

A Better Conversion of LTL Formulas to Symbolic Automata

Kristin Y. Rozier *

NASA Ames Research Center, Moffett Field CA, 94035, Kristin.Y.Rozier@nasa.gov

Abstract. Symbolic model checking has demonstrated more scalability and reliability than explicit model checking, and is used frequently for industrial verification. Yet, the issue of efficient construction of symbolic automata for LTL formulas has been largely neglected, while explicit translation of LTL to automata has been studied extensively. We show that algorithmic ideas from explicit-state LTL-to-automata translators, as well as translations for other logics, are applicable to the symbolic domain. We propose a new algorithmic-portfolio approach for improved translation from LTL to symbolic automata. We provide experimental results comparing our implementation with the front-end translators built into CadenceSMV and NuSMV and demonstrate that we can consistently achieve better performance, even with a very simple heuristic for choosing a symbolic encoding.

1 Introduction

In model checking, the negation of the specification is translated into a Büchi automaton, combined with the system model, and then checked for non-emptiness [31]. A fair path found in this combined model represents a counterexample: a trace where the system violates its specification. For a system of size n and a Linear Temporal Logic (LTL) specification of size m , LTL model checking takes time $n2^{O(m)}$ [19]. As LTL model checking is PSPACE-complete [27], the exponential blow-up in the size of the specification seems inherent, posing an algorithmic challenge for implementors. This motivated extensive research on dealing with this exponential blow-up [30]. Many researchers focused on optimizing the construction of automata from LTL formulas. An extensive survey of different automata construction algorithms is provided in [25]. The main focus of this research area has been on explicit construction of automata from LTL formulas, where the goal of optimization is to reduce the size of the constructed automata. It is shown in [25] that the difference in performance between the different translators can be quite dramatic, having a major impact on the performance of explicit-state model checking. One major finding in [25] is that among all LTL-to-automata translators, SPOT [10] is the only one that proved to be an industrial-strength tool.

Symbolic model checkers describe both the system model and property automaton symbolically: states are viewed as truth assignments to Boolean state variables and the transition relation is defined as a conjunction of Boolean constraints on pairs of current and next states [4]. The model checker uses a BDD-based fixpoint algorithm to find a fair path in the model-automaton product [11]. CadenceSMV¹ [20] and NuSMV² [5]

* Thanks to Moshe Y. Vardi for many insightful comments. Work contributing to this paper was completed at Rice University, Cambridge University, and NASA, and was supported in part by the Shared University Grid at Rice (SUG@R), funded by NSF under Grant EIA-0216467, and a partnership between Rice University, Sun Microsystems, and Sigma Solutions, Inc.

¹ http://www.cadence.com/company/cadence_labs_research.html

² <http://nusmv.irst.itc.it/>

both evolved from the original Symbolic Model Verifier developed at CMU [21]. These tools, or similar industrial tools (e.g., RuleBase [1]), are widely used in industrial hardware verification. Both tools support LTL model checking via a symbolic translation of LTL to automata, and then use CTL model checking (in a very simple way) to check the existence of a fair path in the product [4]. In symbolic model checking, a typical optimization heuristic is to reduce the number of state variables. An optimized translation from LTL formulas to symbolic *generalized Büchi automata* (GBAs), improving on the construction described in [4], was given in [7]. We refer to this construction as *CGH*. CGH is essentially the standard today, and is the basis for the implementations of LTL model checking in both CadenceSMV and NuSMV. Modern symbolic translations of industrial specification languages, cf. [6], are also based on the CGH translation. Surprisingly, there has been practically no follow-up research on this topic since [7].

Another major finding in [25] is that in the context of LTL satisfiability checking (see below), symbolic model checking dramatically outperforms explicit model checking. This motivated us to revisit the issue of symbolic translation of LTL to automata.

In this paper, we improve upon the performance of the CGH algorithm. We bring new symbolic LTL translation techniques from two sources. First, we show that techniques from explicit LTL translation are applicable to symbolic LTL translation. Specifically, we show that formula rewriting techniques, cf. [28], are significant in symbolic LTL translation. In particular, using *negation-normal form* (NNF), may be better than using *Boolean-normal form* (BNF). Also, we show that using *transition-based generalized Büchi automata* (TGBAs) rather than GBAs [8], which is the technique underlying SPOT [10], brings an advantage in symbolic LTL translation. Second, borrowing from symbolic modal satisfiability solving [22], we show that “sloppy” encoding (essentially, corresponding to dual-rail encoding) can be superior to “fussy” encoding (corresponding to single-rail encoding). Finally, as the importance of variable ordering in BDD-based symbolic model checking is well known [16], we show that tailoring variable ordering to LTL formulas also yields performance improvements.

We focus our experimental evaluation on LTL satisfiability checking. We showed in [25] that LTL satisfiability checking is a special case of LTL model checking which provides a source of challenging model-checking problems. We also argued there for the importance of LTL satisfiability checking as a form of sanity checking. The importance of *sanity checks* in model checking has been argued in many papers, see [17]. Clearly, if a formal property is valid, then this is certainly due to an error. Similarly, if a formal property is *unsatisfiable*, that is, true in *no* model, then this is also certainly due to an error. Even if each individual property written by the specifier is satisfiable, their conjunction may very well be unsatisfiable, which also indicates an error. Recall that a logical formula φ is valid iff its negation $\neg\varphi$ is not satisfiable. Thus, as a necessary sanity check for debugging a specification, model-checking tools should ensure that both the specification φ and its negation $\neg\varphi$ are satisfiable.

Our experimental evaluation, across a spectrum of challenging LTL formulas, indicates that no symbolic translation technique is superior across the whole spectrum. We do present, however, a successful algorithmic portfolio approach (cf. [18]) to symbolic LTL translation. We suggest a simple heuristic for selecting a symbolic translation technique based on the structure of the given LTL formula. Our experiments with this simple heuristic demonstrate that its performance significantly dominates the native translation of CadenceSMV. (We cannot compare our translation to NuSMV’s due to technical reasons explained in the paper.)

The structure of this paper is as follows. We review the CGH algorithm of [7] in §2, and uncover some semantic subtleties in the syntax of CadenceSMV and NuSMV

in §3. Next, in §4, we describe the TGBA encoding. We discuss our various symbolic translation techniques. in §5. After covering our experimental methods in §6, we present our results in §7, followed by a discussion (§8).

2 Preliminaries

Support for LTL specifications was added to previously CTL-only symbolic model checkers after CGH demonstrated an efficient encoding of LTL formulas as symbolic automata. Essentially, CGH reduces LTL model checking to checking for the nonemptiness of a Kripke structure with fairness constraints, which can be checked by a CTL model checker. Our definition of a TGBA-based symbolic automaton is derived from their method for creating a GBA-based automaton.

2.1 Another Look at LTL Model Checking [7]: Basic Algorithm Overview

Input: An LTL formula containing only atomic propositions, \neg , \vee , X , and U .

1. Negate f and form the set AP_f of atomic propositions of f .
2. Build el_list , the list of elementary formulas in f :

$el(p) = \{p\}$ if $p \in AP$.	$el(\neg g) = el(g)$.
$el(g \vee h) = el(g) \cup el(h)$.	$el(Xg) = \{Xg\} \cup el(g)$.
$el(g \ U \ h) = \{X(g \ U \ h)\} \cup el(g) \cup el(h)$.	
3. Declare a new variable EL_{X_g} for each formula Xg in the list el_list .
4. Add fairness constraints to the SMV input model:

$$\{sat(\neg(g \ U \ h) \vee h) \mid g \ U \ h \text{ occurs in } f\}$$
5. Construct the characteristic function S_h for each subformula h in $\neg f$:

$S_h = p$	if p is an atomic proposition.
$S_h = EL_h$	if h is elementary formula Xg in el_list .
$S_h = !S_g$	if $h = \neg g$
$S_h = S_{g1} S_{g2}$	if $h = g1 \vee g2$
$S_h = S_{g2} (S_{g1} \& S_{X(g1 \ U \ g2)})$	if $h = g1 \ U \ g2$
6. Print the SMV program:

```

MODULE main
VAR /*declare a boolean variable for each atomic proposition in f*/
  a: boolean;
  b: boolean;
/*and declare a new variable EL_X_g for each formula (X g) in el_list*/
  EL_X_g: boolean;
DEFINE /*for each S_h in the characteristic function, put a line here*/
  S_a := a;
  S_b := b;
  S_g1 := ...
  S_g2 := ...
TRANS /*for each (X g) in el_list, generate a line here*/
  ( S_X_g1 = next(S_g1) ) &
  ...
  ( S_X_gn = next(S_gn) )
/*for each (g U h) in the parse tree, generate a line like this:*/
FAIRNESS ! S_gUh | S_h
/*end with a SPEC statement*/
SPEC !(S_f & EG true)

```

3 Semantic Subtleties

All symbolic model checkers use the CGH translation and the analysis algorithm of [11], though some add further optimizations, including CadenceSMV and VIS [2]. However, the precise semantics of symbolic automata are not explicitly documented and there are some subtle differences between the implementations of CadenceSMV and NuSMV. Therefore, several subtleties arise when checking nonemptiness of fair Kripke structures. With help from the creators of these tools, we have compiled the following rule-set for creation of symbolic automata from LTL formulas:

There must be at least one initial state. For both NuSMV and CadenceSMV, there is an implicit universal quantifier over all initial states. If there are no initial states, then the formula is automatically “true.” Declaring an initial state is not enough to satisfy this condition. For example, $\text{INIT } (\text{a} \& (\text{!a}))$ specifies that there is no initial state.

Symbolic automata must always have a FAIR statement, even if it is “FAIR true.” CadenceSMV considers terminal paths to be fair when there are no fairness constraints. The semantics with a fairness constraint is the “infinite paths” semantics where states without infinite paths are discarded. Therefore, we must have at least one fairness constraint to prevent the possibility of a model with one initial state and no legal transitions from model checking as “false.” Rather than the classical algorithm of implicitly universally quantifying over all initial states, NuSMV restricts itself to all *fair* initial states. If there are no fair initial states, the formula is automatically “true.”

The SPEC should be $(\text{!}(\varphi \wedge \text{EG true}))$. Both CadenceSMV and NuSMV consider a CTL formula φ to hold in a model M if φ holds in *all* initial states of M . If M has no initial states, then every φ holds in M . Using $\text{SPEC } (\text{!}(\varphi \wedge \text{EG true}))$, if the model is not empty, the counterexample returned is a trace of the model.

$\text{INIT } \varphi \text{ SPEC } (\text{!}(\text{EG true}))$ is not equivalent to $\text{SPEC } (\text{!}(\varphi \wedge \text{EG true}))$. For example, if φ is simply *false* and we check $\text{SPEC } (\text{!}(\text{false} \wedge \text{EG true}))$ over an empty model, a failure means that $(\text{false} \wedge \text{EG true})$ is true in some state. However, if we check instead $\text{INIT } \text{false}$ and $\text{SPEC } (\text{!}(\text{EG true}))$, we actually get a counterexample. Similarly, checking for a finite counterexample using $\text{SPEC } (\text{!}(S_f))$ may produce spurious results.

4 Transition-Based Generalized Büchi Automata (TGBA)

In [25] we demonstrated that SPOT [10], which optimizes LTL-to-automata techniques of other translators (eg [14], [12]), is the only publicly available explicit-state model-checking tool that is somewhat competitive with symbolic tools. We borrow the idea of creating TGBAs from SPOT, where the size of the automaton is reduced by placing the labels on the transitions instead of the states. By pushing the acceptance conditions to each accepting states’ outgoing transitions, we can reduce state-based acceptance to transition-based acceptance with a resulting automaton of less than or equal to the original size. There is, of course, a trade-off: symbolic representations of TGBAs may require more variables than for GBAs. While both representations require the declaration of all of the boolean variables present in the input formula f plus a variable associated with each temporal subformula, transition-based representations also require a variable for f and *promise variables* to assure fairness of \mathcal{U} - and \mathcal{F} -operators.

The TGBA construction [8] implemented in SPOT is based on symbolic computation over the following set of boolean variables comprised of atomic propositions, subformula variables (r_f), and promise variables (a_f):

$$\begin{aligned}
r_p &= p \text{ if } p \text{ is an atomic proposition} \\
r_{\neg g} &= !g \\
r_{g \vee h} &= r_g | r_h \\
r_{g \wedge h} &= r_g \& r_h \\
r_{g \ u \ h} &= r_h | (r_g \& a_{g \ u \ h} \& r_{X(g \ u \ h)}) \\
r_{g \ \mathcal{R} \ h} &= r_h \& (r_g | r_{X(g \ \mathcal{R} \ h)}) = (r_h \& r_g) | (r_g \& r_{X(g \ \mathcal{R} \ h)})
\end{aligned}$$

We extend the explicit-state construction of [8] in terms of the CGH symbolic translation, resulting in the following algorithm.

Input: LTL formula f with operators $\sim, |, \&, \rightarrow, X, U, \mathcal{R}, G, \mathcal{F}$.

1. **Convert the formula into Negation Normal Form. Boolean Normal Form is not allowed here.** A symbolic TGBA can only be created for formulas in NNF because the model checker tries to guess a sequence of values for each of the promise variables. This works for positive U -formulas, but not for negative U -formulas. As an example, consider the formula $\phi = \neg(a \ U \ b)$ and the trace $a=1, b=0, a=1, b=1, \dots$. Clearly, $(a \ U \ b)$ holds in the trace, so ϕ fails in the trace. If, however, we chose $P_{a \ U \ b}$ to be false at time 0, then $EL_{a \ U \ b}$ will be false at time 0, which will mean that ϕ holds at time 0.
2. **Declare a variable EL_g for each elementary subformula Xg .** In other words, create a variable for every subformula occurring after an X -operator.³ There is also a variable EL_f for the whole formula f .
3. **Declare a variable for each $U, \mathcal{R}, \mathcal{F},$ and G -subformula.** This represents a significant deviation from the variables declared for such subformulas in [7] in two ways:
 - (a) The variable is for the subformula, not the next-time of the subformula. For example, for each subformula $g \ U \ h$, we declare $EL_{g \ U \ h}$ instead of $EL_{X(g \ U \ h)}$.
 - (b) There is a special optimization described in [8] for the common $G\mathcal{F}$ juxtaposition, which we treat as a single operator.
4. **Define the acceptance conditions.** Declare a promise variable $P_{g \ U \ h}$ for every subformula $g \ U \ h$, $P_{\mathcal{F} \ g}$ for $\mathcal{F} \ g$, and $P_{G\mathcal{F} \ g}$ for $G\mathcal{F} \ g$.
5. **Construct the characteristic function S_h for each subformula h in $\neg f$.** This step is similar to the [7] algorithm for Boolean operator-rooted subformulas and different for temporal subformulas.⁴

$$\begin{array}{ll}
S_h = p & \text{if } p \in AP \\
S_h = !S_g & \text{if } h = \neg g \\
S_h = S_{g1} | S_{g2} & \text{if } h = g1 \vee g2 \\
S_h = S_{g1} \& S_{g2} & \text{if } h = g1 \wedge g2 \\
S_h = next(EL_g) & \text{if } h = Xg \\
S_h = S_{g2} | (S_{g1} \& P_{(g1 \ U \ g2)} \& (next(EL_{(g1 \ U \ g2)}))) & \text{if } h = g1 \ U \ g2 \\
S_h = S_{g2} \& (S_{g1} | (next(EL_{(g1 \ \mathcal{R} \ g2)}))) & \text{if } h = g1 \ \mathcal{R} \ g2 \\
S_h = S_g \& (next(EL_{(G \ g)})) & \text{if } h = G \ g \\
S_h = S_g | (P_{\mathcal{F} \ g} \& next(EL_{(\mathcal{F} \ g)})) & \text{if } h = \mathcal{F} \ g \\
S_h = (next(EL_{(G\mathcal{F} \ g)})) \& (S_g | P_{G\mathcal{F} \ g}) & \text{if } h = G\mathcal{F} \ g
\end{array}$$

6. **Declare the transitions.** In contrast with [7], our TRANS statements associate an EL-var with its matching S-var like this:

$$\text{TRANS } (\ EL_g = (S_g) \) \text{ for fussy encoding}$$

- TRANS (EL_g -> (S_g)) for sloppy encoding (see §5)
7. **Declare fairness constraints as the negations of the promise variables.**
 FAIRNESS (!P_g) for every promise variable P_g
 FAIRNESS TRUE if there are no promise variables.
 8. **Add the specification SPEC !(EL_f & EG true).**
 9. **Use CadenceSMV to check the resulting symbolic automaton.** In CadenceSMV, next() statements may not be nested or present in INIT, FAIRNESS, or SPEC statements. Our solution is to use EL-variables in INIT and SPEC statements and use Promise vars in FAIRNESS statements. TGBA-formatted symbolic automata cannot be checked using NuSMV because variable definitions (ie DEFINE-statements) may only assign simple expressions composed of state variables[23]. Therefore, NuSMV cannot parse the next() operators in our DEFINE section. While NuSMV ASSIGN-statements do allow next() operators, they must occur alone on the left-hand-side of the assignment, which still excludes our TGBA construction.

Theorem 1 *Let ϕ be an LTL formula. Let A_ϕ be the symbolic TGBA that represents ϕ . Then A_ϕ accepts exactly the infinite words over the alphabet 2^{AP} that satisfy ϕ .*

The correctness of our construction follows directly from the proof of Theorem 1 in [8], and can also be adapted from the proof in [7], both of which are based on induction on the structure of the formula. Our construction changes the set constructions of the proof in [7] because we are now reasoning over transitions instead of states. For example, instead of defining the function $sat(g)$ that describes the set of states that satisfy g for every subformula g in f , such that $sat(g \cup h) = sat(h) \cup (sat(g) \cap sat(x(g \cup h)))$ and adding the fairness constraint $sat(\neg(g \cup h) \vee h)$, we define $sat(g)$ over transitions and let $sat(g \cup h) = sat(h) \cup (sat(g) \cap sat(next(g \cup h)) \cap sat(P_{g \cup h}))$ where, as [8] shows, $P_{g \cup h}$ resolves to $(g \cup h) \wedge \neg h$. In this case, the fairness constraint is $\neg P_{g \cup h}$ which is equivalent to the GBA fairness constraint since $\neg(g \cup h \wedge \neg h) = \neg(g \cup h) \vee h$.

5 Symbolic Automaton Encoding Issues

In addition to defining a TGBA translation, we adapted three other optimizations for creating symbolic automata. CGH describes creating GBAs from LTL formulas in Boolean Normal Form (BNF). We compare BNF-based automata to automata created from Negation Normal Form (NNF) formulas. We borrow sloppy encoding as a method for describing the transition relation from PSV [22], who found it preferable to fussy encoding for the modal logic \mathcal{K} . Furthermore, we explore a small collection of BDD variable ordering schemes and measured their effect on the model checking step.

Definition 1 Boolean Normal Form (BNF) *Traverse the parse tree for ϕ in order, changing the operators of ϕ to only \neg , \vee , \mathcal{X} , \cup , and \mathcal{F} . In other words, replace \wedge , \rightarrow , \mathcal{R} , and \mathcal{G} with their equivalents:*

⁷ Note that there may also be a variable for $EL_{\mathcal{X} g}$ itself if nested \mathcal{X} -operators occur in the formula since CadenceSMV syntax forbids nesting next() calls.

⁸ In practice, this step can be optimized considerably for non-temporally-rooted subformulas. For example, we avoid declaring S_a and just use a .

```

MODULE main
VAR
  /*declare a boolean variable for each atomic proposition in f*/
  a : boolean;
  b : boolean;
  ...
VAR
  /*and declare a new variable for each EL_var in el_list*/
  EL_f : boolean; /*f is the input LTL formula*/
  EL_g1 : boolean;
  EL_g2 : boolean;
  ...
  EL_gn : boolean;
DEFINE
  /*for each S_h in the characteristic function, put a line here*/
  S_g = ...
  S_h = ...
  ...
  /*for each Promise variable, put a line here*/
  P_g_U_h := r_g_U_h & ! S_h
  P_F_g := r_F_g & ! S_g
  P_GF_g := r_GF_g & ! S_g
  ...
TRANS
  /*for each EL-var in el_list, generate a line here*/
  /*Basically, for every X, U, R, G, or F in the parse tree, generate a line*/
  ( EL_g1 = S_g1 ) &
  ...
  ( EL_gn = S_gn )
FAIRNESS (!P_g1)
...
FAIRNESS (!P_gn)
SPEC !(EL_f & EG TRUE)

```

Fig. 1. TGBA symbolic automaton for SMV

<pre> MODULE main /*formula: ((X (a)) & ((b)U (!(a)))*)/ VAR /*declare a boolean variable for each atomic proposition in f*/ a : boolean; b : boolean; VAR /*and declare a new variable EL_X_g for each formula (X g) in el_list generated by a primary operator X, U, R, G, or F*/ EL_X_a : boolean; EL_X_b_U_NOT_a : boolean; DEFINE /*for each S_h in the characteristic function, put a line here*/ S_X_a_AND_b_U_NOT_a := (EL_X_a) & (S_b_U_NOT_a); S_b_U_NOT_a := (!(a)) (b & EL_X_b_U_NOT_a); TRANS /*for each (X g) in el_list, generate a line here*/ (EL_X_a = (next(a))) & (EL_X_b_U_NOT_a = (next(S_b_U_NOT_a))) FAIRNESS (!(S_b_U_NOT_a (!(a))) SPEC !(S_X_a_AND_b_U_NOT_a & EG TRUE) </pre>	<pre> MODULE main /*formula: ((X (a)) & ((b)U (!(a)))*)/ VAR /*declare a boolean variable for each atomic proposition in f*/ a : boolean; b : boolean; VAR /*and declare a new variable for each EL_var in el_list*/ EL_X_a_AND_b_U_NOT_a : boolean; P_b_U_NOT_a : boolean; EL_b_U_NOT_a : boolean; DEFINE /*for each S_h in the characteristic function, put a line here*/ S_X_a_AND_b_U_NOT_a := (S_X_a) & (EL_b_U_NOT_a); S_X_a := (next(a)); S_b_U_NOT_a := ((!(a)) (b & P_b_U_NOT_a & (next(EL_b_U_NOT_a)))); TRANS /*for each EL_var in el_list, generate a line here*/ (EL_X_a_AND_b_U_NOT_a = (S_X_a_AND_b_U_NOT_a)) & (EL_b_U_NOT_a = (S_b_U_NOT_a)) FAIRNESS (!P_b_U_NOT_a) SPEC !(EL_X_a_AND_b_U_NOT_a & EG TRUE) </pre>
--	--

Fig. 2. CadenceSMV symbolic GBA with fussy encoding

Fig. 3. CadenceSMV symbolic TGBA with fussy encoding

Here are two symbolic automata encodings for the sample specification $\phi = ((Xa) \wedge (b \ U \ \neg a))$. Figure 2 matches the GBA-based encoding defined by CGH [7]. Figure 3 follows our TGBA-based encoding, declaring a promise variable for the U -subformula, specifying an elementary variable for the whole formula, and pushing the labels from the states to the transitions.

$$g_1 \wedge g_2 \equiv \neg(\neg g_1 \vee \neg g_2)$$

$$g_1 \rightarrow g_2 \equiv \neg g_1 \vee g_2$$

$$g_1 \mathcal{R} g_2 \equiv \neg(\neg g_1 \mathcal{U} \neg g_2)$$

$$\mathcal{G} g_1 \equiv \neg \mathcal{F} \neg g_1$$

Definition 2 Negation Normal Form (NNF) *Traverse the parse tree in a pre-order fashion, using DeMorgan's laws to push the negations in ϕ inwards until only variables are negated. For LTL formulas, we using the following equivalence rules:*

$$\begin{array}{ll} \neg\neg g \equiv g & \neg(\mathcal{X} g) \equiv \mathcal{X}(\neg g) \\ \neg(g_1 \wedge g_2) \equiv (\neg g_1) \vee (\neg g_2) & \neg(g_1 \mathcal{U} g_2) \equiv (\neg g_1 \mathcal{R} \neg g_2) \\ \neg(g_1 \vee g_2) \equiv (\neg g_1) \wedge (\neg g_2) & \neg(g_1 \mathcal{R} g_2) \equiv (\neg g_1 \mathcal{U} \neg g_2) \\ (g_1 \rightarrow g_2) \equiv (\neg g_1) \vee g_2 & \neg(\Box g) \equiv \Diamond(\neg g) \\ & \neg(\Diamond g) \equiv \Box(\neg g) \end{array}$$

Transition Encoding Methods Once a formula is in Negation Normal Form, we can use sloppy encoding instead of fussy encoding. (NNF is necessary due to the way negative \mathcal{U} -formulas are checked by the fixpoint algorithm.) Here is a list describing the differences between the two:

fussy	sloppy
* symbolic automaton has iff-transitions	* symbolic automaton has if-transitions
* TRANS (EL _g = (S _g))	* TRANS (EL _g -> (S _g))
* describes a <i>non-deterministic</i> automaton with trap-states for the negations of \mathcal{X} -transitions	* describes a smaller, usually more <i>non-deterministic</i> automaton

Variable Ordering CadenceSMV and NuSMV search for a fair path in the model-automaton product using a BDD-based fixpoint algorithm, a process whose efficacy is highly sensitive to variable ordering.[3] Finding an optimal BDD variable ordering is NP-complete and, for any constant $c > 1$, it is NP-Hard to compute a variable ordering to make a BDD at most c times larger than optimal. [26] Therefore, finding good heuristics is key. We experiment three heuristics defined by Koster et al [15] and Tarjan and Yannakakis [29], comparing them to the default heuristics of CadenceSMV, and a simple ordering resulting from a pre-order, depth-first traversal of the variable graph.

We form the variable graph from the parse tree of the input formula by identifying nodes in the parse tree corresponding to the roots of the subformulas for which we declare variables. We connect each variable-labeled vertex to its closest ancestor and descendant(s). Unary operator rooted subformula variables are connected to only one closest descendant while binary operator rooted subformula variables are connected to their closest left and right descendants. Figure 4 displays the variable graph for our example formula. We implemented the four variable ordering schemes listed in Table 1, all of which take the variable graph as input.

Algorithm: Form variable graph from parse tree.

1. Each variable is a vertex in the graph. Vertices are either:
 - * propositions \in AP
 - * Elementary (EL) variables
 - * Promise variables
2. Each vertex has a self-loop.
3. Each vertex, except the root, is connected to its closest ancestor.
4. Each non-leaf vertex is connected to its closest descendant(s).

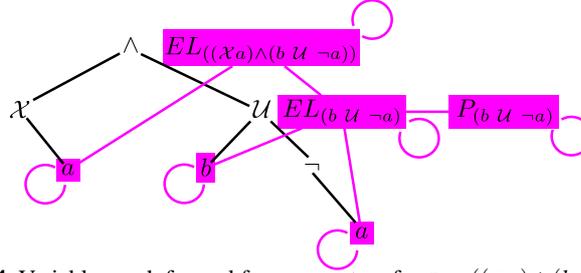


Fig. 4. Variable graph formed from parse tree for $\phi = ((x a) \wedge (b U \neg a))$

Table 1: Variable Ordering Schemes

Pre-order, Depth First Search	naïve	formed directly from a pre-order, depth first traversal of the syntax DAG of the input LTL formula
Lexicographic Breadth First Search, Variant Perfect	LEX_P	Koster et al’s [15] variant of [24] which triangulates the variable graph focusing on either perfection or minimality, labels vertices with their already ordered neighbors in decreasing order by position, and chooses the highest lexicographically-labeled vertex next.
Lexicographic Breadth First Search, Variant Minimal	LEX_M	
Maximum Cardinality Search	MCS	Koster et al’s [15] variant of [29] which also triangulates the variable graph but selects the vertex adjacent to the highest number of ordered vertices next. We seed MCS with an initial vertex, chosen either to have the maximum or minimum degree.

6 Experimental Methods

Test Methods Each test was performed in two steps. First, we applied our LTL-to-symbolic automaton translation to the negation of the input formula. Second, each output automaton and variable ordering file pair was checked by CadenceSMV or NuSMV. When we did not specify a specific variable order, we let the tools use their default variable ordering heuristics to try to find an optimal one. To check whether a LTL formula ϕ is satisfiable, we model check $\neg\phi$ against a universal SMV model. For example, if $\phi = ((X a) \wedge (b U \neg a))$, we provide the following input to CadenceSMV:

```

module main () {
  a : boolean;
  b : boolean;
  assert ~(((X a) & (b U (~ a))));
  FAIRNESS TRUE; }

```

SMV negates the specification, $\neg\phi$, symbolically compiles ϕ into $A_{\neg\phi}$, and conjoins $A_{\neg\phi}$ with the universal model M . If $A_M, \neg\phi$ is not empty, then SMV finds a fair path that satisfies ϕ . In this way, SMV acts as both a symbolic compiler and a search engine.

Platform We ran all tests on Shared University Grid at Rice (SUG@R), an Intel Xeon compute cluster.⁵ SUG@R is comprised of 134 SunFire x4150 nodes, each with two quad-core Intel Xeon processors running at 2.83GHz and 16GB of RAM per processor. The OS is Red Hat Enterprise 5 Linux with the 2.6.18 kernel. Each test was run with exclusive access to one node. Times were measured using the Unix `time` command.

Input Formulas Utilizing the benchmarks established by [25], we tested the algorithms using three types of scalable formulas: random formulas, counter formulas, and pattern formulas. Definitions for these formulas are repeated for convenience in Appendix A.

⁵ <http://rcsg.rice.edu/sugar/>

7 Experimental Results

Our experiments demonstrate that the novel encoding methods we have introduced significantly improve the translation of LTL formulas to symbolic automata, as measured in time to check the resulting automata for satisfiability. However, no one encoding method consistently dominates for all types of formulas. Instead, we find that different combinations of encoding methods are better suited to different formulas. Therefore, we recommend using a portfolio approach to algorithm selection [18] where different translations are used depending on the structure of the input formula. We call our tool PANDA for “Portfolio Approach to Navigate the Design of Automata.”

NNF encodings outperform BNF encodings. The one exception to our portfolio approach is the normal form we use for the input formula. We found that symbolic automata generated from NNF formulas consistently dominated those created from BNF formulas. Figure 7 provides one example of this. Using NNF has the added benefit that it allows us to also employ sloppy encoding and generate TGBAs. Therefore, we recommend a portfolio approach which always generates NNF-based automata.

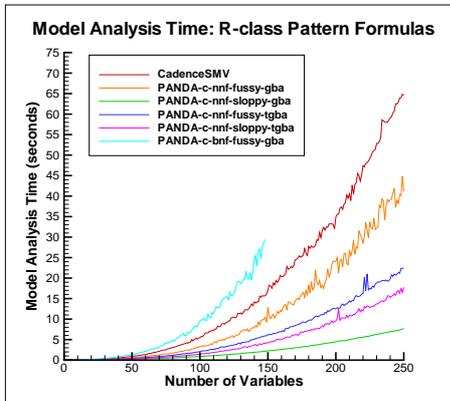


Fig. 5. CadenceSMV

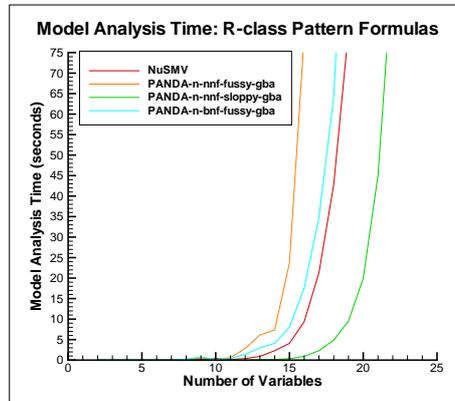


Fig. 6. NuSMV

Performance Results: $R(n) = \bigwedge_{i=1}^n (\Box \Diamond p_i \vee \Diamond \Box p_{i+1})$

Sloppy encoding shows promise for temporally-heavy formulas. While we did not see a significant improvement in model analysis time for Boolean formulas, sloppy encoding dramatically improved the performance of both CadenceSMV and NuSMV in many of our tests of temporal formulas. Figure 7 demonstrates that for both GBAs and (in the case of CadenceSMV) TGBAs, using sloppy encoding can shorten model analysis time.

TGBAs translate smaller automata into smaller model analysis times. We found that the automata encoding optimizations used for TGBAs, such as the special treatment of the \mathcal{GF} operator and use of promise variables, translated into better performance. Figure 7 shows the dramatic improvement we saw when encoding \mathcal{U} -heavy formulas while 8 demonstrates optimizing for \mathcal{GF} operators. From here on, we compare our tool using CadenceSMV as a back-end checker to using CadenceSMV alone. We found our automata with NuSMV as a back end produced different timing results than the same automata with CadenceSMV as a back end. As our objective is to compare LTL-to-automata algorithms and not model checking software, and NuSMV cannot check all of our automata, we focus on CadenceSMV.

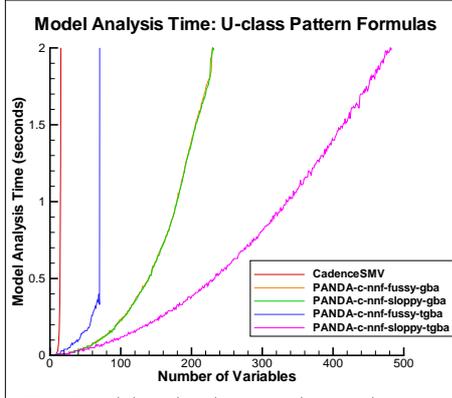


Fig. 7. $U(n) = (\dots (p_1 \ \bar{u} \ p_2) \ \bar{u} \ \dots) \ \bar{u} \ p_n$

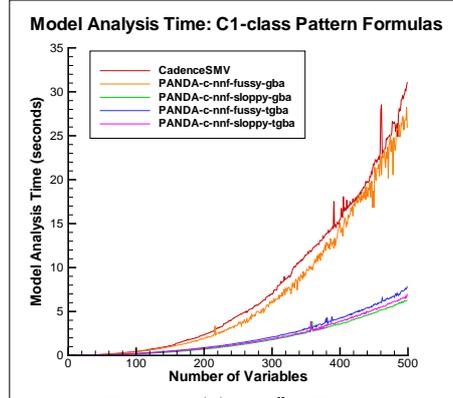


Fig. 8. $C_1(n) = \bigvee_{i=1}^n \square \diamond p_i$

Also, we found TGBAs fared well when model size was a problem. For example, the state space for our counter formulas grows exponentially with the number of bits in the counter. Figure 9 shows that when CadenceSMV and GBA-based encodings fail to create a symbolic automata/model check due to the large size of the state space, the smaller TGBA encodings were able to stay in the game.

LEXP variable ordering benefits TGBA encodings.

While several of the variable orderings we tested matched or even slightly exceeded CadenceSMV’s build-in variable ordering heuristics, none of them ever bested CadenceSMV significantly or consistently on a symbolic GBA. Figure 7 shows a fairly typical example of the results we obtained using our variable orderings with symbolic GBA. Frequently, either the LEXP, as in this case, or naïve variable orderings would perform very similarly to our GBA encoding with the CadenceSMV default ordering. The other variable orderings nearly always performed worse. This graph is continued to the right. (Observe the line representing the NNF, fussy, GBA encoding in the upper left-hand corner of Figure 7.) Here we observe a typical spread caused by varying the variable ordering on symbolic TGBAs. All of the variable orderings we tested performed similar to, or worse than, the default CadenceSMV heuristic except for LEXP, which improves performance slightly.

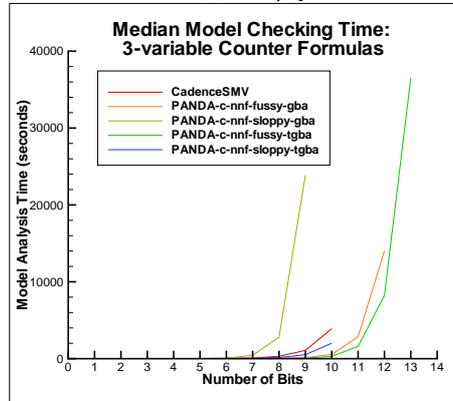


Fig. 9. 3-Variable Counters

Though CadenceSMV appears to have good ordering heuristics for GBAs, the performance of our symbolic TGBAs was frequently improved by using LEXP variable ordering. Figure 13 demonstrates the most dramatic display we found of this phenomenon using our R_2 pattern formulas. Since CadenceSMV does not accept \mathcal{R} operators, we eliminated them from our LTLSEPCs using the conversion $(g_1 \mathcal{R} g_2) \equiv \neg(\neg g_1 \ \bar{u} \ \neg g_2)$. Our symbolic TGBAs grew to 923 variables before triggering the message token too large, exceeds YYLMAX Command exited with non-zero status 2.

7.1 Objective Function

From our pattern formula experiments, we devised a very simple objective function to choose an encoding based on the operators present in the input formula. We make no

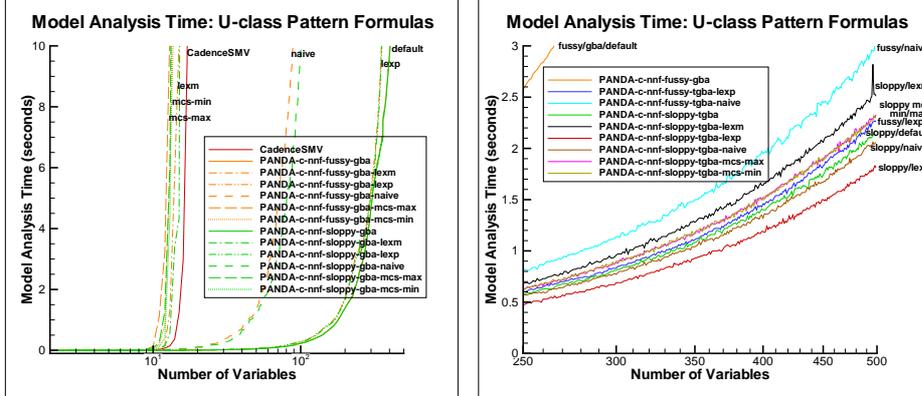


Fig. 10. Performance Results: $U(n) = (\dots(p_1 \cup p_2) \cup \dots) \cup p_n$

claim that this heuristic is optimal; in fact we are sure it is not. However, our aim here is simply to demonstrate that our novel encodings translate to a significant performance improvement in the symbolic domain as they did in the domains from which we derived them. It costs us nothing to count the operators in the input specification as we parse it. Therefore, we get a very simple heuristic which bests the average model analysis time of CadenceSMV without incurring any overhead for the kind of deep structural analysis required for ensuring an optimal encoding choice every time. We employ a portfolio approach to selecting an encoding based on the following guidelines:

Operator Count Shows	We Choose
Boolean operator dominated	NNF, fussy, GBA, default ordering
More \cup 's and \mathcal{R} 's	NNF, sloppy, TGBA, LEXP ordering
Many \mathcal{F} 's, \mathcal{G} , combined with \cup 's or \mathcal{R} 's	NNF, sloppy, GBA, default ordering
More $\mathcal{G}\mathcal{F}$'s in combination	NNF, sloppy, TGBA, default ordering
Otherwise	NNF, fussy, GBA, default ordering

Our basic approach is to use sloppy encoding on more temporally difficult formulas and fussy encoding on temporally easier ones. We always convert our input formulas to NNF since that consistently dominated encodings derived from BNF formulas and gives us the option to use sloppy or TGBA encodings. We construct a TGBA when the formula is dominated by binary temporal operators or combinations (ie $\mathcal{G}\mathcal{F}$) and construct a GBA otherwise. We use the LEXP variable ordering with our TGBA encodings to improve performance over the default ordering heuristics for those formulas heavy with binary temporal operators. Figures 11 and 12 show that even with our overly simplistic objective function to guide our portfolio approach, we significantly decrease the CadenceSMV model analysis time for both random and counter formulas.

8 Discussion

Too little attention has been paid to the issue of efficient construction of symbolic automata for LTL formulas. We defined new algorithms for accomplishing this task. Our experiments showed that no one symbolic translation was superior. Thus, we proposed a new algorithmic-portfolio approach. The effectiveness of this approach is evident: even when we use a very simple heuristic for choosing encodings we can significantly dominate the native translation of CadenceSMV. We demonstrated that the increased non-determinism of sloppy encoding can outperform the traditional fussy encoding and

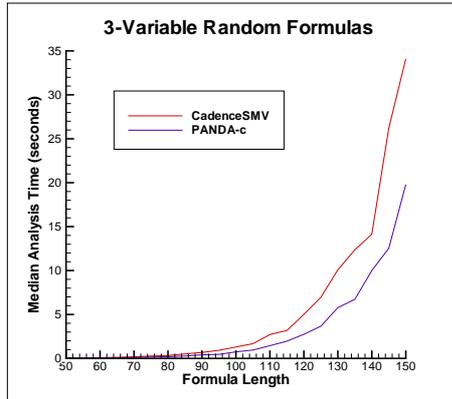


Fig. 11. 3-Variable Random Formulas

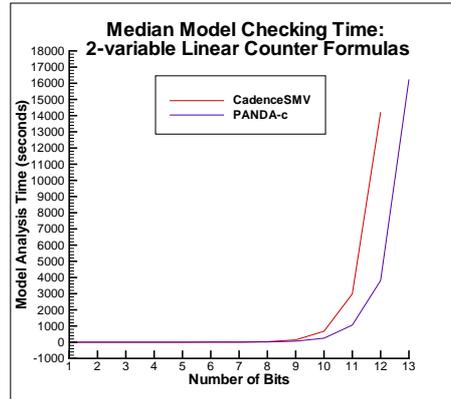


Fig. 12. 2-Variable Linear Counters

that the smaller size of TGBAs can translate to shorter analysis time. We found NNF specifications translate to symbolic automata more efficiently than BNF. Finally, we demonstrated that variable ordering schemes derived from the structure of the specification can also improve performance.

By defining and showing the effectiveness of new symbolic automata encoding techniques, we have only scratched the surface of the possible uses for our tool set. Our simple operator-count heuristic leaves room for a comprehensive study of the relationship between formula structure, symbolic automaton size, and the efficiency of different symbolic automaton encoding techniques, perhaps culminating in a set of formula classes to match specifications with their optimal symbolic automaton encodings with minimal overhead. This kind of in-depth structural analysis would certainly be beneficial, though it was beyond the scope of the experiments in this paper, where the aim of our experiments was simply to support our claim that our novel encoding techniques are useful. Due to the complexity of LTL model checking, which is exponential in the size of the specification, algorithmic advancements, including our better conversion of LTL to automata, can make the difference between whether LTL model checking is practical for the verification of industrial systems or not.

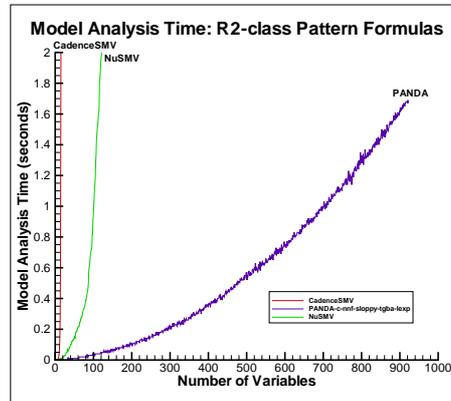


Fig. 13. $R_2(n) = (..(p_1 \mathcal{R} p_2) \mathcal{R} \dots) \mathcal{R} p_n$

References

- [1] Ilan Beer, Shoham Ben-David, Cindy Eisner, and Avner Landver. Rulebase: An industry-oriented formal verification tool. *Design Automation Conf. 0*, pages 655–660, 1996.
- [2] R.K. Brayton, G.D. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.-T. Cheng, S. Edwards, S. Khatri, T. Kukimoto, A. Pardo, S. Qadeer, R.K. Ranjan, S. Sarwary, T.R. Shiple, G. Swamy, and T. Villa. VIS: a system for verification and synthesis. In *CAV, Proc. 8th Int'l Conf*, LNCS 1102, pages 428–432. Springer, 1996.
- [3] R.E. Bryant. Graph-based algorithms for boolean-function manipulation. *IEEE Trans. on Computers C-35*, (8), 1986.

- [4] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Inform. and Computation* 98, (2):142–170, Jun 1992.
- [5] A. Cimatti, E.M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new symbolic model checker. *STTT* 2, (4):410–425, 2000.
- [6] Alessandro Cimatti, Marco Roveri, and Stefano Tonetta. Syntactic optimizations for ps1 verification. In *TACAS*, pages 505–518, 2007.
- [7] E. M. Clarke, O. Grumberg, and K. Hamaguchi. Another look at LTL model checking. *Formal Methods in System Design* 10, (1):47–71, 1997.
- [8] J-M. Couvreur. On-the-fly verification of linear temporal logic. In *Proc. FM*, pages 253–271, 1999.
- [9] N. Daniele, F. Guinchiglia, and M.Y. Vardi. Improved automata generation for linear temporal logic. In *CAV, Proc. 11th Int'l Conf*, LNCS 1633, pages 249–260. Springer, 1999.
- [10] A. Duret-Lutz and D. Poirinaud. SPOT: An extensible model checking library using transition-based generalized büchi automata. In *MASCOTS*, pages 76–83, 2004.
- [11] E.A. Emerson and C.L. Lei. Efficient model checking in fragments of the propositional μ -calculus. In *LICS, 1st Symp.*, pages 267–278, Cambridge, Jun 1986.
- [12] P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In *CAV, Proc. 13th Int'l Conf*, LNCS 2102, pages 53–65. Springer, 2001.
- [13] J. Geldenhuys and H. Hansen. Larger automata and less work for LTL model checking. In *Model Checking Software, 13th SPIN*, LNCS 3925, pages 53–70. Springer, 2006.
- [14] D. Giannakopoulou and F. Lerda. From states to transitions: Improving translation of LTL formulae to Büchi automata. In *FORTE, Proc of 22 IFIP Int'l Conf*, Nov 2002.
- [15] A. M. C. A. Koster, H. L. Bodlaender, and S. P. M. van Hoesel. Treewidth: Computational experiments. ZIB-Report 01–38, ZIB, 2001.
- [16] K.S.Brace, R.L.Rudell, and R.E.Bryant. Efficient implementation of a BDD package. In *DAC*, pages 40–45. ACM, 1990.
- [17] O. Kupferman. Sanity checks in formal verification. In *concur06*, lncs 4137, pages 37–51. springer, 2006.
- [18] Kevin Leyton-brown, Eugene Nudelman, Galen Andrew, Jim Mcfadden, and Yoav Shoham. A portfolio approach to algorithm selection. In *In IJCAI-03*, pages 1542–1543, 2003.
- [19] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *popl85*, pages 97–107, 1985.
- [20] K. McMillan. The SMV language. Technical report, Cadence Berkeley Lab, 1999.
- [21] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
- [22] G. Pan, U. Sattler, and M.Y. Vardi. BDD-based decision procedures for K. In *Proc. 18th Int'l CADE*, LNCS 2392, pages 16–30. Springer, 2002.
- [23] A. Cimatti R. Cavada, C. Jochim, G. Keighren, E. Olivetti, M. Pistore, M. Roveri, and A. Tchaltev. Nusmv 2.4 user manual. Technical report, CMU and ITC-irst, 2005.
- [24] Donald J. Rose, Robert Endre Tarjan, and George S. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM J. Comput.* 5, (2):266–283, 1976.
- [25] K.Y. Rozier and M.Y. Vardi. LTL satisfiability checking. In *Proc. 14th Workshop on Model Checking Software (SPIN)*, LNCS 4595, pages 149–167. Springer, 2007.
- [26] Detlef Sieling. The nonapproximability of obdd minimization. *Information and Computation* 172, pages 103–138, 1998.
- [27] A.P. Sistla and E.M. Clarke. The complexity of propositional linear temporal logic. *J. ACM* 32, pages 733–749, 1985.
- [28] F. Somenzi and R. Bloem. Efficient Büchi automata from LTL formulae. In *CAV, Proc. 12th Int'l Conf*, LNCS 1855, pages 248–263. Springer, 2000.
- [29] Robert E. Tarjan and Mihalis Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM J. Comput.* 13, (3):566–579, 1984.
- [30] M.Y. Vardi. Automata-theoretic model checking revisited. In *vmcai07*, LNCS 4349, pages 137–150. Springer, 2007.
- [31] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *lics86*, pages 332–344, Cambridge, Jun 1986.

Appendix A: Input Formulas

Utilizing the benchmarks established by [25], we tested the algorithms using three types of scalable formulas: random formulas, counter formulas, and pattern formulas. All tools were applied to the same formulas and the timing results were compared. Unlike in [25], correctness of the SAT/UNSAT answers was not an issue. The only cases where the tools disagreed were attributable to the semantic subtleties covered in §3.

Random Formulas In order to cover as much of the problem space as possible, we generated random formulas as in [9]. We created sets of 500 formulas varying the number of variables, N , from 1 to 3, and the length of the formula, L , from 5 to 200. We chose from the operator set $\{\neg, \vee, \wedge, X, \mathcal{U}, \mathcal{R}, \mathcal{G}, \mathcal{F}, \mathcal{G}\mathcal{F}\}$. (We included the combination $\mathcal{G}\mathcal{F}$ as a single operator because that combination occurs so frequently in industrial safety properties.) To create formulas with both a nontrivial temporal structure and a nontrivial Boolean structure, the probability of choosing a temporal operator was $P = 0.5$. Other choices were decided uniformly. All formulas were generated prior to testing, so each tool was run on the *same* formulas.

Counter Formulas To measure performance on scalable, temporally complex formulas with large state spaces, we tested our algorithms on formulas that describe n -bit binary counters with increasing values of n . We know precisely the unique counterexample for each counter formula and the requisite number of states for the automaton. We tested four constructions of binary counter formulas, varying two factors: number of variables and nesting of X 's. These formulas were originally defined in [25].

We can represent a binary counter using two variables: a counter variable and a marker variable to designate the beginning of each new counter value. Alternatively, we can use 3 variables, adding a variable to encode carry bits, which eliminates the need for \mathcal{U} -connectives in the formula. We can nest X 's to provide more succinct formulas or express the formulas using a conjunction of un-nested X -sub-formulas.

Let b be an atomic proposition. Then a computation π over b is a word in $(2^{\{0,1\}})^\omega$. By dividing π into blocks of length n , we can view π as a sequence of n -bit values, denoting the sequence of values assumed by an n -bit counter starting at 0, and incrementing successively by 1. To simplify the formulas, we represent each block b_0, b_1, \dots, b_{n-1} as having the most significant bit on the right and the least significant bit on the left. For example, for $n = 2$ the b blocks cycle through the values 00, 10, 01, and 11. For technical convenience, we use an atomic proposition m to mark the blocks. That is, we intend m to hold at point i precisely when $i = 0 \pmod n$.

For π to represent an n -bit counter, the following properties need to hold:

- 1) The marker consists of a repeated pattern of a 1 followed by $n-1$ 0's.
- 2) The first n bits are 0's.
- 3) If the least significant bit is 0, then it is 1 n steps later and the other bits do not change.
- 4) All of the bits before and including the first 0 in an n -bit block flip their values in the next block; the other bits do not change.

For $n = 4$, these properties are captured by the conjunction of the following formulas:

1. $(m) \ \&\& \ (\ [](m \rightarrow ((X(!m)) \ \&\& \ (X(X(!m))) \ \&\& \ (X(X(X(!m)))) \ \&\& \ X(X(X(X(m))))))$
2. $(!b) \ \&\& \ (X(!b)) \ \&\& \ (X(X(!b))) \ \&\& \ (X(X(X(!b))))$
3. $[]((m \ \&\& \ !b) \rightarrow (X(X(X(X(b)))) \ \&\& \$

