

A Multi-Encoding Approach for LTL Symbolic Satisfiability Checking

Kristin Y. Rozier^{1,2*} and Moshe Y. Vardi²

¹ NASA Ames Research Center, Moffett Field CA, 94035, USA.

Kristin.Y.Rozier@nasa.gov, <http://ti.arc.nasa.gov/profile/kyrozier/>

² Rice University, Houston, Texas 77005, USA.

vardi@cs.rice.edu, <http://www.cs.rice.edu/~vardi/>

Abstract. Formal behavioral specifications written early in the system-design process and communicated across all design phases have been shown to increase the efficiency, consistency, and quality of the system under development. To prevent introducing design or verification errors, it is crucial to test specifications for *satisfiability*. Our focus here is on specifications expressed in linear temporal logic (LTL).

We introduce a novel encoding of symbolic transition-based Büchi automata and a novel, “sloppy,” transition encoding, both of which result in improved scalability. We also define novel BDD variable orders based on tree decomposition of formula parse trees. We describe and extensively test a new multi-encoding approach utilizing these novel encoding techniques to create 30 encoding variations. We show that our novel encodings translate to significant, sometimes exponential, improvement over the current standard encoding for symbolic LTL satisfiability checking.

1 Introduction

In *property-based design* formal properties, written in temporal logics such as LTL [35], are written early in the system-design process and communicated across all design phases to increase the efficiency, consistency, and quality of the system under development [39, 41]. Property-based design and other design-for-verification techniques capture design intent precisely, and use formal logic properties both to guide the design process and to integrate verification into the design process [28]. The shift to specifying desired system behavior in terms of formal logic properties risks introducing specification errors in this very initial phase of system design, raising the need for *property assurance* [34, 39].

The need for checking for errors in formal LTL properties expressing desired system behavior first arose in the context of model checking, where *vacuity checking* aims at reducing the likelihood that a property that is satisfied by the model under verification is an erroneous property [3, 31]. Property assurance is more challenging at the initial phases of property-based design, before a model of the implementation has been specified. *Inherent vacuity checking* is a set of sanity checks that can be applied to a

* Work contributing to this paper was completed at Rice University, Cambridge University, and NASA, was supported in part by the Shared University Grid at Rice (SUG@R), and was funded by NSF under Grant EIA-0216467, NASA’s Airspace Systems Program, and a partnership between Rice University, Sun Microsystems, and Sigma Solutions, Inc.

set of temporal properties, even before a model of the system has been developed, but many possible errors cannot be detected by inherent vacuity checking [22].

A stronger sanity check for a set of temporal properties is LTL *realizability* checking, in which we test whether there is an open system that satisfies all the properties in the set [36], but such a test is very expensive computationally. In LTL *satisfiability* checking, we test whether there is a closed system that satisfies all the properties in the set. The satisfiability test is weaker than the realizability test, but its complexity is lower; it has the same complexity as LTL model checking [44]. In fact, LTL satisfiability checking can be implemented via LTL model checking; see below.

Indeed, the need for LTL satisfiability checking is widely recognized [16, 27, 29, 32, 40]. Foremost, it serves to ensure that the behavioral description of a system is internally consistent and neither over- or under-constrained. If an LTL property is either *valid*, or *unsatisfiable* this must be due to an error. Consider, for example, the specification *always*($b_1 \rightarrow \textit{eventually } b_2$), where b_1 and b_2 are propositional formulas. If b_2 is a tautology, then this property is valid. If b_2 is a contradiction, then this property is unsatisfiable. Furthermore, the collective set of properties describing a system must be satisfiable, to avoid contradictions between different requirements. Satisfiability checking is particularly important when the set of properties describing the design intent continues to evolve, as properties are added and refined, and have to be checked repeatedly. Because of the need to consider large sets of properties, it is critical that the satisfiability test be *scalable*, and able to handle complex temporal properties. This is challenging, as LTL satisfiability is known to be PSPACE-complete [44].

As pointed out in [40], satisfiability checking can be performed via model checking: a *universal model* (that is, a model that allows all possible traces) does not satisfy a linear temporal property $\neg f$ precisely when f is satisfiable. In [40] we explored the effectiveness of model checkers as LTL satisfiability checkers. We compared there the performance of explicit-state and symbolic model checkers. Both use the automata-theoretic approach [48] but in a different way. Explicit-state model checkers translate LTL formulas to Büchi automata explicitly and then use an explicit graph-search algorithm [13]. For satisfiability checking, the construction of the automaton is the more demanding task. Symbolic model checkers construct symbolic encodings of automata and then use a symbolic nonemptiness test. The symbolic construction of the automaton is easy, but the nonemptiness test is computationally demanding. The extensive set of experiments described in [40] showed that the symbolic approach to LTL satisfiability is significantly superior to the explicit-state approach in terms of scalability.

In the context of explicit-state model checking, there has been extensive research on optimized construction of automata from LTL formulas [14, 15, 23, 25, 26, 43, 45, 46], where a typical goal is to minimize the size of constructed automata [47]. Optimizing the construction of symbolic automata is more difficult, as the size of the symbolic representation does not correspond directly to its optimality. An initial symbolic encoding of automata was proposed in [8], but the optimized encoding we call *CGH*, proposed by Clarke, Grumberg, and Hamaguchi [12], has become the de facto standard encoding. CGH encoding is used by model checkers such as CadenceSMV and NuSMV, and has been extended to symbolic encodings of industrial specification languages [11]. Surprisingly, there has been little follow-up research on this topic.

In this paper, we propose novel symbolic LTL-to-automata translations and utilize them in a new multi-encoding approach to achieve significant, sometimes exponential, improvement over the current standard encoding for LTL satisfiability checking. First we introduce and prove the correctness of a novel encoding of symbolic automata inspired by optimized constructions of explicit automata [14, 26]. While the CGH encoding uses *Generalized Büchi Automata* (GBA), our new encoding is based on *Transition-Based Büchi Automata* (TGBA). Second, inspired by work on symbolic satisfiability checking for modal logic [33], we introduce here a novel *sloppy* encoding of symbolic automata, as opposed to the *fussy* encoding used in CGH. Sloppy encoding uses looser constraints, which sometimes results in smaller BDDs. The sloppy approach can be applied both to GBA-based and TGBA-based encodings, provided that one uses negation-normal form (NNF), [45], rather than the Boolean normal form (BNF) used in CGH. Finally, we introduce several new variable-ordering schemes, based on tree decomposition of the LTL parse tree, inspired by observations that relate tree decompositions to BDD variable ordering [19]. The combination of GBA/TGBA, fussy/sloppy, BNF/NNF, and different variable orders yields a space of 30 possible configurations of symbolic automata encodings. (Not all combinations yield viable configurations.)

Since the value of novel encoding techniques lies in increased *scalability*, we evaluate our novel encodings in the context of LTL satisfiability checking, utilizing a comprehensive and challenging collection of widely-used benchmark formulas [9, 16, 27, 40]. For each formula, we perform satisfiability checking using all 30 encodings. (We use CadenceSMV as our experimental platform.) Our results demonstrate conclusively that no encoding performs best across our large benchmark suite. Furthermore, no single approach—GBA vs. TGBA, fussy vs. sloppy, BNF vs. NNF, or any one variable order, is dominant. This is consistent with the observation made by others [1, 47], that in the context of symbolic techniques one typically does not find a “winning” algorithmic configuration. In response, we developed a multi-encoding tool, PANDA, which runs several encodings in parallel, terminating when the first process returns. Our experiments demonstrate conclusively that the multi-encoding approach using the novel encodings invented in this paper achieves substantial improvement over CGH, the current standard encoding; in fact PANDA significantly bested the native LTL model checker built into CadenceSMV.

The structure of this paper is as follows. We review the CGH encoding [12] in Section 2. Next, in Section 3, we describe our novel symbolic TGBA encoding. We introduce our novel sloppy encoding and our new methods for choosing BDD variable orderings and discuss our space of symbolic encoding techniques in Section 4. After setting up our scalability experiment in Section 5, we present our test results in Section 6, followed by a discussion in Section 7. Though our construction can be used with different symbolic model checking tools, in this paper, we follow the convention of [12] and give examples of all constructions using the SMV syntax.

2 Preliminaries

We assume familiarity with LTL [18]; For convenience, Appendix A defines LTL semantics. We use two normal forms:

Definition 1 Boolean Normal Form (BNF) rewrites the input formula to use only \neg , \vee , \mathcal{X} , \mathcal{U} , and \mathcal{F} . In other words, we replace \wedge , \rightarrow , \mathcal{R} , and \mathcal{G} with their equivalents:

$$\begin{aligned} g_1 \wedge g_2 &\equiv \neg(\neg g_1 \vee \neg g_2) & g_1 \mathcal{R} g_2 &\equiv \neg(\neg g_1 \mathcal{U} \neg g_2) \\ g_1 \rightarrow g_2 &\equiv \neg g_1 \vee g_2 & \mathcal{G} g_1 &\equiv \neg \mathcal{F} \neg g_1 \end{aligned}$$

Definition 2 Negation Normal Form (NNF) pushes negation inwards until only atomic propositions are negated, using the following rules:

$$\begin{aligned} \neg \neg g &\equiv g & \neg(\mathcal{X} g) &\equiv \mathcal{X}(\neg g) \\ \neg(g_1 \wedge g_2) &\equiv (\neg g_1) \vee (\neg g_2) & \neg(g_1 \mathcal{U} g_2) &\equiv (\neg g_1 \mathcal{R} \neg g_2) \\ \neg(g_1 \vee g_2) &\equiv (\neg g_1) \wedge (\neg g_2) & \neg(g_1 \mathcal{R} g_2) &\equiv (\neg g_1 \mathcal{U} \neg g_2) \\ (g_1 \rightarrow g_2) &\equiv (\neg g_1) \vee g_2 & \neg(\mathcal{G} g) &\equiv \mathcal{F}(\neg g) \\ & & \neg(\mathcal{F} g) &\equiv \mathcal{G}(\neg g) \end{aligned}$$

In automata-theoretic model checking, we represent LTL formulas with Büchi automata.

Definition 3 A Generalized Büchi Automaton (GBA) is a quintuple $(Q, \Sigma, \delta, Q_0, F)$, where:

- $\delta \subseteq Q \times \Sigma \times Q$ is a transition relation.
- Q is a finite set of states.
- $Q_0 \subseteq Q$ is a set of initial states.
- Σ is a finite alphabet.
- $F \subseteq 2^Q$ is a set of accepting state sets.

A run of a Büchi automaton A over an infinite trace $\pi = \pi_0, \pi_1, \pi_2, \dots \in \Sigma$ is a sequence q_0, q_1, q_2, \dots of states such that $q_0 \in Q_0$, and $\langle q_i, \pi_i, q_{i+1} \rangle \in \delta$ for all $i \geq 0$. A accepts π if the run over π visits states in every set in F infinitely often. We denote the set of infinite traces accepted by A by $\mathcal{L}_\omega(A)$.

A trace satisfying LTL formula f is an infinite run over the alphabet $\Sigma = 2^{Prop}$, where $Prop$ is the underlying set of atomic propositions. We denote by $models(f)$ the set of traces satisfying f . The next theorem relates the expressive power of LTL to that of Büchi automata.

Theorem 1 [49] *Given an LTL formula f , we can construct a generalized Büchi automaton $A_f = \langle Q, \Sigma, \delta, Q_0, F \rangle$ such that $|Q|$ is in $2^{O(|f|)}$, $\Sigma = 2^{Prop}$, and $\mathcal{L}_\omega(A_f)$ is exactly $models(f)$.*

This theorem reduces LTL satisfiability checking to automata-theoretic nonemptiness checking, as f is satisfiable iff $models(f) \neq \emptyset$ iff $\mathcal{L}_\omega(A_f) \neq \emptyset$.

LTL satisfiability checking relates to LTL model checking as follows. We use a *universal model* M that generates all traces over $Prop$ such that $\mathcal{L}_\omega(M) = (2^{Prop})^\omega$. The code for this model appears in [40] and Appendix B. We now have that M does not satisfy $\neg f$ iff f is satisfiable. We use a symbolic model checker to check the formula $\neg f$ against M ; f is satisfiable precisely when the model checker finds a counterexample.

CGH encoding In this paper we focus on LTL to symbolic Büchi automata compilation. We recap the CGH encoding [12], which assumes that the formula f is in BNF, and then forms a symbolic GBA. We first define the *CGH-closure* of an LTL formula f as the set of all subformulas of f (including f itself), where we also add the formula $\mathcal{X}(g \mathcal{U} h)$

for each subformula of the form $g \mathcal{U} h$. The \mathcal{X} -formulas in the CGH-closure of f are called *elementary* formulas.

We declare a Boolean SMV variable $EL_{\mathcal{X}g}$ for each elementary formula $\mathcal{X}g$ in the CGH-closure of f . Also, each atomic proposition in f is declared as a Boolean SMV variable. We define an auxiliary variable S_h for every formula h in the CGH-closure of f . (Auxiliary variables are substituted away by SMV and do not required allocated BDD variables.) The characteristic function for an auxiliary variable S_h is defined as follows:

$$\begin{aligned} S_h &= p \text{ if } p \in AP & S_h &= \neg S_g \text{ if } h = \neg g & S_h &= S_{g_1} | S_{g_2} & \text{if } h = g_1 \vee g_2 \\ S_h &= EL_h \text{ if } h \text{ is a formula } \mathcal{X}g & S_h &= S_{g_2} | (S_{g_1} \& S_{\mathcal{X}(g_1 \mathcal{U} g_2)}) & \text{if } h = g_1 \mathcal{U} g_2 \end{aligned}$$

We now generate the SMV model M_f :

```

MODULE main
VAR
  a: boolean; /*declare a Boolean var for each atomic prop in f */
  EL_Xg: boolean; /*declare a Boolean var for every formula Xg in the CGH-closure*/
DEFINE /*auxiliary vars according to characteristic function */
  S_h := ...
TRANS /*for every formula Xg in the CGH-closure, add a transition constraint*/
  (S_Xg = next(S_g))
FAIRNESS !S_gUh | S_h /*for each subformula gUh */
FAIRNESS TRUE /*or a generic fairness condition otherwise*/
SPEC !(S_f & EG true) /*end with a SPEC statement*/

```

The traces of M_f correspond to the accepting runs of A_f , starting from arbitrary states. Thus, satisfiability of f corresponds to nonemptiness of M_f , starting from an initial state. We can model check such nonemptiness with SPEC $!(S_f \ \& \ EG \ true)$. A counterexample is an infinite trace starting at a state where S_f holds. Thus, the model checker returns a counterexample that is a trace satisfying f .

Remark 1 *While the syntax we use is shared by CadenceSMV and NuSMV, the precise semantics of CTL model checking in these model checkers is not fully documented and there are some subtle but significant differences between the two tools. Therefore, we use CadenceSMV semantics here and describe these subtleties in Appendix C.*

3 A Symbolic Transition-Based Generalized Büchi Automata (TGBA) Encoding

We now introduce a novel symbolic encoding, referred to as TGBA, inspired by the explicit-state transition-based Generalized Büchi automata of [26]. Such automata are used by SPOT [17], which was shown experimentally [40] to be the best explicit LTL translator for satisfiability checking.

Definition 4 *A Transition-Based Generalized Büchi Automaton (TGBA) is a quintuple $(Q, \Sigma, \delta, Q_0, F)$, where:*

- $\delta \subseteq Q \times \Sigma \times Q$ is a transition relation.
- $Q_0 \subseteq Q$ is a set of initial states.
- Q is a finite set of states.
- $F \subseteq 2^\delta$ is a set of accepting transitions.
- Σ is a finite alphabet.

A run of a TGBA over an infinite trace $\pi = \pi_0, \pi_1, \pi_2, \dots \in \Sigma$ is a sequence $\langle q_0, \pi_0, q_1 \rangle, \langle q_1, \pi_1, q_2 \rangle, \langle q_2, \pi_2, q_3 \rangle, \dots$ of transitions in δ such that $q_0 \in Q_0$. The automaton accepts π if it has a run over π that traverses some transition from each set in F infinitely often.

The next theorem relates the expressive power of LTL to that of TGBAs.

Theorem 2 [14,26] *Given an LTL formula f , we can construct a TGBA $A_f = \langle Q, \Sigma, \delta, Q_0, F \rangle$ such that $|Q|$ is in $2^{O(|f|)}$, $\Sigma = 2^{Prop}$, and $\mathcal{L}_\omega(A_f)$ is exactly $\text{models}(f)$.*

Expressing acceptance conditions in terms of transitions rather than states enables a significant reduction in the size of the automata corresponding to LTL formulas [14,26].

Our new encoding of symbolic automata, based on TGBAs, assumes that the input formula f is in NNF. (This is due to the way that the satisfaction of \mathcal{U} -formulas is handled by means of promise variables; see below.) As in CGH, we first define the closure of an LTL formula f . In the case of TGBAs, however, we simply define the closure to be the set of all subformulas of f (including f itself). Note that, unlike in the CGH encoding, \mathcal{U} - and \mathcal{F} -formulas do not require the introduction of new \mathcal{X} -formulas.

The set of elementary formulas now contains: f ; all \mathcal{U} -, \mathcal{R} -, \mathcal{F} -, \mathcal{G} -, and \mathcal{GF} -subformulas in the closure of f , as well as all subformulas g where $\mathcal{X}g$ is in the closure of f . Note that we treat the common \mathcal{GF} combination as a single operator.

Again, we declare a Boolean SMV variable EL_g for every elementary formula g as well as Boolean variables for each atomic proposition in f . In addition, we declare a Boolean SMV promise variable P_g for every \mathcal{U} -, \mathcal{F} -, and \mathcal{GF} -subformula in the closure. These formulas are used to define fairness conditions. Intuitively, P_g holds when g is a promise for the future that is not yet fulfilled. If P_g does not hold, then the promise must be fulfilled immediately. To ensure satisfaction of eventualities we require that each promise variable P_g is false infinitely often. The TGBA encoding creates fewer EL variables than the CGH encoding, but it does add promise variables.

Again, we define an auxiliary variable S_h for every formula h in the closure of f . The characteristic function for S_h is defined as in the CGH encoding, with the following changes:

$$\begin{aligned} S_h &= S_{g_1} \& S_{g_2} \text{ if } h = g_1 \wedge g_2 \\ S_h &= \text{next}(EL_g) \text{ if } h = \mathcal{X}g \\ S_h &= S_{g_2} | (S_{g_1} \& P_{g_1} \mathcal{U} g_2 \& (\text{next}(EL_{g_1} \mathcal{U} g_2))) \text{ if } h = g_1 \mathcal{U} g_2 \\ S_h &= S_{g_2} \& (S_{g_1} | (\text{next}(EL_{g_1} \mathcal{R} g_2))) \text{ if } h = g_1 \mathcal{R} g_2 \\ S_h &= S_g \& (\text{next}(EL_{\mathcal{G}g})) \text{ if } h = \mathcal{G}g \\ S_h &= S_g | (P_{\mathcal{F}g} \& \text{next}(EL_{\mathcal{F}g})) \text{ if } h = \mathcal{F}g \\ S_h &= (\text{next}(EL_{\mathcal{GF}g})) \& (S_g | P_{\mathcal{GF}g}) \text{ if } h = \mathcal{GF}g \end{aligned}$$

Since we reason directly over the temporal subformulas of f (and not over $\mathcal{X}g$ for temporal subformula g as in CGH), the transition relation associates elementary formulas with matching elements of our characteristic function. Finally, we generate our symbolic TGBA; here is our SMV model M_f :

```
MODULE main
VAR /*declare a boolean variable for each atomic proposition in f*/
  a : boolean;
  ...
VAR /*declare a new variable for each elementary formula*/
  EL_f : boolean; /*f is the input LTL formula*/
```

```

EL_g1 : boolean; /*g is an X-, F-, U-, or GF-formula*/
...
DEFINE /*characteristic function definition*/
  S_g = ...
...
TRANS /*for each EL-var, generate a line here*/
  ( EL_g1 = S_g1 ) & /*a line for every EL variable*/
...
FAIRNESS (!P_g1) /*fairness constraint for each promise variable*/
...
FAIRNESS TRUE /*only needed if there are no promise variables*/
SPEC !(EL_f & EG TRUE)

```

Symbolic TGBAs can only be created for NNF formulas because the model checker tries to guess a sequence of values for each of the promise variables to satisfy the subformulas, which does not work for negative \mathcal{U} -formulas. (This is also the case for explicit state model checking; SPOT also requires NNF for TGBA encoding [14].) Consider the formula $f = \neg(a \mathcal{U} b)$ and the trace $a=1, b=0, a=1, b=1, \dots$. Clearly, $(a \mathcal{U} b)$ holds in the trace, so f fails in the trace. If, however, we chose $P_a\mathcal{U}b$ to be false at time 0, then $EL_a\mathcal{U}b$ is false at time 0, which means that f holds at time 0. The correctness of our construction is summarized by the following theorem.

Theorem 3 *Let M_f be the SMV program made by the TGBA encoding for LTL formula f . Then M_f does not satisfy the specification $!(EL_f \ \& \ EG \ true)$ iff f is satisfiable.*

The proof of this theorem appears in Appendix D.

4 A Set of 30 Symbolic Automata Encodings

Our novel encodings are combinations of four components: (1) Normal Form: BNF or NNF, described above, (2) Automaton Form: GBA or TGBA, described above, (3) Transition Form: fussy or sloppy, described below, and (4) Variable Order: default, naive, LEXP, LEXM, MCS-MIN, MCS-MAX, described below. In total, we have 30 novel encodings, since BNF can only be used with fussy-encoded GBAs, as explained below. CGH corresponds to BNF/fussy/GBA; we encode this combination with all six variable orders.

Automaton Form As discussed earlier, CGH is based on GBA, in combination with BNF. We can combine, however, GBA also with NNF. For this, we need to expand the characteristic function for symbolic GBAs in order to form them from NNF formulas:

$$\begin{aligned}
 S_h &= S_{g_1} \& S_{g_2} \text{ if } h = g_1 \wedge g_2 & S_h &= S_g \& S_{\mathcal{X}_{(\mathcal{G}g)}} \text{ if } h = \mathcal{G}g \\
 S_h &= S_{g_2} \& (S_{g_1} | S_{\mathcal{X}_{(g_1 \ \mathcal{R} \ g_2)}}) \text{ if } h = g_1 \ \mathcal{R} \ g_2 & S_h &= S_g | S_{\mathcal{X}_{(\mathcal{F}g)}} \text{ if } h = \mathcal{F}g
 \end{aligned}$$

Since our focus here is on symbolic encoding, PANDA, unlike CadenceSMV, does not apply formula rewriting and related optimizations; rather, PANDA’s symbolic automata are created directly from the given normal form of the formula. Formula rewriting may lead to further improvement in PANDA’s performance.

Sloppy Encoding: A Novel Transition Form CGH employs iff-transitions, of the form $TRANS \ (EL_g = (S_g))$. We refer to this as *fussy* encoding. For formulas in NNF, we can use only-if transitions of the form $TRANS \ (EL_g \rightarrow (S_g))$, which we refer to as *sloppy* encoding. A similar idea was shown to be useful in the context of modal satisfiability

solving [33]. Sloppy encoding increases the level of non-determinism, yielding a looser, less constrained encoding of symbolic automata, which in many cases results in smaller BDDs. A side-by-side example of the differences between GBA and TGBA encodings (demonstrating the sloppy transition form) for formula $f = ((\mathcal{X}a) \& (b \mathcal{U} (!a)))$ is given in Figures 1-2.

```

MODULE main
/*formula: ((X (a )) & ((b )U (!(a ))))*/
VAR /*a Boolean var for each prop in f*/
  a : boolean;
  b : boolean;
VAR /*a var EL_X_g for each formula (X g) in
  el_list w/primary op X, U, R, G, or F*/
  EL_X_a : boolean;
  EL_X_b_U_NOT_a : boolean;
DEFINE
/*each S_h in the characteristic function*/
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 /*a line for each (X g) in el_list*/
  ( 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)

```

Fig. 1. NNF/sloppy/GBA encoding for CadenceSMV

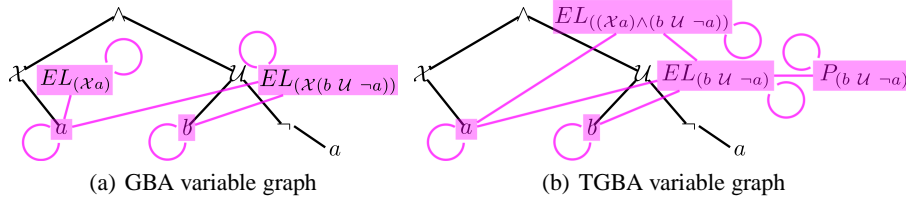
```

MODULE main
/*formula: ((X (a )) & ((b )U (!(a ))))*/
VAR /*a Boolean var for each prop in f*/
  a : boolean;
  b : boolean;
VAR /*a var 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
/*each S_h in the characteristic function*/
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 /*a line for each EL_var in el_list*/
  ( 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)

```

Fig. 2. NNF/sloppy/TGBA encoding for CadenceSMV

A New Way of Choosing BDD Variable Orders Symbolic model checkers search for a fair trace in the model-automaton product using a BDD-based fixpoint algorithm, a process whose efficacy is highly sensitive to variable order [7]. Finding an optimal BDD variable order is NP-hard, and good heuristics for variable ordering are crucial.

Fig. 3. Graphs in (a) and (b) were both formed from the parse tree for $f = ((\mathcal{X}a) \wedge (b \mathcal{U} \neg a))$.

Recall that we define state variables in the symbolic model for only certain subformulas: $p \in AP$, EL_g , and P_g for some subformulas g . We form the variable graph by identifying nodes in the input-formula parse tree that correspond to the primary operators of those subformulas. Since we declare different variables for the GBA and TGBA encodings, the variable graph for a formula f may vary depending on the automaton form we choose. Figure 3 displays the GBA and TGBA variable graphs for an example formula, overlaid on the parse tree for this formula. We connect each variable-labeled

vertex to its closest variable-labeled vertex descendant(s), skipping over vertices in the parse tree that do not correspond to state variables in our automaton construction. We create one node per subformula variable, irrespective of the number of occurrences of the subformula; for example, we create only one node for the proposition a in Figure 3.

We implement five variable ordering schemes, all of which take the variable graph as input. We compare these to the *default* heuristic of CadenceSMV. The *naïve* variable order is formed directly from a pre-order, depth-first traversal of the variable graph. We derive four additional variable-ordering heuristics by repurposing node-ordering algorithms designed for graph triangulation [30].³ We use two variants of a lexicographic breadth-first search algorithm: variants *perfect* (LEXP) and *minimal* (LEXM). LEXP labels each vertex in the variable graph with its already-ordered neighbors; the unordered vertex with the lexicographic largest label is selected next in the variable order. LEXM operates similarly, but labels unordered vertices with both their neighbors and also all vertices that can be reached by a path of unordered vertices with smaller labels. The maximum-cardinality search (MCS) variable ordering scheme differs in the vertex selection criterion, selecting the vertex in the variable graph adjacent to the highest number of already ordered vertices next. We seed MCS with an initial vertex, chosen either to have the *maximum* (MCS-MAX) or *minimum* (MCS-MIN) degree.

5 Experimental Methodology

Test Methods Each test was performed in two steps. First, we applied our symbolic encodings to the input formula. Second, each symbolic automaton and variable order file pair was checked by CadenceSMV. Since encoding time is minimal and heavily dominated by model-analysis time (the time to check the model for nonemptiness to determine LTL satisfiability) we focus exclusively on the latter here.

Platform We ran all tests on Shared University Grid at Rice (SUG@R), an Intel Xeon compute cluster.⁴ 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, 2.6.18 kernel. Each test was run with exclusive access to one node. Times were measured using the Unix `time` command.

Input Formulas We employed a widely-used [9, 16, 27, 40] collection of benchmark formulas, established by [40]. All encodings were tested using three types of scalable formulas: random, counter, and pattern. Definitions of these formulas are repeated for convenience in Appendix B. Our test set includes 4 counter and 9 pattern formula variations, each of which scales to a large number of variables, and 60,000 random formulas.

Correctness In addition to proving the correctness of our algorithm, the correctness of our implementation was established by comparing for every formula in our large benchmark suite, the results (either SAT or UNSAT) returned by all encodings studied here, as well as the results returned by CadenceSMV for checking the same formula as an LTL specification for the universal model. We never encountered an inconsistency.

³ Graph triangulation implementation coded by the Kavradi Lab at Rice University.

⁴ <http://rcsg.rice.edu/sugar/>

6 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 nonemptiness and the size of the state space we can check. No single encoding, however, consistently dominates for all types of formulas. Instead, we find that different encodings are better suited to different formulas. Therefore, we recommend using a multi-encoding approach, a variant of the multi-engine approach [37], of running all encodings in parallel and terminating when the first job completes. We call our tool PANDA for “Portfolio Approach to Navigate the Design of Automata.”

Seven configurations are not competitive While we can not predict the best encodings, we can reliably predict the worst. The following encodings were never optimal for any formulas in our test set. Thus, out of our 30 possible encodings, we rule out these seven:

- BNF/fussy/GBA/LEXM (essentially CGH with LEXM)
- NNF/fussy/GBA/LEXM
- NNF/fussy/TGBA/MCS-MAX
- NNF/sloppy/GBA/LEXM
- NNF/sloppy/TGBA/MCS-MAX
- NNF/sloppy/GBA/LEXM
- NNF/sloppy/TGBA/MCS-MIN

NNF is the best normal form, most (but not all) of the time. NNF encodings were always better for all counter and pattern formulas; see, for example, Figure 4. Figure 5 demonstrates the use of both normal forms in the optimal encodings chosen by PANDA for random formulas. BNF encodings were occasionally significantly better than NNF; the solid point in Figure 5 corresponds to a formula for which the best BNF encoding was more than four times faster than the best NNF encoding. NNF was best much more often than BNF, likely because using NNF has the added benefit that it allows us to employ our sloppy encoding and TGBAs, which often carry their own performance advantages.

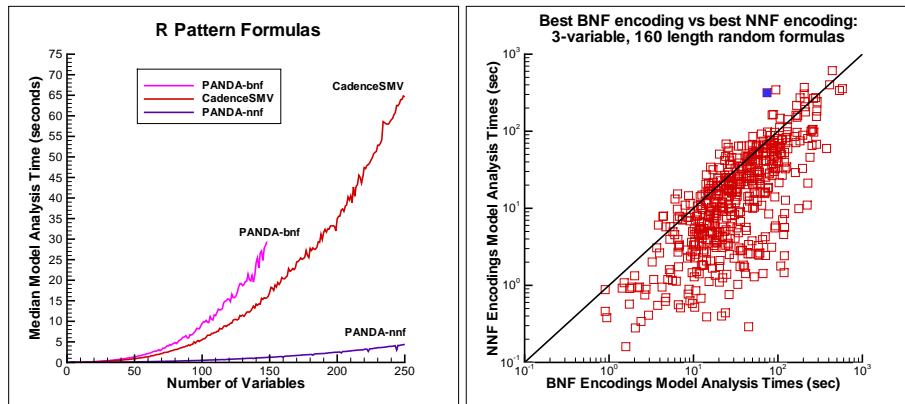


Fig. 4. Median model analysis time for **Fig. 5.** Best encodings of 500 3-variable, 160 length random formulas. Points fall below the NNF/sloppy/GBA/naïve, CadenceSMV, and diagonal when NNF is better. the best BNF encoding.

No automaton form is best. Our TGBA encodings dominated for R_2 , S , and U pattern formulas and both types of 3-variable counter formulas. For instance, the log-scale plot in Figure 6 shows that PANDA’s median model analysis time for R_2 pattern formulas grows subexponentially as a function of the number of variables, while CadenceSMV’s median model analysis time for the same formulas grows exponentially. (The best of PANDA’s GBA encodings is also graphed for comparison.) GBA encodings are better for other pattern formulas, both types of 2-variable counter formulas, and the majority of random formulas; Figure 7 demonstrates this trend for 180 length random formulas.

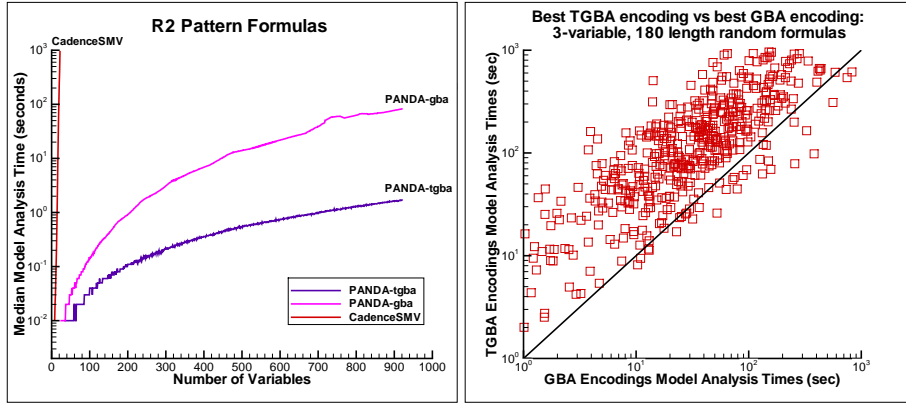


Fig. 6. $R_2(n) = (..(p_1 \mathcal{R} p_2) \mathcal{R} \dots) \mathcal{R} p_n$. **Fig. 7.** Best encodings of 500 3-variable, 180 PANDA’s NNF/sloppy/TGBA/LEXP encoding length random formulas.

scales better than the best GBA encoding, NNF/sloppy/GBA/naïve, and exponentially better than CadenceSMV.

No transition form is best Sloppy is the best transition form for all pattern formulas. For instance, the log-scale plot of Figure 8 illustrates that PANDA’s median model analysis time for U pattern formulas grows subexponentially as a function of the number of variables, while CadenceSMV’s median model analysis time for the same formulas grows exponentially. Fussy encoding is better for all counter formulas. The best encodings of random formulas were split between fussy and sloppy. Figure 9 demonstrates this trend for 140 length random formulas.

No variable order is best, but LEXM is worst. The best encodings for our benchmark formula set were split between five variable orders. The naïve and default orders proved optimal for more random formulas than the other orders. Figure 10 demonstrates that neither the naïve order nor the default order is better than the other for random formulas. The naïve order was optimal for E , Q , R , U_2 , and S patterns. MCS-MAX is optimal for 2- and 3-variable linear counters. The LEXP variable order dominated for C_1 , C_2 , U , and R_2 pattern formulas, as well as for 2- and 3-variable counter formulas, yet it was rarely best for random formulas. Figure 11 demonstrates the marked difference in scalability

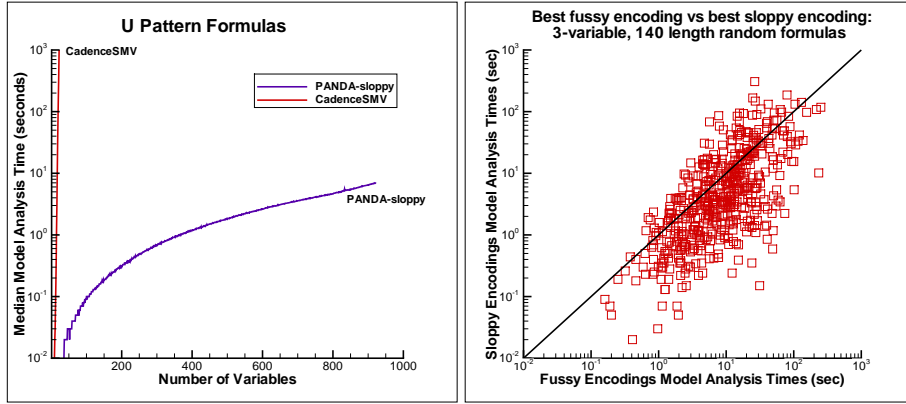


Fig. 8. $U(n) = (\dots(p_1 \mathcal{U} p_2) \mathcal{U} \dots) \mathcal{U} p_n$. **Fig. 9.** Best encodings of 500 3-variable, 140 PANDA’s NNF/sloppy/TGBA/LEXP scalables length random formulas. Points fall below the diagonal when sloppy encoding is best. CadenceSMV.

provided by using the LEXP order over running CadenceSMV on 3-variable counter formulas. We can analyze much larger models with PANDA using LEXP than with the native CadenceSMV encoding before memory-out. We never found the LEXM order to be the single best encoding for any formula.

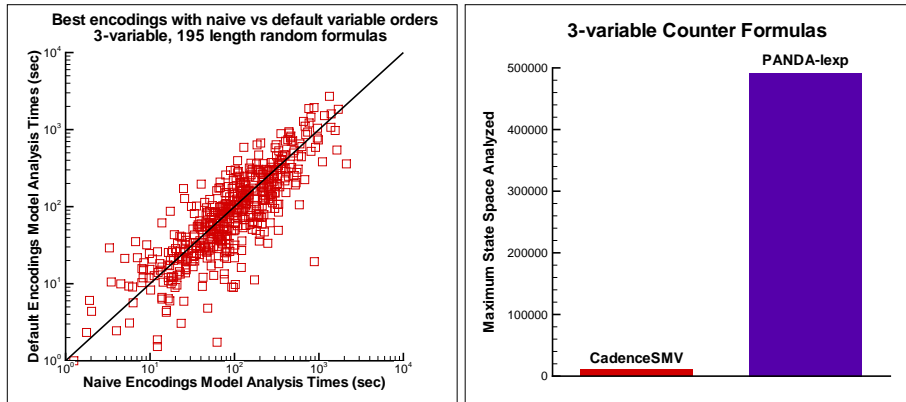


Fig. 10. Best encodings of 500 3-variable, 195 length random formulas. Points fall above the diagonal when naïve variable order is best. **Fig. 11.** Maximum states analyzed before space-out. CadenceSMV quits at 10240 states. PANDA’s NNF/fussy/TGBA/LEXP scales to 491520 states.

A formula class typically has a best encoding, but predictions are difficult While each of our pattern and counter formulas had a best (or a pair of best) encodings, which remained consistent as we scaled the formulas, we found that we could not reliably predict the best encoding using any statistics gathered from parsing, such as operator

counts or ratios. For example, we found that the best encoding for a pattern formula was not necessarily the best for a randomly-generated formula comprised of the same temporal operators. We surmise that the best encoding is tied to the structure of the formula on a deeper level; developing an accurate heuristic is left to future work.

There is no single best encoding; a multi-encoding approach is clearly superior We implement a novel multi-encoding approach: our new PANDA tool creates several encodings of a formula and uses a symbolic model checker to check them for satisfiability in parallel, terminating when the first check completes. Our experimental data supports this multi-encoding approach. Figures 4, 6, and 8 highlight the significant decrease in CadenceSMV model analysis time for R , R_2 , and U pattern formulas, while Figure 11 demonstrates increased scalability in terms of state space using counter formulas. Altogether, we demonstrate that a multi-encoding approach is dramatically more scalable than the current state-of-the-art. The increase in scalability is dependant on the specific formula, though for some formulas PANDA’s model analysis time is exponentially better than CadenceSMV’s model analysis time for the same class of formulas.

7 Discussion

This paper brought attention to the issue of scalable construction of symbolic automata for LTL formulas in the context of LTL satisfiability checking. We defined novel encodings and novel BDD variable orders for accomplishing this task. We explored the impact of these encodings, comprised of combinations of normal forms, automaton forms, transition forms, and combined with variable orders. We showed that each can have a significant impact on performance. At the same time, we showed that no single encoding outperforms all others and showed that a multi-encoding approach yields the best result, consistently outperforming the native translation of CadenceSMV.

We do not claim to have exhaustively covered the space of possible encodings of symbolic automata. Several papers on the automata-theoretic approach to LTL describe approaches that could be turned into alternative encodings of symbolic automata, cf. [5, 21, 23, 42]. The advantage of the multi-encoding approach we introduced here is its *extensibility*; adding additional encodings is straightforward. The multi-encoding approach can also be combined with different back ends. In this paper we used CadenceSMV as a BDD-based back end; using another symbolic back end (cf. [16]) or a SAT-based back end (cf. [4]) would be an alternative approach, as both BDD-based and SAT-based back ends require symbolic automata. Since LTL serves as the basis for industrial languages such as PSL and SVA, the encoding techniques studied here may also serve as the basis for novel encodings of such languages, cf. [10, 11].

In this paper we examined our novel symbolic encodings of LTL in the context of satisfiability checking. An important difference between satisfiability checking and model checking is that in the former we expect to have to handle much larger formulas, since we need to consider the conjunction of properties. Also, in model checking the size of the symbolic automata can be dwarfed by the size of the model under verification. Thus, the issue of symbolic encoding of automata in the context of model checking deserves a separate investigation.

References

1. N. Amla, X. Du, A. Kuehlmann, R.P. Kurshan, and K.L. McMillan. An analysis of SAT-based model checking techniques in an industrial environment. In *CHARME*, LNCS 3725, pages 254–268. Springer, 2005.
2. P. Arcaini, A. Gargantini, and E. Riccobene. Automatic review of abstract state machines by meta property verification. In *NFM, NASA/CP-2010-216215*, pages 4–13, April 2010.
3. I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. Efficient detection of vacuity in ACTL formulas. *FMSD 18*, (2):141–162, 2001.
4. A. Biere, C. Artho, and V. Schuppan. Liveness checking as safety checking. In *FMICS 66:2*, ENTCS, 2002.
5. R. Bloem, A. Cimatti, I. Pill, and M. Roveri. Symbolic implementation of alternating automata. *IJFCS 18*, (4):727–743, 2007.
6. R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. Automatic hardware synthesis from specifications: A case study. In *DATE*, pages 1188–1193, 2007.
7. R.E. Bryant. Graph-based algorithms for Boolean-function manipulation. *IEEE TC C-35*, (8):677–691, 1986.
8. 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.
9. J. Cichon, A. Czubak, and A. Jasinski. Minimal Büchi automata for certain classes of LTL formulas. *DepCoS 0*, pages 17–24, 2009.
10. A. Cimatti, M. Roveri, S. Semprini, and S. Tonetta. From PSL to NBA: A modular symbolic encoding. In *FMCAD*, 2006.
11. A. Cimatti, M. Roveri, and S. Tonetta. Syntactic optimizations for PSL verification. In *TACAS*, pages 505–518, 2007.
12. E. M. Clarke, O. Grumberg, and K. Hamaguchi. Another look at LTL model checking. *Formal Methods in System Design 10*, (1):47–71, 1997.
13. C. Courcoubetis, M.Y. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. In *CAV*, LNCS 531, p233–242. Springer, 1990.
14. J-M. Couvreur. On-the-fly verification of Linear Temporal Logic. In *FM*, p253-271, 1999.
15. N. Daniele, F. Guinchiglia, and M.Y. Vardi. Improved automata generation for Linear Temporal Logic. In *CAV*, LNCS 1633, pages 249–260. Springer, 1999.
16. M. De Wulf, L. Doyen, N. Maquet, and J. Raskin. Antichains: Alternative algorithms for LTL satisfiability and model-checking. In *TACAS*, pages 63–77, 2008.
17. A. Duret-Lutz and D. Poitrenaud. SPOT: An extensible model checking library using Transition-Based Generalized Büchi Automata. In *MASCOTS*, pages 76–83, 2004.
18. E.A. Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science*, volume B, chapter 16, pages 997–1072. Elsevier, MIT Press, 1990.
19. A. Ferrara, G. Pan, and M. Y. Vardi. Treewidth in verification: Local vs. global. In *LPAR*, LNCS 3835, pages 489–503. Springer, 2005.
20. E. Filiot, N. Jin, and J-F. Raskin. An antichain algorithm for ltl realizability. In *CAV*, pages 263–277, 2009.
21. M. Fisher. A normal form for temporal logics and its applications in theorem-proving and execution. *J. Log. Comput. 7*, (4):429–456, 1997.
22. D. Fisman, O. Kupferman, S. Sheinvald-Faragy, and M.Y. Vardi. A framework for inherent vacuity. In *Haifa Verification Conference*, LNCS 5394, pages 7–22. Springer, 2008.
23. P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In *CAV*, LNCS 2102, pages 53–65. Springer, 2001.
24. J. Geldenhuys and H. Hansen. Larger automata and less work for LTL model checking. In *SPIN*, LNCS 3925, pages 53–70. Springer, 2006.

25. R. Gerth, D. Peled, M.Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of Linear Temporal Logic. In *PSTV*, pages 3–18. Chapman & Hall, Aug 1995.
26. D. Giannakopoulou and F. Lerda. From states to transitions: Improving translation of LTL formulae to Büchi automata. In *FORTE*, Nov 2002.
27. V. Goranko, A. Kyrilov, and D. Shkatov. Tableau tool for testing satisfiability in LTL: Implementation and experimental analysis. *ENTCS 262*, pages 113–125, 2010.
28. A. Habibi and S. Tahar. Design for verification of SystemC transaction level models. In *Design, Automation and Test in Europe*, pages 560–565. IEEE, 2005.
29. Y. Kesten, Z. Manna, H. McGuire, and A. Pnueli. A decision algorithm for full propositional temporal logic. In *CAV, LNCS 697*, pages 97–109. Springer, 1993.
30. A. M. C. A. Koster, H. L. Bodlaender, and S. P. M. van Hoesel. Treewidth: Computational experiments. ZIB-Report 01–38, ZIB, 2001.
31. O. Kupferman and M.Y. Vardi. Vacuity detection in temporal model checking. *STTT 4*, (2):224–233, Feb 2003.
32. S. Merz and A. Sezgin. Emptiness of Linear Weak Alternating Automata. Technical report, LORIA, December 2003.
33. G. Pan, U. Sattler, and M.Y. Vardi. BDD-based decision procedures for K. In *CADE, LNCS 2392*, pages 16–30. Springer, 2002.
34. I. Pill, S. Semprini, R. Cavada, M. Roveri, R. Bloem, and A. Cimatti. Formal analysis of hardware requirements. In *DAC*, pages 821–826. ACM, 2006.
35. A. Pnueli. The temporal logic of programs. In *IEEE FOCS*, pages 46–57, 1977.
36. A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *POPL*, p179–190, 1989.
37. L. Pulina and A. Tacchella. A self-adaptive multi-engine solver for quantified Boolean formulas. *Constraints 14*, (1):80–116, 2009.
38. A. Cimatti, R. Cavada, C. Jochim, G. Keighren, E. Olivetti, M. Pistore, M. Roveri, and A. Tchaltsev. NuSMV 2.4 user manual. Technical report, CMU and ITC-irst, 2005.
39. M. Roveri. Novel techniques for property assurance. Technical report, PROSYD deliverable 1.2/2, 2004.
40. K.Y. Rozier and M.Y. Vardi. LTL satisfiability checking. In *Model Checking Software (SPIN)*, LNCS 4595, pages 149–167. Springer, 2007.
41. S. Ruah, A. Fedeli, C. Eisner, and M. Moulin. Property-driven specification of VLSI design. Technical report, PROSYD deliverable 1.1/1, 2005.
42. K. Schneider. Improving automata generation for Linear Temporal Logic by considering the automaton hierarchy. In *LPAR*, pages 39–54. Springer, 2001.
43. R. Sebastiani and S. Tonetta. “More deterministic” vs. “smaller” Büchi automata for efficient LTL model checking. In *CHARME*, pages 126–140. Springer, 2003.
44. A.P. Sistla and E.M. Clarke. The complexity of Propositional Linear Temporal Logic. *J. ACM 32*, pages 733–749, 1985.
45. F. Somenzi and R. Bloem. Efficient Büchi automata from LTL formulae. In *CAV, LNCS 1855*, pages 248–263. Springer, 2000.
46. X. Thirioux. Simple and efficient translation from LTL formulas to Büchi automata. *ENTCS 66*, (2):145–159, 2002.
47. M.Y. Vardi. Automata-theoretic model checking revisited. In *VMCAI, LNCS 4349*, pages 137–150. Springer, 2007.
48. M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *LICS*, pages 332–344, Cambridge, Jun 1986.
49. M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation 115*, (1):1–37, Nov 1994.

Appendix A: LTL Semantics

Linear Temporal Logic (LTL) formulas are composed of a finite set $Prop$ of atomic propositions, the Boolean connectives \neg , \wedge , \vee , and \rightarrow , and the temporal connectives \mathcal{U} (until), \mathcal{R} (release), \mathcal{X} (next time), \mathcal{G} (globally), and \mathcal{F} (in the future). We define LTL formulas inductively:

Definition 5 For every $p \in Prop$, p is a formula. If g , g_1 , and g_2 are formulas, then so are:

$$\neg g \quad g_1 \wedge g_2 \quad g_1 \vee g_2 \quad g_1 \rightarrow g_2 \quad \mathcal{X}g \quad g_1 \mathcal{U} g_2 \quad g_1 \mathcal{R} g_2 \quad \mathcal{G}g \quad \mathcal{F}g$$

LTL formulas describe the behavior of the variables in $Prop$ over a linear series of time steps starting at the present time (time step zero) and extending infinitely into the future. We satisfy such formulas over *computations*, which are functions that assign truth values to the elements of $Prop$ at each time step [18].

Definition 6 We interpret LTL formulas over computations of the form $\pi : \omega \rightarrow 2^{Prop}$, where ω is used in the standard way to denote the set of non-negative integers. We define $\pi, i \models f$ (computation π at time instant $i \in \omega$ satisfies LTL formula f) as follows:

- $\pi, i \models p$ for $p \in Prop$ if $p \in \pi(i)$.
- $\pi, i \models g_1 \wedge g_2$ if $\pi, i \models g_1$ and $\pi, i \models g_2$.
- $\pi, i \models \neg g$ if $\pi, i \not\models g$.
- $\pi, i \models \mathcal{X}g$ if $\pi, i+1 \models g$.
- $\pi, i \models g_1 \mathcal{U} g_2$ if $\exists j \geq i$, such that $\pi, j \models g_2$ and $\forall k, i \leq k < j$, we have $\pi, k \models g_1$.
- $\pi, i \models g_1 \mathcal{R} g_2$ if $\forall j \geq i$, if $\pi, j \not\models g_2$, then $\exists k, i \leq k < j$, such that $\pi, k \models g_1$.
- $\pi, i \models \mathcal{F}g$ if $\exists j \geq i$, such that $\pi, j \models g$.
- $\pi, i \models \mathcal{G}g$ if $\forall j \geq i, \pi, j \models g$.

We take $models(f)$ to be the set of computations that satisfy f at time 0, i.e., $\{\pi : \pi, 0 \models f\}$.

Appendix B: Universal Model and Benchmark Formulas From [40]

In [40] we showed how to relate LTL satisfiability checking to LTL model checking. Suppose we have a *universal model* M that generates all traces over its set of atomic propositions, $Prop$; that is, we have that $\mathcal{L}_\omega(M) = (2^{Prop})^\omega$. Then M does *not* satisfy $\neg f$ if and only if f is satisfiable. We can use a model checker to check whether f is satisfiable by checking the formula $\neg f$ against the universal model; f is satisfiable precisely when the model checker finds a counterexample.

For example, if $f = (\mathcal{X}(a \mathcal{U} b))$, we provide the following input to CadenceSMV⁵:

```

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

```

⁵ The model for NuSMV is nearly identical. We need to add FAIRNESS to guarantee that the model checker returns an infinite trace, which may not happen if there are no FAIRNESS statements in the model.

The model checker negates the specification, $\neg f$, compiles the negated specification, f , into the symbolic automaton A_f , and conjoins A_f with the universal model. If A_f is not empty, the model checker finds a fair trace, which satisfies the formula f .

Input Formulas Utilizing the benchmarks established by [40], we evaluated our set of encodings using three types of scalable formulas: random formulas, counter formulas, and pattern formulas. We experimented with both CadenceSMV and NuSMV as backends for our encodings. Unlike in [40], correctness of the SAT/UNSAT answers was not an issue. The only cases where we found disagreement were attributable to the subtle semantic differences between CadenceSMV and NuSMV covered in Appendix C.

Random Formulas In order to cover as much of the problem space as possible, we generated random formulas as in [15]. We created 60 sets of 500 formulas each varying the number of variables from 1 to 3, and the length of the formula from 5 to 200. We refer to a formula with a total of 150 operators and atomic propositions as a “150 length random formula.” We chose from the operator set $\{\neg, \vee, \wedge, \mathcal{X}, \mathcal{U}, \mathcal{R}, \mathcal{G}, \mathcal{F}, \mathcal{GF}\}$. (We included the combination \mathcal{GF} as a single operator because that combination occurs so frequently in industrial specifications.) To create formulas with both a nontrivial temporal structure and a nontrivial Boolean structure, the probability of choosing a temporal operator was 0.5. Other choices were decided uniformly. All formulas were generated prior to testing for repeatability.

Counter Formulas To measure performance on scalable, temporally complex formulas with large state spaces, we tested our encodings 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 \mathcal{X} ’s.

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 \mathcal{X} ’s to provide more succinct formulas or express the formulas using a conjunction of un-nested \mathcal{X} -sub-formulas. These formulas were originally defined in [40].

Pattern Formulas We evaluated the efficacy of each encoding on specific temporal operators using the eight patterns of scalable formulas defined by [24] plus one we defined and call R_2 .

$$\begin{aligned}
S(n) &= \bigwedge_{i=1}^n \mathcal{G} p_i, & E(n) &= \bigwedge_{i=1}^n \mathcal{F} p_i, & Q(n) &= \bigwedge (\mathcal{F} p_i \vee \mathcal{G} p_{i+1}), \\
U(n) &= (\dots (p_1 \mathcal{U} p_2) \mathcal{U} \dots) \mathcal{U} p_n, & U_2(n) &= p_1 \mathcal{U} (p_2 \mathcal{U} (\dots p_{n-1} \mathcal{U} p_n) \dots), \\
C_1(n) &= \bigvee_{i=1}^n \mathcal{GF} p_i, & C_2(n) &= \bigwedge_{i=1}^n \mathcal{GF} p_i.
\end{aligned}$$

$$R(n) = \bigwedge_{i=1}^n (\mathcal{GF} p_i \vee \mathcal{FG} p_{i+1}), \quad R_2(n) = (\dots (p_1 \mathcal{R} p_2) \mathcal{R} \dots) \mathcal{R} p_n.$$

Since CadenceSMV does not accept \mathcal{R} operators, we eliminated them from our LTLSEPCs using the conversion $(g_1 \mathcal{R} g_2) \equiv \neg(\neg g_1 \mathcal{U} \neg g_2)$.

Appendix C: CadenceSMV and NuSMV Semantic Subtleties

We encountered one large, and several more subtle but still impactful differences between the implementations of CadenceSMV and NuSMV when testing our novel encodings. Most significantly, TGBA-encoded symbolic automata cannot be checked using NuSMV because variable definitions in terms of DEFINE-statements may only assign simple expressions composed of state variables. [38]. Therefore, NuSMV cannot parse the next() operators in our DEFINE section. NuSMV does not offer alternative ways to define variables (e.g. ASSIGN-statements) which allow our TGBA construction.⁶

Though NuSMV cannot parse our TGBA-encoded automata, for all of the encodings we could check with both SMV variants, we saw similar, significant improvements when running PANDA with that model checker as a back-end, versus running the model checker alone. As expected, we found our automata with NuSMV as a back end produced different timing results than the same automata with CadenceSMV as a back end, just as running each of the tools alone produced different timing results, though neither SMV was always faster. Both tools agreed on the satisfiability of a given LTL formula 100% of the time. Also, the results (either SAT or UNSAT) returned by CadenceSMV and NuSMV always agreed with the results of running all 30 encodings of the same formula. However, in order to compare our novel encodings across both SMV back-ends, we had to account for several non-intuitive subtle semantic differences between CadenceSMV and NuSMV.

While both basically use the CGH encoding, the precise semantics of CTL model checking in CadenceSMV and NuSMV are not explicitly documented and the subtle, often unexpected, differences between the implementations of the two tools complicates the problem of creating alternative encodings. The following rules are necessary to work around the several subtleties that arise when checking nonemptiness of fair symbolic Büchi automata:

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, INIT (a&(!a)) specifies that there is no initial state. Similar semantic subtleties have impacted related work with SMV, such as model checking temporal logic meta-property specifications for abstract state machines [2], where the problem that this unexpected quantification over initial states means that $M \models EF(\varphi) \not\equiv M \not\models AG(\neg\varphi)$.

⁶ Thanks to Viktor Schuppan and the members of the NuSMV team for allowing our construction in future versions of NuSMV.

Symbolic automata must always have a FAIR statement, even if it is “FAIR true.” CadenceSMV considers terminal traces to be fair when there are no fairness constraints. A fair trace is defined to be a maximal trace on which all fairness constraints are true infinitely often, which includes terminal traces in models without fairness constraints. The semantics with a fairness constraint is the “infinite traces” semantics where states without infinite traces 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 $(\varphi \wedge \text{EG true})$. 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 SPEC $(\varphi \wedge \text{EG true})$, if the model is not empty, the counterexample returned is a trace of the model.

INIT φ SPEC (EG true) is not equivalent to SPEC $(\varphi \wedge \text{EG true})$. For example, if φ is simply *false* and we check SPEC $(\text{false} \wedge \text{EG true})$, then the check will pass (i.e. there will be no counterexample indicating satisfiability), since no trace satisfies *false*. If, however, we state INIT *false* and check SPEC (EG true) , then the check will fail and a counterexample will be returned, which is clearly not what we intended. Similarly, checking for a finite counterexample using SPEC (S_f) may produce spurious results.

Our symbolic TGBAs can be checked by CadenceSMV but not by NuSMV. 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. 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.

Appendix D: Proof of Theorem 3

Theorem 3 *Let M_f be the SMV program generated by our symbolic TGBA encoding for an LTL formula f . Then M_f does not satisfy the specification $(\text{EL}_f \ \& \ \text{EG true})$ iff f is satisfiable.*

Proof. We prove each direction in turn.

Only if: If f is satisfiable then the specification $(\text{EL}_f \ \& \ \text{EG true})$ does not hold in M_f .

If f is satisfiable, there is a trace $\pi = \pi_0, \pi_1, \dots \in (2^{\text{Prop}})^\omega$ such that $\pi, 0 \models f$ (trace π at time instant 0 satisfies f), where *Prop* is the set of atomic propositions occurring in f . To show that $(\text{EL}_f \ \& \ \text{EG true})$ does not hold in M_f , we need to exhibit an infinite trace π' of M_f such that EL_f holds at point 0 of π' . A trace π' of M_f is a trace $\pi' = \pi'_0, \pi'_1, \dots \in (2^{\text{Var}(f)})^\omega$, where *Var*(f) is the set of variables of M_f , consisting of:

- the atomic propositions $Prop$,
- the variable EL_g for each elementary formula g of f ,
- and a promise variable P_g for each \mathcal{U} , \mathcal{F} , and \mathcal{GF} subformula of f .

Note that $Prop \subseteq Var(f)$. We define π' as a conservative extension of π ; that is, $\pi'_i \cap Prop = \pi_i$, for all $i \geq 0$. We define this extension as follows:

- for each elementary formula g of f , we have that $EL_g \in \pi'_i$ iff $\pi, i \models g$ (trace π at time instant $i \in \omega$ satisfies subformula g),
- for a subformula g of f , of the form $h' \mathcal{U} h$, $\mathcal{F} h$ or $\mathcal{GF} h$, we have that $P_g \in \pi'_i$ iff $\pi, i \models g$, but $\pi, i \not\models h$.

Note that since f is an elementary formula of itself and $\pi, 0 \models f$, we immediately have that $EL_f \in \pi'_0$.

It remains to show that π' is a trace of M_f . To that end we first extend π' . Let $Var'(f)$ be the set of *all* variables in M_f , including auxiliary variables. We define $\pi'' \in (2^{Var'(f)})^\omega$ as a conservative extension of π' ; that is, $\pi''_i \cap Var(f) = \pi'_i$, for all $i \geq 0$. We define this extension as follows: for each subformula g of f , we have that $S_g \in \pi''_i$ iff $\pi, i \models g$.

We now need to show that all the statements of M_f hold in π'' . Each TRANS statement $EL_g = S_g$ holds trivially, as for each elementary formula g of f , we have that $EL_g \in \pi''_i$ iff $\pi, i \models g$ iff $S_g \in \pi''_i$. The DEFINE statements for \wedge , \vee , \mathcal{X} , \mathcal{R} , and \mathcal{G} hold because of the basic properties of the propositional and temporal connectives [18]. For \mathcal{F} -, \mathcal{U} -, and \mathcal{GF} -subformulas, we also have to take into account promise variables. Consider, for example, a subformula h of the form $\mathcal{F} g$, for which the DEFINE statement is $S_h = S_g | (P_{\mathcal{F} g} \& next(EL_{\mathcal{F} g}))$. Suppose $S_h \in \pi''_i$, which means that $\pi, i \models \mathcal{F} g$. We know that $\pi, i \models \mathcal{F} g$ iff either $\pi, i \models g$ or both $\pi, i \not\models g$ and $\pi, i + 1 \models h$. If $\pi, i \models g$, then $S_g \in \pi''_i$. If $\pi, i \not\models g$, then $P_{\mathcal{F} g} \in \pi''_i$ and $EL_h \in \pi''_{i+1}$. Conversely, if $S_g \in \pi''_i$, then $\pi, i \models g$, which entails, $\pi, i \models \mathcal{F} g$, and, consequently, $S_h \in \pi''_i$. Also, if $EL_h \in \pi''_{i+1}$, then $\pi, i + 1 \models h$, and, consequently, $\pi, i \models h$ and $S_h \in \pi''_i$. The arguments for \mathcal{U} - and \mathcal{GF} -subformulas are similar.

Finally, we need to show that the FAIRNESS statement holds. That is, for each promise variable P_h there are infinitely many i 's such that $P_h \notin \pi'_i$. Assume, for example, that h is the subformula $\mathcal{F} g$. Suppose to the contrary that the FAIRNESS statement fails. That is, there is some $i_0 \geq 0$ such that $P_h \in \pi'_i$ for all $i \geq i_0$. But this means that $\pi, i_0 \models \mathcal{F} g$, and $\pi, i \not\models g$ for all $i \geq i_0$, which is impossible. The arguments for \mathcal{U} - and \mathcal{GF} -subformulas are similar.

If: If M_f does not satisfy the specification $!(EL_f \ \& \ EG \ \text{true})$ then f is satisfiable.

Suppose the specification $!(EL_f \ \& \ EG \ \text{true})$ does not hold in M_f . Then, by definition, there is an infinite trace $\pi' = \pi'_0, \pi'_1, \dots \in (2^{Var(f)})^\omega$ of M_f such that EL_f holds at point 0 of π' , where $Var(f)$, as defined above, corresponds to the set of atomic propositions in f , elementary formula variables, and promise variables. To show that f is satisfiable, we show that the infinite trace $\pi = \pi_0, \pi_1, \dots \in (2^{Prop})^\omega$, defined by $\pi_i = \pi'_i \cap Prop$ for $i \geq 0$, satisfies f at time 0; that is, $\pi, 0 \models f$.

Again, we first define $\pi'' \in (2^{Var'(f)})^\omega$ as a conservative extension of π' , where $Var'(f)$ includes *all* variables in M_f , including auxiliary variables. The DEFINE statements define each auxiliary variable S_h in the characteristic function in terms of the set

of variables in $Var(f)$, supplemented with an auxiliary variable S_g corresponding to each subformula g of h . Since π'' is defined over $Var'(f)$, the values for the auxiliary variables can be defined uniquely using the characteristic function.

We now prove that for every subformula h of f and $\forall i \geq 0$ we have that $S_h \in \pi''(i)$ entails $\pi, i \models h$. The proof is by induction on h .

Base case: For $h = p \in Prop$, we have that $S_h = p$. So $S_h \in \pi''(i)$ iff $p \in \pi(i)$ iff $\pi, i \models p$. If $h = \neg p$, then we have that $S_h = !p$. So $S_h \in \pi''(i)$ iff $S_p \notin \pi''(i)$ iff $\pi, i \not\models p$ iff $\pi, i \models h$,

Induction step: Assume the claim holds for subformulas g, g_1, g_2 of f .

- $h = g_1 \wedge g_2$.
We have that $S_h = S_{g_1} \& S_{g_2}$. So $S_h \in \pi''(i)$ iff $S_{g_1} \in \pi''(i)$ and $S_{g_2} \in \pi''(i)$, which entails $\pi, i \models g_1$ and $\pi, i \models g_2$, which entails $\pi, i \models h$.
- $h = g_1 \vee g_2$.
We have that $S_h = S_{g_1} \vee S_{g_2}$. So $S_h \in \pi''(i)$ iff $S_{g_1} \in \pi''(i)$ or $S_{g_2} \in \pi''(i)$, which entails $\pi, i \models g_1$ or $\pi, i \models g_2$, which entails $\pi, i \models h$.
- $h = Xg$.
We have that $S_h = next(EL_g)$. So $S_h \in \pi''(i)$ iff $EL_g \in \pi''(i+1)$ iff (by the TRANS statement) $S_g \in \pi''(i+1)$, which, by induction, entails $\pi, i+1 \models g$, which entails $\pi, i \models h$.
- $h = g_1 \mathcal{U} g_2$.
We have that $S_h = S_{g_2} | (S_{g_1} \& P_h \& next(EL_h))$. Suppose $S_h \in \pi''(i)$. Then either $S_{g_2} \in \pi''(i)$ or $S_{g_1} \in \pi''(i)$, $P_h \in \pi''(i)$, and $EL_h \in \pi''(i+1)$. Note that, by the TRANS statement, $EL_h \in \pi''(i+1)$ iff $S_h \in \pi''(i+1)$. Thus, S_{g_2} can be “postponed,” at the cost of maintaining S_{g_1} and P_h . But the FAIRNESS statement implies that there is some $j : j \geq i$ where $P_h \notin \pi''(j)$. Choose the smallest such j . Then we have that $S_{g_2} \in \pi''(j)$, and $\forall k : i \leq k < j$, we have that $S_{g_1} \in \pi''(k)$. By induction, $\pi, j \models g_2$, and $\forall k, i \leq k < j$, we have that $\pi, k \models g_1$. It follows that $\pi, i \models h$.
- $h = g_1 \mathcal{R} g_2$.
We have that $S_h = S_{g_2} \& (S_{g_1} | next(EL_h))$. So $S_h \in \pi''(i)$ iff $S_{g_2} \in \pi''(i)$ and either $S_{g_1} \in \pi''(i)$ or $EL_h \in \pi''(i+1)$. Note that, by the TRANS statement, $EL_h \in \pi''(i+1)$ iff $S_h \in \pi''(i+1)$. It follows that S_{g_2} is “propagated” until “released” by S_{g_1} . That is, if $S_h \in \pi''(i)$ then $\forall j : j \geq i$, either $S_{g_2} \in \pi''(j)$ or there is some $k : i \leq k < j$, such that $S_{g_1} \in \pi''(k)$. By induction, for all $j : j \geq i$, either $\pi, j \models g_2$ or there is some $k : i \leq k < j$, such that $\pi, k \models g_1$. It follows that $\pi, i \models h$.
- $h = \mathcal{G}g$.
We have that $S_h = S_g \& next(EL_h)$. So $S_h \in \pi''(i)$ iff $S_g \in \pi''(i)$ and $EL_h \in \pi''(i+1)$ iff $g \in \pi(i)$ and (by the TRANS statement) $S_{\mathcal{G}g} \in \pi''(i+1)$. It follows that S_g is continually propagated. That is, if $S_h \in \pi''(i)$, then, for all $j : j \geq i$, we have that $S_g \in \pi''(j)$. By induction, for all $j : j \geq i$, we have that $\pi, j \models g$. It follows that $\pi, i \models h$.
- $h = \mathcal{F}g$.
We have that $S_h = S_g | (P_h \& next(EL_h))$. Suppose $S_h \in \pi''(i)$. Then either $S_g \in \pi''(i)$ or $P_h \in \pi''(i)$ and $EL_h \in \pi''(i+1)$. Note that, by the TRANS statement, $EL_h \in \pi''(i+1)$ iff $S_h \in \pi''(i+1)$. Thus, S_g can be “postponed,” at the cost of maintaining P_h . But the FAIRNESS statement implies that there is some $j : j \geq i$ where $P_h \notin \pi''(j)$.

Choose the smallest such j . Then we have that $S_g \in \pi''(j)$. By induction, $\pi, j \models g$. It follows that $\pi, i \models h$.

– $h = \mathcal{GF}g$.

We have that $S_h = ((next(EL_h) \& (S_g | P_h))$. Suppose $S_h \in \pi''(i)$. Then $EL_h \in \pi''(i+1)$ and either $S_g \in \pi''(i)$ or $P_h \in \pi''(i)$. Note that, by the TRANS statement, $EL_h \in \pi''(i+1)$ iff $S_h \in \pi''(i+1)$. Thus, for all $j : j \geq i$ we have that $S_h \in \pi''(j)$. Again, we note that S_g can be “postponed” at the cost of maintaining P_h . But the FAIRNESS statement implies that for every $j : j \geq i$ there is some $k : k \geq j$ where $P_h \notin \pi''(k)$. For each $j \geq i$, choose smallest such k ; call it k_j . Then we have that $S_g \in \pi''(k_j)$. By induction, $\pi, k_j \models g$. It follows that $\pi, i \models h$.

By assumption $EL_f \in \pi''(0)$. By the TRANS statement it follows that $S_f \in \pi''(0)$, and therefore $\pi, 0 \models f$. Therefore, f is satisfiable.

Appendix E: Application Benchmarks

In order to demonstrate further that our PANDA encoding outperforms the native encoding of CadenceSMV for real-life LTL satisfiability checking, we also tested both tools on a set of application benchmarks, comprised of formulas used to specify actual systems. Our application benchmark formulas come from six sources:⁷

1. **acacia demo-v22**: 10 formulas
2. **acacia demo-v3**: 6 formulas
3. **acacia example**: 25 formulas
4. **alaska szymanski**: 4 formulas
5. **anзу amba**: 8 formulas
6. **anзу genbuf**: 10 formulas

The *acacia demo-v22*, *acacia demo-v3*, and *acacia example* formulas are specifications for systems such as arbiters and traffic-light controllers, distributed with the Acacia tool⁸, as developed for a study on LTL realizability and synthesis [20]. The *alaska szymanski* formulas⁹ were developed as liveness properties for the Szymanski mutual exclusion protocol for LTL satisfiability and model-checking [16]. The Anzu¹⁰ benchmarks are sets of formulas used for synthesizing industrial hardware systems from specifications, combined into monolithic formulas for the purpose of satisfiability checking [6]. The *anзу amba* formulas are specifications for advanced microcontroller bus architectures while the *anзу genbuf* specifications describe generalized buffers.

We applied PANDA and CadenceSMV to these 63 application benchmark formulas. PANDA completed 51 formulas before spacing out, while CadenceSMV completed 45 formulas before spacing out. The comparison of the performance of CadenceSMV and PANDA-best on these application benchmark formulas is plotted in Figure 12 using a

⁷ Thanks to Viktor Schuppan for suggesting these sources, providing some of the formulas in SMV format, and constructing the Anzu formula combinations.

⁸ http://www.antichains.be/acacia/src/acacia_9_linux_i386.tar.gz

⁹ http://www.antichains.be/alaska/tacas08_experiments.zip

¹⁰ http://www.iaik.tugraz.at/content/research/design_verification/anзу/

classical cactus plot: the x-axis shows how many instances were checked in time less than or equal to the runtime given on the y-axis, presuming they are run in parallel. PANDA solved more formulas and in less time than CadenceSMV.

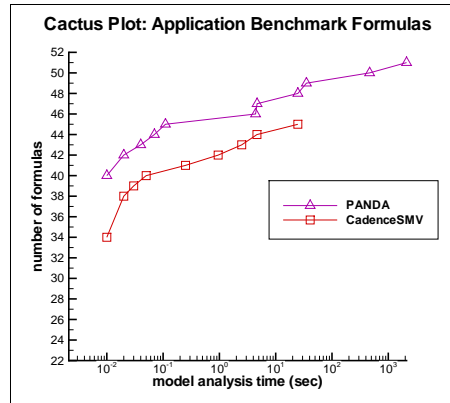


Fig. 12. Cactus plot: median model analysis time over all application benchmarks for CadenceSMV and the best PANDA encoding.