Bebop: A Symbolic Model Checker for Boolean Programs

Thomas Ball and Sriram K. Rajamani

Software Productivity Tools
Microsoft Research
`http://www.research.microsoft.com/slam/`

Abstract. We present the design, implementation and empirical evaluation of `Bebop`—a symbolic model checker for boolean programs. `Bebop` represents control flow explicitly, and sets of states implicitly using BDDs. By harnessing the inherent modularity in procedural abstraction and exploiting the locality of variable scoping, `Bebop` is able to model check boolean programs with several thousand lines of code, hundreds of procedures, and several thousand variables in a few minutes.

1 Introduction

Boolean programs are programs with the usual control-flow constructs of an imperative language such as C but in which all variables have boolean type. Boolean programs contain procedures with call-by-value parameter passing and recursion, and a restricted form of control nondeterminism.

Boolean programs are an interesting subject of study for a number of reasons. First, because the amount of storage a boolean program can access at any point is finite, questions of reachability and termination (which are undecidable in general) are decidable for boolean programs.[1] Second, as boolean programs contain the control-flow constructs of C, they form a natural target for investigating model checking of software. Boolean programs can be thought of as an abstract representation of C programs that explicitly captures correlations between data and control, in which boolean variables can represent arbitrary predicates over the unbounded state of a C program. As a result, boolean programs are useful for reasoning about temporal properties of software, which depend on such correlations.

We have created a model checker for boolean programs called `Bebop`. Given a boolean program B and a statement s in B, `Bebop` determines if s is reachable in B (informally stated, s is reachable in B if there is some initial state such that if B starts execution from this state then s

[1] Boolean programs are equivalent in power to push-down automaton, which accept context-free languages.

```
       decl g;

       main()
       begin
         decl h;
[6]      h := !g;
[7]      A(g,h);
[8]      skip;
[9]      A(g,h);
[10]     skip;
[11]     if (g) then
[12] R:    skip;
         else
[14]       skip;
         fi
       end

       A(a1,a2)
       begin
[20]     if (a1) then
[21]       A(a2,a1);
[22]       skip;
         else
[24]       g := a2;
         fi
       end
```

```
bebop v1.0: (c) Microsoft Corporation.
Done creating bdd variables
Done building transition relations

Label R reachable by following path:

Line 12         State g=1 h=0
Line 11         State g=1 h=0
Line 10         State g=1 h=0
    Line 22     State g=1 a1=1 a2=0
        Line 24 State g=1 a1=0 a2=1
        Line 20 State g=1 a1=0 a2=1
    Line 21     State g=1 a1=1 a2=0
    Line 20     State g=1 a1=1 a2=0
Line 9          State g=1 h=0
Line 8          State g=1 h=0
    Line 22     State g=1 a1=1 a2=0
        Line 24 State g=1 a1=0 a2=1
        Line 20 State g=1 a1=0 a2=1
    Line 21     State g=1 a1=1 a2=0
    Line 20     State g=1 a1=1 a2=0
Line 7          State g=1 h=0
Line 6          State g=1
```

Fig. 1. The **skip** statement labelled `R` is reachable in this boolean program, as shown by the output of the `Bebop` model checker.

eventually executes). If statement s is reachable, then `Bebop` produces a shortest trace leading to s (that possibly includes loops and crosses procedure boundaries).

Example. Figure 1 presents a boolean program with two procedures (`main` and a recursive procedure `A`). In this program, there is one global variable g. Procedure `main` has a local variable h which is assigned the complement of g. Procedure `A` has two parameters. The question is: is label `R` reachable? The answer is yes, as shown by the output of `Bebop` on the right. The tool finds that `R` is reachable and gives a shortest trace (in reverse execution order) from `R` to the first line of `main` (line 6). The indentation of a line indicates the depth of the call stack at that point in the trace. Furthermore, for each line in the trace, `Bebop` outputs the state of the variables (in scope) just before the line. The trace shows that in order to reach label `R`, by this trace of lines, the value of g initially must be 1.[2] Furthermore, the trace shows that the two calls that main makes to procedure `A` do not change the value of g. We re-emphasize that this is a shortest trace witnessing the reachability of label `R`.

Contributions. We have adapted the interprocedural dataflow analysis algorithm of Reps, Horwitz and Sagiv (RHS) [RHS95,RHS96] to decide the reachability status of a statement in a boolean program. A core idea of the RHS algorithm is to efficiently compute "summaries" that record the input/output behavior of a procedure. Once a summary $\langle I, O\rangle$ has been computed for a procedure **pr**, it is not necessary to reanalyze the body of **pr** if input context I arises at another call to **pr**. Instead, the summary for **pr** is consulted and the corresponding output context O is used. We use Binary Decisions Diagrams (BDDs) to symbolically represent these summaries, which are binary relationships between sets of states.

In the program of Figure 1, our algorithm computes the summary $s = \langle \{g = 1, h = 0\}, \{g' = 1, h' = 0\}\rangle$ when procedure `A` is first called (at line 7) with the state $\{g = 1, h = 0\}$. This summary will be "installed" at all calls to `A` (in particular, the call to `A` at line 9). Thus, when the state $I = \{g = 1, h = 0\}$ propagates to the call at line 9, the algorithm finds that the summary s matches and will use it to "jump over" the call to `A` rather than descending into `A` to analyze it again.

A key point about `Bebop` that distinguishes it from other model checkers is that it exploits the locality of variable scopes in a program. The time and space complexity of our algorithm is $O(E \times 2^k)$ where E is the number of edges in the interprocedural control-flow graph of the boolean program[3] and k is the maximal number of variables *in scope* at any program point in the program. In the example program of Figure 1 there are a total of 4 variables (global g, local h, and formals $a1$ and $a2$). However, at any statement, at most three variables are in scope (in `main`, g and h; in `A`, g, $a1$, and $a2$).

So, for a program with g global variables, and a maximum of l local variables in any procedure, the running time is $O(E \times 2^{g+l})$. If the number of variables in scope is held constant then the running time of `Bebop` grows as function of the number of statements in the program (and not the total number of variables). As a result, we have been able to model check boolean programs with several thousand lines of code, and several

[2] Note that g is left unconstrained in the initial state of the program. If a variable's value is unconstrained in a particular trace then `Bebop` does not output it. Thus, it is impossible for g to be initially 0 and to follow the same trace. In fact, for this example, label `R` is not reachable if g initially is 0.

[3] E is linear in the number of statements in the boolean program.

thousand variables in a few minutes (the largest example we report in Section 4 has 2401 variables).

A second major idea in `Bebop` is to use an explicit control-flow graph representation rather than encode the control flow of a boolean program using BDDs. This implementation decision is an important one, as it allows us to optimize the model checking algorithm using well-known techniques from compiler optimization. We explain two such techniuqes —live ranges and modication/reference analysis— to reduce the number of variables in support of the BDDs that represent the reachable states at a program point.

Overview. Section 2 presents the syntax and semantics of boolean programs. Section 3 describes our adaption of the RHS algorithm to use BDDs to solve the reachability problem for boolean programs. Section 4 evaluates the performance of `Bebop`. Section 5 reviews related work and Section 6 looks towards future work.

2 Boolean Programs

2.1 Syntax

Figure 2 presents the syntax of boolean programs. We will comment on noteworthy aspects of it here. Boolean variables are either global (if they are declared outside the scope of a procedure) or local (if they are declared inside the scope of a procedure). Since there is only one type in the boolean programming language, variable declarations need not specify a type. Variables are statically scoped, as in C. A variable identifier is either a C-style identifer or an arbitrary string between the characters "{" and "}". The latter form is useful for creating boolean variables with names denoting predicates in another language (such as {`*p==*q`}).

There are two constants in the language: 0 (false) and 1 (true). Expressions are built in the usual way from these constants, variables and the standard logical connectives.

The statement sub-language (*stmt*) is very similar to that of C, with a few exceptions. Statements may be labelled, as in C. A parallel assignment statement allows the simultaneous assignment of a set of values to a set of variables. Procedure calls use call-by-value parameter passing.[4] There

[4] Boolean programs support return values from procedures, but to simplify the technical presentation we have omitted their description here. A return value of a procedure can be modelled with a single global variable, where the global variable is assigned immediately preceding a return and copied immediately after the return into the local state of the calling procedure.

<table>
<tr><th colspan="3">Syntax</th><th>Description</th></tr>
<tr><td>prog</td><td>::=</td><td>decl * proc *</td><td>A program is a list of global variable declarations followed by a list of procedure definitions</td></tr>
<tr><td>decl</td><td>::=</td><td>decl id + ;</td><td>Declaration of variables</td></tr>
<tr><td>id</td><td>::=</td><td>[a-zA-Z_] [a-zA-Z0-9_] *</td><td>An identifier can be a regular C-style identifier</td></tr>
<tr><td></td><td>|</td><td>{ string }</td><td>or a string of characters between '{' and '}'</td></tr>
<tr><td>proc</td><td>::=</td><td>id (id *) begin decl * sseq end</td><td>Procedure definition</td></tr>
<tr><td>sseq</td><td>::=</td><td>lstmt +</td><td>Sequence of statements</td></tr>
<tr><td>lstmt</td><td>::=</td><td>stmt</td><td></td></tr>
<tr><td></td><td>|</td><td>id : stmt</td><td>Labelled statement</td></tr>
<tr><td>stmt</td><td>::=</td><td>skip ;</td><td></td></tr>
<tr><td></td><td>|</td><td>print (expr +) ;</td><td></td></tr>
<tr><td></td><td>|</td><td>goto id ;</td><td></td></tr>
<tr><td></td><td>|</td><td>return ;</td><td></td></tr>
<tr><td></td><td>|</td><td>id + := expr + ;</td><td>Parallel assignment</td></tr>
<tr><td></td><td>|</td><td>if (decider) then sseq else sseq fi</td><td>Conditional statement</td></tr>
<tr><td></td><td>|</td><td>while (decider) do sseq od</td><td>Iteration statement</td></tr>
<tr><td></td><td>|</td><td>assert (decider) ;</td><td>Assert statement</td></tr>
<tr><td></td><td>|</td><td>id (expr *) ;</td><td>Procedure call</td></tr>
<tr><td>decider</td><td>::=</td><td>?</td><td>Non-deterministic choice</td></tr>
<tr><td></td><td>|</td><td>expr</td><td></td></tr>
<tr><td>expr</td><td>::=</td><td>expr binop expr</td><td></td></tr>
<tr><td></td><td>|</td><td>! expr</td><td>Negation</td></tr>
<tr><td></td><td>|</td><td>(expr)</td><td></td></tr>
<tr><td></td><td>|</td><td>id</td><td></td></tr>
<tr><td></td><td>|</td><td>const</td><td></td></tr>
<tr><td>binop</td><td>::=</td><td>'|' | '&' | '∧' | '=' | '!=' | '⇒'</td><td>Logical connectives</td></tr>
<tr><td>const</td><td>::=</td><td>0 | 1</td><td>False/True</td></tr>
</table>

Fig. 2. The syntax of boolean programs.

are three statements that can affect the control flow of a program: **if**, **while** and **assert**. Note that the predicate of these three statements is a *decider*, which can be used to model non-determinism. A *decider* is either a boolean expression which evaluates (deterministically) to 0 or 1, or "?", which evaluates to 0 or 1 non-deterministically.

2.2 Statements, Variables and Scope

The term *statement* denotes an instance that can be derived from the nonterminal *stmt* (see Figure 2). Let B be a boolean program with n statements and p procedures. We assign a unique *index* to each statement in B in the range $1 \ldots n$ and a unique index to each procedure in B in the range $n+1 \ldots n+p$. Let s_i denote the statement with index i.

To simplify presentation of the semantics, we assume that variable names and statement labels are globally unique in B. Let $V(B)$ be the set of all variables in B. Let $\mathit{Globals}(B)$ be the set of global variables of B. Let $\mathit{Formals}_B(i)$ be the set of formal parameters of the procedure that contains s_i. Let $\mathit{Locals}_B(i)$ be the set of local variables and formal parameters of the procedure that contains s_i. For all i, $1 \leq i \leq n$, $\mathit{Formals}_B(i) \subseteq \mathit{Locals}_B(i)$. Let $\mathit{InScope}_B(i)$ denote the set of all variables of B whose scope includes s_i. For all i, $1 \leq i \leq n$, $\mathit{InScope}_B(i) = \mathit{Locals}_B(i) \cup \mathit{Globals}(B)$.

2.3 The Control-flow Graph

This section defines the control-flow graph of a boolean program. Since boolean programs contain arbitrary intra-procedural control flow (via the **goto**), it is useful to present the semantics of boolean programs in terms of their control-flow graph rather than their syntax. To make the presentation of the control-flow graph simpler, we make the minor syntactic restriction that every call c to a procedure **pr** in a boolean program is immediately followed by a skip statement $\mathbf{skip}_c$.

The control-flow graph of a boolean program B is a directed graph $G_B = (V_B, \mathit{Succ}_B)$ with set of vertices $V_B = \{1, 2, \ldots, n+p+1\}$ and successor function $\mathit{Succ}_B : V_B \to 2^{V_B}$. The set V_B contains one vertex for each statement in B (vertices $1 \ldots n$) and one vertex $\mathit{Exit}_{\mathbf{pr}}$ for every procedure **pr** in B (vertices $n+1 \ldots n+p$). In addition, V_B contains a vertex $\mathit{Err} = n+p+1$ which is used to model the failure of an **assert** statement. For any procedure **pr** in B, let $\mathit{First}_B(\mathbf{pr})$ be the index of the first statement in **pr**. For any vertex $v \in V_B - \{\mathit{Err}\}$, let $\mathit{ProcOf}_B(v)$ be the index of the procedure containing v.

The successor function Succ_B is defined in terms of the function $\mathit{Next}_B : \{1, 2, \ldots, n\} \to \{1, 2, \ldots, n+p\}$ which maps statement indices to their lexical successor if one exists, or to the exit node of the containing procedure otherwise. $\mathit{Next}_B(i)$ has a recursive definition based on the syntax tree of B (see Figure 2). In this tree, each statement has an *sseq* node as its parent. The sequence of statements derived from the *sseq* parent of statement s_i is called the *containing sequence* of s_i. If s_i is not the last statement in its containing sequence then $\mathit{Next}_B(i)$ is the index of the statement immediately following s_i in this sequence. Otherwise, let a be the closest ancestor of s_i in the syntax tree such that (1) a is a *stmt* node, and (2) a is not the last statement in a's containing sequence. If such a node a exists, then $Next_B(i)$ is the index of the statement immediately following a in its containing sequence. Otherwise, $Next_B(s_i) = Exit_{\mathbf{pr}}$, where $\mathbf{pr} = ProcOf_B(i)$.

If s_j is a procedure call, we define $ReturnPt_B(j) = Next_B(j)$ (which is guaranteed to be a **skip** statement because of the syntactic restriction we previously placed on boolean programs).

We now define $Succ_B$ using $Next_B$ and $ReturnPt_B$. For $1 \leq i \leq n$, the value of $Succ_B(i)$ depends on the statement s_i, as follows:

- If s_i is "**goto** L" then $Succ_B(i) = \{j\}$, where s_j is the statement labelled L.
- If s_i is a parallel assignment, **skip** or **print** statement then $Succ_B(i) = \{Next_B(i)\}$.
- If s_i is a **return** statement then $Succ_B(i) = \{Exit_{\mathbf{pr}}\}$, where $\mathbf{pr} = ProcOf_B(i)$.
- If s_i is an **if** statement then $Succ_B(v) = \{Tsucc_B(i), Fsucc_B(i)\}$, where $Tsucc_B(i)$ is the index of the first statement in the **then** branch of the **if** and $Fsucc_B(i)$ is the index of the first statement in the **else** branch of the **if**.
- If s_i is a **while** statement then $Succ_B(i) = \{Tsucc_B(i), Fsucc_B(i)\}$, where $Tsucc_B(i)$ is the first statement in the body of the **while** loop and $Fsucc_B(i) = Next_B(i)$.
- If s_i is an **assert** statement then $Succ_B(i) = \{Tsucc_B(i), Fsucc_B(i)\}$, where $Tsucc_B(i) = Next_B(i)$ and $Fsucc_B(i) = n + p + 1$ (the Err vertex).
- If s_i is a procedure call to procedure **pr** then $Succ_B(i) = First_B(\mathbf{pr})$.

We now define $Succ_B(i)$ for $n+1 \leq i \leq n+p$ (that is, for the $Exit$ vertices associated with the p procedures of B). Given exit vertex $Exit_{\mathbf{pr}}$ for some procedure **pr**, we have

$$Succ_B(Exit_{\mathbf{pr}}) = \{ReturnPt_B(j) \mid \text{statement } s_j \text{ is a call to } \mathbf{pr}\ \}$$

Finally, $Succ_B(Err) = \{\}$. That is, the vertex Err has no successors.

The control-flow graph of a boolean program can be constructed in time and space linear $n + p$, the number of statements and procedures in the program.

2.4 A Transition System for Boolean Programs

For a set $V \subseteq V(B)$, a *valuation* Ω to V is a function that associates every boolean variable in V with a boolean value. Ω can be extended to expressions over V (see *expr* in Figure 2) in the usual way. For example, if $V = \{x, y\}$, and $\Omega = \{(x, 1), (y, 0)\}$ then $\Omega(x|y) = 1$. For any function $f : D \to R$, $d \in D$, $r \in R$, $f[d/r] : D \to R$ is defined as $f[d/r](d') = r$ if $d = d'$, and $f(d')$ otherwise. For example, if $V = \{x, y\}$, and $\Omega = \{(x, 1), (y, 0)\}$ then $\Omega[x/0] = \{(x, 0), (y, 0)\}$.

A *state* η of B is a pair $\langle i, \Omega \rangle$, where $i \in V_B$ and Ω is a valuation to the variables in $InScope_B(i)$. $States(B)$ is the set of all states of B. Intuitively, a state contains the program counter (i) and values to all the variables visible at that point (Ω). Note that our definition of state is different from the conventional notion of a program state, which includes a call stack. The projection operator Γ maps a state to its vertex: $\Gamma(\langle i, \Omega \rangle) = i$. We can extend Γ to operate on sequences of states in the usual way.
We define a set $\Sigma(B)$ of terminals:

$$\begin{aligned}\Sigma(B) = \{\sigma\} \ \cup \ \{ \ \langle \mathbf{call}, i, \Delta \rangle, \langle \mathbf{ret}, i, \Delta \rangle \mid & \exists j \in V_B, s_j \text{ is a procedure call}, \\ & i = ReturnPt_B(j), \text{ and} \\ & \Delta \text{ is a valuation to } Locals_B(j)\}\end{aligned}$$

It is clear that $\Sigma(B)$ is finite since all variables in B are boolean variables. Terminals are either σ, which is a place holder, or triples that are introduced whenever there is a procedure call in B. The first component of the triple is either **call** or **ret**, corresponding to the actions of a call to and return from that procedure, the second is the return point of the call, and the third component keeps track of values of local variables of the calling procedure at the time of the call.

We use $\eta_1 \xrightarrow{\alpha}_B \eta_2$, to denote that B can make an α-labeled transition from state η_1 to state η_2. Formally, $\eta_1 \xrightarrow{\alpha}_B \eta_2$ holds if $\eta_1 = \langle i_1, \Omega_1 \rangle \in States(B)$, $\eta_2 = \langle i_2, \Omega_2 \rangle \in States(B)$, and $\alpha \in \Sigma(B)$, where the conditions on η_1, η_2 and α for each statement construct are shown in Table 1. We explain the table below:

- The transitions for **skip**, **print**, **goto** and **return** are the same. All these statements have exactly one control-flow successor. For vertices v such that $Succ_B(v) = \{w\}$, we define $sSucc_B(v) = w$. Each statement passes control to its single successor $sSucc_B(i_1)$ and does not change the state of the program.
- The transition for parallel assignment again passes control to the sole successor of the statement and the state changes in the expected manner.
- The transitions for **if**, **while** and **assert** statements are identical. If the value of the decider d associated with the statement is ? then

i_1	α	i_2	Ω_2
skip **print** **goto** **return**	$\alpha = \sigma$	$i_2 = sSucc_B(i_1)$	$\Omega_2 = \Omega_1$
$x_1, \ldots, x_k :=$ $e_1, \ldots, e_k$	$\alpha = \sigma$	$i_2 = sSucc_B(i_1)$	$\Omega_2 = \Omega_1[x_1/\Omega_1(e_1))]$ $\cdots[x_k/\Omega_1(e_k)]$
$\mathbf{if}(d)$ $\mathbf{while}(d)$ $\mathbf{assert}(d)$	$\alpha = \sigma$	if $d\ =\ ?$ $i_2 \in Succ_B(i_1)$ if $\Omega_1(d)\ =\ 1$ $i_2 = Tsucc_B(i_1)$ if $\Omega_1(d)\ =\ 0$ $i_2 = Fsucc_B(i_1)$	$\Omega_2 = \Omega_1$
$\mathbf{pr}(e_1, \ldots, e_k)$	$\alpha = \langle \mathbf{call}, \mathit{ReturnPt}_B(i_1), \Delta \rangle$, $\Delta(x) = \Omega_1(x),\ \forall\ x \in \mathit{Locals}_B(i_1)$	$i_2 = \mathit{First}_B(\mathbf{pr})$	$\Omega_2(x_i) = \Omega_1(e_i)$, $\forall\ x_i \in \mathit{Formals}_B(i_2)$ $\Omega_2(g) = \Omega_1(g)$, $\forall\ g \in \mathit{Globals}(B)$
$\mathit{Exit}_{\mathbf{pr}}$	$\alpha = \langle \mathbf{ret}, i_2, \Delta \rangle$	$i_2 \in Succ_B(i_1)$	$\Omega_2(g) = \Omega_1(g)$, $\forall\ g \in \mathit{Globals}(B)$ $\Omega_2(x) = \Delta(x)$, $\forall\ x \in \mathit{Locals}_B(i_2)$

Table 1. Conditions on the state transitions $\langle i_1, \Omega_1 \rangle \xrightarrow{\alpha}_B \langle i_2, \Omega_2 \rangle$, for each vertex type of i_1. See the text for a full explanation.

the successor is chosen non-deterministically from the set $Succ_B(i_1)$. Otherwise, d is a boolean expression and is evaluated in the current state to determine the successor.

- The transition for a call statement s_{i_1} contains the α label

$$\langle \mathbf{call}, \mathit{ReturnPt}_B(i_1), \Delta \rangle$$

  where Δ records the values of the local variables at i_1 from the state Ω_1. The next state, Ω_2 gives new values to the formal parameters of the called procedure based on the values of the corresponding actual arguments in state Ω_1. Furthermore, Ω_2 is constrained to be the same as Ω_1 on the global variables.
- Finally, the transition for an exit vertex $i_1 = \mathit{Exit}_{\mathbf{pr}}$ has $\alpha = \langle \mathbf{ret}, i_2, \Delta \rangle$, where i_2 must be a successor of i_1. The output state Ω_2 is constrained as follows: Ω_2 must agree with Ω_1 on all global variables; Ω_2 must agree with Δ on the local variables in scope at i_2.

2.5 Trace Semantics

We now are in a position to give a trace semantics to boolean programs based on a context-free grammar $\mathcal{G}(B)$ over the alphabet $\Sigma(B)$ that spec-

1. $S \to MS$	**4.** $\forall\langle \mathbf{call}, i, \Delta\rangle, \langle \mathbf{ret}, i, \Delta\rangle \in \Sigma(B)$:
2. $\forall\langle \mathbf{call}, i, \Delta\rangle \in \Sigma(B)$:	$M \to \langle \mathbf{call}, i, \Delta\rangle\ M\ \langle \mathbf{ret}, i, \Delta\rangle$
$S \to \langle \mathbf{call}, i, \Delta\rangle\ S$	**5.** $M \to MM$
3. $S \to \epsilon$	**6.** $M \to \epsilon$
	7. $M \to \sigma$

Table 2. The production rules $Rules(B)$ for grammar $\mathcal{G}(B)$.

ifies the legal sequences of calls and returns that a boolean program B may make.

A context-free grammar $\mathcal{G}$ is a 4-tuple $\langle N, T, R, S\rangle$, where N is a set of nonterminals, T is a set of terminals, R is a set of production rules and $S \in N$ is a start symbol. For each program B, we define a grammar $\mathcal{G}(B) = \langle\{S, M\}, \Sigma(B), Rules(B), S\rangle$, where $Rules(B)$ is defined by the productions of Table 2.

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

We assume that B contains a distinguished procedure named **main**, which is the initial procedure that executes. A state $\eta = \langle i, \Omega\rangle$ is *initial* if $i = First_B(\mathbf{main})$ (all variables can take on arbitrary initial values). A finite sequence $\overline{\eta} = \eta_0 \overset{\alpha_1}{\to}_B \eta_1 \overset{\alpha_2}{\to}_B \cdots \eta_{m-1} \overset{\alpha_m}{\to}_B \eta_m$ is a *trajectory* of B if (1) for all $0 \le i < m$, $\eta_i \overset{\alpha_i}{\to}_B \eta_{i+1}$, and (2) $\alpha_1 \ldots \alpha_m \in \mathcal{L}(\mathcal{G}(B))$. A trajectory $\overline{\eta}$ is called an *initialized trajectory* if η_0 is an initial state of B. If $\overline{\eta}$ is an initialized trajectory, then its projection to vertices $\Gamma(\eta_0), \Gamma(\eta_1), \ldots, \Gamma(\eta_n)$ is called a *trace* of B. The semantics of a boolean program is its set of traces. A state η of B is *reachable* if there exists an initialized trajectory of B that ends in η. An vertex $v \in V_B$ is *reachable* if there exists a trace of B that ends in vertex v.

3 Boolean Programs Reachability via Interprocedural Dataflow Analysis and BDDs

In this section, we present an interprocedural dataflow analysis that, given a boolean program B and its control-flow graph $G_B = (V_B, Succ_B)$, determines the reachability status of every vertex in V_B. We describe and present the algorithm, show how it can be extended to report short trajectories (when a vertex is found to be reachable), and describe several optimizations that we plan to make to the algorithm.

3.1 The RHS Algorithm, Generalized

As discussed in the Introduction, we have generalized the interprocedural dataflow algorithm of Reps-Horwitz-Sagiv (RHS) [RHS95,RHS96]. The main idea of this algorithm is to compute "path edges" that represent the reachability status of a vertex in a control-flow graph and to compute "summary edges" that record the input/output behavior of a procedure. We (re)define path and summary edges as follows:

Path edges. Let v be a vertex in V_B and let $e = First_B(ProcOf_B(v))$. A *path edge* incident into a vertex v, is a pair of valuations $\langle \Omega_e, \Omega_v \rangle$,[5] such that(1) there is a initialized trajectory $\overline{\eta}_1 = \langle First_B(\mathbf{main}), \Omega\rangle \ldots \langle e, \Omega_e \rangle$, and (2) there is a trajectory $\overline{\eta}_2 = \langle e, \Omega_e\rangle \ldots \langle v, \Omega_v\rangle$ that does not contain the exit vertex $Exit_{ProcOf_B(v)}$ (exclusive of v itself). For each vertex v, $PathEdges(v)$ is the set of all path edges incident into v.

A summary edge is a special kind of path edges that records the behavior of a procedure.

Summary edges. Let c be a vertex in V_B representing a procedure call with corresponding statement $s_c = \mathbf{pr}(e_1, e_2, ...e_k)$. A *summary edge* associated with c is a pair of valuations $\langle \Omega_1, \Omega_2 \rangle$, such that all the local variables in $Locals_B(c)$ are equal in Ω_1 and Ω_2, and the global variables change according to some path edge from the entry to the exit of the callee. Suppose P is the set of path edges at $Exit_{\mathbf{pr}}$. We define $Lift_c(P, \mathbf{pr})$ as the set of summary edges obtained by "lifting" the set of path edges P to the call c, while respecting the semantics of the call and return

[5] The valuations Ω_e and Ω_v are defined with respect to the set of variables $V = InScope_B(e) = InScope_B(v)$.

transitions from Table 1. Formally

$$\begin{aligned}
\mathit{Lift}_c(P, \mathbf{pr}) = \{\langle \Omega_1, \Omega_2\rangle \mid & \exists \langle \Omega_i, \Omega_o\rangle \in P, \text{ and} \\
& \forall x \in \mathit{Locals}_B(c) : \Omega_1(x) = \Omega_2(x), \text{ and} \\
& \forall x \in \mathit{Globals}(B) : (\Omega_1(x) = \Omega_i(x)) \wedge (\Omega_2(x) = \Omega_o(x)), \text{ and} \\
& \forall \text{ formals } y_j \text{ of } \mathbf{pr} \text{ and actuals } e_j : \Omega_1(e_j) = \Omega_i(y_j)\}
\end{aligned}$$

For each vertex v in Call_B, $\mathit{SummaryEdges}(v)$ is the set of summary edges associated with v. As the algorithm proceeds, $\mathit{SummaryEdges}(v)$ is incrementally computed for each call site. Summary edges are used to avoid revisiting portions of the state space that have already been explored, and enable analysis of programs with procedures and recursion.

Let Call_B be the set of vertices in V_B that represent call statements. Let Exit_B be the set of exit vertices in V_B. Let Cond_B be the set of vertices in V_B that represent the conditional statements **if**, **while** and **assert**.

Transfer Functions. With each vertex v such that $s_v \notin \mathit{Cond}_B \cup \mathit{Exit}_B$, we associate a transfer function $\mathit{Transfer}_v$. With each vertex $v \in \mathit{Cond}_B$, we associate two transfer functions $\mathit{Transfer}_{v,true}$ and $\mathit{Transfer}_{v,false}$. The definition of these functions is given in Table 3. Given two sets of pairs of valuations, S and T, $\mathit{Join}(S,T)$ is the image of set S with respect to the transfer function T. Formally $\mathit{Join}(S,T) = \{\langle \Omega_1, \Omega_2\rangle \mid \exists \Omega_j . \langle \Omega_1, \Omega_j\rangle \in S \wedge \langle \Omega_j, \Omega_2\rangle \in T\}$. During the processing of calls, in addition to applying the transfer function, the algorithm uses the function *SelfLoop* which takes a set of path edges, and makes self-loops with the targets of the edges. Formally, $\mathit{SelfLoop}(S) = \{\langle \Omega_2, \Omega_2\rangle \mid \exists \langle \Omega_1, \Omega_2\rangle \in S\}$.

Our generalization of the RHS algorithm is shown in Figure 3. The algorithm uses a worklist, and computes path edges and summary edges in a directed, demand-driven manner, starting with the entry vertex of **main** (the only vertex initially known to be reachable). In the algorithm, path edges are used to compute summary edges, and vice versa. In our implementation, we use BDDs to represent transfer functions, path edges, and summary edges. As is usual with BDDs, a boolean expression e denotes the set of states $\Omega_e = \{\Omega | \Omega(e) = 1\}$. A set of pairs of states can easily be represented with a single BDD using primed versions of the variables in $V(B)$ to represent the variables in the second state. Since transfer functions, path edges, and summary edges are sets of pairs of states, we can represent and manipulate them using BDDs.

Upon termination of the algorithm, the set of path edges for a vertex v is empty iff v is not reachable. If v is reachable, we can generate a shortest trajectory to v, as described in the next section.

```
global
PathEdges, SummaryEdges, WorkList

procedure Propagate(v,p)
begin
  if p ⊈ PathEdges(v) then
    PathEdges(v) := PathEdges(v) ∪ p
    Insert v into WorkList fi
  fi
end

procedure Reachable(G_B)
begin
  for all v ∈ V_B do PathEdges(v) := {}
  for all v ∈ Call_B do SummaryEdges(v) := {}
  PathEdges(First_B(main)) :=
    {⟨Ω, Ω⟩ | Ω is any valuation to globals and local variables of main}

  WorkList := {First_B(main)}
  while WorkList ≠ ∅ do
    remove vertex v from WorkList
    switch (v)
      case v ∈ Call_B:
        Propagate(sSucc_B(v),SelfLoop(Join(PathEdges(v), Transfer_v)))
        Propagate(ReturnPt_B(v),Join(PathEdges(v), SummaryEdges(v)))
      case v ∈ Exit_B:
        for each w ∈ Succ_B(v) do
          let
            c ∈ Call_B such that w = ReturnPt_B(c) and
            s = Lift_c(PathEdges(v), ProcOf_B(v))
          in
            if s ⊈ SummaryEdges(c) then
              SummaryEdges(c) := SummaryEdges(c) ∪ s
              Propagate(w,Join(PathEdges(c), SummaryEdges(c)));
          ni
      case v ∈ Cond_B:
        Propagate(Tsucc_B(v), Join(PathEdges(v), Transfer_{v,true}))
        Propagate(Fsucc_B(v), Join(PathEdges(v), Transfer_{v,false}))
      case v ∈ V_B − Call_B − Exit_B − Cond_B:
        let p = Join(PathEdges(v), Transfer_v) in
          for each w ∈ Succ_B(v) do
            Propagate(w,p)
        ni
end
```

Fig. 3. The model checking algorithm.

v	$\mathit{Transfer}_v$
skip **print** **goto** **return**	$\lambda\langle\Omega_1,\Omega_2\rangle.(\Omega_2=\Omega_1)$
$x_1,\ldots,x_k :=$ $e_1,\ldots,e_k$	$\lambda\langle\Omega_1,\Omega_2\rangle.(\Omega_2=\Omega_1[x_1/\Omega_1(e_1))]\cdots[x_k/\Omega_1(e_k)])$
if(d) **while**(d) **assert**(d)	$\mathit{Transfer}_{v,true}=\lambda\langle\Omega_1,\Omega_2\rangle.((\Omega_1(d)=1)\wedge(\Omega_2=\Omega_1))$ $\mathit{Transfer}_{v,false}=\lambda\langle\Omega_1,\Omega_2\rangle.((\Omega_1(d)=0)\wedge(\Omega_2=\Omega_1))$
pr$(e_1,\ldots,e_k)$	$\lambda\langle\Omega_1,\Omega_2\rangle.(\Omega_2=\Omega_1[x_1/\Omega_1(e_1)]\ldots[x_k/\Omega_1(e_k)])$, where $x_1,\ldots,x_k$ are the formal parameters of **pr**.

Table 3. Transfer functions associated with vertices. These are derived directly from the transition rules given in Table 1

3.2 Generating a Shortest Trajectory to a Reachable Vertex

We now describe a simple extension to the algorithm of Figure 3 to keep track of the length of the shortest *hierarchical trajectory* needed to reach each state, so that if vertex v is reachable, we can produce a shortest initialized hierarchical trajectory that ends in v.

A hierarchical trajectory can "jump over" procedure calls using summary edges. Formally, a finite sequence $\overline{\eta}=\eta_0\xrightarrow{\alpha_1}_B\eta_1\xrightarrow{\alpha_2}_B\cdots\eta_{m-1}\xrightarrow{\alpha_m}_B\eta_m$ is a *hierarchical trajectory* of B if for all $0\leq i<m$, (1) either $\eta_i\xrightarrow{\alpha_i}_B\eta_{i+1}$, or $\eta_i=\langle v_i,\Omega_i\rangle$, $\eta_{i+1}=\langle v_{i+1},\Omega_{i+1}\rangle$, $\alpha_i=\sigma$, $v_i\in \mathit{Call}_B$ and $\langle\Omega_i,\Omega_{i+1}\rangle\in \mathit{SummaryEdges}(v_i)$, and (2) $\alpha_1\ldots\alpha_m\in\mathcal{L}(\mathcal{G}(B))$.

Let v be a vertex and let $e=\mathit{First}_B(\mathit{ProcOf}_B(v))$. For a path edge $\langle\Omega_e,\Omega_v\rangle\in \mathit{PathEdges}(v)$ let $W(\langle\Omega_e,\Omega_v\rangle)$ be the set of all hierarchical trajectories that start from **main**, enter into the procedure $\mathit{ProcOf}_B(v)$ with valuation Ω_e and then reach v with valuation Ω_v without exiting $\mathit{ProcOf}_B(v)$. Note that a hierarchical trajectory in $W(\langle\Omega_e,\Omega_v\rangle)$ is comprised of intraprocedural edges, summary edges, and edges that represent calling a procedure (but not the edges representing a return from a procedure). Instead of keeping all the path edges incident on v as a single set $\mathit{PathEdges}(v)$, we partition it into a set of sets

$$\{\mathit{PathEdges}_{r_1}(v),\mathit{PathEdges}_{r_2}(v),\ldots,\mathit{PathEdges}_{r_k}(v)\}$$

where a path edge $\langle\Omega_e,\Omega_v\rangle$ is in $\mathit{PathEdges}_{r_j}(v)$ iff the shortest hierarchical trajectory in $W((\langle\Omega_e,\Omega_v\rangle)$ has length r_j. The set $\{r_1,r_2,\ldots,r_k\}$ is called the set of *rings* associated with v.

We use rings to generate shortest hierarchical trajectories. If vertex v is reachable, we find the smallest ring r such that $\mathit{PathEdges}_r(v)$ exists.

Then we pick an arbitrary path edge $\langle \Omega_e, \Omega_v \rangle$ from $PathEdges_r(v)$, and do the following depending on the type of vertex v:

- If $v \neq First_B(ProcOf_B(v))$ then we have two subcases:
  - If s_v is not a **skip** immediately following a call, then we look for a predecessor u of v such that there exists path edge $\langle \Omega_e, \Omega_u \rangle$ in $PathEdges_{r-1}(u)$, and $Join(\{\langle \Omega_e, \Omega_u \rangle\}, Transfer_u)$ contains $\langle \Omega_e, \Omega_v \rangle$.
  - If s_v is a **skip** immediately following a call (say at vertex u), then we look for a path edge $\langle \Omega_e, \Omega_u \rangle$ in $PathEdges_{r-1}(u)$ such that $Join(\{\langle \Omega_e, \Omega_u \rangle\}, SummaryEdges(u))$ contains $\langle \Omega_e, \Omega_v \rangle$.
- If $v = First_B(ProcOf_B(v))$, then it should be the case that $e = v$, and $\Omega_v = \Omega_e$. We find a caller u of $ProcOf_B(v)$, and suitably "lift" Ω_v to a suitable path edge in $PathEdges(u)$. Formally, we find a vertex $u \in Call_B$ such that s_u is a call to procedure $ProcOf_B(v)$, and there exists path edge $\langle \Omega'_e, \Omega_u \rangle$ in $PathEdges_{r-1}(u)$ satisfying $Transfer_u(\langle \Omega_u, \Omega_v \rangle)$.

Repeating this process with the vertex u and the path edge found in $PathEdges_{r-1}(u)$, we are guaranteed to reach the entry of **main** in r steps. We may traverse over summary edges in the process. However, we can expand the summary edges on demand, to produce a hierarchical error trajectory, as shown in the `Bebop` output in Figure 1.

3.3 Optimizations

The basic algorithm described above has been implemented in the `Bebop` model checker. In this section, we describe a few optimizations based on ideas from compiler optimization [ASU86] that should substantially reduce the size of the BDDs needed to perform the analysis.

Live Ranges. If for some path starting at a vertex v in the control-flow graph G_B, the variable x is used before being defined, then variable x is said to be *live* at v. Otherwise, x is said to be *dead* at v, for its value at v will not flow to any other variable. If variable x is not live at vertex v then we need not record the value of x in the BDD for $PathEdges(v)$. Consider the following boolean program

```
void main()
begin
    decl a,b,c,d,e,f;
L1: a := b|c;     // {b,c,e} live at L1
L2: d := a|e;     // {a,e} live at L2
L3: e := d|e;     // {d,e} live at L3
L4: f := d;       // {d} live at L4
end
```

This program declares and refers to six variables, but at most three variables are live at any time. For example, at the statement labelled `L1` only the values of the variables b, c, and e can flow to the statements after `L1`. As a result, the BDD for the first statement need not track the values of the variables a or d.

MOD/REF sets. A traditional "MOD/REF" (modification/reference) analysis of a program determines the variables that are modified and/or referenced by each procedure **pr** (and the procedures it calls transitively). Let **pr** be a procedure in B such that **pr** nor any of the procedures it calls (transitively) modifies or references global variable g. Although g may be in scope in **pr**, and may in fact be live within **pr**, the procedure **pr** cannot change the value of g. As a result, all that is needed is to record that g remains unchanged, for any summary of **pr**.

4 Evaluation

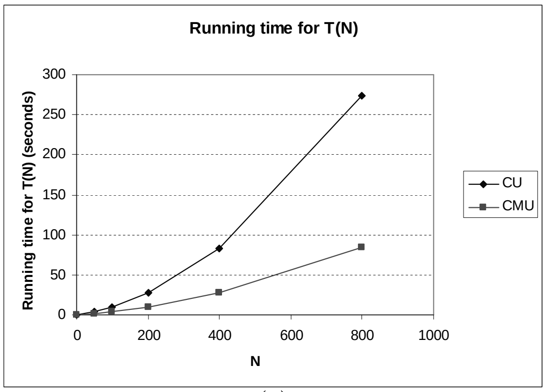
In this section, we present an evaluation of `Bebop` on a series of synthetic programs derived from the template T shown in Figure 4. The template allows us to generate boolean programs $T(N)$ for $N > 0$. The boolean program $T(N)$ has one global variable g and $N+1$ procedures —a procedure `main`, and N procedures of the form `level<i>` for $0 < i \leq N$. For $0 < j < N$, the two instances of `<stmt>` in the body of procedure `level<j>` are replaced by a call to procedure `level<j+1>`. The two instances of `<stmt>` in the body of procedure `level<N>` are replaced by `skip`.

As a result, a boolean program $T(N)$ has $N+1$ procedures, where `main` calls `level1` twice, `level1` calls `level2` twice, etc. At the beginning of each `level` procedure, a choice is made depending on the value of g. If g is 1 then a loop is executed that implements a three bit counter over the local variables a, b, and c. If g is 0 then two calls in succession are made to the next `level` procedure. In the last `level` procedure, if g is 0 then two **skip** statements are executed. At the end of each `level` procedure, the global variable g is negated. Every program $T(N)$ generated from this template has four variables visible at any program point, regardless of N. Note that g is not initialized, so `Bebop` will explore all possible values for g.

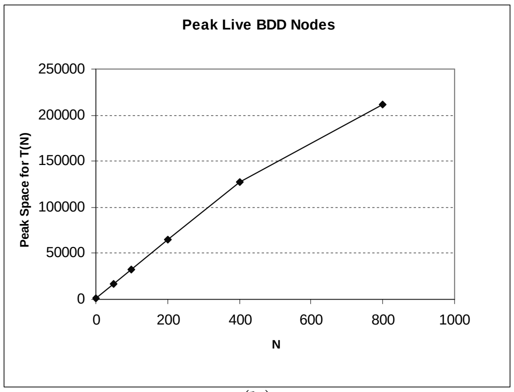
We ran `Bebop` to compute the reachable states for boolean programs $T(N)$ in $0 < N \leq 800$, and measured the running time, and peak memory used. Figure 4(a) shows how the running time of `Bebop` (in seconds) varies

```
decl g;
void main()
begin
  level1();
  level1();
  if(!g) then
    reach: skip;
  else
    skip;
  fi
end

void level<i>()
begin
  decl a,b,c;
  if(g) then
    a,b,c := 0,0,0;
    while(!a|!b|!c) do
      if (!a) then
        a := 1;
      elsif (!b) then
        a,b := 0,1;
      elsif (!c) then
        a,b,c := 0,0,1;
      fi
    od
  else
    <stmt>; <stmt>;
  fi
  g := !g;
end
```



(a)



(b)

Fig. 4. Boolean program template T for performance test and performance results.

with N. Figure 4(b) shows how the peak memory usage of `Bebop` varies with N.

The two curves in Figure 4(a) represent two different BDD packages: CU is the CUDD package from Colorado University [Som98] and CMU is the BDD package from Carnegie Mellon University [Lon93]. We note that the program $T(800)$ has 2401 variables. Model checking of this program takes a minute and a half with the CMU package and four and a half minutes with the CUDD package. Both times are quite reasonable considering the large number of variables (relative to traditional uses of BDDs). The space measurements in Figure 4(b) are taken from the CUDD package, which provides more detailed statistics of BDD space usage.

We expected the peak memory usage to increase linearly with N. The sublinear behavior observed in Figure 4(b) is due to more frequent garbage collection at larger N. We expected the running time also to increase linearly with N. However, Figure 4(a) shows that the running time increases quadratically with N. The quadratic increase in running time was unexpected, since the time complexity of model checking program $T(N)$ is $O(N)$ (there are 4 variables in the scope of any program point). By profiling the runs and reading the code in the BDD packages, we found that the quadratic behavior arises due to an inefficiency in the implementation of `bdd_substitute` in the BDD package. `Bebop` calls `bdd_substitute` in its "inner loop", since variable renaming is an essential component of its forward image computation. While model checking $T(N)$, the BDD manager has $O(N)$ variables, but we are interested in substituting $O(n)$ variables, for a small n, using `bdd_substitute`. Regardless of n, we found that `bdd_substitute` still consumes $O(N)$ time. Both the CUDD and CMU packages had this inefficiency. If this inefficiency is fixed in `bdd_substitute`, we believe that the running time of `Bebop` for $T(N)$ will vary linearly with N.

5 Related Work

Model checking for finite state machines is a well studied problem, and several model checkers —SMV [McM93], Murϕ [Dil96], SPIN [HP96], COSPAN [HHK96], VIS [BHSV$^+$96] and MOCHA [AHM$^+$98]— have been developed. Boolean programs implicitly have an unbounded stack, which makes them identical in expressive power to pushdown automata. The model checking problem for pushdown automata has been studied before [SB92] [BEM97] [FWW97]. Model checkers for push down automata have also been written before [EHRS00]. However, unlike boolean programs, these approaches abstract away data, and concentrate only on control. As a result spurious paths can arise in these models due to information loss about data correlations.

The connections between model checking, dataflow analysis and abstract interpretation have been explored before [Sch98] [CC00]. The RHS algorithm [RHS95,RHS96] builds on earlier work in interprocedural dataflow analysis from [KS92] and [SP81]. We have shown how this algorithm can be generalized to work as a model checking procedure for boolean programs. Also, our choice of hybrid representation of the state space in `Bebop`—an explicit representation of control flow and an implicit BDD-based representation of path edges and summary edges— is novel.

Exploiting design modularity in model checking has been recognized as a key to scalability of model checking [AH96] [AG00] [McM97] [HQR98]. The idea of harnessing the inherent modularity in procedural abstraction, and exploiting locality of variable scoping for efficiency in model checking software is new, though known in the area of dataflow analysis [RHS96]. Existing model checkers neither support nor exploit procedural abstraction. As a result, existing approaches to extract models from software are forced to inline procedure definitions at their points of invocation [CDH+00], which could lead to explosion in both the size of the model and the number of variables.

6 Future Work

`Bebop` is part of a larger effort called SLAM[6], in progress at Microsoft Research, to extract abstract models from code and check temporal properties of software. We are currently implementing a methodology that uses boolean programs, and an iterative refinement process using path simulation to model check critical portions of operating system code [BR00].

References

[AG00] A. Alur and R. Grosu. Modular refinement of hierarchic reactive modules. In *POPL 00: Principles of Programming Languages*. ACM Press, 2000.

[AH96] R. Alur and T.A. Henzinger. Reactive modules. In *LICS 96: Logic in Computer Science*, pages 207–218. IEEE Computer Society Press, 1996.

[AHM+98] R. Alur, T.A. Henzinger, F.Y.C. Mang, S. Qadeer, S.K. Rajamani, and S. Tasiran. MOCHA : Modularity in model checking. In *CAV 98: Computer Aided Verification*, LNCS 1427, pages 521–525. Springer-Verlag, 1998.

[ASU86] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, MA, 1986.

[BEM97] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *CONCUR 97: Concurrency Theory*, LNCS 1243, pages 135–150. Springer-Verlag, 1997.

[BHSV+96] R.K. Brayton, G.D. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.-T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R.K. Ranjan, S. Sarwary, T.R. Shiple, G. Swamy, and T. Villa. VIS: A System for Verification and Synthesis. In *CAV 96: Computer Aided Verification*, LNCS 1102, pages 428–432. Springer-Verlag, 1996.

[BR00] T. Ball and S. K. Rajamani. Boolean programs: A model and process for software analysis. Technical Report MSR-TR-2000-14, Microsoft Research, February 2000.

[Bry86] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[6] `http://www.research.microsoft.com/slam/`

[CC00] P. Cousot and R. Cousot. Temporal abstract interpretation. In *POPL 00: Principles of Programming Languages*. ACM Press, 2000.

[CDH+00] James Corbett, Matthew Dwyer, John Hatcliff, Corina Pasareanu, Robby, Shawn Laubach, and Hongjun Zheng. Bandera : Extracting finite-state models from java source code. In *ICSE 2000: International Conference on Software Engineering*, 2000.

[Dil96] D. L. Dill. The Murϕ Verification System. In *CAV 96: Computer Aided Verification*, LNCS 1102, pages 390–393. Springer-Verlag, 1996.

[EHRS00] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. Technical Report TUM-I0002, SFB-Bericht 342/1/00 A, Technische Universitat Munchen, Institut fur Informatik, February 2000.

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

[HHK96] R.H. Hardin, Z. Har'El, and R.P. Kurshan. COSPAN. In *CAV 96: Computer Aided Verification*, LNCS 1102, pages 423–427. Springer-Verlag, 1996.

[HP96] G.J. Holzmann and D.A. Peled. The State of SPIN. In *CAV 96: Computer Aided Verification*, LNCS 1102, pages 385–389. Springer-Verlag, 1996.

[HQR98] T.A. Henzinger, S. Qadeer, and S.K. Rajamani. You assume, we guarantee: methodology and case studies. In *CAV 98: Computer Aided Verification*, LNCS 1427, pages 440–451. Springer-Verlag, 1998.

[KS92] J. Knoop and B. Steffen. The interprocedural coincidence theorem. In *CC 92: Compiler Construction*, LNCS 641, pages 125–140, Springer-Verlag, 1992.

[Lon93] D. Long. Cmu bdd package. `http://emc.cmu.edu/pub`, Carnegie Melon University, 1993.

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

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

[RHS95] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL 95: Principles of Programming Languages*, pages 49–61. ACM Press, 1995.

[RHS96] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167:131–170, 1996.

[SB92] B. Steffen and O. Burkart. Model checking for context-free processes. In *CONCUR 92: Concurrency Theory*, LNCS 630, pages 123–137. Springer-Verlag, 1992.

[Sch98] D.A. Schmidt. Data flow analysis is model checking of abstract interpretation. In *POPL 98: Principles of Programming Languages*, pages 38–48. ACM Press, 1998.

[Som98] F. Somenzi. Colorado university decision diagram package. `ftp://vlsi.colorado.edu/pub`, University of Colorado, Boulder, 1998.

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