parameter ($A$, $B$, and $C$), a square node denotes states in which the parameter has value **false**, while a round node denotes states in which the parameter has value **true**.

return from procedure C must go to [6], and the value of variable $a$ must be "restored" to **false**. The transition from [14] to [6] is labeled with $\langle \mathbf{ret}, [6], \{a = \mathbf{false}\}\rangle$. The pair of terminals $\langle \mathbf{call}, [6], \{a = \mathbf{false}\}\rangle$ and $\langle \mathbf{ret}, [6], \{a = \mathbf{false}\}\rangle$ are matched by the grammar for this program. Of particular interest is the transition from [13] to [14], in which the value of $g$ changes from **false** to **true**. As a result, the value of the formal parameter $b$ in $B$ is **false** when $B$ is called at line [3] and the value of parameter $c$ in procedure $C$ is **true** when $C$ is called from $B$ at line [8]. □

Let $m_1 \geq |InScope(l)|$ for all $l \in L(B)$, and $m_2 \geq |Locals(l)|$ for all $l \in L(B)$. Let $C$ be the number of procedure calls in $B$. Then $|N_B|$ is $O(|L(B)| \times 3^{m_1})$, and $|\mathcal{G}(B)|$ is $O(C \times 3^{m_2})$. Thus the complexity of our algorithm for checking if $l$ is reachable in $B$ is $O((|L(B)| \times C)^3 \times 3^{m_1+m_2})$. If $B$ has a constant number of global variables then the complexity is asymptotically exponential in the maximum number of local variables (over all procedures) in $B$. Thus, the model checking exploits the inherent modularity from procedural abstraction.

## 4 Constructing Boolean Program Abstractions

It is undecidable to check if a program $P$ in language $\mathbf{X}$ can reach a label $l \in L(P)$. Even if $P$ only has variables over finite domains then model checking over $P$ may be impractical if $P$ is large (although the previous section shows that the problem is decidable if a program's variables have finite domains). In such situations, it is useful to construct abstractions of $P$. Let $P$ and $Q$ be $\mathbf{X}$ programs such that $L(P) = L(Q)$. We say that $P$ *refines* $Q$, or $Q$ *abstracts* $P$, written $P \preceq Q$, if every trace of $P$ is a trace of $Q$.

The most naive abstraction of $P$ is defined by the *control skeleton* of $P$, written $\mathit{Skel}(P)$. $\mathit{Skel}(P)$ is a boolean program that is constructed by:

- deleting all variable declarations from $P$;
- replacing all assignments and asserts in $P$ with **skip** statements;
- replacing all boolean expressions in **if** and **while** statements of $P$ with ?;
- retaining all procedure calls in $P$ (after deleting the actual parameter list) and deleting all formal parameter lists.

By construction, $P \preceq \mathit{Skel}(P)$.

Let $P$ be an $\mathbf{X}$ program and $E$ be a finite set of boolean expressions over variables in $P$ and constants in $\mathbf{X}$. Section 4.1 describes the construction of a boolean program $\mathcal{B}(P, E)$, such that $P \preceq \mathcal{B}(P, E)$. Section 4.2 formalizes what a path simulator does in terms of strongest postconditions. Recall that we use a path simulator to provide a set of conditions that "explain" the infeasibility of a path $p$ in $P$. Section 4.3 shows how to construct $E$, and a set of "annotations" $A$ (defined later) so as to make path $p$ infeasible in the boolean program $\hat{\mathcal{B}}(P, E, A)$. Section 4.4 describes how to iteratively refine the boolean program using these results.

To simplify the technical presentation in Sections 4.1– 4.4 we assume that $P$ is in $\mathbf{X}$-normal form (defined below) and is a single procedure program. The extension to multiple procedures is described in Section 4.5. An $\mathbf{X}$ program $P$ is in $\mathbf{X}$-*normal form* if all the following hold:

- every **assert** statement in $P$ is followed by a **skip** statement;
- all boolean expressions in **if** and **while** statements of $P$ are ?;
- all assignment statements in $P$ assign to a single variable.

If $P$ is not in $\mathbf{X}$-normal form, it is easy to transform it to $P'$ in $\mathbf{X}$-normal form without changing its semantics, using the following transformations: the statement "$\mathbf{if}(e)\ \{\ A\ \}\ \mathbf{else}\ \{\ B\ \}$" is transformed to

$$\mathbf{if}(?)\ \{\ \mathbf{assert}(e); \mathbf{skip}; A\ \}\ \mathbf{else}\ \{\ \mathbf{assert}(!e); \mathbf{skip}; B\ \}$$

The statement "$\mathbf{while}(e)\ \{\ S\ \}$" is transformed to

$$\mathbf{while}(?)\ \{\ \mathbf{assert}(e); \mathbf{skip}; S\ \}\ \mathbf{assert}(!e); \mathbf{skip};$$

Parallel assignments are transformed to a sequence of single assignments using temporary variables to ensure that the semantics of parallel assignment is respected. For example $x, y := y, x$ is transformed to $\mathit{tmp}1 := y; \mathit{tmp}2 := x; x := \mathit{tmp}1; y := \mathit{tmp}2;$. In general $P'$ has additional labels and statements. However, for any $l$ in $L(P)$, $l$ is reachable in $P$ iff $l$ is reachable in $P'$.

| Statement(s) in $P$ | Translation in $\mathcal{B}(P,E)$ |
|---|---|
| [i] **if** $('?')$ | [i] **if** $('?')$ |
| [i] **while** $('?')$ | [i] **while** $('?')$ |
| [i] $x := e$ | [i] $b_1, \ldots, b_n := \mathcal{I}(x := e, e_1), \ldots, \mathcal{I}(x := e, e_n)$ |
| [i] **assert**$(e)$<br>[j] **skip** | [i] **assert**$(!\mathcal{F}(!e))$<br>[j] $b_1, \ldots, b_n := \mathcal{I}(\textbf{assert}(e), e_1), \ldots, \mathcal{I}(\textbf{assert}(e), e_n)$ |
| [i] **skip** | [i] **skip** |

**Table 3.** Transforming statements in a source program $P$ into a boolean program $\mathcal{B}(P,E)$.

## 4.1 Constructing $\mathcal{B}$(P,E)

Let $E = \{e_1, e_2, \ldots, e_n\}$ be a set of boolean expressions over variables in $P$ and constants in **X**. Then $\mathcal{B}(P,E)$ has $n$ boolean variables $V_B = \{b_1, b_2, \ldots b_n\}$, and the same control structure as $P$. Intuitively, we wish to maintain that whenever $b_i =$ **true**(**false**) at a given state of a trajectory in $\mathcal{B}(P,E)$, $e_i$ evaluates to **true**(**false**) at the corresponding state of the corresponding trajectory in $P$. However, if $b_i$ evaluates to ? at a given state in $\mathcal{B}(P,E)$ then there is no guarantee of what the value of $e_i$ is at the corresponding state in $P$.

Initially, all the $b_i$ are set to ? in $\mathcal{B}(P,E)$. Each statement $s$ in $P$ (with the exception of asserts) is transformed to a corresponding statement in $\mathcal{B}(P,E)$. An **assert** statement is transformed to two statements in $\mathcal{B}(P,E)$ (which is why we required the **skip** statement after every **assert** in **X**-normal form). Table 3 shows the translation and uses the notation defined below. (Our translation is quite naive, as we conservatively assume that an assignment statement can affect every $e_i$ (and thus every $b_i$). Optimizing the translation remains as future work.)

For a statement $s$ and a boolean expression $f$, let $WP(s,f)$ denote the *weakest precondition* [Gri81] of $f$ with respect to statement $s$. Given an expression $f$, let $f[x \leftarrow e]$ denote the expression obtained by substituting each occurrence of $x$ in $f$ by expression $e$ (with appropriate renaming to avoid name capture). $WP$ for assignments and asserts is defined as follows:

$$\begin{aligned} WP(x := e, f) &= f[x \leftarrow e] \\ WP(\textbf{assert}(e), f) &= e \Rightarrow f \end{aligned}$$

A *minterm* over $V_B$ is a conjunction $c_{i_1} \wedge c_{i_2} \cdots \wedge c_{i_k}$, where each $c_{i_j} \in \{b_{i_j}, !b_{i_j}\}$ for some $b_{i_j} \in V_B$. For any boolean variable $b_i \in V_B$, let $\mathcal{E}(b_i)$ denote the corresponding boolean expression $e_i$ and let $\mathcal{E}(!b_i)$ denote the corresponding boolean expression $!e_i$. Extend $\mathcal{E}$ to minterms and disjunctions of minterms in the usual way. For any boolean expression $e$, let $\mathcal{F}(e)$ denote the largest disjunction of minterms over $V_B$ such that $\mathcal{E}(\mathcal{F}(e))$ implies $e$. The function $\mathcal{E}(\mathcal{F}(e))$ represents the weakest expression over $E$ that implies $e$. The computation of $\mathcal{F}$ is exponential in $n$, in the worst-case. However, by considering minterms over pairs or triples of variables only, we can compute a conservative approximation to $\mathcal{F}$ in $O(n^2)$ or $O(n^3)$. Given two boolean expressions $e$ and $f$, we define the function $\mathcal{H}$:

$$\mathcal{H}(e,f) = \begin{array}{ll} \textbf{true} & \text{if } e \\ \textbf{false} & \text{if } f \\ ? & \text{otherwise} \end{array}$$

Note that for $\mathcal{H}(e,f)$ to be well defined, $e$ and $f$ should never be **true** simultaneously. We ensure that this condition is satisfied whenever we use $\mathcal{H}$. Given a statement $s$ and a boolean expression $e$, let $\mathcal{I}(s,e) = \mathcal{H}(\mathcal{F}(WP(s,e)), \mathcal{F}(WP(s,!e)))$. Intuitively, $\mathcal{I}(s,e)$ denotes the truth value for the boolean variable $b$ corresponding to $e$, after executing statement $s$. We illustrate the translations of the assignment and **assert** statements in the following two examples:

**Examples**. Let $E = \{(x=1),(x=2),(x \leq 3)\}$ and let $V_B = \{b_1, b_2, b_3\}$ be the three boolean variables with $\mathcal{E}(b_1) = (x=1)$, $\mathcal{E}(b_2) = (x=2)$, and $\mathcal{E}(b_3) = (x \leq 3)$. Consider the assignment statement $x := x+1$. The following table shows the weakest preconditions $WP(x := x+1, e)$, and $WP(x := x+1, !e)$ for each $e \in E$. The table also shows the strengthening of the weakest preconditions with respect to the boolean variables $b_1$, $b_2$ and $b_3$.

| | $e = (x=1)$ | $e = (x=2)$ | $e = (x \leq 3)$ |
|---|---|---|---|
| $WP(x := x+1, e)$ | $x = 0$ | $x = 1$ | $x \leq 2$ |
| $\mathcal{F}(WP(x := x+1, e))$ | **false** | $b_1$ | $b_1 \vee b_2$ |
| $\mathcal{E}(\mathcal{F}(WP(x := x+1, e)))$ | **false** | $x = 1$ | $x = 1 \vee x = 2$ |
| $WP(x := x+1, !e)$ | $x \neq 0$ | $x \neq 1$ | $x \geq 3$ |
| $\mathcal{F}(WP(x := x+1, !e))$ | $b_1 \vee b_2 \vee !b_3$ | $!b_1 \vee b_2 \vee !b_3$ | $!b_3$ |
| $\mathcal{E}(\mathcal{F}(WP(x := x+1, !e)))$ | $x = 1 \vee x = 2 \vee x > 3$ | $x \neq 1 \vee x = 2 \vee x > 3$ | $x > 3$ |

We explain the entries in the last column of the table. First, consider $WP(x := x+1, (x \leq 3)) = x \leq 2$. The condition $(x \leq 2)$ is not in $E$. However, we have that $(x=1) \vee (x=2) \Rightarrow (x \leq 2)$. Therefore, if $b_1 \vee b_2$ is **true** before the execution of $x := x+1$ then $b_3$ should be **true** afterwards. Now consider $WP(x := x+1, !(x \leq 3)) = x \geq 3$. The condition $(x \geq 3)$ is not in $E$. However, we have that $!(x \leq 3) \Rightarrow (x \geq 3)$. Therefore, if $b_3$ is **false** before $x := x+1$ then $b_3$ should be **false** after $x := x+1$. Further note that both $b_1 \vee b_2$ and $!b_3$ cannot be **true** at the same time (since this would imply that $x = 1 \wedge x = 2$ and $x > 3$ could be true at the same time). Thus, the effect of the assignment statement $x := x+1$ on variable $b_3$ is given by:

$$b_3 := \mathcal{I}(x := x+1, (x \leq 3)) \quad = \quad \begin{array}{ll} \textbf{true} & \text{if } (b_1 \vee b_2) \\ \textbf{false} & \text{if } !b_3 \\ ? & \text{otherwise} \end{array}$$

The next example illustrates the translation of **assert** statements. Let $E = \{(x < y),(x < z),(y < z)\}$, and let $V_B = \{b_4, b_5, b_6\}$ with $\mathcal{E}(b_4) = (x < y)$, $\mathcal{E}(b_5) = (x < z)$, and $\mathcal{E}(b_6) = (y < z)$. Consider the assert statement **assert**$(y < z)$. It can be checked that $\mathcal{F}(!(y < z)) = !b_6 \vee (b_4 \wedge !b_5)$. If the **assert** statement **assert**$(!\mathcal{F}(!(y < z)))$ fails in $\mathcal{B}(P, E)$, we are guaranteed that **assert**$(y < z)$ fails in $P$. If **assert**$(y < z)$ succeeds in $P$, we must take into account how the truth of $e$ impacts the truth values of the expressions in $E$ and their corresponding variables in $V_B$. This is captured by the second statement in the translation (which replaces the **skip** statement following the **assert**). For the three expressions in $E$ and their corresponding boolean variables we have that

– $\mathcal{F}(y < z \Rightarrow x < y) = b_4$, and $\mathcal{F}(y < z \Rightarrow !(x < y)) = !b_4 \vee !b_5$. Note that if both $b_4$ and $!b_4 \vee !b_5$ are simultaneously **true**, then it must be the case that $b_4 \wedge !b_5$ is **true**, which implies that **assert**$(!\mathcal{F}(!(y < z)))$ should have failed. If **assert**$(!\mathcal{F}(!(y < z)))$ passes, then $b_4$ and $!b_4 \vee !b_5$ cannot be true simultaneously. Thus, the translation for $b_4$ is:

$$b_4 \quad := \quad \mathcal{I}(\textbf{assert}(y < z), (x < y)) \quad = \quad \begin{array}{ll} \textbf{true} & \text{if } b_4 \\ \textbf{false} & \text{if } !b_4 \vee !b_5 \\ ? & \text{otherwise} \end{array}$$

– $\mathcal{F}(y < z \Rightarrow x < z) = b_5 \vee b_4$ and $\mathcal{F}(y < z \Rightarrow !(x < z)) = !b_5$ In this case, the truth of $(y < z)$ implies the truth of $(x < z)$ if $(x < z)$ or $x < y$ already is true, and the truth of $(y < z)$ implies the truth of $!(x < z)$ if $!(x < z)$ is already true. Note that if $b_5 \vee b_4$ and $!b_5$ are simultaneously **true**, then it must be the case that $b_4 \wedge !b_5$ is **true**, which is not possible if $\textbf{assert}(!\mathcal{F}(!(y < z)))$ passes. Thus, the translation for $b_5$ is:

$$b_5 \quad := \quad \mathcal{I}(\textbf{assert}(y < z), (x < z)) \quad = \quad \begin{array}{ll} \textbf{true} & \text{if } b_5 \vee b_4 \\ \textbf{false} & \text{if } !b_5 \\ ? & \text{otherwise} \end{array}$$

Since $\mathcal{F}(y < z \Rightarrow y < z) = \textbf{true}$, the translation for $b_6$ is:

$$b_6 \quad := \quad \mathcal{I}(\textbf{assert}(y < z), (y < z)) \quad = \textbf{true}$$

□

We have the following results:

**Lemma 1.** *Let $P$ be an $\mathbf{X}$ program in $\mathbf{X}$-normal form, and let $E = \{e_1, e_2, \ldots, e_n\}$ be a set of expressions over $V(P)$ and constants in $\mathbf{X}$. Let $B = \mathcal{B}(P, E)$ be the boolean program abstraction with variables $\{b_1, b_2, \ldots, b_n\}$ (where $b_i$ corresponds to $e_i$, for $i \leq i \leq n$). Then, for every trajectory $\overline{\eta} = \langle l_0, \Omega_0 \rangle \rightarrow_P \langle l_1, \Omega_1 \rangle \cdots \rightarrow_P \langle l_{N-1}, \Omega_{N-1} \rangle \rightarrow_P \langle l_N, \Omega_N \rangle$ of $P$, there exists a trajectory $\overline{\eta}' = \langle l_0, \Omega'_0 \rangle \rightarrow_B \langle l_1, \Omega'_1 \rangle \cdots \rightarrow_B \langle l_{N-1}, \Omega'_{N-1} \rangle \rightarrow_B \langle l_N, \Omega'_N \rangle$ of $B$, such that for every boolean variable $b_i$ in $B$, and $0 \leq k \leq N$, we have that*

$$((\Omega'_k(b_i) = \textbf{true}) \Rightarrow (\Omega_k(e_i) = \textbf{true})) \text{ and } ((\Omega'_k(b_i) = \textbf{false}) \Rightarrow)(\Omega_k(e_i) = \textbf{false}))$$

*Proof.* By induction over $k$. $\Omega'_0(b_i)$ is ? for all $b_i$, so the base case $k = 0$ is easy. For the induction, let $s = S_P(l_{k-1})$ ($s$ is the last statement executed on the trajectory thus far). There are four cases to consider:

– $s$ is an assignment statement: Suppose $\Omega'_k(b_i)$ is **true** (a similar proof can be done for the case in which $\Omega'_k(b_i)$ is **false**). Since $\Omega'_k(b_i)$ is **true** some minterm in $\mathcal{F}(WP(s, e_i))$ was **true** in $\Omega'_{k-1}$ (Note that no minterm can be in both $\mathcal{F}(WP(s, e_i))$ and $\mathcal{F}(WP(s, !e_i))$). Let that minterm be $c_{j_1} \wedge c_{j_2} \cdots \wedge c_{j_m}$. Let $f_{j_r} = \mathcal{E}(c_{j_r})$. By induction, for each $0 \leq r \leq m$, $\Omega_{k-1}(f_{j_r}) = \Omega'_k(c_{j_r}) = \textbf{true}$. By the definition of $\mathcal{F}$ we have $f_{j_1} \wedge f_{j_2} \cdots \wedge f_{j_m} \Rightarrow WP(s, e_i)$. Thus, $\Omega_k(e_i)$ is **true** as well.
– $s$ is of the form $\textbf{assert}(e)$: In this case, we need to prove if **assert** in $\mathcal{B}(P, E)$ fails then the **assert** in $P$ fails as well. If $\textbf{assert}(!\mathcal{F}(!e))$ fails, it follows that some minterm in $\mathcal{F}(!e)$ was **true** in $\Omega'_{k-1}$. Let that minterm be $c_{j_1} \wedge c_{j_2} \cdots \wedge c_{j_m}$. Let $f_{j_r} = \mathcal{E}(c_{j_r})$. By induction, for each $0 \leq r \leq m$, $\Omega_{k-1}(f_{j_r}) = \Omega'_{k-1}(c_{j_r}) = \textbf{true}$. By the definition of $\mathcal{F}$, $f_{j_1} \wedge f_{j_2} \cdots \wedge f_{j_m} \Rightarrow !e$. It follows that $\Omega_{k-1}(!e) = \textbf{true}$ and $\textbf{assert}(e)$ in $P$ fails.

– $s$ is a **skip** that immediately follows **assert**$(e)$: Suppose $\Omega'_k(b_i)$ is **true** (a similar proof can be done for the case in which $\Omega'_k(b_i)$ is **false**). Since $\Omega'_k(b_i)$ is **true** some minterm $m$ in $\mathcal{F}(e \Rightarrow e_i)$ was **true** in $\Omega'_{k-1}$. Note that $m$ cannot be in $\mathcal{F}(e \Rightarrow !e_i)$, because that would imply that $\mathcal{E}(m) \Rightarrow !e$, which is not possible since **assert**$(!\mathcal{F}(!e))$ passed in $\mathcal{B}(P,E)$. Let $m$ be $c_{j_1} \wedge c_{j_2} \cdots \wedge c_{j_m}$. By induction , for each $0 \leq r \leq m, \Omega_{k-1}(f_{j_r}) = \mathbf{true}$. Further $\Omega_{k-1}(e)$ is **true**, because the assertion **assert**$(e)$ just passed in $P$. Since $f_{j_1} \wedge f_{j_2} \cdots \wedge f_{j_m} \Rightarrow (e \Rightarrow e_i)$, we have that $\Omega_{k-1}(e_i) = \mathbf{true}$. Since $s$ is a **skip**, we have $\Omega_k = \Omega_{k-1}$, and $\Omega_k(e_i) = \mathbf{true}$.
– $s$ does not fall into any of the above categories: Since $\Omega_k = \Omega_{k-1}$, and $\Omega'_k = \Omega'_{k-1}$, we are done.

□

**Theorem 1.** *Let $P$ be an* **X** *program in* **X***-normal form, and let $E = \{e_1, e_2, \ldots, e_n\}$ be a set of expressions over $V(P)$ and constants. Then $P \preceq \mathcal{B}(P,E)$.*

*Proof.* Follows from Lemma 1. □

## 4.2 Formalizing Path Simulation using Strongest Postconditions

A given error trace $\gamma$ in the boolean program $\mathcal{B}(P,E)$ may not be a trace in $P$. We use the technology of path simulation (symbolic evaluation of a single path through $P$) to find a set of expressions $E$ such that $\gamma$ is not a trace of $\mathcal{B}(P,E)$. This section formalizes our requirements on a path simulator using the foundation of strongest postconditions. The strongest postcondition formulation we present is not the traditional one [Gri81], which we review quickly here, but can be shown to be equivalent to it. The classic Hoare-logic formulation of strongest postconditions for assignment and assert statements are defined as follows.

$$\begin{array}{lcl} SP_H(f, x := e) & = & \exists x'. f[x \leftarrow x'] \wedge (x = e[x \leftarrow x']) \\ SP_H(f, \mathbf{assert}(e)) & = & f \wedge e \end{array}$$

We note that if $f$ and $e$ do not contain any occurrences of the variable $x$ then the first definition can be simplified to: $SP_H(f, x := e) = f \wedge (x = e)$. For a sequence of statements, $p = s_1, s_2, \ldots, s_n$, we have:

$$SP_H(f,p) \quad = \quad SP_H(\ldots SP_H(SP_H(f, s_1), s_2) \ldots, s_n)$$

We formalize paths and their relationship to traces. Let $P$ be an **X** program. A sequence $p$ of statements $s_1, s_2, \ldots, s_n$ is a *path* of $P$ if there is a trace $\gamma = l_0, l_1, \ldots, l_m$ of $Skel(P)$ such that $m > n$, and $p$ is the subsequence of $S_P(\gamma) = S_P(l_0), S_P(l_1), \ldots, S_P(l_m)$ containing only assertions and assignments. (Since $P$ is a single-procedure program in **X**-normal form $S_P(\gamma)$ contains only assertions, assignments, **skip**, **if** statements, and **while** statements. Further, the **if** and **while** statements have '?' as their deciders. The path $p$ is obtained from $S_P(\gamma)$ by eliminating **skip**, **if** and **while** statements). We say that $\gamma$ is a *witnessing trace* to the path $p$. Let $\hat{\Gamma}$ be a function that maps statement indices in $p$ to their corresponding labels in the witnessing trace $\gamma$. A path $p$ is *feasible* if its witnessing trace $\gamma$ is a trace of $P$ ($\gamma$ is by definition a trace of $Skel(P)$). Otherwise, path $p$ is *infeasible* in $P$.

**Uses and definitions.** Given an expression $e$, let $use(e)$ denote the set of variables used in $e$. For every statement $s$, we define two sets of variables $use(s)$ and $def(s)$. The set $use(s)$ denotes the set of variables whose values are used by $s$, and $def(x)$ denotes the set of variables for which new values are defined by $s$. If $s$ is an assert statement **assert**$(e)$, then $use(s) = use(e)$, and $def(s) = \emptyset$. If $s$ is an assignment statement $x_1, x_2, \ldots, x_k := e_1, e_2, \ldots, e_k$, then $use(s) = use(e_1) \cup use(e_2) \cdots \cup use(e_k)$, and $def(s) = \{x_1, x_2, \ldots, x_k\}$. Let $p = s_1, s_2, \ldots, s_n$ be a path of $P$. Let $V(p)$ be the set of variables *referenced* (used or defined) in $p$. Let $\Theta(p)$ be a set of symbolic constants in a one-to-one correspondence with the variables of $V(p)$: $\Theta(p) = \{\theta_{x,p} | x \in V(p)\}$. Let $Exp$ denote the (infinite) set of expressions over $\Theta(p)$ and the constants in $\mathbf{X}$.

**Contexts.** A *context* of the path simulator is a triple $\langle \hat{\Omega}, \Pi, \Phi \rangle$, where $\hat{\Omega}$ is a partial function $V(p) \rightarrow Exp$ called the *store*, $\Pi \subseteq V(p) \times Exp$ is a set called the *history*, and $\Phi$ is a set of boolean expressions from $Exp$ called the *conditions*. $\hat{\Omega}$ represents the current valuation to $V(p)$, $\Pi$ represents the past valuations to $V(p)$, and $\Phi$ represents the constraints introduced by expressions in assert statements. $\hat{\Omega}$ is extended to expressions over $V(p)$ in the usual way. Given a store $\hat{\Omega}$ and a set of variables $X \subseteq V(p)$, let $undef(\hat{\Omega}, X) = \{\langle x, \theta_{x,p} \rangle | x \in X, \hat{\Omega}(x) \text{ not defined}\}$. We use $undef(\hat{\Omega}, X)$ to assign variable $x \in X$ its symbolic constant $\theta_{x,p}$ when $x$ is not defined in $\hat{\Omega}$.

The *strongest postcondition* $SP$ maps a context and a statement to a new context:

$$SP(\langle \hat{\Omega}, \Pi, \Phi \rangle, x := e) \quad = \quad \textbf{let } \bar{\Omega} = \hat{\Omega} \cup undef(\hat{\Omega}, use(e)) \textbf{ in}$$
$$\langle \bar{\Omega}[x/\bar{\Omega}(e)],\ \Pi \cup \{\langle x, \bar{\Omega}(x) \rangle | \bar{\Omega}(x) \text{ defined}\},\ \Phi \rangle$$

$$SP(\langle \hat{\Omega}, \Pi, \Phi \rangle, assert(e)) \quad = \quad \textbf{let } \bar{\Omega} = \hat{\Omega} \cup undef(\hat{\Omega}, use(e)) \textbf{ in}$$
$$\langle \bar{\Omega},\ \Pi,\ \Phi \cup \bar{\Omega}(e) \rangle$$

As before, for a path $p = s_1, s_2, \ldots, s_n$, we have:

$$SP(\langle \hat{\Omega}, \Pi, \Phi \rangle, p) \quad = \quad SP(\ldots SP(SP(\langle \hat{\Omega}, \Pi, \Phi \rangle, s_1), s_2) \ldots, s_n)$$

For a pair $\langle x, e \rangle$ in $V(p) \times Exp$, let $\mathcal{C}(\langle x, e \rangle)$ denote the boolean expression $(x = e)$. We generalize $\mathcal{C}$ to sets of pairs and sets of expressions in the usual way. Given a context $\langle \hat{\Omega}, \Pi, \Phi \rangle$, let $\mathcal{C}(\langle \hat{\Omega}, \Pi, \Phi \rangle) = \bigwedge_{e \in (\mathcal{C}(\hat{\Omega}) \cup \Phi)} e$. Let $\emptyset^3$ abbreviate the empty context $\langle \emptyset, \emptyset, \emptyset \rangle$. There are three interpretations of a context $\langle \hat{\Omega}, \Pi, \Phi \rangle = SP(\emptyset^3, p)$ that are of interest:

- The boolean expression $F = \bigwedge_{c \in \Phi} c$ represents the conditions imposed by the asserts in the path $p$. If $F$ is not satisfiable (i.e., a decision procedure shows that $F$ cannot be satisfied; satisfiability is computationally hard but there are good heuristics for efficiently checking satisfiability) then $p$ is infeasible in $P$. A decision procedure actually may report three possibilities: "satisfiable", "not satisfiable", "don't know".
- The boolean expression $\mathcal{C}(SP(\emptyset^3, p))$ is logically equivalent to $SP_H(true, p)$, where we extend the alphabet of logical discourse to include the symbolic constants of $p$ as well as the variables of $p$. The difference is that the traditional strongest postcondition does not perform forward substitution of symbolic values for variables in the interpretation of expressions.
- As the next section shows, the set of expressions $E = \mathcal{C}(\hat{\Omega}) \cup \mathcal{C}(\Pi) \cup \Phi$, where $\langle \hat{\Omega}, \Pi, \Phi \rangle = SP(\emptyset^3, p)$, is sufficient to make path $p$ infeasible in $\mathcal{B}(P, E)$.

**Example.** Consider the following path $p$ and how each statement transforms the context $\langle \hat{\Omega}, \Pi, \Phi \rangle$:

| $p$ | $\hat{\Omega}$ | $\Pi$ | $\Phi$ |
|---|---|---|---|
| `assert(b>0);` | $\langle b, \theta_{b,p} \rangle$ | | $\theta_{b,p} > 0$ |
| `c := b+b;` | $\langle c, 2\theta_{b,p} \rangle$ | | |
| `a := b;` | $\langle a, \theta_{b,p} \rangle$ | | |
| `a := a-1;` | $\langle a, \theta_{b,p} - 1 \rangle$ | $\langle a, \theta_{b,p} \rangle$ | |
| `assert(a<b);` | | | $\theta_{b,p} - 1 < \theta_{b,p}$ |
| `assert(c=a);` | | | $2\theta_{b,p} = (\theta_{b,p} - 1)$ |

At the first statement the variable $b$ is undefined in $\hat{\Omega}$, so it is assigned the symbolic constant $\theta_{b,p}$. The **assert** statement also constrains the value of $\theta_{b,p}$ to be greater than zero, which is reflected in the $\Phi$ column. The next two assignment statements simply add to $\hat{\Omega}$. The assignment statement "`a:=a-1`" moves the pair $\langle a, \theta_{b,p} \rangle$, representing the old value of $a$, from $\hat{\Omega}$ to $\Pi$ and adds $\langle a, \theta_{b,p} - 1 \rangle$ to $\hat{\Omega}$. The final context of the path $\langle \hat{\Omega}, \Pi, \Phi \rangle = SP(\emptyset^3, p)$ is thus:

$$\begin{array}{lcl} \hat{\Omega} & = & \{\langle a, \theta_{b,p} - 1 \rangle, \langle b, \theta_{b,p} \rangle, \langle c, 2\theta_{b,p} \rangle\} \\ \Pi & = & \{\langle a, \theta_{b,p} \rangle\} \\ \Phi & = & \{(\theta_{b,p} > 0), (\theta_{b,p} - 1 < \theta_{b,p}), (2\theta_{b,p} = \theta_{b,p} - 1)\} \end{array}$$

This path is infeasible because $(\theta_{b,p} > 0) \Rightarrow (2\theta_{b,p} \neq \theta_{b,p} - 1)$. In comparison, the traditional Hoare-logic strongest postcondition of $p$ is:

$$SP_H(\mathbf{true}, p) = (b > 0) \wedge (c = b + b) \wedge [\exists a' : (a' = b) \wedge (a = a' - 1)] \wedge (a < b) \wedge (c = a)$$

Eliminating $a'$ yields: $(b > 0) \wedge (c = b + b) \wedge (a = b - 1) \wedge (a < b) \wedge (c = a)$. We add the equality $b = \theta_{b,p}$ to get $(b = \theta_{b,p}) \wedge (b > 0) \wedge (c = b + b) \wedge (a = b - 1) \wedge (a < b) \wedge (c = a)$. Performing forward substitutions yields:

$$\begin{array}{l} (b = \theta_{b,p}) \wedge (\theta_{b,p} > 0) \wedge (c = 2\theta_{b,p}) \wedge (a = \theta_{b,p} - 1) \wedge \\ (\theta_{b,p} - 1 < \theta_{b,p}) \wedge (2\theta_{b,p} = \theta_{b,p} - 1) \end{array}$$

which is equivalent to $\mathcal{C}(\langle \hat{\Omega}, \Pi, \Phi \rangle)$. □

## 4.3 Eliminating Infeasible Paths in a Boolean Program

A path $p$ contains *loops* if its witnessing trace has two occurrences of the same label in $L(P)$. In the following, we assume that the given path $p$ does not contain loops. If $p$ contains loops, we can unroll the loops in $P$ a finite number of times to create an equivalent program $P'$ and equivalent path $p'$, such that $p'$ does not contain loops.

In this section, we show that if path $p$ is infeasible in program $P$ then $E = \mathcal{C}(\hat{\Omega}) \cup \mathcal{C}(\Pi) \cup \Phi$, where $\langle \hat{\Omega}, \Pi, \Phi \rangle = SP(\emptyset^3, p)$, is a sufficient set of conditions to make $p$ infeasible in $\mathcal{B}(P, E)$. However, since expressions in $E$ can refer to symbolic constants in $\Theta_p$, we need to introduce "annotations" to relate these symbolic constants to the boolean program. Given $P$ and $E$, let $E' \subseteq E$ be the expressions of the form $(x = \theta_{x,p})$ in $E$. A pair $a \in L(P) \times E'$ is called an *annotation*. Annotations are used to force values to variables of the boolean program $\mathcal{B}(P, E)$. Let $\langle l, e_i \rangle$ be an annotation, where $e_i$ is $(x = \theta_{x,p})$, and let $b_i$ be the boolean variable such that $\mathcal{E}(b_i) = e_i$. Let $b_{j_1}, b_{j_2}, \ldots, b_{j_k}$ be the maximal list of boolean variables not including $b_i$, such that each of the expressions $\mathcal{E}(b_{j_1}), \mathcal{E}(b_{j_2}), \ldots, \mathcal{E}(b_{j_k})$ reference $\theta_{x,p}$. *Applying* $\langle l, e_i \rangle$ to $\mathcal{B}(P, E)$ involves introducing the additional statement "$b_i, b_{j_1}, b_{j_2}, \ldots, b_{j_k} := \textbf{true}, ?, \ldots, ?$" immediately preceding the label $l$ in $\mathcal{B}(P, E)$. Given $P$, $E$, and a set $A$ of annotations, let $\hat{\mathcal{B}}(P, E, A)$ denote the program obtained by applying every annotation $a \in A$ to $\mathcal{B}(P, E)$.

Since $E$ could refer to symbolic constants that are not present in $P$, we also need to generalize the definition of *WP*, as was done with *SP*. Let $E = \{e_1, e_2, \ldots, e_n\}$ be a set of expressions and $V_B = \{b_1, b_2, \ldots, b_n\}$ be the corresponding set of boolean variables. Recall that $E = \mathcal{C}(\hat{\Omega}) \cup \mathcal{C}(\Pi) \cup \Phi$, where $\langle \hat{\Omega}, \Pi, \Phi \rangle = SP(\emptyset^3, p)$. Let us (re)interpret $\mathcal{F}(WP(s, e))$, where $s$ is an assignment or assert statement, as the set $M$ of all minterms $c_{i_1} \wedge c_{i_2} \cdots \wedge c_{i_k}$ such that $SP_H(f_{i_1} \wedge f_{i_2} \cdots \wedge f_{i_k}, s) \Rightarrow e$, where $f_{i_j} = \mathcal{E}(c_{i_j})$ (Note that we are making use of the equivalence of *SP* and $SP_H$ here to interpret the strongest postcondition with respect to a logical formula rather than a context).

**Example.** Let $E = \{x = \theta_{x,p}, y = \theta_{y,p}\}$ and let $V_B = \{b_x, b_y\}$ be the corresponding boolean variables. Now, consider $WP(\textbf{assert}(x < y), \theta_{x,p} < \theta_{y,p})$. By the Hoare-logic formulation (see Section 4.2), we have

$$WP(\textbf{assert}(x < y), \theta_{x,p} < \theta_{y,p}) = (x < y) \Rightarrow (\theta_{x,p} < \theta_{y,p})$$

Under the new interpretation of *WP*, we have that

$$\mathcal{E}(\mathcal{F}(WP(\textbf{assert}(x < y), \theta_{x,p} < \theta_{y,p}))) = \underbrace{(x = \theta_{x,p}) \wedge (y = \theta_{y,p})}$$

because

$$\begin{aligned} &SP_H(\underbrace{(x = \theta_{x,p}) \wedge (y = \theta_{y,p})}, \textbf{assert}(x < y)) = \underbrace{(x = \theta_{x,p}) \wedge (y = \theta_{y,p})} \wedge (x < y) \\ &\Rightarrow \\ &(\theta_{x,p} < \theta_{y,p}) \end{aligned}$$

□

Given a path $p = s_1, s_2, \ldots, s_n$ and $1 \leq j \leq n$, let $upex(p, j) = \{x | x \in use(s_j) \text{ and } \forall i < j, x \notin (use(s_i) \cup def(s_i))\}$.

**Theorem 2.** *Let $P$ be an* **X** *program in* **X***-normal form, and let $p$ be a path in $P$ such that $\mathcal{C}(SP(\emptyset^3, p))$ is not satisfiable. Let $SP(\emptyset^3, p) = \langle \hat{\Omega}, \Pi, \Phi \rangle$, and let $E_p = \mathcal{C}(\hat{\Omega}) \cup \mathcal{C}(\Pi) \cup \Phi$. Let $A_p = \{\langle \hat{\Gamma}(j), x = \theta_{x,p} \rangle | x \in upex(p, j)\}$ be a set of annotations. Then the following statements hold: (1) $P \preceq \hat{\mathcal{B}}(P, E_p, A_p)$, and (2) $p$ is infeasible in $\hat{\mathcal{B}}(P, E_p, A_p)$.*

*Proof.* Let $E_p = \{e_1, e_2, \ldots, e_n\}$, and let $V(\hat{\mathcal{B}}(P, E_p, A_p)) = \{b_1, b_2, \ldots, b_n\}$. Let $p'$ be an arbitrary path of $P$. We can prove by induction on the length of $P$ (similar to the proof of

Lemma 1) that whenever a boolean variable $b_i$ is **true** (**false**) in $\hat{\mathcal{B}}(P, E_p, A_p)$, then $e_i$ is **true** (**false**) in the context of the path simulator simulating the same path in $P$. This proves (1).

We say that an expression $e$ is in the context $(\hat{\Omega}, \Pi, \Phi)$ if $e \in \mathcal{C}(\hat{\Omega}) \cup \Phi$. Let $\overline{\eta}$ be the trajectory in $\hat{\mathcal{B}}(P, E_p, A_p)$ such that $\Gamma(\overline{\eta})$ is the witnessing trace for the path $p$ in $P$. To prove (2), we prove that if $e_i$ ($!e_i$) is in context of the path simulator simulating $p$ then $b_i$ is **true** (**false**) in the corresponding state of the trajectory $\overline{\eta}$ in $\hat{\mathcal{B}}(P, E_p, A_p)$. The base case is trivial since path simulation begins with the empty context $\emptyset^3 3$. Let $(\hat{\Omega}_1, \Pi_1, \Phi_1)$ be the context of the path simulator before executing some statement $s$ and let $(\hat{\Omega}_2, \Pi_2, \Phi_2) = SP((\hat{\Omega}_1, \Pi_1, \Phi_1), s)$. Let $\eta_1 = \langle l_1, \Omega_1 \rangle$ be the state of $\hat{\mathcal{B}}(P, E_p, A_p)$ before executing the translation of $s$ and $\eta_2 = \langle l_2, \Omega_2 \rangle$ be the state of $\hat{\mathcal{B}}(P, E_p, A_p)$ after executing the translation of $s$, in the trajectory $\overline{\eta}$. Since the translation introduces annotations, a single statement in $s$ could translate to multiple statements in $\hat{\mathcal{B}}(P, E_p, A_p)$. Suppose $e_i \in E_p$ is in the context $(\hat{\Omega}_2, \Pi_2, \Phi_2)$ (if $!e_i$ is in the context $(\hat{\Omega}_2, \Pi_2, \Phi_2)$, we can handle it similarly). There must exist $f_{j_1}, f_{j_2} \ldots f_{j_m}$ such that

- $f_{j_1} \wedge f_{j_2} \cdots \wedge f_{j_m} \Rightarrow WP(s, e_i)$.
- for each $0 \leq r \leq m$, $f_{j_i} \in \{e_{j_i}, !e_{j_i}\}$ for some $e_{j_i} \in E_p$.
- for each $0 \leq r \leq m$, either
  - $f_{j_r}$ is in $(\hat{\Omega}_1, \Pi_1, \Phi_1)$,or
  - $f_{j_r}$ is of the form $x = \theta_{x,p}$ and $\langle x, \theta_{x,p} \rangle \in undef(\hat{\Omega}_1, use(s))$.

Consider $f_{j_r}$ for each $0 \leq r \leq m$. Suppose $f_{j_r}$ is in $(\hat{\Omega}_1, \Pi_1, \Phi_1)$. If $f_{j_r} = e_{j_r}$, then $\Omega_1(b_{j_r})$ is **true** by induction. If $f_{j_r} =!e_{j_r}$, then then $\Omega_1(b_{j_r})$ is **false** by induction. If $f_{j_r}$ is of the form $x = \theta_{x,p}$, and $\langle x, \theta_{x,p} \rangle \in undef(\hat{\Omega}_1, use(s))$, then a statement "$b_{j_r}$ := **true**" should have been introduced in $\hat{\mathcal{B}}(P, E_p, A_p)$ as a result of applying the annotation for $(x = \theta_{x,p})$ in $A_p$. Thus, $b_{j_r}$ is set to **true** in $\hat{\mathcal{B}}(P, E_p, A_p)$ by executing this statement. Since $f_{j_1} \wedge f_{j_2} \cdots \wedge f_{j_m} \Rightarrow WP(s, e_i)$, and each $e_{j_r} \in E_p$ for $0 \leq r \leq m$, we know that $b_i$ is set to **true** in $\hat{\mathcal{B}}(P, E_p, A_p)$ after executing the translation of $s$. Note that we cannot claim this along some other path $p'$ because, $e_i$ might become **true** because of some other reasons that we have not modeled. □

## 4.4 Iterating the Refinement Process

The previous sections showed how to complete one iteration of the refinement process. The next natural question is how to iterate this process. The following theorem shows the way:

**Theorem 3.** *Let $P$ be an* **X** *program in* **X***-normal form, and let $E_1, E_2$ be sets of expressions over $V(P)$ and constants, such that $E_1 \subseteq E_2$. Then $P \preceq \mathcal{B}(P, E_2) \preceq \mathcal{B}(P, E_1)$.*

*Proof.* Silimar to proof of Lemma 1. □

This theorem states that $\mathcal{B}$ is a monotone function with respect to the $\preceq$ relation. Thus, every iteration of the refinement simply adds to the set of expressions $E$ (and corresponding boolean variables) in order to rule out more infeasible paths in $P$.

We now address the termination properties of the refinement process. The process will terminate if one of the two following conditions is met:

– Model checking of the boolean program $B$ shows that label $l$ is not reachable in $B$.
– A path $p$ (corresponding to a path in $B$ that reaches label $l$) is found that is feasible in $P$.
– The decision procedure cannot determine whether path $p$ is feasible or infeasible in $P$. In this case, the process terminates with the result "don't know".

If the path $p$ is found to be infeasible. In this case, the process iterates. Because there may be an unbounded number of infeasible paths in $P$, it is possible that the process may iterate forever due to this case. However, at any time, we may halt the process and the current boolean program is guaranteed to abstract $P$. Premature termination corresponds to giving a "don't know" answer.

## 4.5 Transforming Multi-Procedure Programs

We now consider how to extend the above techniques when $P$ is a multi-procedure program in **X**-normal form. There are several basic issues to address:

– How do we extend path simulation to interprocedural paths?
– How do we determine the scope of boolean variables (as global variables, local variables or formal parameters) in the boolean program?
– How do we translate the source program into a boolean program in the presence of procedure calls?

Path simulation and strongest postconditions generalize to interprocedural paths via a simple two-step transformation to an interprocedural path containing calls:

– All instances of the formal parameters and local variables in a procedure invocation are renamed in the path to be unique to that invocation;
– The procedure call $A(e_1, e_2, \cdots, e_n)$ is transformed to a parallel assignment $x_1, x_2, \cdots, x_n := e_1, e_2, \cdots, e_n$, where the $x_i$ are the (renamed) formal parameters of $A$.

Thus, the transformation faithfully models the call-by-value parameter passing of the **X** language. Each statement in the transformed path has a corresponding statement in the original interprocedural path.

Consider the program at the top of Figure 6. The path corresponding to the sequence of labels $[1, 5-7, 2-3, 5]$ is infeasible. The table in Figure 6 shows the transformed interprocedural path (**skip** and **return** statements have been omitted since they have no effect on $SP$), and the effect of each statement on $\hat{\Omega}$ and $\Phi$ (in this example, $\Pi = \emptyset$ at all times). Just before the second instance of label [5], the conditions over the symbolic constants are $\theta_x < \theta_y$. Thus, the condition introduced by the assert statement at the second instance label [5], $\theta_y < \theta_x$ is contradictory. By the results of the previous sections, the set of boolean variables $\{xTx, yTy, b1x, b2y, b1y, b2x, xLy, yLx\}$ where

$$\begin{array}{llll}
\mathcal{E}(xTx) = (x = \theta_x) & \mathcal{E}(b1x) = (b1 = \theta_x) & \mathcal{E}(b1y) = (b1 = \theta_y) & \mathcal{E}(xLy) = (\theta_x < \theta_y) \\
\mathcal{E}(yTy) = (x = \theta_y) & \mathcal{E}(b2y) = (b2 = \theta_y) & \mathcal{E}(b2x) = (b2 = \theta_x) & \mathcal{E}(yLx) = (\theta_y < \theta_x)
\end{array}$$

```
    x,y: int;
    A() {
[1]   B(x,y);
[2]   skip;
[3]   B(y,x);
[4]   skip;
    }

    B(b1,b2: int) {
[5]   assert(b1<b2);
[6]   skip;
[7]   return;
    }
```

| Statement | $\hat{\Omega}$ | $\Phi$ |
|---|---|---|
| `[1] b1.1, b2.1 := x, y;` | $\langle x, \theta_x \rangle$<br>$\langle y, \theta_y \rangle$<br>$\langle b1.1, \theta_x \rangle$<br>$\langle b2.1, \theta_y \rangle$ | |
| `[5] assert(b1.1<b2.1);` | | $\theta_x < \theta_y$ |
| `[3] b1.2, b2.2 := y, x;` | $\langle b1.2, \theta_y \rangle$<br>$\langle b2.2, \theta_x \rangle$ | |
| `[5] assert(b1.2<b2.2);` | | $\theta_y < \theta_x$ |

```
    xTx,yTy: bool;
    xLy,yLx: bool;
    A() {
      xTx,xLy,yLx := true,?,?;
      yTy,xLy,yLx := true,?,?;
[1]   B(xTx,false,yTy,false);
[2]   skip;
[3]   B(false,yTy,false,xTx);
[4]   skip;
    }
```

```
    B(b1x, b1y, b2y, b2x: bool) {
[5]   assert( ! ( (b1x & b2y & !xLy) | (b1x & b2x) |
        (b1y & b2x & !yLx) | (b1y & b2y) ) );
[6]   xLy,yLx :=
        ℋ (xLy|(b1x&b2y),!xLy|yLx|(b1y&b2x)),
        ℋ (yLx|(b1y&b2x),!yLx|xLy|(b1x&b2y));
[7]   return;
    }
```

**Fig. 6.** A program with an interprocedural infeasible path (as shown in the table on the right), and the corresponding boolean program (bottom).

is sufficient to rule out the infeasible path in the boolean program $\mathcal{B}(P, E, A)$: Note that we have collapsed the unique instances of the formal parameters back to the single parameter name from which those instances were generated. For example, `b1.1` and `b1.2` reduce back to `b1`.

Given the set of conditions to model, and their corresponding boolean variables, we must choose the scope of these boolean variables in the boolean program. Let $\langle \hat{\Omega}, \Pi, \Phi \rangle = SP(\emptyset^3, p)$, let $E' = \mathcal{C}(\hat{\Omega}) \cup \mathcal{C}(\Pi)$, and let $E = E' \cup \Phi$. Expressions in $E'$ are of the form $(x = e)$, where $e$ is an expression over symbolic constants and constants of the **X** language. Expressions in $\Phi$ only refer to symbolic constants, and constants in the **X** language (no program variables). Thus, given an expression $f \in E$, and its corresponding boolean variable $b_f$, we have the following rules for determining the scope of $b_f$:

- If $f \in \Phi$ then $b_f$ is a global variable (we conservatively assume that every symbolic constant introduced lives forever, and thus has global scope).
- Otherwise, $f$ is of the form $x = e$. There are three subcases, depending on the scope of $x$:
  - If $x$ is a global variable then $b_f$ is a global variable.
  - If $x$ is a formal parameter of procedure $Q$ in program $P$, then $b_f$ is a formal parameter of $Q$ in the boolean program.
  - Finally, if $x$ is a local variable of procedure $Q$ in program $P$, then $b_f$ is a local variable of $Q$ in the boolean program.

In our example, this means that the boolean variables $\{xTx, yTy, xLy, yLx\}$ are global, while the variables $\{b1x, b1y, b2y, b2x\}$ are formal parameters of procedure $B$. Note that a single parameter in the original program (such as $b1$ in procedure $B$) can give rise to multiple parameters in the boolean program (namely, $b1x$ and $b1y$).

The translation of the source program $P$ into the boolean program must be extended to handle procedure calls. As with the single procedure case, if there is a loop in the path $p$ (i.e., there is more than one instance of a label $l$) then some amount of finite inlining may be needed in order to make the path loop-free. In our example, there is a repeated label ([5]) but in this case inlining is not needed (because no annotations are required in procedure $B$).

We have shown how the formal parameters of a procedure in the boolean program $\mathcal{B}(P, E, A)$ are determined: for each formal parameter $fp$ of procedure $Q$ in the source program, the corresponding procedure $Q$ in $\mathcal{B}(P, E, A)$ has one formal parameter for each expression in $E$ of the form $(fp = e)$. The remaining issue in translating a procedure call is to determine the actual parameters to a call in $\mathcal{B}(P, E, A)$. Given a pair $(ap, fp)$ of actual parameter $ap$, and corresponding formal parameter $fp$ (in a call to procedure $Q$ in the source program), and an expression of the form $(fp = e)$ from $E$, the actual parameter corresponding to the formal parameter for $(fp = e)$ is $\mathcal{I}(fp := ap, (fp = e))$, with the following restriction. We restrict the set of expressions $E$ from which a minterm is constructed to be those that are visible at the particular call to $Q$ that we are translating.

In our example, procedure $B$ has the formal parameter list $b2x, b2y, b3y, b3x$. Let us consider the procedure call $B(x, y)$, which is modeled by the parallel assignment $b1, b2 := x, y$. It is easy to see that $\mathcal{I}(b1 := x, (b1 = \theta_x))) = xTx$. Now, consider $\mathcal{I}(b1 := x, (b1 = \theta_y))$. Given the expressions in $E$, there is no way to make $b1 = \theta_y$ true through the assignment $b1 := x$. Thus, $\mathcal{I}(b1 := x, (b1 = \theta_y)) = \textbf{false}$.

The translations (from Table 3) are unchanged, with the exception that the construction of minterms in $\mathcal{F}$ respect the scope of the boolean variables corresponding to the expressions in $E$. The bottom of Figure 6 shows the resulting boolean program.

## 5 Minimizing "Explanations" of Infeasible Paths

Given an infeasible path $p$, we now show how to find a smaller set of expressions $E'$ and annotations $A'$ to $P$ such that path $p$ is infeasible in $\hat{\mathcal{B}}(P, E', A')$. Intuitively, we wish to eliminate statements from an infeasible path $p$ to see if the resulting path is still infeasible. However, we wish to do so in a way that is "consistent" with $p$. The motivation behind doing this is to reduce the number of boolean variables needed to model the infeasibility of a path. Moreover, fewer boolean variables that are needed, the more likely other infeasible paths will be ruled eliminated.

### 5.1 Consistent Path Projections

Let path $p = s_1, s_2, \ldots, s_n$, as before. Path $q$ is a *projection* of path $p$ if there is a sequence of integers $i_1, i_2, \ldots, i_m$ such that $q = s_{i_1}, s_{i_2}, \ldots, s_{i_m}$, where $m \leq n$ and for all $j$, $i_j < i_{j+1}$ and $1 \leq i_j \leq n$. If $q$ is a projection of $p$ and $j$ is the index of a statement in $q$, let $j_p$ be the

| $p_1$ | $\hat{\Omega}_1$ | $\Phi_1$ | $p_2$ | $\hat{\Omega}_2$ | $\Phi_2$ | $p_3$ | $\hat{\Omega}_3$ | $\Phi_3$ |
|---|---|---|---|---|---|---|---|---|
| `assert(b>0);` | $\langle b, \theta_{b,1} \rangle$ | $\theta_{b,1} > 0$ | `assert(b>0);` | $\langle b, \theta_{b,2} \rangle$ | $\theta_{b,2} > 0$ | `assert(b>0);` | $\langle b, \theta_{b,3} \rangle$ | $\theta_{b,3} > 0$ |
| `c := b+b;` | $\langle c, 2\theta_{b,1} \rangle$ | | `c := b+b;` | $\langle c, 2\theta_{b,2} \rangle$ | | `c := b+b;` | $\langle c, 2\theta_{b,3} \rangle$ | |
| `z := c;` | $\langle z, 2\theta_{b,1} \rangle$ | | | | | | | |
| `a := b;` | $\langle a, \theta_{b,1} \rangle$ | | `a := b;` | $\langle a, \theta_{b,2} \rangle$ | | | | |
| `a := a-1;` | $\langle a, \theta_{b,1} - 1 \rangle$ | | `a := a-1;` | $\langle a, \theta_{b,2} - 1 \rangle$ | | | | |
| `x := b;` | $\langle x, \theta_{b,1} \rangle$ | | | | | | | |
| `assert(a<b);` | | $(\theta_{b,1} - 1) < \theta_{b,1}$ | `assert(a<b);` | | $(\theta_{b,2} - 1) < \theta_{b,2}$ | `assert(a<b);` | $\langle a, \theta_{a,3} \rangle$ | $\theta_{a,3} < \theta_{b,3}$ |
| `assert(c=a);` | | $2\theta_{b,1} = (\theta_{b,1} - 1)$ | `assert(c=a);` | | $2\theta_{b,2} = (\theta_{b,2} - 1)$ | `assert(c=a);` | | $2\theta_{b,3} = \theta_{a,3}$ |

**Fig. 7.** Three paths annotated with their store ($\hat{\Omega}$) and conditions over the symbolic expressions ($\Phi$). Path $p_1$ is infeasible. Paths $p_2$ and $p_3$ are both ICPPs of $p_1$. In addition, $p_3$ is an ICPP of $p_2$.

index of the corresponding statement in $p$, namely $i_j$. A *$q$-annotation* to a path $p$ is a pair $\langle i, (x = \theta_{x,q}) \rangle$ such that $i$ is an integer, $1 \leq i \leq n$. A $q$-annotation differs from an annotation (defined in Section 4.3) in that the first component is an integer index, and not a label. Given a path $p = s_1, \ldots, s_n$ and an $q$-annotation $a = \langle i, (x = \theta_{x,q}) \rangle$, *applying* $a$ to $p$ inserts the parallel assignment "$x := \theta_{x,q}$" immediately before statement $s_i$ in $q$. Given a path $p$ and a set $A_q$ of $q$-annotations, let $p_{A_q}$ denote the path obtained by applying every $q$-annotation $a \in A_q$ to $p$.

Path $q$ is a *consistent path projection* (or CPP) of path $p$ if (1) $q$ is a projection of $p$, and (2) there exists a set $A_q$ of $q$-annotations such that: $\mathcal{C}(SP(\emptyset^3, p_{A_q})) \Rightarrow \mathcal{C}(SP(\emptyset^3, q))$. Let $p$ be an infeasible path ending with an **assert** statement such that every proper prefix of $p$ is feasible. Path $q$ is an *infeasible consistent path projection* (ICPP) of an infeasible path $p$ if (1) $q$ contains the last **assert** statement of $p$ (2) $q$ is infeasible, and (3) $q$ without its last **assert** statement is a CPP of $p$ without its last **assert** statement.

**Example.** Path $p_1$ in Figure 7 is infeasible. In the two columns to the right of $p_1$ are symbolic values of the variables (column $\hat{\Omega}_1$) and the conditions over these values (column $\Phi_1$). For each statement in $p_1$, the columns show the additions to $\hat{\Omega}_1$ and $\Phi_1$ that the statement makes. Note that the statement "`a:=a-1`" replaces the existing binding for variable $a$. Column $\Phi_1$ clearly explains the infeasibility, as $\theta_{b,1} > 0 \Rightarrow 2\theta_{b,1} \neq (\theta_{b,1} - 1)$. Path $p_2$ is an ICPP of $p_1$, as it does not contain the assignments to variables $z$ and $x$ that are in $p_1$, which are not referenced in any of the **assert** statements.[1] Path $p_3$ is an ICPP of $p_1$ as well as $p_2$.[2] This path shows that the exact value of variable $a$ is not important in explaining the infeasibility. That is, $(\theta_{b,3} > 0 (\wedge (\theta_{a,3} < \theta_{b,3}) \Rightarrow 2\theta_{b,3} \neq \theta_{a,3}$. We show why $p_3$ is an ICPP of $p_2$, which by transitivity shows that $p_3$ is an ICPP of $p_1$. Let $p'_2$ and $p'_3$ be paths $p_2$ and $p_3$ without their last **assert** statements. Let $A_{p'_3} = \{ \langle 1, (b = \theta_{b,3}) \rangle, \ \langle 5, (a = \theta_{a,3}) \rangle \}$. Applying this set of $p'_3$-annotations to path $p'_2$ yields the program $p''_2$:

[1] The ICPP $p_2$ of $p_1$ could be formed using traditional program slicing on $p_1$ since the expression "`a<b`" is not (transitively) flow dependent on the assignment "`x:=b`" or "`z:=c`".

[2] In this case, traditional program slicing could not be used to construct $p_3$ from $p_2$, as the expression "`a<b`" is (transitively) flow dependent on the assignments "`a:=a-1`" and "`a:=b`".

```
b := $\theta_{b,3}$;
assert (b>0);
c := b+b;
a := b;
a := a-1;
a := $\theta_{a,3}$;
assert(a<b);
```

It is straightforward to see that $\mathcal{C}(SP(\emptyset^3,p''_2)) \Rightarrow \mathcal{C}(SP(\emptyset^3,p'_3))$ This example shows that reinterpreting the symbolic value of $a$ in the statement "`assert(a<b);`" of $p'_2$ as $\theta_{a,3}$ (rather than $\theta_{b,2}-1$) allows the infeasibility of the path $p_2$ to be explained by a weaker (larger) set of states. □

### 5.2 Syntactically Consistent Path Projections

We now present a syntactic dependency condition that gives us a procedure for constructing CPPs and ICPPs. Given a statement $s_j$ in path $p$, let

$$del(s_j) = \{s_j\} \cup \{del(s_i) | i < j, def(s_j) \cap (use(s_i) \cup def(s_i)) \neq \emptyset\}$$

If statement $s$ in path $p$ is an assignment "$x := e$", then $del(s)$ contains $s$ and the $del$ set of every statement preceding $s$ in $p$ that uses or defines variable $x$. If $s$ is an **assert** statement then $del(s) = \{s\}$, as an **assert** does not assign to any variable. Path $q$ is a *syntactically consistent path projection* (or SCPP) of path $p$ if (1) $q$ is a projection of $p$ and (2) if statement $s$ (from $p$) is not in $q$ then none of the statements in $del(s)$ are in $q$.

**Example.** In Figure 7, paths $p_2$ and $p_3$ are SCPPs of path $p_1$. Furthermore, path $p_3$ is an SCPP of $p_2$. In path $p_1$, the $del$ set of the statement "`a:=a-1`" is { `a:=b, a:=a-1` }, which means that in order to form an SCPP the statement "`a:=b`" must be removed from $q$ whenever the statement "`a:=a-1`" is removed from $q$. On the other hand, the $del$ set of the statement "`assert(a<b)`" consists of the statement itself. □

To establish that every CPP of $p$ is an SCPP of $p$, we define the notion of control points. A *control point* is a point of control between two statements in a path (a control point also exists before the first statement of a path and after the last statement of a path). If path $p = s_1, ..., s_n$, then $p$ has $n+1$ control points $c_0, ..., c_n$, where $c_0$ is the control point before statement $s_1$ and, for $i > 0$, $c_i$ is the control point after statement $s_i$. Given a control point $c$, let $seq(p,c)$ be the sequence of statements in $p$ preceding $c$. If $q$ is a projection of $p$, then for every control point $c$ in $p$ there is a corresponding control point $q(c)$ in $q$.

**Lemma 2.** *Let p be the path $s_1, ..., s_n$ with control points. $c_0, ..., c_n$. A variable $x$ is in $upex(p,j)$*[3] *iff $\langle x, \theta_{x,p}\rangle \in undef(\hat{\Omega}, use(s_j))$, where $(\hat{\Omega}, \Pi, \Phi) = SP(\emptyset^3, seq(p, c_{j-1}))$.*

The proof of the lemma is straightforward. It is used in proving the following theorem.

[3] See Section 4.3 for the definition of *upex*.

**Theorem 4.** *For any two paths $p$ and $q$, if $q$ is an SCPP of $p$ then $q$ is a CPP of $p$.*

*Proof.* Let $A_q = \{\langle j_p, (x = \theta_{x,q})\rangle \mid x \in \mathit{upex}(q, j)\}$. Let path $p' = p_{A_q}$ be a sequence of statements $s_1, ..., s_n$ with $n+1$ corresponding control points $c_0, ..., c_n$. It is clear that since $q$ is a projection of $p$ it is also a projection of $p'$. We prove, by induction on $k$, that

$$\mathit{SP}(\emptyset^3, \mathit{seq}(p', c_k)) \supseteq \mathit{SP}(\emptyset^3, \mathit{seq}(q, q(c_k)))$$

from which it follows directly that $\mathcal{C}(\mathit{SP}(\emptyset^3, \mathit{seq}(p', c_k))) \Rightarrow \mathcal{C}(\mathit{SP}(\emptyset^3, \mathit{seq}(q, q(c_k))))$.

*Base Case*: Consider the initial control point $c_0$ in $p'$. Because both $c_0$ and $q(c_0)$ are the first control points in their respective paths, $\mathit{SP}(\emptyset^3, \mathit{seq}(p', c_0)) = \mathit{SP}(\emptyset^3, \mathit{seq}(q, q(c_0))) = \emptyset^3$. So we are done with the base case.

*Induction Step:* Consider a control point $c_k$ in $p'$ such that $i > 0$. Let $P'_k = (\hat{\Omega}_{p'_k}, \Pi_{p'_k}, \Phi_{p'_k}) = \mathit{SP}(\emptyset^3, \mathit{seq}(p', c_k))$ and let $Q_k = (\hat{\Omega}_{q_k}, \Pi_{q_k}, \Phi_{q_k}) = \mathit{SP}(\emptyset^3, \mathit{seq}(q, q(c_k)))$. By the Induction Hypothesis, we have that $P'_k \supseteq Q_k$. There are two cases to consider: $s_{k+1}$ is in $q$ or $s_{k+1}$ is not in $q$. In the first case, we want to show that

$$P'_{k+1} = \mathit{SP}(P'_k, s_{k+1}) \supseteq Q_{k+1} = \mathit{SP}(Q_k, s_{k+1}))$$

We perform a case analysis on statement $s_{k+1}$ to show that $P'_{k+1} \supseteq Q_{k+1}$:

- Statement $s_{k+1}$ is an assignment statement "$x := e$": We first show that $\hat{\Omega}_{p'_{k+1}} \supseteq \hat{\Omega}_{q_{k+1}}$. Let $\bar{\Omega}_{p'_k} = \hat{\Omega}_{p'_k} \cup \mathit{undef}(\hat{\Omega}_{p'_k}, \mathit{use}(e))$ and $\bar{\Omega}_{q_k} = \hat{\Omega}_{q_k} \cup \mathit{undef}(\hat{\Omega}_{q_k}, \mathit{use}(e))$, per the definition of *SP* for the assignment statement. Consider a variable $y \in \mathit{use}(e)$. By Lemma 2, we have $y \in \mathit{upex}(q)$ iff $\langle y, \theta_{y,q}\rangle \in \mathit{undef}(\hat{\Omega}_{q_k}, \mathit{use}(e))$, and thus in $\bar{\Omega}_{q_k}$. Since $p' = p_{A_q}$ there also must be an assignment statement preceding $s_{k+1}$ in $p'$ due to applying the annotation that assigns $\theta_{y,q}$ to $y$. This means that $\langle y, \theta_{y,q}\rangle$ is in $\bar{\Omega}_{p'_k}$ as well. Since $\bar{\Omega}_{p'_k} \supseteq \bar{\Omega}_{q_k}$, it follows that

  $$\bar{\Omega}_{p'_k}[x/\bar{\Omega}_{p'_k}(e)] \supseteq \bar{\Omega}_{q_k}[x/\bar{\Omega}_{q_k}(e)]$$

  Now, we need only show that $\Pi_{p'_{k+1}} \supseteq \Pi_{q_{k+1}}$( since the $\Phi$ components remain unchanged by *SP* for assignment statements). This follows straightforwardly from the fact that $\bar{\Omega}_{p'_k} \supseteq \bar{\Omega}_{q_k}$.
- Statement $s_{k+1}$ is an **assert** statement "**assert**$(e)$": The analysis follows that of the preceding paragraph. We again arrive at the point that $\bar{\Omega}_{p'_k} \supseteq \bar{\Omega}_{q_k}$, which means that $\hat{\Omega}_{p'_{k+1}} \supseteq \hat{\Omega}_{q_{k+1}}$, since for **assert** statements $\hat{\Omega}_{p'_{k+1}} = \bar{\Omega}_{p'_k}$ and $\hat{\Omega}_{q_{k+1}} = \bar{\Omega}_{q_k}$. As a result, it is clear that

  $$\Phi_{p'_{k+1}} = \Phi_{p'_k} \cup \{\bar{\Omega}_{p'_k}(e)\} \supseteq \Phi_{q_{k+1}} = \Phi_{q_k} \cup \{\bar{\Omega}_{q_k}(e)\}$$

  Since **assert** statements do not affect the $\Pi$ components, we are done.

Suppose $s_{k+1}$ is not in $q$. In this case, we must show that $\mathit{SP}(P'_k, s_{k+1}) \supseteq Q_k$. Because of the assignment statements introduced by the $q$-annotations there are three cases to consider:

- Statement $s_{k+1}$ is an assignment statement "$x := e$" originally in $p$: By construction of SCPPs, since $s_{k+1}$ is not in $q$, no statement in $\mathit{seq}(q, q(c_k))$ can refer to the variable $x$. This means that $Q_k$ contains no element referring to $x$. As a result, it is clear that $P'_{k+1} \supseteq Q_k$.

– Statement $s_{k+1}$ is an assignment statement introduced into $p'$ by the $q$-annotations: Consider any of the variables $x$ assigned to in $s_{k+1}$. As before, no statement in $seq(q, q(c_k))$ can refer to the variable $x$ (otherwise $x$ would not be assigned to in $s_{k+1}$ as a result of applying $A_q$).
– Statement $s_{k+1}$ is an **assert** statement "**assert**$(e)$": Trivial, from the definition of $SP$.

□

### 5.3 A Greedy Algorithm for Constructing ICPPs

The following Lemma about CPPs (and thus SCPPs) gives us a way to construct ICPPs.

**Lemma 3.** *If $p$ is a feasible path of assignment statements and* **assert** *statements and $q$ is a CPP of $p$, then $q$ is feasible.*

*Proof.* Follows directly from the definition of CPPs. □

We give the following greedy algorithm for computing an ICPP of an infeasible path $p$:

```
path ICPP (path $p$) {
    for each statement $s_i$ in $p$ {
        $p' = p \setminus del(s_i)$;
        if $infeasible(p')$ then
            return(ICPP ($p'$))
    }
    return($p$)
}
```

Once a (locally) small ICPP $q$ has been found, the path simulator is used to retrieve the context $(\hat{\Omega}, \Pi, \Phi) = SP(\emptyset^3, q)$ that contains the desired conditions $E_q$, and associated annotations $A_q$ for the construction of $\hat{\mathcal{B}}(P, E_q, A_q)$.

## 6 Related Work

Model checking for state machines is a well studied problem, and several model checkers — SMV [McM93], Mur$\phi$ [Dil96], SPIN [HP96], COSPAN [HHK96], and MOCHA [AHM$^+$98]— have been developed. Several notions of refinement, including trace containment, have been studied before [Mil71] [AL88] [Kur94]. Boolean programs can be viewed as abstract interpretations of the underlying program [CC77]. The connections between model checking, dataflow analysis and abstract interpretation have been explored before [Sch98] [CC00]. The model checking problem for pushdown automata has been studied before [SB92] [BEM97] [FWW97]. The CFL reachability framework from [RHS95] builds on earlier work in interprocedural dataflow analysis from [KS92] and [SP81]. However, our notion of trace semantics for programs with unbounded recursion, and the use of CFL reachability as a model checking procedure for boolean programs are new. Exploiting design modularity in model checking has been recognized as a key to scalability of model checking [AH96] [AG00] [McM97] [HQR98]. However, the idea of using a boolean program to harness the inherent modularity in procedural abstraction, and exploiting locality of variable scoping for efficiency in model checking is new.

Program slicing is a projection operation on programs that preserves the execution behavior of statements in the projection [Wei82]. The semantics of slicing generally guarantees that the projection will have "identical" behavior to the original program, if started in an "identical" state. This form of slicing is used by Hatcliff and Dwyer [HD99] in their work on constructing models from programs. If we apply their algorithm to the example program $P$ of Figure 1, slice would contain all the variables. In particular, the slice would determine that the variable *level* is relevant in determining the reachability status of 10, which is not true. As our results on consistent path projections show, if one's goal is to compute projections of a path that *abstract* its behavior (rather than preserve it), one could slice more aggressively. Godefroid et al.'s work on closing open system uses a program slicing operation for a different goal [CGJ98]. In this work, a sequential open program $P$ (interacting with an environment $E$) is transformed into a program $P'$ such that for any environment $E$, the set of traces of $P'$ is a superset of the set of traces of $E \| P$.

## 7 Conclusion and Future Work

We have presented a process for abstracting programs based on the model of boolean programs, which are sequential programs with procedure calls and boolean variables. The theory of context-free-language reachability yields a model checking algorithm for boolean programs. An initial boolean program $B$ representing the source program $P$ is incrementally refined with respect to a particular reachability query in $P$. The presence of infeasible paths in $P$ may lead the model checker to report false positive errors in $B$. We have shown how to incrementally refine $B$ by introducing boolean variables to rule out the infeasible path. In addition, the use of consistent path projections allows us to reduce the number of boolean variables needed to eliminate an infeasible path.

There are several directions to pursue in the future:

- *Language extensions.* We are considering how to extend our framework to handle pointers, function pointers, dynamic memory allocation and concurrency. A single level of indirection through pointers allows two names in a program to refer to the same storage cell. We believe that we can accommodate single-level pointers in our framework without incorporating pointers into boolean programs.
- *Efficient algorithms.* All of the algorithms in this paper will have to be refined in order to make them practical. The CFL reachability algorithm can be optimized in a number of ways. For example, we have found a way to use symbolic model checking techniques (BDDs in particular) to encode the exploded graph of Section 3 and perform model checking of boolean programs symbolically. We are also exploring direct ways of creating infeasible consistent path projections from an infeasible path (rather than running the path simulator over and over again on different consistent path projections) based on connections between consistent path projections and Stalmarck's method. Our refinement process will benefit also from any improvements in the decision procedures that path simulators use to determine the feasibility of a path.

– *Experimental Results.* Our plans are to implement these algorithms for a subset of the C language and experiment with them in the domain of device drivers. We will report on this work in a later paper. A key issue that the experimental results will address is how many infeasible paths are ruled out by each iteration of the refinement process. Our process is inefficient if only one path is ruled out per iteration. However, because of related empirical evidence in the area of branch prediction that shows that branches in programs are highly correlated with one another [PSR92], we expect many infeasible paths to be eliminated per iteration. Consider $N$ if-then-else statements in a chain. There are $2^N$ potential paths. Suppose that two of the branches, $P$ and $Q$ in the chain are strongly correlated (that is, $P \Leftrightarrow Q$. This means that there can be at most $2^{N-1}$ feasible paths in the chain. By finding a single pair of strongly correlated branches, we have the potential to eliminate a large set of infeasible paths in one iteration.
– *Refinement.* Our definition of refinement ($P \preceq B$) restricts the label sets of $P$ and $B$ to be equivalent. This was natural, as we were concerned with constructing $B$ from $P$ automatically. Given a program $P$, we construct abstractions $B$ such that $P \preceq B$ by construction. It also is natural to ask for two arbitrary boolean programs, $B_1$ and $B_2$, whether or not $B_1$ refines $B_2$, by suitably generalizing the notion of refinement. Given the similarity of this question to the emptiness question of intersection of context-free languages, we conjecture that it is undecidable.

## Acknowledgements

We thank Manuel Fähndrich for many enlightening discussions about the operation of path simulators and for his insights into context-free reachability. Thanks also to the other members of the Software Productivity Tools Group at Microsoft Research for their input, and to Todd Millstein for his detailed comments on this paper.

## References

[AG00] A. Alur and R. Grosu. Modular refinement of hierarchic reactive modules. In *Proceedings of the Twenty Seventh Annual Symposium on Principles of Programming Languages.* ACM Press, 2000.

[AH96] R. Alur and T.A. Henzinger. Reactive modules. In *Proceedings of the 11th Annual Symposium on Logic in Computer Science*, pages 207–218. IEEE Computer Society Press, 1996.

[AHM+98] R. Alur, T.A. Henzinger, F.Y.C. Mang, S. Qadeer, S.K. Rajamani, and S. Tasiran. MOCHA : Modularity in model checking. In A. Hu and M. Vardi, editors, *CAV 98: Computer Aided Verification*, Lecture Notes in Computer Science, pages 521–525. Springer-Verlag, 1998.

[AL88] M. Abadi and L. Lamport. The existence of refinement mappings. In *Proceedings of the 3rd Annual Symposium on Logic in Computer Science*, pages 165–175. IEEE Computer Society Press, 1988.

[BEM97] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *CONCUR'97: Concurrency theory*, volume 1243 of *Lecture Notes in Computer Science (LNCS)*, pages 135–150. Springer-Verlag, 1997.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for the static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth Annual Symposium on Principles of Programming Languages.* ACM Press, 1977.

[CC00] P. Cousot and R. Cousot. Temporal abstract interpretation. In *Proceedings of the Twenty Seventh Annual Symposium on Principles of Programming Languages.* ACM Press, 2000.

[CGJ98] C. Colby, P. Godefroid, and L. J. Jagadeesan. Automatically closing open reactive programs. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI'98)*, pages 345–357, June 1998.

[CL96] C. Colby and P. Lee. Trace-based program analysis. In *Conference Record of the Twenty-Third ACM Symposium on Principles of Programming Languages*, pages 195–207, January 1996.

[DE82] R. B. Dannenberg and G. W. Ernst. Formal program verification using symbolic execution. *IEEE Transactions on Software Engineering*, SE-8(1):43–52, January 1982.

[Dil96] D. L. Dill. The Mur$\phi$ Verification System. In R. Alur and T.A. Henzinger, editors, *CAV 96: Computer Aided Verification*, Lecture Notes in Computer Science 1102, pages 390–393. Springer-Verlag, 1996.

[FWW97] A. Finkel, B. Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. In *INFINITY' 97: Verification of Infinite-state Systems*, July 1997.

[Gri81] D. Gries. *The Science of Programming*. Springer-Verlag, New York, 1981.

[HD99] J. Hatcliff and M. Dwyer. Slicing software for model construction. In *Proceedings of the 1999 ACM Workshop of Partial Evaluation and Program Manipulation (BRICS Notes Series NS-99-1)*, January 1999.

[HHK96] R.H. Hardin, Z. Har'El, and R.P. Kurshan. COSPAN. In R. Alur and T.A. Henzinger, editors, *CAV 96: Computer Aided Verification*, Lecture Notes in Computer Science 1102, pages 423–427. Springer-Verlag, 1996.

[HP96] G.J. Holzmann and D.A. Peled. The State of SPIN. In R. Alur and T.A. Henzinger, editors, *CAV 96: Computer Aided Verification*, Lecture Notes in Computer Science 1102, pages 385–389. Springer-Verlag, 1996.

[HQR98] T.A. Henzinger, S. Qadeer, and S.K. Rajamani. You assume, we guarantee: methodology and case studies. In A. Hu and M. Vardi, editors, *CAV 98: Computer Aided Verification*, Lecture Notes in Computer Science, pages 440–451. Springer-Verlag, 1998.

[KS92] J. Knoop and B. Steffen. The interprocedural coincidence theorem. In P. Pfahler U. Kastens, editor, *Proceedings of the 4th International Conference on Compiler Construction (CC'92), Paderborn (Germany)*, volume 641 of *Lecture Notes in Computer Science (LNCS)*, pages 125–140, Heidelberg, Germany, 1992. Springer-Verlag.

[Kur94] R.P. Kurshan. *Computer-aided Verification of Coordinating Processes.* Princeton University Press, 1994.

[McM93] K.L. McMillan. *Symbolic Model Checking: An Approach to the State-Explosion Problem.* Kluwer Academic Publishers, 1993.

[McM97] K.L. McMillan. A compositional rule for hardware design refinement. In O. Grumberg, editor, *CAV 97: Computer-Aided Verification*, Lecture Notes in Computer Science 1254, pages 24–35. Springer-Verlag, 1997.

[Mil71] R. Milner. An algebraic definition of simulation between programs. In *Proceedings of the 2nd International Joint Conference on Artificial Intelligence*, pages 481–489. The British Computer Society, 1971.

[PSR92] S-T. Pan, K. So, and J. T. Rahmeh. Improving the accuracy of dynamic branch prediction using branch correlation. *ACM SIGPLAN Notices*, 27(9):76–84, September 1992. Proceedings of the 5th International Conference on Architectural Support for Programmming Languages and Operating Systems.

[RHS95] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages*, pages 49–61, January 1995.

[SB92] B. Steffen and O. Burkart. Model checking for context-free processes. In *CONCUR'92, Stony Brook (NY)*, volume 630 of *Lecture Notes in Computer Science (LNCS)*, pages 123–137, Heidelberg, Germany, 1992. Springer-Verlag.

[Sch98] D.A. Schmidt. Data flow analysis is model checking of abstract interpretation. In *Proceedings of the Twenty Fifth Annual Symposium on Principles of Programming Languages*, pages 38–48. ACM Press, 1998.

[SP81] M. Sharir and A. Pnueli. Two approaches to interprocedural data dalow analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–233. Prentice-Hall, 1981.

[SRW99] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Proceedings of the 26th ACM SIGPLAN-SIGACT on Principles of Programming Languages*, pages 105–118, January 1999.

[Wei82] M. Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, July 1982.