A SAT characterization of boolean-program correctness

K. Rustan M. Leino
Microsoft Research, One Microsoft Way, Redmond, WA 98052, USA
leino@microsoft.com

Abstract. Boolean programs, imperative programs where all variables have type boolean, have been used effectively as abstractions of device drivers (in Ball and Rajamani's SLAM project). To find errors in these boolean programs, SLAM uses a model checker based on binary decision diagrams (BDDs). As an alternative checking method, this paper defines the semantics of boolean programs by weakest solutions of recursive weakest-precondition equations. These equations are then translated into a satisfiability (SAT) problem. The method uses both BDDs and SAT solving, and it allows an on-the-fly trade-off between symbolic and explicit-state representation of the program's initial state.

0 Introduction

Boolean programs are imperative computer programs where all variables have type boolean. They have been found to be useful abstractions of low-level systems software. In particular, boolean programs have been used as abstractions of device drivers, as used in Ball and Rajamani's SLAM project [2]. In this paper, I show how to translate a boolean program into a logical formula that (a) is satisfiable if and only if there is a path from the beginning of the program to an error and (b) can be used as input to a satisfiability (SAT) solver.

The translation starts with the semantics of the boolean program, which I define as the weakest solution to a set of weakest-precondition equations. The main challenge is then to transform the second-order fixpoint characterization into a first-order propositional formula. I present an algorithm for doing so. The algorithm is founded on three mathematical ingredients, which I explain.

The translation into a SAT formula uses binary decision diagrams (BDDs) [5]. A simple mathematical equality allows the symbolic representation of the program's initial values to be changed, on the fly, into a representation that models these boolean values explicitly. Because of this explicit representation, the BDDs used can be made arbitrarily simple.

The paper proceeds as follows. Sec. 1 defines boolean programs and their weakest-precondition semantics. Sec. 2 presents the three mathematical ingredients on which the translation is based. Sec. 3 then presents the translation algorithm. Sec. 4 discusses the complexity of the algorithm, brings out its symbolic-versus-explicit-state trade-offs, and reports on some preliminary experiments. The paper then wraps up with some related work and a conclusion.

1 Boolean programs and their semantics

1.0 Syntax

A boolean program has the form given in Figure 0 (*cf.* [1]). A program consists of a set of variables and a sequence of named blocks. Each block consists of a sequence of statements followed by a **goto** statement that declares a set of successor blocks (the absence of a **goto** statement indicates the empty set of successor blocks). Program execution begins in

$$
\begin{array}{rcl}
\mathit{Prog} & ::= & \mathbf{var}\ \mathit{Id}^*\ ;\ \mathit{Block}^+ \\
\mathit{Block} & ::= & \mathit{LabelId}\!:\ \mathit{Stmt}^*\ [\ \mathbf{goto}\ \mathit{LabelList}\ ;\ ] \\
\mathit{Stmt} & ::= & \mathit{Id}^+ := \mathit{Expr}^+\ ;\ \mid\ \mathbf{assume}\ \mathit{Expr}\ ;\ \mid\ \mathbf{assert}\ \mathit{Expr}\ ; \\
\mathit{LabelList} & ::= & \mathit{LabelId}\ \mid\ \mathit{LabelList}\ \mathbf{or}\ \mathit{LabelId} \\
\mathit{Expr} & ::= & \mathbf{false}\ \mid\ \mathbf{true}\ \mid\ \mathit{Id}\ \mid\ \neg \mathit{Expr}\ \mid\ \mathit{Expr} \vee \mathit{Expr}\ \mid\ \mathit{Expr} \wedge \mathit{Expr}
\end{array}
$$

Fig. 0. Grammar of boolean programs

the first given block. At the end of a block, a successor block is picked arbitrarily (nondeterministically) and execution continues there. If the set of successor blocks is empty, execution terminates.

All variables have type boolean (hence the name *boolean program*). In addition to assignment statements, there are $\mathbf{assume}$ and $\mathbf{assert}$ statements. Executing these statements when their conditions evaluate to $\mathbf{true}$ is equivalent to a no-op. Executing an $\mathbf{assume}$ statement when its condition evaluates to $\mathbf{false}$ causes the execution to terminate successfully. Executing an $\mathbf{assert}$ statement when its condition evalutes to $\mathbf{false}$ causes the execution to *go wrong*, an undesirable behavior. A program is erroneous if an execution of it can go wrong; otherwise, it is correct.

For the purpose of checking the program for errors, this simple language is sufficiently expressive to encode common conditional and iterative statements. For example, an if statement $\mathbf{if}\ E\ \mathbf{then}\ S\ \mathbf{else}\ T\ \mathbf{end}$ is encoded as:

$$
\begin{array}{llll}
0: & \mathbf{goto}\ 1\ \mathbf{or}\ 2\ ; & 2: & \mathbf{assume}\ \neg E\ ;\ T\ ;\ \mathbf{goto}\ 3\ ; \\
1: & \mathbf{assume}\ E\ ;\ S\ ;\ \mathbf{goto}\ 3\ ; & 3: &
\end{array}
$$

and a loop $\mathbf{while}\ E\ \mathbf{do}\ S\ \mathbf{end}$ is encoded as:

$$
\begin{array}{llll}
0: & \mathbf{goto}\ 1\ \mathbf{or}\ 2\ ; & 2: & \mathbf{assume}\ \neg E\ ;\ \mathbf{goto}\ 3\ ; \\
1: & \mathbf{assume}\ E\ ;\ S\ ;\ \mathbf{goto}\ 0\ ; & 3: &
\end{array}
$$

Here is a simple example program, which I use as a running example throughout the paper:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathbf{var}\ x, y\ ;} \\
A\!: & x := \mathbf{true}\ ;\ \mathbf{goto}\ B\ ; \\
B\!: & \mathbf{assert}\ x\ ;\ x := x \wedge y\ ;\ \mathbf{goto}\ B\ \mathbf{or}\ C\ ; \\
C\!: &
\end{array}
\tag{0}
$$

The program contains an error, because if y is initially $\mathbf{false}$, then x will be $\mathbf{false}$ in a second iteration of block B, causing the $\mathbf{assert}$ statement to go wrong.

1.1 Semantics

I define the semantics of a boolean program in terms of *weakest preconditions* [6]. For any sequence of statements S and postcondition Q (that is, Q characterizes some set of post-states of S), $wp(S, Q)$ characterizes those pre-states from which execution of S is guaranteed:

– not to go wrong, and

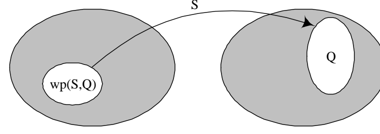

Fig. 1. $wp(S, Q)$ denotes the largest set of pre-states with the following property: starting from any state in $wp(S, Q)$, execution of S does not go wrong, and, if the execution terminates, it does so in a state satisfying Q.

– either the execution does not terminate or it terminates in a state satisfying Q.[0]

Pictorially (see Figure 1), $wp(S, Q)$ is the largest set of pre-states from which execution of S never goes wrong and terminates only in Q.

The weakest preconditions of statement sequences are defined as follows, for any expression E, statement sequences S and T, and postcondition Q:

$$\begin{array}{rcl} wp(\epsilon, Q) & = & Q \\ wp(S\ T, Q) & = & wp(S, wp(T, Q)) \\ wp(x := E;, Q) & = & Q[x := E] \\ wp(\textbf{assume}\ E;, Q) & = & E \Rightarrow Q \\ wp(\textbf{assert}\ E;, Q) & = & E \wedge Q \end{array} \tag{1}$$

where ϵ denotes the empty statement sequence, $S\ T$ is the sequence S followed by sequence T, and $Q[x := E]$ denotes the capture-free substitution of E for x in Q.

The semantics of a block is defined by the weakest precondition of the block statements with respect to the semantics of the block's successor blocks. To formalize the definition, I introduce for each block a boolean function on the block's pre-state, a function whose name I take to be the same as the name of the block. The function represents the semantics of the block. Formally, for any block B, the equation defining the function B is, for any w:

$$B(w) \;=\; wp(S, \bigwedge_{G \in \mathcal{G}} G(w)) \tag{2}$$

where w denotes the program variables, S is the sequence of statements of block B, and $\mathcal{G}$ is the set of successor blocks of B. That is, $B(w)$ is defined as $wp(S, Q)$ where Q is the conjunction of $G(w)$ for every block G in $\mathcal{G}$. Thus, for any pre-state w of B (that is, for any value of the program variables w), $B(w)$ is $\textbf{true}$ if and only if execution of the program beginning at block B from state w is guaranteed not to go wrong.

Example program (0) gives rise to three functions, A, B, C, defined as follows, for any x, y:

$$\begin{array}{rcl} A(x, y) & = & wp(x := \textbf{true},\ B(x, y)) \\ B(x, y) & = & wp(\textbf{assert}\ x\ ;\ x := x \wedge y,\ B(x, y) \wedge C(x, y)) \\ C(x, y) & = & wp(\epsilon,\ \textbf{true}) \end{array}$$

which, after expanding the wp and making the quantification over x, y explicit, gives:

$$\begin{array}{l} (\forall\, x, y :: A(x, y) \;=\; B(\textbf{true}, y)\ ) \\ (\forall\, x, y :: B(x, y) \;=\; x \wedge B(x \wedge y, y) \wedge C(x \wedge y, y)\ ) \\ (\forall\, x, y :: C(x, y) \;=\; \textbf{true}\ ) \end{array} \tag{3}$$

[0] Unlike Dijkstra's original definition of weakest preconditions, I ignore non-terminating executions. That is, as long as an execution does not go wrong, I consider it correct, even if it goes on forever.

The set of equations formed by the definitions of the boolean functions may not have a unique solution, a fact that arises from the presence of loops in boolean programs. For example, the following two closed-form definitions of A, B, C both satisfy equations (3):

$$\begin{array}{ll} (\forall\, x, y \,::\, A(x,y) = \mathbf{false}\,) & (\forall\, x, y \,::\, A(x,y) = y\,) \\ (\forall\, x, y \,::\, B(x,y) = \mathbf{false}\,) & (\forall\, x, y \,::\, B(x,y) = x \wedge y\,) \\ (\forall\, x, y \,::\, C(x,y) = \mathbf{true}\,) & (\forall\, x, y \,::\, C(x,y) = \mathbf{true}\,) \end{array} \qquad (4)$$

To be precise about the semantics equations, let us indicate which variables are the unknowns, which I shall do by writing them in front of the equations. For the example, the unknowns are A, B, C, so from now on I write (3) as:

$$\begin{array}{lrcl} A, B, C: & (\forall\, x, y \,::\, A(x,y) & = & B(\mathbf{true}, y)\,) \\ & (\forall\, x, y \,::\, B(x,y) & = & x \wedge B(x \wedge y, y) \wedge C(x \wedge y, y)\,) \\ & (\forall\, x, y \,::\, C(x,y) & = & \mathbf{true}\,) \end{array} \qquad (5)$$

Of the various solutions to the equations we have set up, which solution do we want? The weakest solution. That is, the largest solution. That is, the solution where the unknowns are functions that return $\mathbf{true}$ as often as possible. The reason for this is that we want these weakest-precondition equations to characterize *all* the pre-states from which executions do not go wrong, that is, the *largest* set of pre-states from which executions do not go wrong.

To illustrate, consider the second line of the left-hand solution in (4). This line says that *if* the program is started from block B in a state satisfying $\mathbf{false}$,[1] then any resulting execution will be correct. However, since this is not the *weakest* solution, the line says nothing about starting executions from states *not* satisfying $\mathbf{false}$. The second line of the right-hand solution in (4) says that if the program is started from block B in a state satisfying $x \wedge y$, then any resulting execution will be correct. Since this *is* the weakest solution, we also have that if the program is started from B in a state *not* satisfying $x \wedge y$, then there exists an execution that goes wrong.

In summary, the semantics of a boolean program is the weakest solution to the set of equations prescribed, for every block B in the program, by (2).

Weakest solution versus weakest fixpoint. Before going on, let me take the opportunity to clear up a possible confusion, namely the issue of weakest *solution* versus weakest *fixpoint*. The distinction between the two is analogous to the distinction between the result of a function and the function itself. The weakest solution (in the unknown a) to: $a : a = f(a)$ is the weakest fixpoint of f. Likewise, the weakest solution to a set of equations:

$$a, b, c: \quad a = f(a,b,c) \qquad b = g(a,b,c) \qquad c = h(a,b,c)$$

is the weakest fixpoint of f, g, h.[2]

To view the semantics of a boolean program as a weakest fixpoint, we formulate equations (5) equivalently as:

$$\begin{array}{lrcl} A, B, C: & A & = & \mathcal{F}(A,B,C) \\ & B & = & \mathcal{G}(A,B,C) \\ & C & = & \mathcal{H}(A,B,C) \end{array} \qquad (6)$$

[1] An impossibility, but never mind that.

[2] That is, the fixpoint of the tuple function $[f, g, h]$, defined for any 3-tuple w as $[f,g,h](w) = \langle f(w), g(w), h(w)\rangle$, with the partial order of tuples defined pointwise as $\langle a,b,c\rangle \sqsubseteq \langle a',b',c'\rangle \;\equiv\; a \le a' \wedge b \le b' \wedge c \le c'$.

where $\mathcal{F}, \mathcal{G}, \mathcal{H}$ are defined as follows, for any functions A, B, C :

$$\begin{array}{rcl} \mathcal{F}(A,B,C) & = & (\lambda\, x, y :: B(\mathbf{true}, y)\,) \\ \mathcal{G}(A,B,C) & = & (\lambda\, x, y :: x \wedge B(x \wedge y, y) \wedge C(x \wedge y, y)\,) \\ \mathcal{H}(A,B,C) & = & (\lambda\, x, y :: \mathbf{true}\,) \end{array}$$

That is, $\mathcal{F}$, $\mathcal{G}$, and $\mathcal{H}$ are defined to take three functions (here called A, B, C) as arguments and to return one function (here denoted by λ -expressions). Since we are now dealing with functions that map functions to functions, it may sometimes be confusing as to which kind of function I'm referring to; therefore, I shall refer to $\mathcal{F}, \mathcal{G}, \mathcal{H}$ as *functional transformations* when doing so provides clarification. Under formulation (6), the semantics of the boolean program is the weakest fixpoint of $\mathcal{F}, \mathcal{G}, \mathcal{H}$.

According to a well-known theorem by Knaster and Tarski (see, *e.g.*, [7]), a unique weakest fixpoint to the functional transformations $\mathcal{F}, \mathcal{G}, \mathcal{H}$ exists (equivalently, a unique weakest solution to (5) exists), provided $\mathcal{F}, \mathcal{G}, \mathcal{H}$ are monotonic. To see that the functional transformations are indeed monotonic, we just need to remember that they come from the right-hand side of (2). In particular, in the right-hand side of (2), the arguments to the functional transformations (that is, the functions G) occur conjoined in the second argument of the wp . And indeed, wp is monotonic in its second argument (see definition (1) or [6]).[3]

Program correctness. Given the semantics of a boolean program, as defined above, the program is correct if and only if $(\forall\, w :: start(w)\,)$ is a valid formula, where $start$ is the program's start block and w denotes the program's variables. That is, the program is correct if, for every initial state w , its executions are guaranteed not to go wrong. Conversely, the program contains an error if and only if $(\exists\, w :: \neg start(w)\,)$, that is, if $\neg start(w)$ is satisfiable.

Applying SAT. Now that we have defined the semantics and correctness criterion of boolean programs, we may get the feeling that the problem of checking the program for correctness can be performed by a SAT solver. This is an enticing prospect, given the recent impressive progress in producing efficient SAT solvers, for example like GRASP [8] and Chaff [9]. Indeed, the equations that define the program semantics are boolean formulas, and the existence of an error comes down to the satisfiability of a boolean formula. However, the problem also involves components not addressed by SAT solvers, namely the fact that the semantics is defined in terms of the weakest solution to the set of equations and the fact that the equations involve functions.

In the next section, I go through the mathematical ingredients that let us overcome these roadblocks to applying SAT techniques.

2 Three mathematical ingredients

2.0 Equations over a closed set of function terms

The first mathematical ingredient, computing a set of equations over a closed set of function terms, allows us to eliminate the universal quantifications in the semantics equations.

[3] Note that the existence of a weakest fixpoint requires the *functional transformations* ($\mathcal{F}, \mathcal{G}, \mathcal{H}$) to be monotonic. In particular, the existence of the weakest fixpoint does not rely on the *arguments* of the functional transformations (the functions A, B, C) to be monotonic. That's good, because the functions A, B, C are generally not monotonic.

To eliminate the quantifi cations in a set of equations like:

$$\begin{array}{llcl} A, B, C: & (\forall\, x, y :: A(x, y) & = & \ldots) \\ & (\forall\, x, y :: B(x, y) & = & \ldots) \\ & (\forall\, x, y :: C(x, y) & = & \ldots) \end{array}$$

we can form a new set of equations by appropriate instantiations of the universal quantifi cations. In general, we need to instantiate the quantifi cations with all possible values, but if we are interested in evaluating only some particular expression, then we may be able to do with fewer instantiations. Let's do only as many instantiations as required for every mentioned function instantiation, or *function term* as I shall call them, to appear as the left-hand side of some equation. That is, let's instantiate the equations until we get a set of quantifi er-free equations over a closed set of function terms.

For example, suppose for (5) that we have an interest in the expression $\neg A(k, m)$, as indeed we do if k and m denote initial values of the program variables x and y. Then, we instantiate the fi rst quantifi cation of (5), the one that defi nes A, with $x, y := k, m$, producing $A(k, m) \;=\; B(\mathbf{true}, m)$. This equation mentions two function terms, $A(k, m)$ and $B(\mathbf{true}, m)$, but the second of these does not appear as a left-hand side. Therefore, we continue the closure computation and instantiate the second quantifi cation of (5) with $x, y := \mathbf{true}, m$:

$$B(\mathbf{true}, m) \;=\; \mathbf{true} \;\wedge\; B(\mathbf{true} \wedge m, m) \;\wedge\; C(\mathbf{true} \wedge m, m)$$

This in turn requires two more instantiations:

$$\begin{array}{rcl} B(\mathbf{true} \wedge m, m) & = & \mathbf{true} \wedge m \wedge B(\mathbf{true} \wedge m \wedge m, m) \wedge C(\mathbf{true} \wedge m \wedge m, m) \\ C(\mathbf{true} \wedge m, m) & = & \mathbf{true} \end{array}$$

Observe that the function terms appearing in the right-hand sides of these equations also appear as left-hand sides of the equations we've produced so far. This observation hinges on the fact that the function arguments of the new right-hand function terms for B and C, namely $\mathbf{true} \;\wedge\; m \;\wedge\; m$ and m, are the same as the arguments $\mathbf{true} \;\wedge\; m$ and m, respectively, for which we already have instantiations of B and C. Thus, we have now reached a closed set of function terms. Summarizing the example, if we are interested in the solution to (5) only to evalute $\neg A(k, m)$, then we can equivalently consider the solution to the following quantifi er-free equations:

$$\begin{array}{lrcl} A, B, C: & A(k, m) & = & B(\mathbf{true}, m) \\ & B(\mathbf{true}, m) & = & B(m, m) \;\wedge\; C(m, m) \\ & B(m, m) & = & m \;\wedge\; B(m, m) \;\wedge\; C(m, m) \\ & C(m, m) & = & \mathbf{true} \end{array} \tag{7}$$

where I have allowed myself to simplify the boolean expressions involved. In particular, the value of $\neg A(k, m)$ according to the weakest solution of (5) is the same as the value of $\neg A(k, m)$ according to the weakest solution of (7).

In general, there may be more solutions to the computed quantifi er-free set of equations than to the original set of equations. For example, (7) leaves $B(\mathbf{false}, \mathbf{true})$ unconstrained, whereas (5) constrains it to be $\mathbf{false}$ (see (4)). However, the function term $B(\mathbf{false}, \mathbf{true})$ is apparently irrelevant to the value of $\neg A(k, m)$, which we took to be the only expression of interest.

It is important to note that, unlike the bound x and y in (5), the occurrences of k and m in (7) are *not* bound. Rather, k and m are free variables in (7).

Another thing to note is that we could have terminated the closure computation above earlier if we had had particular boolean values for k and m. For example, if m had the value $\mathtt{true}$, then the function term $B(m, m)$ would be the same as $B(\mathbf{true}, m)$, so we could have stopped the closure computation earlier. But if we are interested in representing k and m symbolically, like we did in this example so they can represent *any* initial program state, then we treat these function terms as potentially being different.

To actually perform this closure computation mechanically, we need to be able to compare two function terms for (symbolic) equality. This can be done by comparing (the names of the functions and) the arguments, which in turn can be done using BDDs [5], because BDDs provide a canonical representation of boolean expressions. The closure computation does terminate, because the number of function terms to be defined is finite: there is one function name for each block, and each function argument is a purely propositional combination of the variables k and m representing the initial program state (function-term arguments never contain nested function terms, see (2) and (1)). Sec. 4 takes a closer look at the actual complexity.

2.1 Point functions

The second mathematical ingredient, viewing a function as the union of a collection of point functions, allows us to eliminate functions from the semantics equations.

Any function $f\colon W \to U$ can be viewed as a union of $|W|$ nullary functions that I shall refer to as *point functions*. For every value w in the domain of f, we define a point function $f_w\colon () \to U$ by $f_w() = f(w)$. In principle, we can think of f as being defined by a table with two columns with elements from W and U, respectively. Each row of the table gives the value of f (shown in the right column) for a particular argument (shown in the left column). Each row then corresponds to what I'm calling a point function. By the way, being nullary functions, point functions are essentially just variables, so let's just drop the parentheses after them.

Symbolically, a recursive function (or a set of recursive functions) can be defined as the weakest (say) solution of a set of equations; equivalently, the function (or set of functions) can be defined by its (their) point functions, which in turn are defined as the weakest solution to a set of equations where the point functions are expressed in terms of each other. For example, if f is a function on the booleans, then the weakest solution to:

$$f: \quad (\forall\, w :: f(w) \;=\; f(\mathbf{false}) \lor f(w)\,) \tag{8}$$

can equally well be expressed as the weakest solution to:

$$f_{\mathbf{false}}, f_{\mathbf{true}}: \qquad f_{\mathbf{false}} = f_{\mathbf{false}} \lor f_{\mathbf{false}} \qquad f_{\mathbf{true}} = f_{\mathbf{false}} \lor f_{\mathbf{true}}$$

Note that these equations have two unknowns, $f_{\mathbf{false}}$ and $f_{\mathbf{true}}$, whereas (8) has only one unknown, f.

The set of equations that we arrived at in the previous subsection (the ones over a closed set of function terms) can be viewed as constraining a set of point functions. That is, we can think of each function term as being a point function. Viewing function terms as point

functions, the equations (7) take the form:

$$A_{k,m}, B_{\mathbf{true},m}, B_{m,m}, C_{m,m}: \quad \begin{array}{rcl} A_{k,m} & = & B_{\mathbf{true},m} \\ B_{\mathbf{true},m} & = & B_{m,m} \wedge C_{m,m} \\ B_{m,m} & = & m \wedge B_{m,m} \wedge C_{m,m} \\ C_{m,m} & = & \mathbf{true} \end{array} \tag{9}$$

Before leaving the topic of point functions, there's another issue worth analyzing. Because we allow variables like k and m in the instantiations of the functions, the conversion into point functions may produce several variables for what is really the same point function. For example, equations (7) contain the function terms $B(\mathbf{true}, m)$ and $B(m, m)$. From the fact that B is a function, we immediately know that if m happens to have the value $\mathbf{true}$, then these two function terms evaluate to the same value. But by representing the two functions terms as two separate variables, the point functions $B_{\mathbf{true},m}$ and $B_{m,m}$, we no longer have the guarantee that $B_{\mathbf{true},m}$ and $B_{m,m}$ are equal if m and $\mathbf{true}$ are. Indeed, because we may have introduced several point functions for a function term, the conversion from function terms to point functions may have enlarged the set of solutions to the equations.[4] Luckily, we are interested only in the *weakest* solution, which has not changed. Here's why it has not changed: The quantifi er-free equations are produced as instantiations of *one* set of semantics equations. Therefore, if the function for a block B is instantiated in more than one way, say like $B(w')$ and $B(w'')$, then the quantifi er-free equations will constrain $B(w')$ and $B(w'')$ in the same way for the case where w' and w'' have the same value. Because the function terms $B(w')$ and $B(w'')$ have the same constraints, the corresponding point functions $B_{w'}$ and $B_{w''}$ also have the same constraints (again, in the case where w' and w'' have the same value). And since the point functions $B_{w'}$ and $B_{w''}$ have the same constraints, the set of solutions for each of these point functions is the same, and, in particular, the weakest solution of each is then the same.

2.2 Inward computation of fixpoints

The third mathematical ingredient, the inward compution of a fi xpoint expression, allows us to eliminate all but the weakest of the solutions to the set of equations.

Let's review the standard way of computing the weakest fi xpoint of a function. If a function F on a complete lattice is *meet-continuous* —that is, if it has no infi nite descending chains, or, equivalently, if it distributes meets over any nonempty, linear set of elements—

[4] Suppose a boolean program with one variable and start block L gives rise to the following set of quantifier-free equations:

$$J, L, M: \quad \begin{array}{rcl} J(k) & = & L(k) \wedge M(k) \\ M(k) & = & k \wedge M(k) \end{array} \qquad \begin{array}{rcl} L(k) & = & M(\mathbf{true}) \\ M(\mathbf{true}) & = & M(\mathbf{true}) \end{array}$$

Then the corresponding point-function equations:

$$J_k, L_k, M_k, M_{\mathbf{true}}: \quad \begin{array}{rcl} J_k & = & L_k \wedge M_k \\ M_k & = & k \wedge M_k \end{array} \qquad \begin{array}{rcl} L_k & = & M_{\mathbf{true}} \\ M_{\mathbf{true}} & = & M_{\mathbf{true}} \end{array}$$

admit the solution $J_k = M_k = \mathbf{false}$, $L_k = M_{\mathbf{true}} = \mathbf{true}$ regardless of the value of k. (The letter k occurs in the subscripts of the point functions, but these occurrences do not denote the variable k. Rather, the subscripts are just part of the names of the four point-function variables.) In constrast, the *function-term* equations admit this solution *only* if k does not have the value $\mathbf{true}$.

then the weakest fixpoint of F is $F^d(\top)$ for some sufficiently large natural number d, where $\top$ is the top element of the lattice. Stated differently, the weakest fixpoint can be computed by iteratively applying F, starting from $\top$. The number d, which I shall call the *fixpoint depth* of F, is bounded from above by the *height* of the lattice, that is, the maximum number of strictly-decreasing steps required to get from $\top$ to $\bot$, the bottom element of the lattice. In general, the height of a lattice may be infinite. But *if* the lattice itself is finite, then the height is also finite. For a finite lattice, any monotonic function is also meet-continuous.[5] For the particular case of the lattice of booleans, $(\{\mathbf{false}, \mathbf{true}\}, \Rightarrow)$, the height is 1, and so for any monotonic function F on the booleans, $F(\mathbf{true})$ always gives the weakest fixpoint of F.

There are two basic ways to arrive at the weakest fixpoint of a meet-continuous function:

outward by starting at $\top$ and iteratively applying the function, either until the current value is unaffected by another application of the function, at which point the fixpoint has been detected, or until the number of times the function has been applied equals the fixpoint depth; and

inward by recursively invoking the function until the fixpoint depth is reached, at which point any further recursive invocations are replaced by $\top$.

To illustrate the two ways, suppose (an upper bound on) the fixpoint depth is known and that we're happy to apply the function that many times without checking to see if we have reached the fixpoint. In particular, suppose we want to compute the weakest fixpoint of a meet-continuous function F with a fixpoint depth of 2. Using the outward way, our computation of an expression for the fixpoint goes like this:

$$\begin{array}{ll} & \top \\ \leadsto & \quad \{ \text{ apply } F \text{ to } \top \text{ (completing 1 application) } \} \\ & F(\top) \\ \leadsto & \quad \{ \text{ apply } F \text{ to } F(\top) \text{ (completing 2 applications) } \} \\ & F(F(\top)) \end{array}$$

Using the inward way, our computation of an expression for the fixpoint goes like this, where $\star$ is a placeholder for an subexpression to be produced:

$$\begin{array}{ll} & \star \\ \leadsto & \quad \{ \text{ replace } \star \text{ by } F(\star) \text{ (completing 1 nested invocation) } \} \\ & F(\star) \\ \leadsto & \quad \{ \text{ replace } \star \text{ by } F(\star) \text{ (completing 2 nested invocations) } \} \\ & F(F(\star)) \\ \leadsto & \quad \{ \text{ replace } \star \text{ by } \top\text{, since } \star \text{ is nested within 2 invocations of } F \ \} \\ & F(F(\top)) \end{array}$$

Note that the outward way produces $\top$ first and the outermost F last, whereas the inward way produces the outermost F first and $\top$ last. Both the outward way and the inward way arrive at the same expression for the weakest fixpoint of F.

To compute the weakest fixpoint of a *set* of functions using the inward way, it suffices to keep track of the nesting level of each function, I conjecture. For example, consider a set of equations like:

$$a, b : \qquad a \;=\; F(a, b) \qquad b \;=\; G(a, b) \tag{10}$$

[5] because monotonicity is equivalent to the property of distributing meets over any nonempty, linear, *finite* set of elements [7]

where F and G are meet-continuous functions of their arguments, where F for a fixed second argument has a fixpoint depth of 2, and where G for a fixed first argument has a fixpoint depth of 1. The weakest solution of a is computed as follows, where the two-digit subscripts of a and b indicate the nesting level within F's and G's, respectively:

$$
\begin{array}{ll}
 & a_{00} \\
\rightsquigarrow & \quad \{ \quad a = F(a,b) \quad \} \\
 & F(a_{10}, b_{10}) \\
\rightsquigarrow & \quad \{ \quad a = F(a,b) \text{ and } b = G(a,b) \quad \} \\
 & F(F(a_{20}, b_{20}), G(a_{11}, b_{11})) \\
\rightsquigarrow & \quad \{ \text{ replace } a_{20} \text{ and } b_{11} \text{ with } \top \text{, and apply } a = F(a,b) \text{ to } a_{11} \\
 & \qquad \text{and } b = G(a,b) \text{ to } b_{20} \quad \} \\
 & F(F(\top, G(a_{21}, b_{21})), G(F(a_{21}, b_{21}), \top)) \\
\rightsquigarrow & \quad \{ \text{ replace } a_{21} \text{ and } b_{21} \text{ with } \top \quad \} \\
 & F(F(\top, G(\top,\top)), G(F(\top,\top),\top))
\end{array}
\tag{11}
$$

Back to boolean programs and our example. The weakest solution to the set of equations (9) is the weakest fixpoint of the functional transformations implicit in the equations. These functional transformations return point functions, which in our application are nullary boolean functions, that is, boolean variables. The result of each functional transformation is therefore a boolean, so each functional transformation, restricted to a single varying argument, has a fixpoint depth of at most 1. According to our third mathematical ingredient, we can use the inward way to produce a set of equations whose one and only solution is the weakest solution to (9). Note that this final set of equations consists only of boolean variables and boolean operators; there are no quantifiers, no functions, and no weakest solutions to worry about.

We are now ready to write the algorithm for doing what we've done in the running example.

3 Algorithm

The algorithm, which is based on the three mathematical ingredients, computes a set of equations, the conjunction of which is satisfiable if and only if the program contains an error. For any block b and arguments u for function b, I write the pair $\langle b, u \rangle$ to represent the mathematical expression $b(u)$, that is, function b applied to the arguments u. In order to check when the fixpoint depth has been reached, the context of an occurrence of such a pair is taken into account, and different occurrences of the same pair may need to be represented by differently named point functions.[6] Therefore, the algorithm labels each pair $\langle b, u \rangle$ with a discriminating value, say s, and I write the labeled pair as a triple $\langle b, u, s \rangle$. The equations computed by the algorithm have these triples as their propositional variables.[7]

The algorithm is shown in Figure 2. The algorithm outputs the set Eqs, given: a program with start block $start$, (symbolic) initial values w of the program's variables, and a procedure $Instantiate$ that for any program block b and argument list u returns the

[6] This is a way in which the algorithm differs from ordinary forward-reachability model checking algorithms.

[7] Depending on the SAT solver used to check the satisfiability of the resulting equations, the equations may first need to converted into conjunctive normal form.

$$
\begin{array}{l}
cnt := 0\,; \quad t, cnt := \langle start, w, cnt \rangle, cnt+1\,; \quad Eqs, TBD := \{\neg t\}, \{\langle t, \epsilon \rangle\}\,; \\
\textbf{while } TBD \neq \emptyset \textbf{ do} \\
\quad \textbf{pick } b, u, s, path \textbf{ such that } \langle \langle b, u, s \rangle, path \rangle \in TBD\,; \\
\quad TBD := TBD \setminus \{\langle \langle b, u, s \rangle, path \rangle\}\,; \\
\quad \textbf{if } \langle b, u \rangle \in path \textbf{ then} \quad Eqs := Eqs \cup \{\langle b, u, s \rangle = \textbf{true}\}\,; \\
\quad \textbf{else} \quad rhs := Instantiate(b, u)\,; \\
\qquad \textbf{foreach } \langle c, v \rangle \in rhs \textbf{ do} \\
\qquad\quad t, cnt := \langle c, v, cnt \rangle, cnt+1\,; \\
\qquad\quad \textbf{replace } \langle c, v \rangle \textbf{ with } t \textbf{ in } rhs\,; \\
\qquad\quad TBD := TBD \cup \{\langle t, path +\!\!+ \langle b, u \rangle \rangle\}\,; \\
\qquad\quad G := G \cup \{\langle b, u, s \rangle \mapsto t\}\,; \\
\qquad \textbf{end}\,; \\
\qquad Eqs := Eqs \cup \{\langle b, u, s \rangle = rhs\}\,; \\
\textbf{end}\,; \ \textbf{end}\,;
\end{array}
$$

Fig. 2. The algorithm for computing a set of boolean equations Eqs whose conjunction is satisfiable if and only if the boolean program under consideration is correct.

right-hand side of the weakest-precondition equation for b, instantiated with u. For example, for the running example, $Instantiate(B, \langle \mathbf{true}, m \rangle)$ would return the formula $\mathbf{true} \wedge \langle B, \langle \mathbf{true} \wedge m, m \rangle \rangle \wedge \langle C, \langle \mathbf{true} \wedge m, m \rangle \rangle$ simplified to taste.[8] To facilitate the simple generation of discriminating values, the algorithm keeps a counter cnt of the number of different triples produced so far. The algorithm keeps a work list TBD of pairs $\langle t, path \rangle$, where t is a triple that is used in Eqs, but not defined, and where the sequence $path$ of function-term instantiations gives the context in which t is used in Eqs. That is, $path$ corresponds to the subscripts in the example calculation (11).

The first branch of the if statement is taken when the fixpoint depth for $\langle b, u \rangle$ has been reached in the context $path$. In this case, I shall refer to $\langle b, u, s \rangle$ as a *fixpoint triple*. In my *implementation* of the algorithm, I use BDDs for the arguments u when comparing $\langle b, u \rangle$ to other pairs $\langle b, \cdot \rangle$ in $path$. The second branch of the if statement adds a discriminating value to each pair in rhs and adds the resulting triples to the work list. The algorithm terminates when the work list is empty.

Variable G represents the edges in a graph whose vertices are the triples produced so far. An edge $t \mapsto t'$ indicates that the equation for t requires a definition for t'. By producing a satisfying assignment with as few negative variables as possible, one can then produce an execution path from the initial state to the error by restricting this graph to the vertices corresponding to negative variables.[9]

The algorithm in Figure 2 constructs the edges G to be a tree. As an optimization, my implementation of the algorithm shares common function terms between the branches of

[8] It seems prudent to apply constant folding to simplify these expressions, since that's simple to do and may reduce the number of function terms that ever make it onto the work list, causing the algorithm to converge more quickly. In my initial experiments, I found constant folding to make a significant difference in the number of equations generated.

[9] Once my implementation has found the existence of an error in a given program, which takes one call to the SAT solver, it performs many more calls to the SAT solver to determine which variables are forced to be $\mathbf{false}$ in a satisfying assignment. The situation could probably be improved by an incremental SAT solver, perhaps like Satire [0], or by some kind of SAT solver that produces this information more readily, perhaps like one based on Stålmarck's saturation algorithm [11].

this tree, resulting in a DAG. That is to say, before calling $Instantiate(b, u)$, my implementation checks if there already is an equation with a left-hand side $\langle b, u, s' \rangle$, for some s'.[10] If there is (and if the sharing is possible, something that depends on conditions more subtle than I have room to describe here), then the equation $\langle b, u, s \rangle = \langle b, u, s' \rangle$ is added to Eqs, that is, $\langle b, u, s \rangle$ is defined to equal $\langle b, u, s' \rangle$. This reduces the number of triples produced, but may cause some fixpoint triples reachable from $\langle b, u, s' \rangle$ to be used as if they were fixpoint triples reachable from the context of $\langle b, u, s \rangle$. To adjust for this, my implementation visits all fixpoint triples now reachable from $\langle b, u, s \rangle$ and, when necessary, redefines them (by replacing the previous right-hand side of **true** by a proper right-hand instantiation of $\langle b, u \rangle$). To avoid having to actually retract an equation placed into Eqs, my implementation simply marks fixpoint triples as such, adding their definitions to Eqs once the main loop has completed and the fixpoint triples that remain really should be equated to **true**. While it does not improve the time spent by the algorithm, this sharing optimization can reduce the number of generated triples.

An interesting possibility that calls for future investigation is the prospect of passing equations to the SAT solver as soon as they are added to Eqs. An incremental SAT solver, perhaps like Satire [0], would then be used. This may have the advantage that, if the program is correct, unsatisfiability may be detected before the fixpoint depth has been reached everywhere.

My implementation handles a richer language of boolean program than that shown in Figure 0, namely the actual input to the Bebop model checker [1]. The richer features are encoded in the core language in Figure 0. For example, if and while statements are encoded as shown in Sec. 1.0, nondeterministic assignments are encoded as nondeterministic jumps to blocks that perform the different possible assignments, and calls to procedures are encoded by inlining.[11]

4 Complexity

The complexity of the algorithm depends crucially on the number of function terms generated, because each function term (or, more precisely, each function term triple) gives rise to a propositional variable. In a program with K variables, a function term consists of a function symbol followed by K boolean-expression arguments. There are 2^{2^K} different boolean expressions in K variables. Thus, if the program has N blocks and K program variables, there are $N \cdot (2^{2^K})^K$ different function terms. The function-term-sharing implementation of the algorithm can therefore produce a doubly-exponential number of different function terms.

However, in practice, I expect most boolean programs to generate fewer equations than the worst case. The table in Figure 3 shows the number of function terms generated for a collection of boolean programs that were generated by SLAM from device drivers for the Microsoft Windows operating system. For these programs, the growth of the number of function terms does not seem outrageous.

It is too early to get meaningful timed performance results from my implementation.

There is a simple way to change the worst-case number of function terms generated. In what I have shown, the arguments to the negated start-block function, which represent

[10] BDDs are used in determining this, too, in particular in comparing the arguments u.

[11] The inlining assumes the procedures to be non-recursive, which is a realistic assumption in the context of device drivers.

name	# blocks	# variables # global	avg./block	# generated function terms
Driver “A”	3903	1	1.0	2840
Driver “B”	4022	2	2.3	2951
Driver “C”	3925	2	2.0	2860
Driver “D”	4487	2	2.0	3124
Driver “E”	3933	4	4.0	2868
Driver “F”	4519	6	9.2	37365
Driver “G”	4521	5	13.4	4395
Driver “H”	6760	18	39.5	14612
Driver “I”	5429	3	4.3	9744
Driver “J”	8693	1	1.0	7250
Driver “K”	4569	7	20.7	29984

Fig. 3. Measurements from a preliminary investigation of the number of function terms generated for some small test programs. The richer boolean programs that these programs encode contain procedures with parameters and local state, which causes the blocks in the encoding to have varying numbers of variables on entry. The table shows both the number of global variables and the average number of variables per block. The boolean programs can contain unreachable blocks, which explains the fact that the number of function terms is sometimes smaller than the number of blocks.

the program's initial state, are given symbolically (k and m in the running example). If, instead of $\neg A(k,m)$, a complete set of explicit boolean values are used, as in:

$$\neg A(\mathbf{false},\mathbf{false}) \;\wedge\; \neg A(\mathbf{false},\mathbf{true}) \;\wedge\; \neg A(\mathbf{true},\mathbf{false}) \;\wedge\; \neg A(\mathbf{true},\mathbf{true})$$

then all function-term arguments will also be explicit boolean values, so there are only $N \cdot 2^K$ different function terms, a *single* exponential.

There is a good reason we don't want to abandon the symbolic initial values in favor of the explicit ones: by using explicit values, we get *at least* 2^K function terms, because that's how many function terms we get for the start block alone. The numbers in Figure 3 show that the symbolic initial values can do better than that.

Interestingly enough, we can adjust the degree to which we use the two argument representations, by using the following simple equality: for any function b , expression e , and lists of expressions E_0 and E_1 , we have:

$$b(E_0,e,E_1) \;=\; (\neg e \;\wedge\; b(E_0,\mathbf{false},E_1)) \;\wedge\; (e \;\wedge\; b(E_0,\mathbf{true},E_1)) \qquad (12)$$

Thus, if e is an expression other than an explicit boolean value, then the algorithm in Figure 2 can choose to set rhs to the right-hand side of (12) instead of invoking the procedure $Instantiate$. The choice of which one to do would be determined heuristically. A possible heuristic is to use (12) whenever the argument e is 'too complicated", as perhaps when the number of variables in e exceeds some threshold, or when e is anything but the symbolic initial value of the program variable corresponding to this function argument. By choosing the latter heuristic, for example, the number of different function terms is $N \cdot 3^K$, a single exponential as in the case of using only explicit boolean values; yet, by using this heuristic, the algorithm begins with just one negated start block function, not an exponential number of them as in the case of using only explicit boolean values.

I have yet to experiment with the symbolic-versus-explicit argument representations in my implementation.

5 Related work

The method of doing model checking that I have presented is perhaps, in the end, most reminiscent of a standard forward reachability algorithm where the weakest-precondition equation for a block gives the next-state relation of that block. In the present method, each function term corresponds to *one* state (the program counter plus the variables in scope at the beginning of the block), as a function of the *arbitrary* initial state represented by the initial-state variables (k and m in the running example). The present method uses both BDDs (in the phase that determines closure of the equations produced, shown in Figure 2) and SAT solving (in the phase that checks the resulting SAT formula). In particular, the expressions in **assume** and **assert** statements, which determine whether or not execution will proceed normally, are not used in the first phase (except as mentioned in footnote 8). Instead, these asserted and assumed conditions are encoded into the final SAT formula, which puts the responsibility of wrestling with path feasibility onto the SAT solver. Consequently, the boolean expressions that the BDD package gets to see are much simpler; indeed, by applying equation (12), one can make the expressions be arbitrarily simple. The cost of this partitioning of work between BDDs and SAT solver is that the first phase may over-approximate the set of function terms; that is, it may produce some function terms that do not correspond to any reachable state. As I've mentioned, the first phase may also need to produce several copies of a function term, since the decision of when a function-term triple is a fixpoint triple depends on the path leading to the function term (*cf.* footnote 6). In contrast, standard forward reachability algorithms only need to compute a reachable set of states.

Another technique of model checking is *bounded model checking* [3], which seeks to find errors by investigating only prefixes of the program's execution paths. This allows the technique to continue the forward simulation without checking if the next states have already been explored, a check that can lead to large BDDs. Indeed, bounded model checking has been targeted especially for SAT solvers, avoiding BDDs altogether. By iteratively increasing the prefix length used, bounded model checking can, at least in principle, establish the correctness of a given program. The incremental nature of the technique and the efficiency of modern SAT solvers have let bounded model checking find many errors quickly. The present method can also avoid large BDDs for determining reachability, but does so without missing any errors. It is also interesting to note that the incremental version of the present method mentioned in Sec. 3 can, in contrast to bounded model checking, establish *correctness* early, whereas finding the presence of errors requires running the present algorithm to completion.

Other SAT-based model checking techniques include the use of induction, which can be used with bounded model checking to make up for the fact that only prefixes of executions are used [10]. Also, standard algorithms for symbolic reachability algorithms can be performed using SAT solving rather than BDDs (see, *e.g.*, [4]).

6 Conclusion

I have presented a precise method for mechanically checking the correctness of boolean programs. The method makes it possible to construct an execution trace leading to an error, if the program contains one. The method uses both BDDs and SAT solving. The method also permits a simple and adjustable trade-off between symbolic and explicit-state modeling of the initial state, a trade-off that can be made dynamically during the checking process.

The preliminary experiments with my current implementation do not rule out practical use of the method, despite the high worst-case complexity of the algorithm. I'm working toward removing some gross inefficiencies of my implementation, so that I can make some meaningful time measurements. I then hope to compare the method with the SLAM BDD-based model checker Bebop [1].

Future work includes tightening up the mathematical account of the algorithm's correctness and resolving the conjecture in Sec. 2.2. I also wish I had a better encoding for the nondeterministic assignments that SLAM's richer boolean-program notation allows. It would be nice to represent these choices symbolically, but the method I've described uses symbolic values only to represent the program's initial state. Finally, I wish I had better solution for procedures, so that there could be more reuse between different invocations of a procedure. Perhaps one can, in some useful way, generate a summary for each procedure and then use it. Or maybe there's a clever encoding that lets one "save" the local variables at a call site and restore the same values after the call.

Acknowledgements. I'm grateful to Sriram Rajamani for some good and inspiring discussions about this problem. I have personally benefitted, and so has this work, from discussions with Byron Cook about SAT solving, with Jacob Lichtenberg about BDD implementations, and with Ernie Cohen about the complexity of the problem. I'm also grateful to the audiences of a talk I gave on this subject at the Eindhoven Technical University and at the IFIP WG 2.4 meeting at Schloß Dagstuhl, for their comments have helped me improve the presentation of the subject. Byron, Jacob, and Viktor Kuncak provided valuable comments on earlier drafts of this paper.

References

0. Fadi Aloul. Satire. `http://www.eecs.umich.edu/~faloul/Tools/satire/`, 2002.
1. Thomas Ball and Sriram K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN 2000*, pages 113–130, 2000.
2. Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 2001*, pages 103–122, 2001.
3. Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *TACAS '99*, pages 193–207, 1999.
4. Per Bjesse and Koen Claessen. SAT-based verification without state space traversal. In *FMCAD 2000*, pages 372–389, 2000.
5. Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.
6. Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.
7. Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics*. Springer-Verlag, 1990.
8. João P. Marques-Silva and Karem A. Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.
9. Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *DAC 2001*, pages 530–535, 2001.
10. Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT-solver. In *FMCAD 2000*, pages 108–125, 2000.
11. Mary Sheeran and Gunnar Stålmarck. A tutorial on Stålmarck's proof procedure for propositional logic. *Formal Methods in System Design*, 16(1):23–58, January 2000.