Analysis of Boolean Programs

Patrice Godefroid
Microsoft Research
pg@microsoft.com

Mihalis Yannakakis
Columbia University
mihalis@cs.columbia.edu

Abstract

Boolean programs are a popular abstract domain for static-analysis-based software model checking. Yet little is known about the complexity of model checking for this model of computation. This paper aims to fill this void by providing a comprehensive study of the worst-case complexity of several basic analyses of Boolean programs, including reachability analysis, cycle detection, LTL, CTL, and CTL* model checking. We present algorithms for these problems and show that our algorithms are all *optimal* by providing matching lower bounds. We also identify particular classes of Boolean programs which are easier to analyse, and compare our results to prior work on context-free and pushdown model checking.

1. Introduction

Boolean programs are programs in which all variables have Boolean type and which can contain recursive procedures. They are a popular abstract domain for static-analysis-based software model checking, pioneered by the SLAM project [4]. SLAM verifies control-flow dominated properties of Windows device drivers by abstracting a C program with a Boolean program generated using *predicate abstraction* (e.g., [20]). The Boolean program contains the same procedures and control flow as the original program, but uses Boolean variables to keep track of the values of predicates over variables of the original program, abstracting its "data part". The level of abstraction can be adjusted iteratively and automatically by changing the finite set of predicates being tracked, using a process sometimes called "Counter-Example Guided Abstraction Refinement" (CEGAR). Since SLAM, other tools have adopted Boolean programs as an abstract domain for software model checking, such as BLAST [22], YASM [21], TERMINATOR [15] and YOGI [19].

The main advantage of Boolean programs compared to finite-state transition systems is that their stack allows a *precise* representation of procedure calls, including recursion, while providing a model of computation for which many interesting properties are still *decidable*. Indeed, Boolean programs have the same expressiveness as pushdown systems [3], for which many properties of interest, such as reachability and temporal-logic model checking, are decidable [7], even though their set of reachable states can be infinite.

Several algorithms for reachability analysis of Boolean programs have been proposed in the literature. For instance, [3] discusses a symbolic model checker for safety properties (reachability analysis) using BDDs as procedure summaries. [17] extends the previous results to Linear Temporal Logic (LTL) model checking, which can also check liveness properties with fairness constraints. [24] discusses how to reduce reachability analysis of Boolean programs to SAT solving. More recently, [6] investigates how to use SAT encodings, instead of BDDs, to represent procedures summaries and to use a QBF solver for reachability analysis.

Yet, despite this prior work, little is known about the complexity of model checking for Boolean programs. Indeed, all the algorithms for analyzing Boolean programs discussed in prior work run in time exponential in the size of the Boolean program, or worse – sometimes runtime complexity is discussed explicitly, sometimes such a discussion is omitted altogether. Moreover, no lower bounds are discussed in prior work on analyzing Boolean programs, to the best of our knowledge.

In contrast, the complexity of model checking for pushdown automata, context-free processes and recursive state machines has been studied extensively in the literature (e.g., [2, 7, 8]). However, Boolean programs can be exponentially more succinct than ordinary pushdown systems or recursive state machines. Therefore, the program complexity of model checking for Boolean programs does not follow directly from prior work on model checking for pushdown systems.

This paper aims to fill this void by providing a comprehensive study of the worst-case complexity of several basic analyses of Boolean programs, including reachability analysis, cycle detection, LTL, CTL and CTL* model checking. We present algorithms (upper bounds) as well as matching lower bounds for all those problems. In other words, all the algorithms presented in this paper are *optimal* in the complexity-theoretic sense. For instance, we show that reachability analysis for Boolean programs is EXPTIME-complete. We also study precisely the program complexity of LTL, CTL and CTL* model checking. Moreover, we identify particular classes of Boolean programs which are easier to analyse.

This paper is organized as follows. In Section 2, we formally define Boolean programs and compare them to other models of computation. In Section 3, we study the complexity of reachability analysis for Boolean programs. We also identify particular program classes for which the complexity is lower, illustrating how various features of Boolean programs contribute to the overall problem complexity. We then discuss cycle detection and LTL model checking in Section 4. In Section 5, we turn to the complexity of model checking for branching-time properties expressed in the temporal logics CTL and CTL*. Section 6 summarizes and discusses insights gained by this work. We conclude in Section 7.

2. Boolean Programs

Boolean programs are imperative programs with the usual constructs of languages like C, that have Boolean variables, and which can use nondeterminism and recursion. [4] describes in detail their syntax and defines their semantics using their control flow graphs.

Boolean programs are essentially recursive state machines extended with a finite set of Boolean variables. Therefore, we will use the terms "Boolean program" and "Extended Recursive State Machine" (ERSM) interchangeably in this paper.

2.1 Syntax

Formally, a (Boolean) *Extended Recursive State Machine (ERSM)* A over a finite alphabet Σ is defined by a tuple $\langle A_1, \ldots, A_k, V \rangle$, where V is a finite set of global Boolean variables and each *procedure* A_i consists of the following pieces:

- A finite set V_i of Boolean variables that are local to the procedure A_i, a tuple $V_i^{in} \subseteq V_i$ of *input variables* and a tuple $V_i^{out} \subseteq V_i$ of *output variables*.
- A finite set N_i of *nodes* and a (disjoint) finite set B_i of *boxes*, or *call sites*.
- A labeling $Y_i : B_i \mapsto \{1, \ldots, k\}$ that assigns to every box an index of one of the procedures (component machines), $A_1, \ldots, A_k$, and a pair of mappings $\beta_i^{in}, \beta_i^{out}$ which assign to each box $b \in B_i$ two tuples $\beta_i^{in}(b), \beta_i^{out}(b)$ of variables in V_i that are respectively the input and output arguments of the recursive call represented by the box b, where $|\beta_i^{in}(b)| = |V_{Y_i(b)}^{in}|$ and $|\beta_i^{out}(b)| = |V_{Y_i(b)}^{out}|$.
- A set of *entry* nodes $En_i \subseteq N_i$, and a set of *exit* nodes $Ex_i \subseteq N_i$.
- A *transition relation* δ_i, where transitions are of the form (u, G, σ, C, v) where (1) the source u is either a node of $N_i \setminus Ex_i$, or a pair (b, x), where b is a box in B_i and x is an exit node in Ex_j for $j = Y_i(b)$; (2) the guard G is a Boolean predicate on the variables in $V_i \cup V$; (3) the label σ is in Σ; (4) the command C assigns new Boolean values to the variables in $V_i \cup V$ as a function of the old values; and (5) the destination v is either a node in N_i or a pair (b, e), where b is a box in B_i and e is an entry node in En_j for $j = Y_i(b)$.

We will use the term *ports* to refer to pairs $(b, e), (b, x)$ consisting of a box b of a procedure A_i and corresponding entry nodes e and exit nodes x of the procedure A_j called by b. We will use the term *vertices* of A_i to refer to its nodes and the ports of its boxes that participate in some transition. We will often refer to a vertex (b, e) as a *call vertex* and (b, x) as a *return vertex*.

We define the *size* $|A|$ of an ERSM A to be the sum of the total numbers of nodes, boxes, transitions and variables of A.

Remarks: 1. In the above definition we have allowed procedures to have multiple entries (initial nodes) and exits (final nodes). In the presence of variables, this is strictly speaking not necessary, i.e., ERSMs where every procedure has a single entry and exit are equally expressive, because we can use extra input and output variables to specify different entries and exits. In fact, in a straightforward translation of the code of a Boolean program to an ERSM, the procedures will have a single entry and exit. A statement like $x := A_j(y)$ in a procedure A_i corresponds to a box b with $Y_i(b) = j$, $\beta_i^{in}(b) = y$, and $\beta_i^{out}(b) = x$. We have allowed multiple entries and exits here for consistency with the definition of standard RSMs that do not have variables [2], where the multiplicity of entries and exits is essential.
2. It is convenient syntactically for procedures to receive inputs and return outputs, although in the presence of global variables it is not really essential to have explicitly input and output variables: a value passed as argument to a procedure can be modeled using a global variable which is assigned the argument value just before the procedure call and then copied immediately after the start of the called procedure into a local variable of that procedure. Similarly, a return value of a procedure can be modeled with a global variable which is assigned the return value just before the return and then copied immediately after the return into the local state of the calling procedure.
3. The syntax of the guards and commands of the transitions in the definition is left flexible. For the complexity upper bounds, we assume that the guards and commands are arbitrary predicates and functions respectively that can be evaluated in polynomial time. For the lower bound constructions, the guards are simple equality conditions, and the commands are simple assignments.
4. In the above definition, all variables are Boolean. More generally, we could define ERSMs whose variables have other domains. If all the variables have finite domains then we can clearly encode them with Boolean variables, and the results of the paper apply.

2.2 Semantics

To define the executions of ERSMs, we first define the global states and transitions associated with an ERSM. Let $\vec{X}$ denote a mapping that associates a value to each variable in a set X of variables. We assume all Boolean variables have a unique default initial value. A (global) *state* of an ERSM $A = \langle A_1, \ldots A_k, V \rangle$ is a tuple $\langle (b_1, \vec{V_1}), \ldots, (b_r, \vec{V_r}), (u, \vec{V_{r+1}}, \vec{V}) \rangle$ where $b_1, \ldots, b_r$ are boxes, $\vec{V_1}, \ldots, \vec{V_r}, \vec{V_{r+1}}$ are value assignments to local variables, u is a node, and $\vec{V}$ assigns a value to every global variable. Equivalently, a state can be viewed as a string, and the set Q of global states of A is $(B \times \vec{V'})^*(N \times \vec{V'} \times \vec{V})$, where $B = \cup_i B_i$, $V' = \cup_i V_i$ and $N = \cup_i N_i$. Consider a state $\langle (b_1, \vec{V_1}), \ldots, (b_r, \vec{V_r}), (u, \vec{V_{r+1}}, \vec{V}) \rangle$ such that $b_i \in B_{j_i}$ for $1 \leq i \leq r$ and $u \in N_j$. Such a state is *well-formed* if $Y_{j_i}(b_i) = j_{i+1}$ and $V_i = V_{j_i}$ for $1 \leq i < r$, and if $Y_{j_r}(b_r) = j$ and $V_{r+1} = V_j$. A well-formed state of this form corresponds to the case when the control is inside the component A_j, which was entered via box b_r of component A_{j_r} (the box b_{r-1} gives the context in which A_{j_r} was entered, and so on). Henceforth, we assume states to be well-formed.

We assume a call-by-value model for the procedure calls. We define a (global) transition relation δ among the global states of A as follows. Let $s = \langle (b_1, \vec{V_1}), \ldots, (b_r, \vec{V_r}), (u, \vec{V_{r+1}}, \vec{V}) \rangle$ be a state with $u \in N_j$ and $b_r \in B_m$. Then, $(s, \sigma, s') \in \delta$ iff one of the following holds:

1. $(u, G, \sigma, C, u') \in \delta_j$ for a node u' of A_j, $G(\vec{V_{r+1}}, \vec{V})$ evaluates to true, $C(\vec{V_{r+1}}, \vec{V}) = (\vec{V_{r+1}}', \vec{V'})$, and $s' = \langle (b_1, \vec{V_1}), \ldots, (b_r, \vec{V_r}), (u', \vec{V_{r+1}}', \vec{V'}) \rangle$.
2. $(u, G, \sigma, C, (b', e)) \in \delta_j$ for a box b' of A_j, $G(\vec{V_{r+1}}, \vec{V})$ evaluates to true, $C(\vec{V_{r+1}}, \vec{V}) = (\vec{V_{r+1}}', \vec{V'})$, and $s' = \langle (b_1, \vec{V_1}), \ldots, (b_r, \vec{V_r}), (b', \vec{V_{r+1}}'), (e, \vec{V_{r+2}}', \vec{V'}) \rangle$, where $\vec{V_{r+2}}'$ denotes an initial value assignment for the local variables in $V_{Y_j(b')}$ of the procedure corresponding to box b', in which the input variables $V_{Y_j(b')}^{in}$ have value equal to the value of the variables $\beta_j^{in}(b')$ in $\vec{V_{r+1}}'$.
3. u is an exit-node of A_j, $((b_r, u), G, \sigma, C, u') \in \delta_m$ for a node u' of A_m, $\hat{V_r}$ is the assignment to the local variables of A_m in which the variables of $\beta_m^{out}(b_r)$ have value equal to that of the output variables V_j^{out} of A_j in $\vec{V_{r+1}}$ and the rest of the variables have the same value as in $\vec{V_r}$, $G(\hat{V_r}, \vec{V})$ evaluates to true, $C(\hat{V_r}, \vec{V}) = (\vec{V_r}', \vec{V'})$, and $s' = \langle (b_1, \vec{V_1}), \ldots, (b_{r-1}, \vec{V_{r-1}}), (u', \vec{V_r}', \vec{V'}) \rangle$.
4. u is an exit-node of A_j, $((b_r, u), G, \sigma, C, (b', e)) \in \delta_m$ for a box b' of A_m, $\hat{V_r}$ is the assignment to the local variables of A_m in which the variables of $\beta_m^{out}(b_r)$ have value equal to that of the output variables V_j^{out} of A_j in $\vec{V_{r+1}}$

```
int[8] x; // 8-bit integer global variable

procedure foo()  // Boolean Program 1
{
  print(``a'');
  if (x>0)
  {
    x = x-1;
    foo();
  }
  print(``b'');
  return;
}

procedure bar(int[8] y)  // Boolean Program 2
{
  print(``a'');
  if (y>0)
  {
    bar(y-1);
  }
  print(``b'');
  return;
}
```

Figure 1. Simple examples of Boolean programs.

and the rest of the variables have the same value as in $\vec{V_r}$, $G(\hat{V_r}, \vec{V})$ evaluates to true, $C(\hat{V_r}, \vec{V}) = (\vec{V_r}', \vec{V'})$, and $s' = \langle (b_1, \vec{V_1}), \ldots, (b_{r-1}, \vec{V_{r-1}}), (b', \vec{V_r}'), (e, \vec{V_{r+1}}', \vec{V'}) \rangle$, where $\vec{V_{r+1}}'$ denotes an initial value assignment for the local variables in $V_{Y_m(b')}$ of the procedure corresponding to box b', in which the input variables $V^{in}_{Y_m(b')}$ have value equal to the value of the variables $\beta^{in}_m(b')$ in $\vec{V_r}'$.

Case 1 is when the control stays within the component A_j, case 2 is when a new component is entered via a box of A_j, case 3 is when the control exits A_j and returns back to A_m, and case 4 is when the control exits A_j and enters a new component via a box of A_m.

The *Labeled Transition System (LTS)* $T_A = (Q, \Sigma, \delta)$ is called the *"unfolding" of* A. The set Q of reachable states can be infinite. Given a state $\langle (b_1, \vec{V_1}), \ldots, (b_r, \vec{V_r}), (u, \vec{V_{r+1}}, \vec{V}) \rangle$, we will sometimes refer to $\langle (b_1, \vec{V_1}), \ldots, (b_r, \vec{V_r}) \rangle$ as the *call stack*, or stack, in that state. For a state s of the LTS T_A and a node v of A, $s \Rightarrow v$ denotes that s can reach some state $\langle (b_1, \vec{V_1}), \ldots, (b_r, \vec{V_r}), (v, \vec{V_{r+1}}, \vec{V}) \rangle$ in T_A whose node is v.

2.3 Examples

In what follows, we will represent ERSMs using pseudo-code, as illustrated in Figure 1. For notational convenience, we will also sometimes use non-boolean variables with bounded domains. For example, the Boolean program composed of procedure `foo` shown in Figure 1 uses an 8-bit integer variable named `x`. Given an initial value n for variable `x`, this program will print the string $a^n b^n$ using n recursive calls to procedure `foo`. Similarly, the second Boolean program composed of procedure `bar` in Figure 1 will also print the string $a^n b^n$ given an initial call to `bar` with value n as argument. These two Boolean programs represent similar ERSMs, with 8 Boolean variables each. The only difference is that the first one uses a global variable `x` while the second uses a local variable `y`.

By definition, the semantics for local and global variables are different: the values of local variables are saved in the stack at each procedure call (they multiply the "stack alphabet") while the values of global variables are never stored on the stack (they multiply the "set of control states").

2.4 Special Cases

ERSMs generalize several other well-known models of computation.

- A *Recursive State Machine (RSM)* is an ERSM with no Boolean variables, i.e., where V and the sets V_i are all empty, the guards G are all vacuously $true$, and the commands C do not modify the value of any variable.
- An *Extended Hierarchical State Machine (EHSM)* is an ERSM with no cycle of recursive calls between the procedures, i.e., where every procedure A_i can only call a procedure A_j with $j > i$, i.e., we have $\forall i : \forall b \in B_i : Y_i(b) > i$.
- A *Hierarchical State Machine (HSM)* is an EHSM with no Boolean variables.
- An *Extended Finite State Machine (EFSM)* is an ERSM (or EHSM) with a single procedure A_1 and no boxes.
- A *Finite State Machine (FSM)* is an EFSM with no Boolean variables.

A procedure or machine A_i is called *single-entry* when it has a single entry node e, i.e., when $En_i = \{e\}$. Similarly, a procedure or machine A_i is called *single-exit* when it has a single exit node x, i.e., when $Ex_i = \{x\}$. An ERSM is single-entry or single-exit if all its procedures are. As mentioned earlier, any ERSM can be transformed to an equivalent single-entry, single-exit ERSM by introducing additional variables. This is not the case for RSMs.

A procedure is called *input/output bounded*, or *I/O bounded* for short, if the number of its input and output variables, the number of entries and exits, and the number of global variables are all upper bounded by a fixed constant (independent of the size of the program). A program (ERSM) is I/O bounded if all its procedures are I/O bounded. This property characterizes programs where there is a constant amount of information communicated between the different procedures.

A procedure A_i is called *acyclic* if the graph $(N_i \cup B_i, E_i)$ is acyclic, where E_i contains an edge from a node u or box b to another node u' or box b' iff δ_i contains a transition from u or a vertex of b to u' or a vertex of b' (regardless of the guard and command of the transition). An ERSM is acyclic iff all its procedures are.

A procedure is called *deterministic* if, for all its vertices, the guards of all its transitions at that vertex are mutually exclusive. In that case, each state of that procedure can have at most one successor state. A program is deterministic if all its procedures are deterministic. Usual programs (without abstraction) are deterministic. This is the case for the two Boolean programs shown in Figure 1.

2.5 Expansion of an ERSM

Given an ERSM $A = \langle A_1, \ldots A_k, V \rangle$, we can construct an RSM $A' = \langle A'_1, \ldots A'_k \rangle$ (without variables) that is equivalent to A, in the sense that their unfoldings T_A and $T_{A'}$ are identical. The RSM A' is in general exponentially larger than A. In particular, if $m = \max_i |V \cup V_i|$ then the size $|A'|$ of the RSM A' is (at most) $|A| \cdot 2^m$.

The translation from an ERSM A to a RSM A' is similar to the standard translation from EFSMs to FSMs, except that call sites and returns have to be taken into account. Let A be an ERSM as previously defined. Define the *scope* of a procedure A_i to be $\text{scope}(A_i) = V \cup V_i$. The RSM A' has one procedure A'_i for each procedure A_i of A. The node set N'_i of A'_i is the Cartesian product of the node set N_i of A_i and the set of valuations of all variables in $\text{scope}(A_i)$. The set of entries En'_i is the Cartesian product of the entry set En_i of A_i and the set of valuations for all variables in scope where all non-input local variables are assigned their initial value. The set Ex'_i of exits is the product of the exit set Ex_i of A_i

and the set of valuations of all variables in scope(A_i). The set of boxes B_i' of A_i' is the product of B_i and the set of valuations of V_i (only the local variables).

The transition relation δ_i' of A_i' is generated from the transition relation δ_i of A_i as follows. Let (u, G, σ, C, v) be a transition of δ_i. We distinguish cases depending on whether u, v are nodes or ports of a box.

Case 1. If $u, v \in N_i$, and η, θ are valuations of scope(A_i) such that $G(\eta)$ is true and $C(\eta) = \theta$, then $((u, \eta), \sigma, (v, \theta))$ is a transition of δ_i'.

Case 2. Suppose $u \in N_i$ and $v = (b, e)$ is a call vertex of a box b that is mapped to A_j. Then δ_i' includes a transition $((u, \eta), \sigma, ((b, \eta'), (e, \theta)))$ provided the following hold: η is a valuation of $V \cup V_i$ such that $G(\eta)$ is true, $\hat{\theta} = C(\eta)$, η' is a valuation of V_i such that $\eta'[V_i] = \hat{\theta}[V_i]$, and θ is a valuation of $V \cup V_j$ such that $\theta[V] = \hat{\theta}[V]$, $\theta[V_j^{in}] = \hat{\theta}[\beta_i^{in}(b)]$, and θ assigns to the non-input local variables of A_j their initial values.

Case 3. Suppose $u = (b, x)$ is a return vertex of a box b that is mapped to A_j, and $v \in N_i$ is a node. Then δ_i' includes a transition $(((b, \eta), (x, \eta')), \sigma, (v, \theta))$ provided the following hold: η is a valuation of V_i, η' is a valuation of $V \cup V_j$, $\hat{\eta}$ is a valuation of $V \cup V_i$ such that $\hat{\eta}[V] = \eta'[V]$, $\hat{\eta}[\beta_i^{out}(b)] = \eta'[V_j^{out}]$, and $\hat{\eta}[V_i \setminus \beta_i^{out}(b)] = \eta[V_i \setminus \beta_i^{out}(b)]$, $G(\hat{\eta})$ is true, and $\theta = C(\hat{\eta})$.

Case 4. Suppose $u = (b, x)$ is a return vertex of a box b mapped to A_j, and $v = (b', e)$ is a call vertex of a box b' mapped to A_l. Then δ_i' includes a transition $(((b, \eta), (x, \eta')), \sigma, ((b', \theta)(e, \theta')))$ provided the following hold: η is a valuation of V_i, η' is a valuation of $V \cup V_j$, $\hat{\eta}$ is a valuation of $V \cup V_i$ such that $\hat{\eta}[V] = \eta'[V]$, $\hat{\eta}[\beta_i^{out}(b)] = \eta'[V_j^{out}]$, and $\hat{\eta}[V_i \setminus \beta_i^{out}(b)] = \eta[V_i \setminus \beta_i^{out}(b)]$, $G(\hat{\eta})$ is true, $\hat{\theta} = C(\hat{\eta})$, θ is a valuation of V_i such that $\theta[V_i] = \hat{\theta}[V_i]$, and θ' is a valuation of $V \cup V_l$ such that $\theta'[V] = \hat{\theta}[V]$, $\theta'[V_l^{in}] = \hat{\theta}[\beta_i^{in}(b')]$, and θ' assigns to the non-input local variables of A_l their initial values.

We call A' the *expanded* RSM corresponding to A. It is easy to verify the following from the definitions.

PROPOSITION 1. *The ERSM A and the RSM A' have the same unfoldings, $T_A = T_{A'}$.*

3. Reachability

Let $Init$ denote a given set of initial states, consisting of some entry nodes together with specified valuations for the variables in the scope of their procedures. Given an ERSM $A = \langle A_1, \ldots A_k, V \rangle$ and such a set $Init$, let $Init \Rightarrow v$ denote that for some $s \in Init$, $s \Rightarrow v$. Our goal in simple reachability analysis is to determine whether a specific target node t is in the set $\{v \mid Init \Rightarrow v\}$ of reachable vertices. In this section, we study the complexity of the reachability analysis problem for ERSMs and several special cases.

3.1 General Case

THEOREM 2. *Reachability analysis for ERSMs is EXPTIME-complete. Furthermore, this holds even for deterministic, acyclic ERSMs.*

Proof: Membership in EXPTIME follows essentially from previous work (e.g., [2, 3]). Given a ERSM A, we can construct the corresponding expanded RSM A', which has size (at most) exponential in A. Since reachability analysis for RSMs can be solved in polynomial time (cubic in the general case, and linear for single-entry or single-exit RSMs to be precise [2]), we obtain an algorithm with EXPTIME complexity overall. (The expansion and reachability analysis can be done together, on the fly, so that only the reachable parts of A' are generated.)

```
procedure Top()
{
  if Acc(q0, 0, Initial Tape)
     then print(``M accepts'');
}

bool Acc(state q, head location h, Tape T)
{
  if (q in QT) then return true;
  if (q in QF) then return false;

  bool res;
  if (q in Q∃) then res = false;
  else res = true; // case (q in Q∀)

  for each (q',s,D) in δM(q,T[h])
  {
    compute the new tape location h' and tape T';
    if (q in Q∃) then res = res ∨ Acc(q',h',T');
    else res = res ∧ Acc(q',h',T');
  }
  return res;
}
```

Figure 2. A Boolean program simulating an alternating PSPACE machine M.

For the hardness part, we reduce the acceptance problem for 1-tape alternating polynomial space machines, which is known to be EXPTIME-complete [10], to reachability analysis of ERSMs. An (1-tape) alternating PSPACE machine M is a tuple $(\Sigma, Q, q_0, \delta_M)$ where Σ is a finite tape alphabet, $Q = Q_\forall \cup Q_\exists \cup Q_T \cup Q_F$ is a finite set of states, where $Q_\forall$ denotes a finite set of *universal* states, $Q_\exists$ denotes a finite set of *existential* states, Q_T denotes a finite set of *accepting* states, and Q_F denotes a finite set of *rejecting* states, the sets $Q_\forall, Q_\exists, Q_T, Q_F$ are all disjoint, q_0 is an initial state in Q, and δ_M is the transition relation which maps every pair (q, a) with $q \in Q_\forall \cup Q_\exists$ and $a \in \Sigma$ (q is the current state and a the symbol in the cell under the tape head) to one or more tuples of the form (q', a', D), where $q' \in Q$ is the next state, $a' \in \Sigma$ is the symbol written in the tape cell under the tape head (the other cells are unchanged), and $D \in \{L, R\}$ is the direction of the next movement of the tape head (one step left or one step right). A *configuration* of M is a tuple (q, h, T) where q is the state, h is the location of the tape head and T the contents of the tape. In the initial configuration, the state is q_0, the head is at cell 0 and the initial tape content T_0 consists of the given input x followed by blanks. The acceptance problem for an alternating PSPACE machine M is: Given an input x, decide whether M accepts x. It is known that this problem is EXPTIME-complete even for 1-tape linear-space bounded machines.

Given any (1-tape) alternating machine M that uses space $c \cdot n$ and input x of length n we can build, in polynomial time (in fact, linear) in the size of M and x, a Boolean program A that simulates M on input x precisely. Figure 2 shows what such a Boolean program looks like. The program uses variables to keep track of the state, the location of the head and the contents of the tape. Initially, the program starts by executing procedure `Top`, which then calls procedure `Acc` with the initial configuration $(q_0, 1, T_0)$ as arguments. The procedure $Acc(q, h, T)$ determines whether the machine M accepts when it starts at state q, with the head located at cell h and the tape contents given by T. In procedure `Acc`, the boolean value $true$ is returned if the current state q is accepting, and $false$ is returned if state q is rejecting. Otherwise, the procedure `Acc` is recursively called with each successor configuration (q', h', T') of (q, h, T) as defined by each tuple

$(q', s, D) \in \delta_M(q, T[h])$ of the transition relation, i.e., $h' = h - 1$ if $D = L$, $h' = h + 1$ if $D = R$, $T'[h] = s$ and $T'[j] = T[j]$ for all $j \neq h$. If $q \in Q_\exists$, `Acc` returns the disjunction, and if $q \in Q_\forall$ the conjunction, over the results obtained from all the recursive calls. The Boolean program shown in the figure is generic: for a particular M and x, a particular instantiation of A can be generated in time $O(|M|n)$, where generic statements like $q \in Q_T$, "for each $(q', T') \in \delta_M(q, T)$", the computation of the new location h' and tape T' etc. will be explicitly expanded in explicit switch/case statements. The constructed program has size $O(|M|n)$. It is easy to see that the Boolean program A will reach the statement print "M accepts" if and only if M has an accepting computation.

The Boolean program of Figure 2 is deterministic and acyclic, so these features do not make any difference in the complexity of ERSM reachability analysis. ■

Note that the procedure Acc in the program of Figure 2 is recursive and passes a linear amount of information in each call. We now show that restricting the use of recursion or the amount of I/O information reduces the complexity to a lower class.

3.2 Special Cases

In the hierarchical case, reachability analysis becomes PSPACE-complete, thus, no worse than simple EFSMs.

THEOREM 3. *Reachability analysis for EHSMs is PSPACE-complete. Furthermore, the problem remains PSPACE-complete for deterministic, acyclic EHSMs.*

Proof: Membership in PSPACE follows from nondeterministically simulating a computation that reaches the target node. In a EHSM $A = \langle A_1, \ldots A_k, V \rangle$, the stack of any computation has at all times polynomial size: the stack consists of a stack of records, one for each procedure that is currently active. The number of active procedures at any time is at most the height k of the hierarchy (so linear), and each record consists of a program location (a box) and the values of the local variables of the procedure at the time it made the recursive call, so all this information can be recorded in polynomial space. Therefore, the reachability problem is in Nondeterministic PSPACE, which is the same as PSPACE by Savitch's theorem.

It is known that reachability analysis is already PSPACE-hard for EFSMs. Hence, PSPACE-hardness for the more general EHSMs follows immediately. We show now that the problem remains PSPACE-complete for EHSMs that are deterministic and acyclic (whereas the problem is in NP for acyclic EFSMs, see below, and hence probably not PSPACE-complete). For this purpose, we reduce Quantified Boolean Formula (QBF) satisfiability (QSAT), known to be PSPACE-complete, to EHSM reachability.

Figure 3 shows a deterministic acyclic hierarchical Boolean program for checking the satisfiability of a QBF formula ψ of the form $\exists x_1 \forall x_2 \exists x_3 \ldots Q x_n \phi(x_1, \ldots, x_n)$. The program has $n+1$ procedures named `SAT[i]` for $0 \leq i \leq n$, and a main top-level procedure `Top`. Each of the `SAT[i]` procedures takes i Boolean arguments as inputs. For $i = n$, `SAT[n]`$(x_1, \ldots, x_n)$ evaluates ϕ on its input and returns the value. For odd $i < n$, `SAT[i]`$(x_1, \ldots, x_i)$ calls `SAT[i+1]` with $(x_1, \ldots, x_i, 0)$ and $(x_1, \ldots, x_i, 1)$, and returns the conjunction $\wedge$ of their values, since x_{i+1} is universally quantified in the given QBF formula ψ. Similarly, for even $i < n$, `SAT[i]`$(x_1, \ldots, x_i)$ calls `SAT[i+1]` with $(x_1, \ldots, x_i, 0)$ and $(x_1, \ldots, x_i, 1)$, and returns the disjunction $\vee$ of their values. Given any QBF formula ψ, we can build such a Boolean program in polynomial time in the size of the formula. It is easy to see that the Boolean program will reach the statement print "ψ is SAT" if and only if the formula ψ is satisfiable. ■

If we are given a bound on the depth d of the hierarchy, then the complexity of reachability analysis for acyclic EHSMs is reduced further to NP-complete.

```
procedure Top()
{
  if SAT[0]()
     then print(``ψ is SAT'');
}

bool SAT[n](bool x_1,...,x_n) // n inputs
{
  return (φ(x_1,...,x_n)); // evaluate φ
}

// if i is odd, x_{i+1} is ∀-quantified in ψ
bool SAT[i](bool x_1,...,x_i) // i inputs
{
  return(SAT[i+1](x_1,...,x_i,0)
         ∧ SAT[i+1](x_1,...,x_i,1));
}

// if i is even, x_{i+1} is ∃-quantified in ψ
bool SAT[i](bool x_1,...,x_i) // i inputs
{
  return(SAT[i+1](x_1,...,x_i,0)
         ∨ SAT[i+1](x_1,...,x_i,1));
}
```

Figure 3. A Boolean program for checking satisfiability of the QBF formula $\psi = \exists x_1 \forall x_2 \exists x_3 \ldots Q x_n \phi(x_1, \ldots, x_n)$.

THEOREM 4. *Reachability analysis for acyclic EHSMs of bounded depth is NP-complete.*

Proof: To prove membership in NP, we can guess an input assignment and follow the execution of the acyclic EHSM of bounded depth d (if the EHSM is nondeterministic, we also guess the nondeterministic choices along the way); since the EHSM is acyclic and hierarchical, the length of this execution path is bounded by n^d where n is the maximum number of nodes and boxes in a procedure and d is the bound of the hierarchy.

To prove the problem is NP-hard, we can reduce propositional logic satisfiability (3SAT) to EFSM reachability. Already for an ordinary nonhierarchical acyclic EFSM, we can construct in linear time an EFSM with no input arguments, which first nondeterministically guesses and assigns values to the variables of a given Boolean formula and then goes on to verify that the assignment satisfies the formula. Alternatively, the EFSM can be deterministic and read as input a variable assignment, which is then evaluated and verified. ■

We now consider the particular case of I/O bounded Boolean programs. This means that there is a limit on how much information can pass to and from each procedure of the program. Each procedure may however have an arbitrary number of internal variables and may be arbitrarily large. We prove that the complexity of reachability analysis for I/O bounded programs is lower than for non-I/O bounded programs.

THEOREM 5. *Reachability analysis for I/O bounded deterministic acyclic EHSMs is in P.*

Proof: For deterministic acyclic programs, only a bound on the number of possible values of the input information for each procedure is sufficient for the result. The input information consists of the values of the input variables and the global variables, and the choice of entry node.

We can prove membership in P using a simple dynamic-programming bottom-up algorithm (we can do it also top-down). Starting from procedures at the deepest level, we consider one by one each possible input assignment, and run the procedure to compute the resulting output information that it will produce (i.e., final

values of the global and output variables, and the final exit node). Since the procedure is deterministic, there is exactly one return value for each input, and because of the acyclicity, it takes a number of steps which is at most linear in the number of transitions in the procedure. These computed input-output pairs are tabulated. Moreover, the number of possible inputs to consider is bounded, by the I/O bounded hypothesis. For higher-level procedures, all the calls to lower-level procedures are replaced by the corresponding single input-output transition that was previously computed and tabulated. Overall, we obtain an algorithm for reachability analysis whose running time is linear in the product of the I/O bound times the number of transitions in the deterministic acyclic EHSM, i.e., polynomial in the size of the EHSM. ■

THEOREM 6. *Reachability analysis for I/O bounded* nondeterministic *acyclic EHSMs is NP-complete.*

Proof: To show membership in NP, we can nondeterministically guess an execution path from the (or one) initial node to a given target node. However, the length of such a path can be exponential in the size of the EHSM, even if the EHSM is acyclic, because this path may contain many repetitions of identical subpaths that correspond to different procedure calls along the way to the same procedure with the same arguments. So instead of writing all these repetitions of subpaths, we only guess and write once every such subpath that is used by the accepting computation. Checking that this compact encoding of an execution path leads to the target node can be done in time linear in the product of the I/O bound times the number of transitions in the acyclic EHSM, i.e., polynomial in the size of the EHSM.

For the NP-hardness, we previously showed in the proof of Theorem 4 that satisfiability of propositional logic can be reduced to reachability for acyclic EFSMs that have no input (or output). ■

THEOREM 7. *Reachability analysis for I/O bounded* cyclic *EHSMs is PSPACE-complete.*

Proof: Membership in PSPACE follows from Theorem 3 for the general case. Since EFSM reachability is already PSPACE-complete, PSPACE-hardness follows immediately. ■

We now show that, in the world of I/O bounded programs, reachability analysis for ERSMs is not more expensive than for EHSMs or just EFSMs.

THEOREM 8. *Reachability analysis for I/O bounded ERSMs is PSPACE-complete.*

Proof: That the problem is PSPACE-hard is again immediate since EFSM reachability is already PSPACE-complete.

We now show that the problem is in PSPACE. The algorithm is as follows. Let C be the ERSM obtained from A by replacing in each component A_i every entry node e by a set of entry nodes $\{(e,\eta)|\eta$ a valuation to $V \cup V_i^{in}\}$, and every exit node x by a set of exit nodes $\{(x,\eta)|\eta$ a valuation to $V \cup V_i^{out}\}$; the boxes and their labeling stay the same, but now have more call and return vertices, and the transitions are adjusted appropriately. Since A is I/O bounded, the size of C is polynomial in $|A|$. Remove all the boxes from all the components of C and replace them by just their call and return vertices. Now we have a collection of EFSMs $C'_1, \ldots, C'_k$. Then we discover iteratively reachabilities between the entries and exits of the EFSMs, and add corresponding "summary edges" between the corresponding call and return vertices in other components. Initially, there are no summary edges. In each iteration, we consider one component EFSM C'_i and apply the PSPACE algorithm for EFSMs to compute all the entry-exit reachabilities. If there are any new reachabilities discovered, we add them as summary edges to connect the call and return nodes of all the boxes that correspond to C'_i in all the components. We iterate until there are no more new entry-exit reachabilities for any component, at which point we have the complete entry-exit reachability information. The number of iterations is at most the number of entry-exit pairs in all the components. To compute all the nodes of the ERSM reachable from the initial set $Init$, construct an EFSM $\hat{C}$ which is the union of all the C'_i together with transitions from each call vertex (b,e) of each box b to the corresponding entry e of $C'_{Y_i(b)}$, and compute the set of nodes that are reachable from $Init$ in this EFSM $\hat{C}$. Clearly, the algorithm uses polynomial space.

■

Most of the results of this section are summarized in Figure 4.

4. LTL Model Checking

We now consider linear time properties expressed in Linear Temporal Logic (LTL) or using Büchi automata. Formulas of LTL are built from a finite set $Prop$ of atomic propositions using the usual Boolean operators $\neg$, $\vee$, $\wedge$, the unary temporal operators X (next), and the binary operator U (until), with the following semantics: A computation satisfies $X\phi$ in a step i iff it satisfies ϕ in the next step $i+1$, and it satisfies $\phi U \psi$ in a step i, if it satisfies ψ in some step $j \geq i$ and it satisfies ϕ in all intermediate step k with $i \leq k < j$. Computations are considered to be infinite (with finite computations repeating their final state forever). A Büchi automaton is a finite (nondeterministic) automaton on infinite words that accepts a word w iff it has a run on w that visits the subset of accepting states infinitely often. Every LTL formula ϕ can be translated to an equivalent Büchi automaton D_ϕ over the alphabet $\Sigma = 2^{Prop}$ (the translation may increase exponentially the size in general). The LTL or automaton model checking problem is to determine whether all computations of a given Kripke structure T (starting from designated initial states) satisfy a given LTL formula ϕ or are accepted by a Büchi automaton D. We refer to [12] for detailed background on LTL, automata and model checking. In our case the Kripke structure is the unfolding T_A of a given ERSM A over $\Sigma = 2^{Prop}$.

All the results for reachability of the last section extend to model checking of all linear time properties, with the same dependency of the complexity on the size of the program (this is called the *program complexity*) in all the cases, i.e., for general ERSMs as well as for their subclasses. Roughly speaking, LTL model checking involves forming the product ERSM $\hat{A}$ of the ERSM with an automaton $D_{\neg\phi}$ representing the negation of the property, and testing whether (the unfolding of) $\hat{A}$ has a reachable cycle that contains an accepting state or has an accepting computation path where the stack grows without bound. Both of these can be solved using suitable reachability problems. The dependence of the complexity on the size of the specification is polynomial for automata specifications and exponential for LTL (as is the case for model checking of even nonrecursive finite state structures). Rather than list the individual results, we state them collectively in the following:

THEOREM 9. *The program complexity of model checking linear time properties of ERSMs is the same as that given for reachability analysis in the last section, for all the considered classes of ERSMs.*

The proof for some of the classes is easy and similar to reachability. For example, for general ERSMs, we can expand the given ERSM A to the exponentially larger RSM A' (only the reachable part has to be generated) and use the polynomial-time algorithm for LTL model checking of RSMs from [2]. For others classes, the proof requires some additional work. Due to space limitations, we will not go through all the special cases. We will just sketch here for illustration only the algorithm for the model checking of I/O bounded ERSM in PSPACE.

Class of Program	Restriction	General Case	I/O Bounded
ERSM		EXPTIME	PSPACE
EHSM		PSPACE	PSPACE
EHSM	nondeterministic acyclic	PSPACE	NP
EHSM	deterministic acyclic	PSPACE	P

Figure 4. Complexity of reachability analysis.

Let A be an I/O bounded ERSM and ϕ an LTL formula. We first construct the automaton $D = D_{\neg\phi}$ for the negation $\neg\phi$ of the specification. Then we construct the product $\hat{A} = A \times D$ of the ERSM A and the automaton D, as follows. $\hat{A}$ is an ERSM, with the same number of components as A, and the same global and local variables for each component. The nodes, entries and exits of each component $\hat{A}_i$ are the cartesian products of the nodes, entries, and exits respectively of A with the states of D, while the boxes $\hat{B}_i$ are the same as B_i with the same labeling Y_i and mappings β_i^{in} and β_i^{out}. The transition relation $\hat{\delta}_i$ is obtained by combining transitions of A_i and D with the same label $\sigma \in \Sigma$. Thus, if A_i has a transition (u, G, σ, C, v) where $u, v \in N_i$ and D has a transition (q, σ, r) then $\hat{A}_i$ has a transition $((u, q), G, \sigma, C, (v, r))$; if $u \in N_i$ and $v = (b, e)$ is a call vertex, then $\hat{A}_i$ has a transition $((u, q), G, \sigma, C, (b, (e, r)))$; similarly if u is a return vertex of a box. Let $\hat{F}$ be the set of nodes (u, q) where $q \in F$ is an accepting state of the automaton D. The given ERSM A violates the property ϕ iff A has a computation that is accepted by D and this happens iff the ERSM $\hat{A}$ has a computation that visits $\hat{F}$ infinitely often.

Note that the ERSM $\hat{A}$ is also I/O bounded (its bound is the product of the bound for A and the number of states of D). From the ERSM $\hat{A}$ we construct another ERSM C as in the proof of Theorem 8 where the entries and exits of the components are combined with the input and output valuations. We construct a collection of EFSMs $C'_1, \ldots, C'_k$, and we perform iteratively reachability computations as before in each EFSM to discover the reachable vertices from each entry, and for each entry-exit reachability we add corresponding summary edges. In addition to this, we determine iteratively for each entry e of each EFSM C'_i which vertices of the component it can reach through a path (in the unfolding, i.e., taking into account the variables) that goes through a node in $\hat{F}$ (in the same or in another component); if there is such a path from an entry to an exit, then we mark the corresponding summary edges as "special".

If the ERSM $\hat{A}$ has a computation that visits $\hat{F}$ infinitely often, then either the stack stays bounded throughout the computation, or the stack grows without bound. One can show that the first case happens iff for some EFSM C'_i there is a cycle (in its unfolding) that contains either a node of $\hat{F}$ or a special summary edge, and which is reachable from the initial states $Init$ of the ERSM; we can test this condition in PSPACE. The second case happens iff the following (ordinary) graph H contains a cycle that includes a special edge: The graph H contains the reachable (from $Init$) entries of all the components of $\hat{A}$ and all the corresponding call vertices of boxes, it has an edge from each call vertex of each box b to the corresponding entry of the component to which b is mapped, and has an edge from each entry e of each component to all the call vertices v in the same component that it can reach, with the edge marked 'special' if e can reach v via a path that goes through a node in $\hat{F}$. With the previously computed information, we can build the graph H and test this condition in polynomial time. Thus, the overall algorithm runs in PSPACE.

5. Branching-Time Properties

We now consider the verification of properties expressed in the branching-time logic CTL [11]. CTL allows quantification over computations of a system, such as "along some computation, eventually p" or "along all computations, eventually p". The temporal logic CTL uses the temporal operators U (until), X (nexttime) and the existential path quantifier E, in addition to the operators $\neg$ (not) and $\vee$ (or). We use the standard abbreviations A (for all paths) for $\neg E \neg$, Fp (eventually p) for $trueUp$, and Gp (always p) for $\neg F \neg p$. See [12] for a detailed description of the syntax and semantics of CTL.

The *CTL model checking problem* is to decide whether a Kripke structure satisfies a CTL formula [11]. In our context, unfoldings of ERSMs will be used as Kripke structures.

THEOREM 10. *The program complexity of CTL model checking for ERSMs is 2EXPTIME-complete.*

Proof: Given an ERSM A, we can build an exponentially larger RSM A' such that their unfoldings T_A and $T_{A'}$ are identical, following the construction used in the proof of Proposition 1. Then, we can use the CTL model checking algorithm for RSMs discussed in [2], whose running time can be exponential in the size of the RSMs. Overall, we thus obtain an algorithm with 2EXPTIME complexity.

To prove 2EXPTIME-hardness, we reduce the acceptance problem for 1-tape alternating exponential space machines, which is known to be 2EXPTIME-complete [10], to CTL model checking of ERSMs.

Let M be a 1-tape alternating machine that uses space 2^n on inputs of length n and let x be a given input of length n. We want to determine if M accepts x. Let $M = (\Sigma, Q, q_0, \delta_M)$ where Σ is the tape alphabet, $Q = Q_\forall \cup Q_\exists \cup Q_T \cup Q_F$ is the set of states, q_0 the initial state and δ_M the transition relation. (See the proof of Theorem 2 for the definition of an alternating machine.) Initially M is in state q_0, the head is in cell 0, and the tape holds x followed by blanks. We let x_j denote the content of cell j in the initial tape, i.e., x_j is the j-th symbol of x for $j = 0, \ldots, n-1$ and x_j is blank for $j \geq n$. We construct the Boolean program shown in Figure 5. The program nondeterministically simulates possible executions of M. It generalizes the Boolean program simulating an alternating PSPACE machine of Figure 2. However, it is more complicated because now the exponentially large tape cannot be passed as argument to procedure `Next`, otherwise the reduction would not be polynomial in the size of x.

The main idea to deal with the exponentially large tape is to store the tape content on the stack of the Boolean program. Given the current state q, tape head location h, and symbol s read at location h from the current tape content T, each call to $\texttt{Next}(q, h, s, d)$ computes the next successor configurations: for each such configuration, it computes its next state q', the next value s' to be stored at location h and defining the next tape content T', and the next tape head location h', following the transition relation δ_M of M. However, `Next` does not copy the content for all the tape cells at locations other than h: instead, it *nondeterministically* guesses those values, which should all be unchanged in the new tape content T'

```
Global variables:
g_s, g_s', s_new: previous/next/temporary symbol in Σ (log(|Σ|) bits)
g_q': current state (log(|Q|) bits)
g_h, g_h': previous/next location for the tape head (n bits)
g_d: depth (is either 0, 1, 2)
j: cell location (n bits) or UNDEF
T[j],T'[j]: symbol in Σ (log(|Σ|) bits) or UNDEF // 2 symbols, not arrays
OK=false, CheckMode=false, Success=false: boolean variables (false by default)

Top()
{
  j=UNDEF; T[j]=UNDEF; T'[j]=UNDEF;
  if Next(q0,0,x0,0) then Success=true
  STOP;
}

bool Next(state q, headLocation h, symbol s, depth d)
{
C:if (nondeterminism) then CheckMode=true; // in C:, EX(CheckMode ∧ AF(OK)) must hold

  if (CheckMode)  // start CheckMode — this is executed at most once!
  { // we check that the last 2 tape contents T and T' (last) are δM−compatible
    if (d==0) then { OK=true; STOP; } // nothing to check
    j= nondeterministically pick a cell location // 0 ≤ j < 2^n — ∀−nondeterminism due to AF(OK)
    return false; // dummy return value in this mode; start popping to get T'[j] and T[j]
  }
  if (q in QT) then return true;
  if (q in QF) then return false;

  boolean result;
  if (q in Q∃) then result=false;
  else result=true; // case where q in Q∀

  boolean ret;
  for each (q',s',D) in δM(q,s) // with s=T[h]
  {
    if (D==L) then h' = h−1 else h' = h+1;  // set h' = new head location
    if (d<2) then g_d'=d+1;  // note: d is either 0, 1 or 2
    else g_d'=d;
    g_q' = q'; g_h' = h';      // global variables for next call of Next()
    g_s = s'; g_h = h;         // global variables for this call of Next()

    if (g_h==0) s_new = g_s;
    else s_new = nondeterministically pick a symbol in Σ; // ∃−nondeterminism

    ret=GuessNextTapeCell(0,s_new);

    if (CheckMode)
    {
      if (T[j]!=UNDEF ∧ T'[j]==UNDEF) then // we got T'[j]
      {
        T'[j]=T[j];
        if (d>0) then return false; // continue popping to get T[j]
        else { T[j]=xj; h'=h; }
      }
      // we are ready to check δM−compatibility at position j
      if ((j!=h') ∧ T'[j]==T[j]) then OK=true; // the tape cell content must be unchanged
      if (j==h') then OK=true;    // nothing to check — case enforced by construction
      STOP;
    }
    if (q in Q∃) then result = result ∨ ret;
    else result = result ∧ ret;
  }
  return result;
}

bool GuessNextTapeCell(tapeLocation i, symbol s)
{
  boolean ret;
  if (g_h'==i) then g_s' = s; // record in g_s' the next symbol read from the next location h'
  if (i<(2^n −1))
  {
    if (g_h==i+1) then s_new = g_s; // new symbol just written at the previous location h
    else s_new = nondeterministically pick a symbol in Σ; // ∃−nondeterminism
    ret=GuessNextTapeCell(i+1,s_new); // put s_new on the stack of the ERSM
  }
  else
    ret=Next(g_q',g_h',g_s',g_d');
  if (CheckMode ∧ i==j) then T[j]=s;
  return ret;
}
```

Figure 5. A nondeterministic Boolean program simulating the computations of an alternating EXPSPACE machine M.

compared to the previous one T. These guesses are marked by "$\exists$-nondeterminism" in the code of Figure 5 and made just before and inside the procedure `GuessNextTapeCell`. This procedure takes a tape location and a symbol as arguments. If the tape location is different from the last tape head location, a symbol is nondeterministically guessed and passed as argument to another call to `GuessNextTapeCell`. This way, all cell contents of all tape contents are recorded on the stack of the Boolean program.

At any time during any execution of the Boolean program, its stack content will thus be a prefix of the regular expression

$$\texttt{Top()(Next(q,h,s,d) GuessNextTapeCell(i,s)}^{2^n}\texttt{)}^*$$

At the beginning of every call of `Next()`, at the program location marked "C:", the Boolean program can nondeterministically decide to *stop* the current computation and to start a `CheckMode` in order to check whether the last 2 tape contents T and T' (which are on the stack) are compatible with the transition relation δ_M. When starting a `CheckMode`, the stack of the Boolean program will be of the form

$$\text{(prefix)}\ldots\texttt{Next1(}\ldots\texttt{)}\ T\ \texttt{Next2(}\ldots\texttt{)}\ T'\ \texttt{Next3(}\ldots\texttt{)}$$

if the `depth` d argument to `Next3` is greater than 1. (The depth parameter is used to deal with the special cases of the initial state with $d = 0$ and when there is a single tape content on the stack with $d = 1$.) In the above expression, T and T' are thus encoded each by exactly 2^n calls to `GuessNextTapeCell(i,s)` (with $0 \leq \texttt{i} < n$). A check started in `Next3` checks whether the 2 tape contents T and T' are δ_M-compatible: at any location j *other than* the tape head location h during the call to `Next2`, the content `T[j]` of jth cell of T should be identical to the content `T'[j]` of jth cell of T'. Note that cell location h during the call to `Next2` is the value of h' at the call to `Next1`; this is the only location where T and T' may vary, and there is no need for a check at that location since the new value `T'[h]` is computed in `Next2` directly from δ_M and is thus correct by construction.

Th δ_M-compatibility check is performed *at one location* by nondeterministically picking one location j (in `Next3`) among the 2^n possible tape locations, then popping off the stack 2^n calls to `GuessNextTapeCell(i,s)` while grabbing the value `T'[j]` when popping the call where `i==j`, then popping `Next2`, then popping off the stack another 2^n calls to `GuessNextTapeCell(i,s)` while grabbing this time the value `T[j]`, and then checking back in `Next1` whether `T'[j]==T[j]`. If this test passes, a Boolean variable `OK` is set to true, otherwise it remains false, and the execution of the Boolean program stops.

To check whether T and T' contain the same values at *all* locations (other than h in `Next2`), we use a CTL formula where all possible locations are universally quantified: the check passes at all locations (other than h in `Next2`) if and only if the CTL formula $AF(\texttt{OK})$ holds when the `CheckMode` starts (i.e., when the Boolean variable `CheckMode` becomes true).

Overall, we can prove the following:

> The EXPSPACE alternating machine M accepts the input x if and only if the Boolean program of Figure 5 satisfies (in its initial configuration) the CTL formula
>
> $$E(\texttt{C} \rightarrow EX(\texttt{CheckMode} \wedge AF(\texttt{OK}))\ U\ \texttt{Success})$$

In the formula, `C` denotes a Boolean variable which is true iff the program is currently at its node marked by "C:", while `CheckMode`, `OK` and `Success` refer to Boolean variables used in Figure 5, and $\rightarrow$ denotes logic implication.

This CTL formula holds if and only if (1) there is a computation of the Boolean program that leads to `Success` (this would be the computation that does the traversal of the accepting tree of the alternating machine M with all the correct configurations), and (2) in this computation, *all* the states have the property that if we did a check it would be `OK`.

As before, the Boolean program shown in Figure 5 is generic: for a particular M and x, a particular instantiation A of this Boolean program can be generated in time linear in the size of M and x, where generic statements like "for each successor (q',T') in δ_M(q,s)" will be explicitly expanded in (linear size) explicit switch/case statements. ■

The 2EXPTIME-hardness proof relies on the Boolean program of Figure 5 to be nondeterministic. Indeed, we now prove that CTL model checking for *deterministic* Boolean programs is "only" EXPTIME-complete.

THEOREM 11. *The program complexity of CTL model checking for* deterministic *ERSMs is EXPTIME-complete.*

Proof: EXPTIME-hardness is immediate since reachability analysis of a target node n_t in a deterministic ERSM can be reduced to model checking of the CTL formula $EF(n_t)$ on the same ERSM, and since Theorem 2 already established that ERSM reachability analysis is EXPTIME-hard, regardless of whether the ERSM is deterministic or not.

We can prove membership in EXPTIME in two ways: One way is to exploit the determinism of the ERSM to reduce CTL to LTL model checking and use the algorithm from Section 4. A second, and better way, which we describe below, is to show that a deterministic ERSM A can be translated into an equivalent but possibly exponentially larger *single-exit* RSM A". Since CTL model checking for single-exit RSMs can be done in time linear in the size of the RSM [2], we obtain an algorithm with EXPTIME complexity in the size of the ERSM. This approach is also better for reachability and LTL model checking of deterministic ERSM, as the running time is linear in the number of reachable vertices of the expanded RSM, rather than cubic as in the general algorithms of Sections 3, 4. We describe the translation of the ERSM A to a single-exit RSM A" below in 3 separate steps for simplicity, although in practice we would perform the steps together, on the fly, to generate only the reachable part, in linear time in its size.

Step 1. We first expand the ERSM A into an equivalent exponentially-larger RSM A' as in Proposition 1. Since the ERSM A is deterministic, so is the resulting RSM A'. This means that every state in every FSM component of the RSM A' has at most one successor. This also implies that each entry node can reach at most one exit node.

Step 2. Next, we make a pass over the RSM A', starting at each initial vertex, to determine the vertices that are reachable and compute for each one of them whether it can reach an exit node, and if so, which one. At all times, reached vertices in each component form a forest (except possibly in the last step when a cycle closes). The edges of the forest include ordinary edges and possibly some summary edges. The roots of the trees are either (i) exit nodes, or (ii) call ports of boxes (the vertices that are on the stack), or (iii) the current vertex. The leaves are entry nodes. Each vertex has (i) a mark bit to mark it when the search reaches it (initially =0), (ii) a label field to label it with an exit when we determine that it can reach an exit of its component and which one (initially = nil), (iii) list of its immediate predecessors in the forest, if any. Initially, only the start node is marked, and there is only the trivial tree with just this node.

The main loop is as follows. Let u be the current vertex.

Case 1. u has an edge (u, v) (ordinary edge or a summary edge).

Subcase 1a. mark[v]=0 and v is not an exit node. Set mark[v]=1 and set the current vertex to v.

Subcase 1b. (mark[v]=1 and label[v]=x for some exit x) or v is an exit node x. If $v = x$ is an unmarked exit node then mark it (mark[v]=1) and label it x (label[v]=x). In either case, label u

and all the nodes in its tree by x (i.e., use the predecessor links to perform a backward search from u and label all the nodes it its tree). If the stack is empty, terminate, else pop the top element (b, e) form the stack, add the summary edge $((b, e), (b, x))$, and let the current vertex be (b, e).

Subcase 1c. mark[v]=1 and label[v]=nil. Pop all the elements from the stack and terminate. (In this case we know that the computation does not terminate. Vertices u, v, all their predecessors and all the vertices on the stack and all their predecessors cannot exit. All these vertices have mark=1 and label=nil, which characterizes at the end the vertices that cannot exit.)

Case 2. u is a call vertex (b, e) with no summary edge.

Subcase 2a. mark[e]=0. Push u on the stack, set mark[e]=1 and set the new current vertex to e.

Subcase 2b. mark[e]=1 and label[e]=x for some exit x. Add the summary edge $((b, e), (b, x))$. Current vertex stays $u = (b, e)$ but now it has an edge and it will follow Case 1.

Subcase 2c. mark[e]=1 and label[e]=nil. Pop all the elements from the stack and terminate. (As in case 1c, we know that the computation does not terminate. So e, u, all the predecessors of u, and all vertices on the stack and their predecessors cannot exit.)

Clearly, every reachable vertex is processed once or twice, so the time is linear in the number of reachable vertices (apart from initialization).

Step 3. We construct a new single-exit RSM A'' that contains one component $A''_{i,x}$ for each reachable exit node x of each component A'_i of the given RSM A'; the component $A''_{i,x}$ contains all vertices of A'_i that can reach the exit node x. Thus, each $A''_{i,x}$ has a single exit node x and possibly many entry nodes. In addition, if a component A'_i of the RSM A' contains reachable nodes that cannot exit, then A'' contains another component $A''_{i,0}$ that contains all such vertices; this component has no exit nodes. Each reachable box b of an original component A'_i is replaced by one or more boxes in the components $A''_{i,x}$ and $A''_{i,0}$ of the new RSM A'', namely one box for each reachable return port of the box b contained in the appropriate component $A''_{i,x}$ and $A''_{i,0}$ (depending on whether the return port can reach an exit of A'_i, and if so which one), and possibly one more box in $A''_{i,0}$ if there are reachable call vertices (b, e) such that the entry e cannot exit its component. Every reachable vertex and edge of the RSM A' belongs to exactly one component of the new RSM A'', and A'' has no other vertices or edges. Thus, the total size of A'' (number of all vertices and edges) is bounded by the number of reachable vertices and edges of A'. (Note that we only include in A'' as vertices the call ports that have some incident edges for this to be true. Moreover, we do not create separate components for each reachable entry of a component A'_i because if many entries reach the same exit then we would replicate the vertices and edges of the component A_i for each entry and the size would become quadratic.)

By construction, the reachable parts of the unfoldings T_A, $T_{A'}$ and $T_{A''}$ are all identical. ■

The three steps of the algorithm used in the previous proof could be interleaved and optimized in practice. First, we do not need to construct the expanded RSM A' explicitly. Instead, starting from the initial nodes we explore the RSM on the fly. Second, with an implicit representation of the ERSM, we would use a search structure R for the reachable vertices instead of the mark-bit array (for example a hash table or a search tree) and just generate the reachable vertices on the fly as needed and insert them into R. Note that we do the reachability search top down in Step 2 (rather than bottom up). The running time of the algorithm is proportional to the number of reachable vertices of A', which for a typical ERSM will be probably a fraction of all the vertices since most of the combinations of variable values are likely not reachable.

The algorithm of Theorem 11 is useful also to reduce the complexity of reachability and LTL model checking for deterministic ERSM, from cubic to linear in the number of (reachable) expanded vertices. (Of course we cannot expect an exponential reduction in view of Theorem 2.)

Thanks to the construction used in Steps 2 and 3 of the previous algorithm, it is easy to prove the following.

THEOREM 12. *CTL model checking for* deterministic *multi-exit RSMs can be done in time linear in the size of the structure.*

Proof: Using Step 2 and Step 3 of the proof of Theorem 11, we can translate a deterministic multi-exit RSM A' into an equivalent single-exit RSM A'' is linear time, and then use the linear-time CTL model checking algorithm for single-exit RSMs of [2] on A''. ■

Obviously, this means that CTL model checking of deterministic multi-exit HSMs can also be done in linear time (since HSMs are special RSMs), in contrast with the general case of nondeterministic multi-exit HSMs for which the program complexity of CTL model checking is known to be PSPACE-complete [1].

In the case of EHSMs, we can show that determinism does not help reduce the program complexity of CTL model checking compared to the nondeterministic case. However, and perhaps surprisingly, the program complexity of CTL model checking for EHSMs is the same as for HSMs: it is also PSPACE-complete.

THEOREM 13. *The program complexity of CTL model checking for EHSMs is PSPACE-complete.*

Proof: Since CTL model checking is more general than reachability analysis and the latter is PSPACE-complete by Theorem 3,the program complexity of CTL model checking for EHSMs is PSPACE-hard.

To prove membership in PSPACE, a key observation is that we can build a fully expanded FSM corresponding to an EHSM such that the FSM has singly exponential size: a state of the FSM consists of a stack of suspended calls and the current vertex and tuple of variable values of the EHSM. The stack has depth at most d, which is the nesting depth of the EHSM (the height of the hierarchy), and each record on the stack consists of the box for the call and the values for the local variables of the component at the time of the call. So if there is a total of n vertices and boxes in the EHSM and k Boolean variables, then a state description for the expanded FSM needs $d(log(n)+k)$ bits, i.e., polynomial space (and the number of states is $n^d \cdot 2^{dk}$). The successor states of any state of the FSM can also be computed in polynomial space. By using the space efficient CTL model checking algorithm of [23] on the expanded FSM (which runs in NLOGSPACE in the size of the FSM), we obtain an algorithm with PSPACE complexity overall. ■
Since EFSMs are special cases of EHSMs, the previous PSPACE upper bound carries over to EFSMs, and we have the following.

COROLLARY 14. *The program complexity of CTL model checking for EFSMs is PSPACE-complete.*

Proof: Follows from Theorem 13 and since EFSM reachability analysis is already PSPACE-complete. ■
Since EFSMs are standard, the last result might be already known, but we do not know if it is stated somewhere in the literature.

Finally we note that all the algorithms of this section apply also to the more powerful branching time logic CTL* (see [12] for a definition) with exactly the same complexity:

THEOREM 15. *The program complexity of CTL* model checking is as follows:*
1. For ERSMs it is 2EXPTIME-complete.
2. For deterministic ERSMs it is EXPTIME-complete.
3. For EHSMs it is PSPACE-complete.

Class of Program	Restriction	LTL	CTL
FSM		Linear	Linear
EFSM		PSPACE	**PSPACE**
HSM		Linear	PSPACE
HSM	deterministic	Linear	**Linear**
EHSM		**PSPACE**	**PSPACE**
EHSM	deterministic	**PSPACE**	**PSPACE**
RSM		Cubic	EXPTIME
RSM	deterministic	**Linear**	**Linear**
ERSM		**EXPTIME**	**2-EXPTIME**
ERSM	deterministic	**EXPTIME**	**EXPTIME**

Figure 6. Complexity bounds in the size of the program. The new bounds from this paper are highlighted in bold.

6. Discussion

6.1 Summary of Results

Figure 4 summarizes the results for reachability and linear time properties. For general Boolean programs (ERSMs) the problems are EXPTIME-complete which means that the analysis provably requires exponential time in the worst-case. Since even reachability of simple EFSMs (which have no recursion) is PSPACE-complete, we cannot hope for better than PSPACE for programs with variables that include EFSMs. As we see, PSPACE suffices for important subclasses including EHSM (hierarchical recursion) and I/O bounded ERSM (bounded communication). For the I/O bounded class, the complexity is reduced further in more restricted cases.

Figure 6 summarizes the results regarding the program complexity of LTL and CTL (and CTL*) model checking for general (nondeterministic) and deterministic ERSMs and EHSMs and their counterparts RSM, HSM that have no variables. New results from this work are highlighted in bold.

From Figure 6, we observe that the program complexity of CTL model checking for deterministic programs is exponentially better than for nondeterministic ones, *except* for EHSMs where the complexity does not change. In practice, this means that whenever it is possible to *hoist* nondeterministic choices in a Boolean programs to its initial states, then the program effectively becomes deterministic and CTL model checking can be exponentially faster in the size of the program.

Figure 7 compares the program complexity of LTL and CTL model checking for the main (no restriction) classes of programs considered in Figure 6. From this figure, we make the following observations.

- Adding Boolean variables (extension "E") to programs increases the program complexity of model checking *except* for HSMs and CTL model checking.
- Adding hierarchy to EFSMs does not increase the program complexity of model checking for LTL or CTL.
- For a fixed program class, CTL model checking can be exponentially more expensive in the size of the program than LTL model checking, *except* in the case of EFSMs and EHSMs (where the complexity remains PSPACE-complete) and in the FSM case (where the complexity is linear in both cases).

6.2 Comparison with Pushdown Model Checking

In [2], it is shown that every *RSM* is bisimilar to a *pushdown system* (also called pushdown automaton), while every *single-exit RSM* is bisimilar to a *context-free system*, which is defined as a pushdown system with only one control state. From [9], this implies that there exist multi-exit RSMs whose unfolding is not bisimilar to any single-exit RSM or context-free system. It is also shown in [2] that

- the LTL model checking problem for RSMs and for pushdown systems are inter-reducible in linear time and logarithmic space, and similarly for CTL and CTL*;
- the LTL model checking problem for single-exit RSMs and for context-free systems are inter-reducible in linear time and logarithmic space, and similarly for CTL and CTL*.

Therefore, the program complexity of model checking for RSMs and pushdown systems is the same, and so is the program complexity of model checking for single-exit RSMs and context-free systems.

Since Boolean programs can be exponentially more succinct than ordinary pushdown systems or recursive state machines, the program complexity of model checking for Boolean programs does not follow directly from prior work on model checking for traditional pushdown systems.

[17] defines "symbolic pushdown systems", which are pushdown systems extended with variables in the control states and the stack symbols, it shows how to derive such a system from a Boolean program, and gives an algorithm for LTL model checking (the algorithm has exponential complexity). No lower bound is given on the complexity of the problem.

6.3 Impact on Logic Encodings

The complexity results presented in our work also shed new light on how to represent classes of Boolean programs using logic, and the abilities and limitations of different logics in this respect.

An approach to symbolic program analysis consists in representing the program by a logic formula, possibly generated incrementally, and then reducing reachability analysis and property checking to a satisfiability or validity check for the corresponding logic performed using a SAT or SMT solver. This is the methodology used in *verification-condition generation* [5, 16, 18] and *SAT/SMT-based bounded model checking* [13, 14].

For a *polynomial-size* logic encoding of a specific class of programs, it is necessary to encode in a sufficiently-expressive logic. For instance, consider the EHSM case. Theorem 3 states that reachability analysis for EHSMs is PSPACE-complete. This suggests that a polynomial-size encoding for EHSMs is possible using a logic like QBF since satisfiability for QBF is also PSPACE-complete. (Such an encoding is indeed possible.) This also proves that a polynomial-size encoding in a less expressive logic, such as propositional logic, is impossible: a (precise) translation from EHSMs to propositional logic may result in formulas that are exponentially larger than the program. In contrast, Theorems 4 and 6 identify specific classes of EHSMs for which reachability analysis is "only" NP-complete and for which precise polynomial-size encodings to propositional logic are possible (as satisfiability for propositional logic is NP-complete).

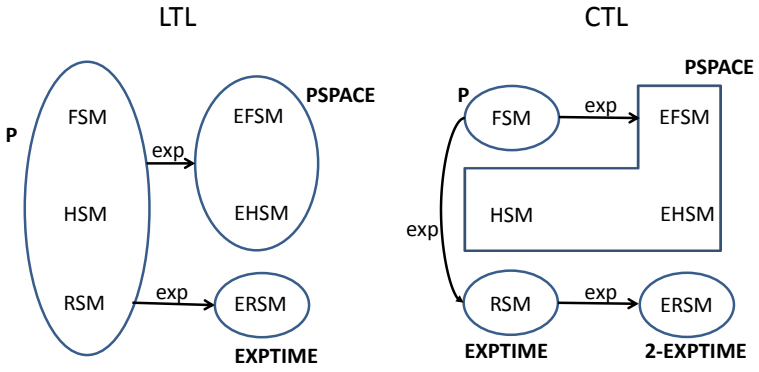

Figure 7. Visual summary for the program complexity of LTL and CTL model checking.

7. Conclusion

Boolean programs are a simple, natural and popular abstract domain for static-analysis-based software model checking. This paper presents the first comprehensive study of the worst-case complexity of several basic analyses of Boolean programs, including reachability analysis, cycle detection, and model checking for the temporal logics LTL, CTL and CTL*. We presented lower and upper bounds for all those problems. We also identified specific classes of Boolean programs which are easier to analyze. These results help explain what features of Boolean programs contribute to the overall worst-case complexity. For instance, nondeterminism does not impact drastically the complexity of reachability analysis for Boolean programs (it increases it only polynomially, see Sections 3 and 5) while it impacts much more significantly (exponentially) the program complexity of CTL model checking (see Section 5).

References

[1] R. Alur and M. Yannakakis. Model Checking of Hierarchical State Machines. *ACM TOPLAS*, 23(3):273–303, 2001.

[2] R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T. Reps, and M. Yannakakis. Analysis of Recursive State Machines. *ACM Trans. on Programming Languages and Systems (TOPLAS)*, 27(4):786–818, 2005.

[3] T. Ball and S. Rajamani. Bebop: A Symbolic Model Checker for Boolean Programs. In *Proceedings of the 7th SPIN Workshop*, pages 113–130, 2000.

[4] T. Ball and S. Rajamani. The SLAM Toolkit. In *Proceedings of CAV'2001*, volume 2102 of *Lecture Notes in Computer Science*, pages 260–264, Paris, July 2001. Springer-Verlag.

[5] M. Barnett and K. R. M. Leino. Weakest Precondition of Unstructured Programs. In *Proceedings of PASTE'05 (Program Analysis For Software Tools and Engineering)*, pages 82–87, 2005.

[6] G. Basler, D. Kroening, and G. Weissenbacher. SAT-based Summarization for Boolean Programs . In *Proceedings of SPIN'2007*, number 4595 in Lecture Notes in Computer Science, pages 131–148, 2007.

[7] A. Bouajjani, J. Esparza, and O. Maler. Reachability Analysis of Pushdown Automata: Application to Model-Checking. In *Proceedings of CONCUR'97*, volume 1243 of *Lecture Notes in Computer Science*, pages 135–150. Springer-Verlag, 1997.

[8] O. Burkart and B. Steffen. Model Checking for Context-Free Processes. In *Proceedings of CONCUR'92*, volume 630 of *Lecture Notes in Computer Science*, pages 123–137. Springer-Verlag, 1992.

[9] B. Caucal and R. Monfort. On the Transition Graphs of Automata and Grammars. In *Graph Theoretic Concepts in Computer Science*, volume 484 of *Lecture Notes in Computer Science*, pages 311–337. Springer-Verlag, 1990.

[10] A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer. Alternation. *Journal of the ACM*, 28(1):114–133, 1981.

[11] E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons using Branching-Time Temporal Logic. In D. Kozen, editor, *Proceedings of the Workshop on Logic of Programs,* Yorktown Heights, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1981.

[12] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[13] E. M. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded Model Checking Using Satisfiability Solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[14] E. M. Clarke, D. Kroening, and K. Yorav. Behavioral Consistency of C and Verilog Programs using Bounded Model Checking. In *Design Automation Conference (DAC)*, pages 368–371. ACM, 2003.

[15] B. Cook, A. Podelski, and A. Rybalchenko. Termination Proofs for Systems Code. In *Proceedings of PLDI'2006*, pages 415–426, 2006.

[16] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Comm. of the ACM*, 18:453–457, 1975.

[17] J. Esparza and S. Schwoon. A BDD-based Model Checker for Recursive Programs. In *Proceedings of CAV'2001*, volume 2102 of *Lecture Notes in Computer Science*, Paris, July 2001. Springer-Verlag.

[18] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended Static Checking for Java. In *Proceedings of PLDI'2002*, pages 234–245, 2002.

[19] P. Godefroid, A. Nori, S. Rajamani, and S. Tetali. Compositional May-Must Program Analysis: Unleashing The Power of Alternation. In *Proceedings of POPL'2010*, pages 43–55, Madrid, January 2010.

[20] S. Graf and H. Saidi. Construction of Abstract State Graphs with PVS. In *Proceedings of CAV'97*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83, Haifa, June 1997. Springer-Verlag.

[21] A. Gurfinkel, O. Wei, and M. Chechik. Yasm: A Software Model Checker for Verification and Refutation. In *Proceedings of CAV'2006*, volume 4144 of *Lecture Notes in Computer Science*, pages 170–174, Seattle, Aug. 2006. Springer-Verlag.

[22] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In *Proceedings of POPL'2002*, pages 58–70, Portland, January 2002.

[23] O. Kupferman, M. Y. Vardi, and P. Wolper. An Automata-Theoretic Approach to Branching-Time Model Checking. *Journal of the ACM*, 47(2):312–360, March 2000.

[24] K. R. M. Leino. A SAT Characterization of Boolean Program Correctness . In *Proceedings of SPIN'2003*, 2003.