Analysis of Boolean Programs

Patrice Godefroid[1] Mihalis Yannakakis[2]

[1] Microsoft Research, pg@microsoft.com
[2] 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 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 [5]. 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., [21]). 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 [23], YASM [22], 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 [4], for which many properties of interest, such as reachability and temporal-logic model checking, are decidable [8], 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, [4] 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. [25] discusses how to reduce reachability

analysis of Boolean programs to SAT solving. More recently, [7] 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., [9, 8, 1, 28]). 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. Furthermore, we study several natural subclasses of Boolean programs and characterize precisely the effects on the complexity of basic restrictions on the structure of the procedures or the type of the recursion: (i) deterministic vs. nondeterministic programs, (ii) hierarchical programs where there is no cycle of mutual recursion between the procedures, (iii) programs where the procedures have a bounded number of input and output arguments. In all the cases, we present algorithms (upper bounds) as well as matching lower bounds for all the problems we consider. In other words, all the algorithms presented in this paper are *optimal* in the complexity-theoretic sense.

Boolean programs correspond to recursive state machines extended with variables (ERSM for short), and can be mapped to ordinary recursive state machines (RSM) that are equivalent but exponentially larger, i.e., the use of variables, besides the syntactical convenience, allows an exponentially more succinct representation than ordinary RSM. Many times this exponential succinctness in representation results in a corresponding exponential increase in the complexity of problems. Indeed there are metatheorems in other domains (e.g., graphs represented succinctly via circuits [27]) showing that under general conditions the succinctness causes an exponential increase in complexity (for example, NP-complete problems become NEXPTIME-complete, P-complete problems become EXPTIME-complete, etc.). However, this is not the case here: the picture is much more varied and rich. As our results show, the succinctness afforded by the use of variables in the extended version of a model (recursive state machines, hierarchical state machines and their subclasses) causes in some cases an exponential jump in complexity (as one may expect), while in other cases the jump is less than exponential, and in yet other cases there is no jump at all. For example, we show that reachability analysis and LTL model checking for Boolean programs (i.e., ERSM) are EXPTIME-complete, while we know that for RSM these problems are P-complete. However, in the hierarchical case, reachability and LTL model checking for Extended Hierarchical State Machines (EHSM) are PSPACE-complete, and not EXPTIME-complete, as one might

expect from the fact that for HSM (Hierarchical State Machines, without variables) these problems are still P-complete, like for RSM. Furthermore, CTL model checking for EHSM and HSM have the same complexity, it is PSPACE-complete, i.e., in this case there is no jump at all.

Similarly, there is also interesting variability in the effects that restrictions on the programs, like determinism, have on the complexity of the problems. For example, reachability analysis for deterministic Boolean programs (ERSM) is EXPTIME-complete, the same as for nondeterministic programs. However, for CTL model checking, determinism reduces the complexity by one exponential: for nondeterministic Boolean programs it is 2EXPTIME-complete, while for deterministic Boolean programs it is still EXPTIME-complete (like reachability).

As a consequence of this richness and variability in the effects of the succinctness afforded by variables and of the restrictions, one has to deal individually with the different problems, models and restrictions, and use appropriate techniques in each case to obtain the correct matching upper and lower bounds.

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. Proofs of theorems are given in the full paper.

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. [5] 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 $Y_i : B_i \to \{1, \ldots, k\}$ 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 $|\beta_i^{in}(b)| = |V^{in}_{Y_i(b)}|$ and $|\beta_i^{out}(b)| = |V^{out}_{Y_i(b)}|$.

- 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 $y := A_j(x)$ in a procedure A_i corresponds to a box b with $Y_i(b) = j$, $\beta_i^{in}(b) = x$, and $\beta_i^{out}(b) = y$. We have allowed multiple entries and exits here for consistency with the definition of standard RSMs that do not have variables [1], 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, we can clearly encode them with Boolean variables, and the results of the paper apply.

In what follows, we will represent ERSMs using pseudo-code.

2.2 Semantics

To define the executions of ERSMs, we first define the global states and transitions associated with an ERSM. Let $\overline{X}$ 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 $\langle (b_1, \overline{V_1}), \ldots, (b_r, \overline{V_r}), (u, \overline{V_{r+1}}, \overline{V}) \rangle$ where $b_1, \ldots, b_r$ are boxes, $\overline{V_1}, \ldots, \overline{V_r}, \overline{V_{r+1}}$ are value assignments to local variables, u is a node, and $\overline{V}$ 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 $(B \times \overline{V'})^*(N \times \overline{V'} \times \overline{V})$, where $B = \cup_i B_i$, $V' = \cup_i V_i$ and $N = \cup_i N_i$. Consider a state $\langle (b_1, \overline{V_1}), \ldots, (b_r, \overline{V_r}), (u, \overline{V_{r+1}}, \overline{V}) \rangle$ 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. Given a state $\langle (b_1, \overline{V_1}), \ldots, (b_r, \overline{V_r}), (u, \overline{V_{r+1}}, \overline{V}) \rangle$, we will sometimes refer to $\langle (b_1, \overline{V_1}), \ldots, (b_r, \overline{V_r}) \rangle$ as the *call stack*, or stack, in that state.

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 $s = \langle (b_1, \overline{V_1}), \ldots, (b_r, \overline{V_r}), (u, \overline{V_{r+1}}, \overline{V}) \rangle$ 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, $G(\overline{V_{r+1}}, \overline{V})$ evaluates to true, $C(\overline{V_{r+1}}, \overline{V}) = (\overline{V_{r+1}}', \overline{V}')$, and $s' = \langle (b_1, \overline{V_1}), \ldots, (b_r, \overline{V_r}), (u', \overline{V_{r+1}}', \overline{V}') \rangle$. This case is when the control stays within the component A_j.
2. $(u, G, \sigma, C, (b', e)) \in \delta_j$ for a box b' of A_j, $G(\overline{V_{r+1}}, \overline{V})$ evaluates to true, $C(\overline{V_{r+1}}, \overline{V}) = (\overline{V_{r+1}}', \overline{V}')$, and $s' = \langle (b_1, \overline{V_1}), \ldots, (b_r, \overline{V_r}), (b', \overline{V_{r+1}}'), (e, \overline{V_{r+2}}', \overline{V}') \rangle$, where $\overline{V_{r+2}}'$ 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 $V^{in}_{Y_j(b')}$ have value equal to the value of the variables $\beta^{in}_j(b')$ in $\overline{V_{r+1}}'$. This case is when a new component is entered via a box of A_j.
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 $\beta^{out}_m(b_r)$ have value equal to that of the output variables V^{out}_j of A_j in $\overline{V_{r+1}}$ and the rest of the variables have the same value as in $\overline{V_r}$, $G(\hat{V_r}, \overline{V})$ evaluates to true, $C(\hat{V_r}, \overline{V}) = (\overline{V_r}', \overline{V}')$, and $s' = \langle (b_1, \overline{V_1}), \ldots, (b_{r-1}, \overline{V_{r-1}}), (u', \overline{V_r}', \overline{V}') \rangle$. This case is when the control exits A_j and returns back to A_m.
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 $\beta^{out}_m(b_r)$ have value equal to that of the output variables V^{out}_j of A_j in $\overline{V_{r+1}}$ and the rest of the variables have the same value as in $\overline{V_r}$, $G(\hat{V_r}, \overline{V})$ evaluates to true, $C(\hat{V_r}, \overline{V}) = (\overline{V_r}', \overline{V}')$, and $s' = \langle (b_1, \overline{V_1}), \ldots, (b_{r-1}, \overline{V_{r-1}}), (b', \overline{V_r}'), (e, \overline{V_{r+1}}', \overline{V}') \rangle$, where $\overline{V_{r+1}}'$ 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 $\beta_m^{in}(b')$ in $\overline{V_r}'$. This case 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. 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 $\langle (b_1, \overline{V_1}), \ldots, (b_r, \overline{V_r}), (v, \overline{V_{r+1}}, \overline{V}) \rangle$ in T_A whose node is v.

2.3 Special Classes

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 Boolean program A is called *input/output bounded*, or *I/O bounded* for short, if the number of the input and output variables of every procedure, and the number of global variables are $O(\log |A|)$ (i.e., upper bounded by $c \cdot \log |A|$ for some fixed constant c). The procedures themselves can be arbitrarily large and complex, and use an arbitrary number of local variables. The I/O bounded property characterizes programs where there is a limited 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.

2.4 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 construction of A' involves combining every vertex of each procedure of A with every valuation for the global and local variables (see the full paper for the detailed construction). 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) $|A| \cdot 2^m$. We call A' the *expanded* RSM corresponding to 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.

Theorem 1. *Reachability analysis for ERSMs is EXPTIME-complete. Furthermore, this holds even for deterministic, acyclic ERSMs.*

Sketch: Membership in EXPTIME follows essentially from previous work (e.g., [4, 1]). 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 [1]), we obtain an algorithm with EXPTIME complexity overall.

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. Figure 1 shows a Boolean program (left) simulating an alternating PSPACE machine. The proof is given in the full paper. ■

The Boolean program of Figure 1 is deterministic and acyclic, so these features do not make a dramatic difference in the complexity of ERSM reachability analysis. Note that the procedure Acc in the program of Figure 1 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.

In the hierarchical case, reachability analysis becomes PSPACE-complete, thus, no worse than simple EFSMs. Note that if we expand the EHSM to an (exponentially larger) HSM and apply the HSM reachability algorithm, the resulting algorithm will have exponential space complexity, and this is probably inherent in that approach since reachability for HSM is P-complete [3].

Theorem 2. *Reachability analysis for EHSMs is PSPACE-complete. Furthermore, the problem remains PSPACE-complete for deterministic, acyclic EHSMs.*

```
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 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;
}
```

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

bool SAT[n](bool x1,...,xn)
{
  return (φ(x1,...,xn)); // evaluate φ
}

// if i is odd, xi+1 is after ∀ in ψ
bool SAT[i](bool x1,...,xi)
{
  return(SAT[i+1](x1,...,xi,0)
         ∧ SAT[i+1](x1,...,xi,1));
}

// if i is even, xi+1 is after ∃ in ψ
bool SAT[i](bool x1,...,xi)
{
  return(SAT[i+1](x1,...,xi,0)
         ∨ SAT[i+1](x1,...,xi,1));
}
```

Fig. 1. Boolean programs simulating an alternating PSPACE machine M (left) and for checking satisfiability of the QBF formula $\psi = \exists x_1 \forall x_2 \exists x_3 \ldots Q x_n \phi(x_1, \ldots, x_n)$ (right).

Sketch: Membership in PSPACE follows from nondeterministically simulating a computation that reaches the target node using polynomial space, and applying Savitch's theorem to make it deterministic. Since reachability analysis is already known to be PSPACE-hard for EFSMs, PSPACE-hardness for the more general EHSMs follows immediately. Moreover, the problem remains PSPACE-complete for EHSMs that are deterministic and acyclic. For this purpose, we reduce Quantified Boolean Formula (QBF) satisfiability (QSAT), known to be PSPACE-complete, to EHSM reachability: Figure 1 shows a deterministic acyclic hierarchical Boolean program (on the right) 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 proof is given in the full paper. ■

For acyclic EFSM and, more generally, for acyclic EHSMs where the depth of the hierarchy is bounded by a constant, the complexity of reachability analysis is reduced further to NP-complete.

Theorem 3. *Reachability analysis for acyclic EHSMs of bounded depth is NP-complete.*

We now consider the subclass of I/O bounded Boolean programs, and show that the complexity is lower.

Theorem 4. *Reachability analysis for I/O bounded deterministic acyclic EHSMs is in P.*

Theorem 5. *Reachability analysis for I/O bounded* nondeterministic *acyclic EHSMs is NP-complete.*

Theorem 6. *Reachability analysis for I/O bounded* cyclic *EHSMs is PSPACE-complete.*

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

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

Fig. 2. Complexity of reachability analysis.

Theorem 7. *Reachability analysis for I/O bounded ERSMs is PSPACE-complete.*

Sketch: The algorithm involves doing first a partial expansion of the ERSM where we only expand in each procedure the input and output variables and the global variables. Then we remove the boxes from the procedures, yielding a collection of EFSMs, and solve iteratively a sequence of EFSM reachability problems to infer incrementally the reachabilities between the expanded entries and exits of the procedures. Finaly we construct a final single EFSM $\hat{C}$ that incorporates the entry-exit reachabilities and interconnects the procedures, and solve an EFSM reachability problem on $\hat{C}$ to compute all the vertices that are reachable from the initial set $Init$. See the full paper for details. ■

Most of the results of this section are summarized in Figure 2.

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). 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 [13] 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. 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 8. *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.*

Due to space constraints we will not give the proofs for the various classes. 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 cases can be solved using suitable reachability problems. The specifics of the algorithms depend on the class of ERSMs; in some cases this is easy, while in others it is nontrivial.

5 Branching-Time Properties

We now consider the verification of properties expressed in the branching-time logic CTL [12]. 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 Ap (for all paths p) for $\neg E \neg p$, Fp (eventually p) for $trueUp$, and Gp (always p) for $\neg F \neg p$. See [13] 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 [12]. In our context, unfoldings of ERSMs will be used as Kripke structures.

Theorem 9. *The program complexity of CTL model checking for ERSMs is 2EXPTIME-complete.*

Sketch: Given an ERSM A, we can build an exponentially larger RSM A' such that their unfoldings T_A and $T_{A'}$ are identical. Then, we can use the CTL model checking algorithm for RSMs discussed in [1], 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. Given an alternating EXPSPACE machine M and an input x, we construct a Boolen program P that simulates the computations of M on x. A problem here is that the exponentially large tape cannot be passed as an argument (unlike the proof of Theorem 1). The main idea to address this is to have the program nondeterministically guess continuously the contents of the tape, cell by cell, and store it in the stack. Another part of the program may nondeterministically at any point stop the computation and backtrack to try to check whether the content of a particular cell is consistent with the previous configuration. The constructed CTL formula φ is a fixed formula that says that there is a computation of the program P that leads to acceptance and if we were to do any check along the way it would turn out ok. We show that the EXPSPACE alternating machine M accepts an input x if and only if P satisfies φ; see the full paper for the details of the construction and the proof. ■

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

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

The proof involves the development of a new efficient algorithm for CTL model checking of deterministic RSM showing the following:

Theorem 11. *CTL model checking for* deterministic *multi-exit RSMs can be done in time linear in the size of the structure.*

Sketch: Given a deterministic multi-exit RSM A we show how to construct in linear time an equivalent single-exit RSM A", and then we use the linear-time algorithm for CTL model checking of single-exit RSMs from [1]. The construction of A" involves two phases. In the first phase, we compute for each initial node incrementally all the reachable vertices, and for each reachable vertex, we compute whether it can reach an exit node of its component and which one. This has to be done carefully to ensure that nonterminating computations are cut off promptly and that every reachabe vertex is processed only at most twice, and thereby achieve linear time in the number of reachable vertices. In the second phase, we construct in linear time from the information of Phase 1 a single-exit RSM A" that contains several procedures for each component of A with the property that every reachable vertex and edge of A appears in exactly one procedure of A", and A" has no other vertices and edges. Furthermore, the reachable parts of the unfoldings of A and A" are identical. See the full paper for the details. ■

Theorem 10 can be shown then by expanding the given deterministic ERSM A to a RSM and applying the algorithm of Theorem 11. The expansion can be done in fact on the fly, only to the extent that is needed, starting from the set $Init$ of initial states, so that the whole CTL model checking algorithm takes time proportional to the number of reachable vertices in the expanded RSM.

The algorithm used in the proof of Theorem 10 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 1.)

Obviously, Theorem 11 implies 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 [3].

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

Since EFSMs are special cases of EHSMs, the previous PSPACE upper bound carries over to EFSMs, and we have the following.

Corollary 1. *The program complexity of CTL model checking for EFSMs 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**

Fig. 3. Complexity bounds in the size of the program. The new bounds from this paper are highlighted in bold.

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 [13] for a definition) with exactly the same complexity:

Theorem 13. *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.

6 Discussion

6.1 Summary of Results

Figure 2 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 3 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 3, 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. On the other hand,

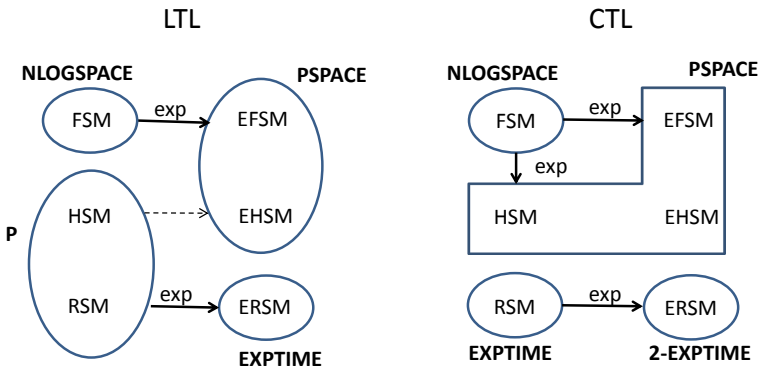

Fig. 4. Visual summary for the program complexity of LTL and CTL model checking.

for LTL model checking, determinism decreases the complexity more modestly, by a polynomial amount.

Figure 4 compares the program complexity of LTL and CTL model checking for the main (no restriction) classes of programs considered in Figure 3. 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. Adding further full recursion increases somewhat the complexity for LTL, but much more drastically (more than exponentially) for 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 [1], it is shown that every *RSM* is bisimilar to a *pushdown system* (also called pushdown automaton). Therefore, the program complexity of model checking for RSMs and pushdown systems is the same. 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. The same comment applies to prior work on hierarchical systems (e.g. [3, 2, 20, 26]).

[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* [16, 18, 6] and *SAT/SMT-based bounded model checking* [11, 14].

For a *polynomial-size* logic encoding of a specific class of programs, it is necessary to use a sufficiently-expressive logic. For instance, consider the EHSM case. Theorem 2 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 3 and 5 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).

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 also studied several natural classes of Boolean programs which are easier to analyze. We presented matching lower and upper bounds for all these problems. The overall picture is quite rich and varied and required a range of different techniques. The 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) while it impacts much more significantly (exponentially) the program complexity of CTL model checking.

Acknowledgments. Research partially supported by NSF Grant CCF-1017955.

References

1. 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.
2. R. Alur, S. Kannan, and M. Yannakakis. Communicating Hierarchical State Machines. In *Proc. ICALP*, pages 169–178. Springer-Verlag, 1999.
3. R. Alur and M. Yannakakis. Model Checking of Hierarchical State Machines. *ACM TOPLAS*, 23(3):273–303, 2001.

4. T. Ball and S. Rajamani. Bebop: A Symbolic Model Checker for Boolean Programs. In *Proceedings of the 7th SPIN Workshop*, pages 113–130, 2000.
5. 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.
6. M. Barnett and K. R. M. Leino. Weakest Precondition of Unstructured Programs. In *Proc. PASTE (Program Analysis For Software Tools and Engineering)*, pages 82–87, 2005.
7. G. Basler, D. Kroening, and G. Weissenbacher. SAT-based Summarization for Boolean Programs . In *Proceedings of SPIN'2007*, LNCS 4595, pages 131–148, 2007.
8. A. Bouajjani, J. Esparza, and O. Maler. Reachability Analysis of Pushdown Automata: Application to Model-Checking. In *Proc. CONCUR*, LNCS 1243, pages 135–150. Springer-Verlag, 1997.
9. O. Burkart and B. Steffen. Model Checking for Context-Free Processes. In *Proceedings of CONCUR'92*, LNCS 630, pages 123–137. Springer-Verlag, 1992.
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, A. Biere, R. Raimi, and Y. Zhu. Bounded Model Checking Using Satisfiability Solving. *Formal Methods in System Design*, 19(1):7–34, 2001.
12. E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons using Branching-Time Temporal Logic. In D. Kozen, editor, *Proc. of the Workshop on Logic of Programs*, volume 131 of *LNCS*, pages 52–71. Springer-Verlag, 1981.
13. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.
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 *Proc. CAV*, LNCS 2102. Springer-Verlag, 2001.
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 *Proc. POPL*, pages 43–55, 2010.
20. S. Goller and M. Lohrey. Fixpoint Logics over Hierarchical Structures. *Theory Comp. Sys.*, 48(1):93–131, 2011.
21. S. Graf and H. Saidi. Construction of Abstract State Graphs with PVS. In *Proceedings of CAV'97*, LNCS 1254, pages 72–83. Springer-Verlag, 1997.
22. A. Gurfinkel, O. Wei, and M. Chechik. Yasm: A Software Model Checker for Verification and Refutation. In *Proc. CAV*, LNCS 4144, pages 170–174. Springer-Verlag, 2006.
23. T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In *Proceedings of POPL'2002*, pages 58–70, Portland, January 2002.
24. 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.
25. K. R. M. Leino. A SAT Characterization of Boolean Program Correctness . In *Proceedings of SPIN*, 2003.
26. M. Lohrey. Model-checking hierarchical structures. *J. Comp. Sys. Sc.*, 78(2):461–490, 2012.
27. C. H. Papadimitriou and M. Yannakakis. A Note on Succinct Representation of Graphs. *Inf. and Comp.*, 71(3):181–185, 1986.
28. I. Walukiewicz. Model Checking CTL Properites of Pushdown Systems. In *Proc. FSTTCS*, pages 127–138, 2000.