A SAT Characterization of Boolean-Program Correctness

K. Rustan M. Leino

Microsoft Research
One Microsoft Way, Redmond, WA 98052, USA
`leino@microsoft.com`
Appears in the SPIN 2003 workshop proceedings, Springer LNCS, copyright Springer-Verlag.

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 procedure-less 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 in the SLAM project [2]. In this paper, I show how to translate a procedure-less 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 procedure-less 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) [6]. 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.

Initial experience with an implementation (called Dizzy) of the method has exhibited poor performance. An unknown portion of this is due to that Dizzy inlines procedures rather than summarizing them (which SLAM's model checker Bebop [1] does). I hope the method presented here stimulates further research, perhaps where the method is used as an intraprocedural component of another checker.

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

Fig. 0. Grammar of the boolean programs in this paper.

Section 1 defines boolean programs and their weakest-precondition semantics. Section 2 presents the three mathematical ingredients on which the translation is based. Section 3 shows the translation algorithm. Section 4 discusses the complexity of the algorithm, brings out its symbolic-versus-explicit-state trade-offs, and reports on some preliminary experiments.

1 Boolean Programs and Their Semantics

Boolean programs in SLAM can have procedures [1]. In this paper, I use a procedure-less language. Procedures can be inlined at call sites (as indeed they are in Dizzy), since recursion is not prevalent in the device drivers that SLAM checks.

1.0 Syntax

In this paper, a boolean program has the form given in Figure 0. A program consists of a set of boolean variables and a set of named blocks. Each block consists of a sequence of statements followed by a $\textbf{goto}$ statement that declares a set of successor blocks (the absence of a $\textbf{goto}$ statement indicates the empty set of successor blocks).

The execution of a program fragment may *terminate* (in some state), *loop forever*, *go wrong*, or *idle*. Going wrong indicates undesirable behavior. A program is erroneous if an execution of it can go wrong; otherwise, it is correct. Idling indicates points in the execution beyond which we are not interested in the program's behavior, such as the beginning of infeasible paths.

Program execution begins in the first given block. Upon termination of the statements in a block, a successor block is picked arbitrarily (nondeterministically) and execution continues there. If the set of successor blocks is empty, execution idles.

In addition to ordinary assignment statements, there are $\textbf{assume}$ and $\textbf{assert}$ statements. Executing these statements when their conditions evaluate to $\textbf{true}$ is equivalent to a no-op. That is, their execution terminates in the same state in which it started. If the condition evaluates to $\textbf{false}$, the $\textbf{assert}$ statement goes wrong whereas the $\textbf{assume}$ statement idles.

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 $\textbf{if}\ E\ \textbf{then}\ S\ \textbf{else}\ T\ \textbf{end}$ is encoded as:

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

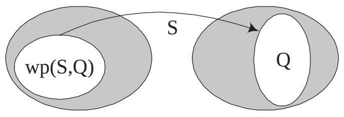

Fig. 1. $wp(S, Q)$ denotes the largest set of pre-states with the following property: starting from any state in $wp(S, Q)$, execution of S does not go wrong, and, if the execution terminates, it does so in a state satisfying Q.

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

```
0:  goto 1 or 2 ;                 2:  assume ¬E ; goto 3 ;
1:  assume E ; S ; goto 0 ;       3:
```

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

```
var x, y ;
A:  x := true ; goto B ;
B:  assert x ; x := x ∧ y ; goto B or C ;          (0)
C:
```

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

1.1 Semantics

I define the semantics of a boolean program in terms of *weakest preconditions* [7]. (The connection between operational semantics and weakest preconditions has been studied elsewhere, *e.g.* [10] and [8].) 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
– 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:

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

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.

[0] Unlike Dijkstra's original definition of weakest preconditions, I distinguish between looping forever and going wrong, and only the latter makes a program erroneous.

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 :

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

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 $\mathbf{true}$ 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 :

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

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

$$\begin{array}{lcl} (\forall\, x, y \ :: \ A(x,y) & = & B(\mathbf{true}, y)\ ) \\ (\forall\, x, y \ :: \ B(x,y) & = & x \wedge B(x \wedge y, y) \wedge C(x \wedge y, y)\ ) \\ (\forall\, x, y \ :: \ C(x,y) & = & \mathbf{true}\ ) \end{array} \tag{3}$$

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, these two closed-form definitions of A, B, C both satisfy equations (3):

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

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:

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

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 happens to *be* 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:

$$\begin{array}{rrcl} a, b, c: & a & = & f(a, b, c) \\ & b & = & g(a, b, c) \\ & c & = & h(a, b, c) \end{array} \qquad (6)$$

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:

$$\begin{array}{rrcl} A, B, C: & A & = & \mathcal{F}(A, B, C) \\ & B & = & \mathcal{G}(A, B, C) \\ & C & = & \mathcal{H}(A, B, C) \end{array} \qquad (7)$$

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 (7), the semantics of the boolean program is the weakest fixpoint of $\mathcal{F}, \mathcal{G}, \mathcal{H}$.

[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 $\langle a, b, c \rangle \sqsubseteq \langle a', b', c' \rangle \;\equiv\; a \leq a' \wedge b \leq b' \wedge c \leq c'$.

According to a celebrated theorem by Knaster and Tarski (see, *e.g.*, [8]), 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 [7]).[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 [11] and Chaff [12]. Indeed, the equations that define the program semantics are boolean formulas, and the existence of an error boils 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 Eliminating quantifications: 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.

To eliminate the quantifications in a set of equations like:

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

we can form a new set of equations by appropriate instantiations of the universal quantifications. In general, we need to instantiate the quantifications with all possible values,

[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.

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 quantifier-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 first quantification of (5), the one that defines 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 quantification 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:

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

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 evaluate $\neg A(k,m)$, then we can equivalently consider the solution to the following quantifier-free equations:

$$\begin{array}{lrcl} A,B,C: & A(k,m) & = & B(\mathbf{true},m) \\ & B(\mathbf{true},m) & = & B(m,m) \wedge C(m,m) \\ & B(m,m) & = & m \wedge B(m,m) \wedge C(m,m) \\ & C(m,m) & = & \mathbf{true} \end{array} \tag{8}$$

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 (8). What we have achieved is a projection of the original equations onto those values relevant to the expression of interest, akin to how *magic sets* can identify relevant instantiations of Horn clauses in logic programs [3].

In general, there may be more solutions to the computed quantifier-free set of equations than to the original set of equations. For example, (8) 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 (8) are *not* bound. Rather, k and m are free variables in (8).

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 $\mathbf{true}$, 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 [6], 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)). Section 4 takes a closer look at the actual complexity.

2.1 Eliminating functions: 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 $f \colon W \rightarrow U$ can be viewed as the 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 $f_w \colon () \rightarrow U$ 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 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 booleans, then the weakest solution to:

$$f : \quad (\forall\, w :: f(w) \;=\; f(\mathbf{false}) \vee f(w)\,) \tag{9}$$

can equally well be expressed as the weakest solution to:

$$\begin{array}{lrcl} f_{\mathbf{false}}, f_{\mathbf{true}} : & f_{\mathbf{false}} & = & f_{\mathbf{false}} \vee f_{\mathbf{false}} \\ & f_{\mathbf{true}} & = & f_{\mathbf{false}} \vee f_{\mathbf{true}} \end{array}$$

Note that these equations have two unknowns, $f_{\mathbf{false}}$ and $f_{\mathbf{true}}$, whereas (9) 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.

$$\begin{array}{rrcl}
J, L, M: & J(k) & = & L(k) \wedge M(k) \\
& L(k) & = & M(\mathbf{true}) \\
& M(k) & = & k \wedge M(k) \\
& M(\mathbf{true}) & = & M(\mathbf{true})
\end{array}
\qquad
\begin{array}{rrcl}
J_k, L_k, M_k, M_{\mathbf{true}}: & J_k & = & L_k \wedge M_k \\
& L_k & = & M_{\mathbf{true}} \\
& M_k & = & k \wedge M_k \\
& M_{\mathbf{true}} & = & M_{\mathbf{true}}
\end{array}$$

Fig. 2. Sets of equations in function-term form (left) and point-function form (right), illustrating that the conversion into point functions can enlarge the solution set.

That is, we can think of each function term as being a point function. Viewing function terms as point functions, the equations (8) take the form:

$$\begin{array}{rrcl}
A_{k,m}, B_{\mathbf{true},m}, B_{m,m}, C_{m,m}: & A_{k,m} & = & B_{\mathbf{true},m} \\
& B_{\mathbf{true},m} & = & B_{m,m} \wedge C_{m,m} \\
& B_{m,m} & = & m \wedge B_{m,m} \wedge C_{m,m} \\
& C_{m,m} & = & \mathbf{true}
\end{array} \tag{10}$$

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 (8) 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 function 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 into point functions may have enlarged the set of solutions to the equations. Suppose a boolean program with one variable and start block J gives rise to the set of quantifier-free equations to the left in Figure 2. Then the corresponding point-function equations, shown to the right in Figure 2, admit the solution where J_k and M_k are $\mathbf{false}$ and L_k and $M_{\mathbf{true}}$ are $\mathbf{true}$, independent of the value of k.[4] In contrast, the *function-term* equations admit this solution *only* if k does not have the value $\mathbf{true}$.

Luckily, the fact that our conversion into point functions may produce several names for the same point function, which may, as we've just seen, enlarge the set of solutions to our equations, need not bother us. The reason is that we are interested only in the *weakest* solution, which has not changed. Here's why it has not changed: The quantifier-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 quantifier-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

[4] 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.

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 Eliminating non-weakest solutions: Abbreviated way of computing fixpoints

The third mathematical ingredient, an abbreviated way of computing an expression for a fixpoint, 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 fixpoint of a function. If a function F on a complete lattice is *meet-continuous* —that is, if it has no infinite descending chains, or, equivalently, if it distributes meets over any nonempty, linear set of elements— 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, because monotonicity is equivalent to the property of distributing meets over any nonempty, linear, *finite* set of elements [8]. 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.

The functional transformations implicit in our set of equations (10) all return point functions, which in our application are nullary boolean functions, that is, booleans. Suppose there are n equations in the set ($n = 4$ for the set of equations (10)). The height of the lattice of n-tuples of booleans (ordered pointwise, see footnote 2) is n. Hence, the standard way of computing fixpoints tells us that we can obtain the weakest solution to the set of equations by iteratively applying the n functional transformations n times starting from $\mathbf{true}$.

The standard way does compute the weakest fixpoint. However, there is opportunity to compute it a bit faster. According to the standard way, the weakest solution of a for boolean functions f, g, h constrained as in (6) is:

$$\begin{aligned} f(&f(f(\top,\top,\top),\ g(\top,\top,\top),\ h(\top,\top,\top)),\\ &g(f(\top,\top,\top),\ g(\top,\top,\top),\ h(\top,\top,\top)),\\ &h(f(\top,\top,\top),\ g(\top,\top,\top),\ h(\top,\top,\top))) \end{aligned} \tag{11}$$

where, for brevity, I have written $\top$ for $\mathbf{true}$. But in fact, nested applications of the functions can be replaced by $\top$, which yields a smaller formula:

$$f(\top,\ g(\top,\top,h(\top,\top,\top)),\ h(\top,g(\top,\top,\top),\top)) \tag{12}$$

In the standard way (11), the tuple top element $\top,\top,\top$ occurs in the formula when the fixpoint depth (namely, 3) of the tuple function $[f,g,h]$ has been reached. In the abbreviated way (12), the boolean top element $\top$ occurs in the formula as soon as the fixpoint depth (namely, 1) of any single function (f, g, or h) has been reached. A proof due to Kuncak shows the standard and abbreviated forms to yield the same value [9].

Let's apply the abbreviated computation of fixpoints to our example (10). Uniquely labeling the subexpressions, we get:

$$\begin{array}{rclrcl} 0{:}A_{k,m} &=& 1{:}B_{\mathbf{true},m} & 3{:}C_{m,m} &=& \mathbf{true} \\ 1{:}B_{\mathbf{true},m} &=& 2{:}B_{m,m} \wedge 3{:}C_{m,m} & 4{:}B_{m,m} &=& \mathbf{true} \\ 2{:}B_{m,m} &=& m \wedge 4{:}B_{m,m} \wedge 5{:}C_{m,m} & 5{:}C_{m,m} &=& \mathbf{true} \end{array} \quad (13)$$

Notice how the subexpression $4{:}B_{m,m}$ is set to $\mathbf{true}$, because $4{:}B_{m,m}$ is an inner application of the point function $B_{m,m}$, enclosed within subexpression $2{:}B_{m,m}$. Notice also how these equations define a unique value of $0{:}A_{k,m}$; there are no more recursive definitions. By treating the symbolic initial values (here, m) and subexpression labels (like $0{:}A_{k,m}$ and $1{:}B_{\mathbf{true},m}$) as propositional variables, we have now produced a SAT formula for the semantics of the starting block of program (0). Thus, by conjoining $\neg\, 0{:}A_{k,m}$, we get a formula that is satisfiable if and only if program (0) is erroneous. And $\neg\, 0{:}A_{k,m} \wedge$ (13) *is* satisfiable, reflecting the presence of an error in program (0).

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. [5] 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 (and the symbolic initial values) as their propositional variables. [6]

The algorithm is shown in Figure 3. It outputs the set Eqs, given: a program with start block start, (symbolic) initial values w of the program's variables, and a procedure $\mathit{Instantiate}$ that for any program block b and argument list u returns the right-hand side of the weakest-precondition equation for b, instantiated with u. For example, for the running example, $\mathit{Instantiate}(B, \langle \mathbf{true}, m\rangle)$ 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.[7] To facilitate the simple generation of discriminating values, the algorithm keeps a counter

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

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

[7] 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.

```
$cnt := 0$ ;    $t, cnt := \langle start, w, cnt\rangle, cnt + 1$ ;    $Eqs, W := \{\neg t\}, \{\langle t, \epsilon\rangle\}$ ;
while $W \neq \emptyset$ do
  pick $b, u, s, cntxt$ such that $\langle\langle b, u, s\rangle, cntxt\rangle \in W$ ;
  $W := W \setminus \{\langle\langle b, u, s\rangle, cntxt\rangle\}$ ;
  if $\langle b, u\rangle \in cntxt$ then  $Eqs := Eqs \cup \{\langle b, u, s\rangle = \mathbf{true}\}$ ;  else
    $rhs := Instantiate(b, u)$ ;
    foreach $\langle c, v\rangle \in rhs$ do
      $t, cnt := \langle c, v, cnt\rangle, cnt + 1$ ;    replace $\langle c, v\rangle$ with $t$ in $rhs$ ;
      $W := W \cup \{\langle t, cntxt \cup \{\langle b, u\rangle\}\rangle\}$ ;    $G := G \cup \{\langle b, u, s\rangle \mapsto t\}$ ;
    end ;
    $Eqs := Eqs \cup \{\langle b, u, s\rangle = rhs\}$ ;
  end ;
end ;
```

Fig. 3. The algorithm for computing a set of boolean equations Eqs whose conjunction is satisfiable if and only if the boolean program under consideration is erroneous.

cnt of the number of different triples produced so far. The algorithm keeps a work list W of pairs $\langle t, cntxt\rangle$, where t is a triple that is used but not defined in Eqs, and where the set $cntxt$ of function terms gives the context in which t is used in Eqs.

The first branch of the if statement is taken when the fixpoint depth for $\langle b, u\rangle$ has been reached in the context $cntxt$. In this case, I shall refer to $\langle b, u, s\rangle$ as a *fixpoint triple* (like $4{:}B_{m,m}$ in (13)). In my *implementation* (Dizzy) of the algorithm, I use BDDs for the arguments u when comparing $\langle b, u\rangle$ to other pairs $\langle b, \cdot\rangle$ in $cntxt$. The second branch of the if statement attaches a discriminating value to each function term in rhs and adds the resulting triples to the work list.

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 restricting graph G to vertices corresponding to negative variables in a satisfying assignment, one can produce an execution trace from the initial state to an error.

The algorithm in Figure 3 has the effect of constructing G to be a tree of triples. As an important optimization, Dizzy shares common triples when possible, resulting in a directed acyclic graph. In my experience, this sharing can reduce the number of triples (and thus propositional variables) by more than 4 orders of magnitude.

An interesting possibility that calls for future investigation is the prospect of periodically, or even immediately, passing to the SAT solver the equations added to Eqs (maybe using an incremental SAT solver like Satire [0]). This has the advantage that, if the program is correct, unsatisfiability may be detected before the fixpoint depth has been reached everywhere.

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 in K variables. In a program with N blocks, there are then $N \cdot (2^{2^K})^K$ different function terms. The triple-sharing implementation of the algorithm can therefore produce a doubly-exponential number of different function terms.

There is a simple way to change the worst-case number of function terms generated: replace the symbolic initial values by a complete set of explicit boolean values. For the example, we would then start with $\neg A(\mathbf{false}, \mathbf{false}) \wedge \neg A(\mathbf{false}, \mathbf{true}) \wedge \neg A(\mathbf{true}, \mathbf{false}) \wedge \neg A(\mathbf{true}, \mathbf{true})$ instead of $\neg A(k, m)$. This would cause all subsequently produced function-term arguments also to be explicit boolean values, so there would only be $N \cdot 2^K$ different function terms, a *single* exponential. However, this representation suffers from always producing *at least* 2^K function terms, because that's how many function terms we get for the start block alone.

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:

$$b(E_0, e, E_1) \;=\; (\neg e \;\wedge\; b(E_0, \mathbf{false}, E_1)) \;\wedge\; (e \;\wedge\; b(E_0, \mathbf{true}, E_1)) \qquad (14)$$

Thus, if e is an expression other than an explicit boolean value, then the procedure $Instantiate$ called in Figure 3 can heuristically use (14) in lieu of (2) to return a definition of $b(E_0, e, E_1)$. For example, (14) may be used 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 using this latter heuristic, 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.)

In practice, I expect most boolean programs to generate fewer equations than the worst case. But initial experiments with Dizzy have not been rosy. SLAM's BDD-based model checker Bebop [1] outperforms Dizzy on most, but not all, examples I've tried from the SLAM regression test suite and actual device drivers for the Microsoft Windows operating system. In too many cases to be practical, Dizzy spends so much time it might as well be looping forever.

It may be that the present method reaches fixpoints too slowly to receive a net benefit from speedy SAT solvers. But straightforward measurements don't compare Dizzy with just the BDD-based aspect of Bebop, because Bebop generates procedure summaries, which enable sharing of work between different procedure invocations, whereas Dizzy inlines procedures, which forsakes work sharing and also dramatically increases the effective number of program variables. Since I don't know how to incorporate procedure summarization in Dizzy, I instead tried crippling Bebop by feeding it the inlined-procedure version of a smallish program. This increased Bebop's running time from 37 seconds to 64 minutes. Dizzy practically loops forever on this program, but the big impact of procedure summarization suggests there's room for further innovation in Dizzy.

5 Related Work and Conclusion

The presented method of doing model checking is perhaps, in the end, most reminiscent of a standard forward reachability algorithm where weakest-precondition equations give the next-state relation of blocks. 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 symbolically. The method uses both BDDs (in the phase that produces the SAT formula, see Figure 3) and SAT solving (in the phase that checks the resulting formula). In particular, the conditions in **assume** and **assert** statements, which determine whether or not execution proceeds normally, are not used in the first phase (except as mentioned in footnote 7). Instead, these conditions are encoded into the final SAT formula, which puts the responsibility of wrestling with path feasibility and errors onto the SAT solver. Consequently, the boolean expressions that the BDD package sees are much simpler; indeed, by applying equation (14), one can make them arbitrarily simple. The cost of this partitioning of work between BDDs and SAT solving 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 5). In contrast, standard forward reachability algorithms represent a state only once, because they consider path feasibility and errors with each step of the algorithm.

Another technique of model checking is *bounded model checking* [4], 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 at the end of Section 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 [14]. Also, standard algorithms for symbolic reachability can be performed using SAT solving rather than BDDs (see, *e.g.*, [5]).

Podelski has considered the characterization of model checking as a system of constraints [13]. In a different context, Tang and Hofmann convert a second-order problem into a first-order problem [15].

In conclusion, I have presented a (sound and complete) method for mechanically checking the correctness of boolean programs. Based on three mathematical ingredients, the method attempts to capitalize on recent advances in SAT solving. The implementation Dizzy shows the method to succeed in using BDDs lightly. Perhaps too lightly, because Dizzy often spends an impractically long time reaching fixpoints, exasperated (caused?) by the act of inlining procedures.

An appealing piece of future work is to consider using the method with some form of procedure summarization. I also hope that the theoretical techniques presented will inspire other future work.

Acknowledgements I'm grateful to Sriram Rajamani for inspiring discussions about this problem. I have personally benefited, and so has this work, from discussions with Byron Cook about SAT solving, with Jakob 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. Viktor Kuncak devised a proof of correctness for the abbreviated way to compute fixpoints [9], and John Dunagan helped me design the triple-sharing algorithm alluded to in Section 3. Byron, Jakob, Viktor, and the referees provided thoughtful and valuable comments on earlier drafts of this paper.

References

0. Fadi Aloul. Satire. `http://www.eecs.umich.edu/~faloul/Tools/satire/`, April 2002.
1. Thomas Ball and Sriram K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN 7*, *LNCS* #1885, pp. 113–130. Springer, 2000.
2. Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 8*, *LNCS* #2057, pp. 103–122. Springer, 2001.
3. François Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. Magic sets and other strange ways to implement logic programs. In *PODS 5*, pp. 1–15. ACM Press, 1986.
4. Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *TACAS '99*, *LNCS* #1579, pp. 193–207. Springer, 1999.
5. Per Bjesse and Koen Claessen. SAT-based verification without state space traversal. In *FMCAD 2000*, *LNCS* #1954, pp. 372–389. Springer, 2000.
6. Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.
7. Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.
8. Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics*. Texts and Monographs in Computer Science. Springer-Verlag, 1990.
9. Viktor Kuncak and K. Rustan M. Leino. On computing the fixpoint of a set of boolean equations. Technical Report MSR-TR-2003-08, Microsoft Research, 2003.
10. M. S. Manasse and C. G. Nelson. Correct compilation of control structures. Technical report, AT&T Bell Laboratories, September 1984.
11. João P. Marques-Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.
12. Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *DAC 2001*, pp. 530–535. ACM, 2001.
13. Andreas Podelski. Model checking as constraint solving. In *SAS 2000*, *LNCS* #1824, pp. 22–37. Springer, 2000.
14. Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT-solver. In *FMCAD 2000*, *LNCS* #1954, pp. 108–125. Springer, 2000.
15. Francis Tang and Martin Hofmann. Generation of verification conditions for Abadi and Leino's logic of objects. In *FOOL 9*, 2002. Electronic proceedings.