LTL Satisfiability Checking

Kristin Y. Rozier[1] * and Moshe Y. Vardi[2]

[1] NASA Langley Research Center, Hampton, Virginia 23681. Kristin.Y.Rozier@nasa.gov
[2] Rice University, Houston, Texas 77005, vardi@cs.rice.edu

Abstract. We report here on an experimental investigation of LTL satisfiability checking via a reduction to model checking. By using large LTL formulas, we offer challenging model-checking benchmarks to both explicit and symbolic model checkers. For symbolic model checking, we use both CadenceSMV and NuSMV. For explicit model checking, we use SPIN as the search engine, and we test essentially all publicly available LTL translation tools. Our experiments result in two major findings. First, most LTL translation tools are research prototypes and cannot be considered industrial quality tools. Second, when it comes to LTL satisfiability checking, the symbolic approach is clearly superior to the explicit approach.

1 Introduction

Model-checking tools are successfully used for checking whether systems have desired properties [11]. The application of model-checking tools to complex systems involves a nontrivial step of creating a mathematical model of the system and translating the desired properties into a formal specification. When the model does not satisfy the specification, model-checking tools accompany this negative answer with a counterexample, which points to an inconsistency between the system and the desired behaviors. It is often the case, however, that there is an error in the system model or in the formal specification. Such errors may not be detected when the answer of the model-checking tool is positive: while a positive answer does guarantee that the model satisfies the specification, the answer to the real question, namely, whether the system has the intended behavior, may be different.

The realization of this unfortunate situation has led to the development of several *sanity checks* for formal verification [29]. The goal of these checks is to detect errors in the system model or the properties. Sanity checks in industrial tools are typically simple, ad hoc, tests, such as checking for enabling conditions that are never enabled [31]. *Vacuity detection* provides a more systematic approach. Intuitively, a specification is satisfied vacuously in a model if it is satisfied in some non-interesting way. For example, the linear temporal logic (LTL) specification $\Box(req \rightarrow \Diamond grant)$ ("every request is eventually followed by a grant") is satisfied vacuously in a model with no requests. While

* Work contributing to this paper was completed at Rice University, Cambridge University, and NASA Langley Research Center, and was supported in part by the Rice Computational Research Cluster (Ada), funded by NSF under Grant CNS-0421109 and a partnership between Rice University, AMD and Cray.

vacuity checking cannot ensure that whenever a model satisfies a formula, the model is correct, it does identify certain positive results as vacuous, increasing the likelihood of capturing modeling and specification errors. Several papers on vacuity checking have been published over the last few years [2, 3, 8, 27, 26, 30, 34, 37], and various industrial model-checking tools support vacuity checking [2, 3, 8].

All vacuity-checking algorithms check whether a subformula of the specification does not affect the satisfaction of the specification in the model. In the example above, the subformula *req* does not affect satisfaction in a model with no request. There is, however, a possibility of a vacuous result that is not captured by current vacuity-checking approaches. If the specification is *valid*, that is, true in *all* models, then model checking this specification always results in a positive answer. Consider for example the specification $\Box(b_1 \rightarrow \Diamond b_2)$, where b_1 and b_2 are propositional formulas. If b_1 and b_2 are logically equivalent, then this specification is valid and is satisfied by all models. Nevertheless, current vacuity-checking approaches do not catch this problem. We propose a method for an additional sanity check to catch exactly this sort of oversight.

Writing formal specifications is a difficult task, which is prone to error just as implementation development is error prone. However, formal verification tools offer little help in debugging specifications other than standard vacuity checking. 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. 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. (For a different approach to debugging specifications, see [1].)

A basic observation underlying our work is that LTL satisfiability checking can be reduced to model checking. Consider a formula φ over a set *Prop* of atomic propositions. If a model M is *universal*, that is, it contains all possible traces over *Prop*, then φ is satisfiable precisely when the model M does *not* satisfy $\neg\varphi$. Thus, it is easy to add a satisfiability-checking feature to LTL model-checking tools.

LTL model checkers can be classified as *explicit* or *symbolic*. Explicit model checkers, such as SPIN [28] or SPOT [15], construct the state-space of the model explicitly and search for a trace falsifying the specification [12]. In contrast, symbolic model checkers, such as CadenceSMV [32], NuSMV [9], or VIS [5], represent the model and analyze it symbolically using binary decision diagrams (BDDs) [7].

LTL model checkers follow the automata-theoretic approach [45], in which the complemented LTL specification is explicitly or symbolically translated to a Büchi automaton, which is then composed with the model under verification; see also [44]. The model checker then searches for a trace of the model that is accepted by the automaton. All symbolic model checkers use the symbolic translation described in [10] and the analysis algorithm of [17], though CadenceSMV and VIS try to optimize further. There has been extensive research over the past decade into explicit translation of LTL to automata[13, 14, 18, 19, 20, 25, 21, 24, 40, 38, 42], but it is difficult to get a clear sense of the state of the art from a review of the literature. Measuring the performance of LTL satisfiability checking enables us to benchmark the performance of LTL model checking tools, and, more specifically, of LTL translation tools.

We report here on an experimental investigation of LTL satisfiability checking via a reduction to model checking. By using large LTL formulas, we offer challenging model-checking benchmarks to both explicit and symbolic model checkers. For symbolic model checking, we used both CadenceSMV and NuSMV. For explicit model checking, we use SPIN as the search engine, and we tested essentially all publicly available LTL translation tools. We used a wide variety of benchmark formulas, either generated randomly, as in [14], or using a scalable pattern (e.g., $\bigwedge_{i=1}^{n} p_i$). LTL formulas typically used for evaluating LTL translation tools are usually too small to offer challenging benchmarks. Note that real specifications typically consist of many temporal properties, whose conjunction ought to be satisfiable. Thus, studying satisfiability of large LTL formulas is quite appropriate.

Our experiments resulted in two major findings. First, most LTL translation tools are research prototypes and cannot be considered industrial quality tools. Many of them are written in scripting languages such as Perl or Python, which has drastic negative impact on their performance. Furthermore, these tools generally degrade gracelessly, often yielding incorrect results with no warning. Among all the tools we tested, only SPOT can be considered an industrial quality tool. Second, when it comes to LTL satisfiability checking, the symbolic approach is clearly superior to the explicit approach. Even SPOT, the best LTL translator in our experiments, was rarely able to compete effectively against the symbolic tools. This result is consistent with the comparison of explicit and symbolic approach to modal satisfiability [35, 36], but is somewhat surprising in the context of LTL satisfiability in view of [39].

Related software, called `lbtt`,[3] provides an LTL-to-Büchi explicit translator testbench and environment for basic profiling. The `lbtt` tool performs simple consistency checks on an explicit tool's output automata, accompanied by sample data when inconsistencies in these automata are detected [41]. Whereas the primary use of `lbtt` is to assist developers of explicit LTL translators in debugging new tools or comparing a pair of tools, we compare performance with respect to LTL satisfiability problems across a host of different tools, both explicit and symbolic.

The structure of the paper is as follows. Section 2 provides the theoretical background for this work. In Section 3, we describe the tools studied here. We define our experimental method in Section 4, and detail our results in Section 5. We conclude with a discussion in Section 6.

2 Theoretical Background

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 $\mathcal{X}$ (next time) $\mathcal{U}$ (until), $\mathcal{R}$ (release), $\Box$ (also called $\mathcal{G}$ for "globally") and $\Diamond$ (also called $\mathcal{F}$ for "in the future"). We define LTL formulas inductively:

[3] `www.tcs.hut.fi/Software/lbtt/`

Definition 1 *For every $p \in Prop$, p is a formula. If φ and ψ are formulas, then so are:*

$\neg\varphi$	$\varphi \wedge \psi$	$\varphi \rightarrow \psi$	$\varphi \mathcal{U} \psi$	$\mathcal{G}\varphi$
	$\varphi \vee \psi$	$\mathcal{X}\varphi$	$\varphi \mathcal{R} \varphi$	$\mathcal{F}\varphi$

LTL formulas describe the behavior of the variables in *Prop* over a linear series of time steps starting at time 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 instant [16].

Definition 2 *We interpret LTL formulas over computations of the form $\pi : \omega \rightarrow 2^{Prop}$. We define $\pi, i \vDash \varphi$ (computation π at time instant $i \in \omega$ satisfies LTL formula φ) as follows:*

$\pi, i \vDash p$ for $p \in Prop$ if $p \in \pi(i)$.
$\pi, i \vDash \varphi \wedge \psi$ if $\pi, i \vDash \varphi$ and $\pi, i \vDash \psi$.
$\pi, i \vDash \neg\varphi$ if $\pi, i \nvDash \varphi$.
$\pi, i \vDash \mathcal{X}\varphi$ if $\pi, i+1 \vDash \varphi$.
$\pi, i \vDash \varphi \mathcal{U} \psi$ if $\exists j \geq i$, such that $\pi, j \vDash \psi$ and $\forall k, i \leq k < j$, we have $\pi, k \vDash \varphi$.
$\pi, i \vDash \varphi \mathcal{R} \psi$ if $\forall j \geq i$, if $\pi, j \nvDash \psi$, then $\exists k$, $i \leq k < j$, such that $\pi, k \vDash \varphi$.

We define $(\mathcal{F}\varphi)$ as $(true\ \mathcal{U}\varphi)$ and $(\mathcal{G}\varphi)$ as $(\neg\mathcal{F}\neg\varphi)$. We take $models(\varphi)$ to be the set of computations that satisfy φ at time 0, i.e., $\{\pi : \pi, 0 \vDash \varphi\}$.

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

Definition 3 *A* **Büchi Automaton** *(BA) is a quintuple $(Q, \Sigma, \delta, q_0, F)$ where:*

Q is a finite set of states.
Σ is a finite alphabet.
$\delta : Q \times \Sigma \rightarrow Q$ is the transition relation.
$q_0 \in Q$ is the initial state.
$F \subseteq Q$ is a set of final states.

A run of a Büchi automaton over an infinite word $w = w_0, w_1, w_2, \ldots \in \Sigma$ is a sequence of states $q_0, q_1, q_2, \ldots \in Q$ such that $\forall i \geq 0$, $\delta(q_i, w_i) = q_{i+1}$. An infinite word w is accepted by the automaton if the run over w visits at least one state in F infinitely often. We denote the set of infinite words accepted by an automaton A by $L_\omega(A)$.

A computation satisfying LTL formula φ is an infinite word over the alphabet $\Sigma = 2^{Prop}$. The next theorem relates the expressive power of LTL to that of Büchi automata.

Theorem 1. [46] *Given an LTL formula φ, we can construct a Büchi automaton $A_\varphi = \langle Q, \Sigma, \delta, q_0, F \rangle$ such that $|Q|$ is in $2^{O(|\varphi|)}$, $\Sigma = 2^{Prop}$, and $L_\omega(A_\varphi)$ is exactly $models(\varphi)$.*

This theorem reduces LTL satisfiability checking to automata-theoretic nonemptiness checking, as φ is satisfiable iff $models(\varphi) \neq \emptyset$ iff $L_\omega(A_\varphi) \neq \emptyset$.

We can now relate LTL satisfiability checking to LTL model checking. Suppose we have a *universal model*, M, that generates all computations over its atomic propositions; that is, we have that $L + \omega(M) = (2^{Prop})^\omega$. We now have that M does *not* satisfy $\neg\varphi$ if and only if φ is satisfiable. Thus, φ is satisfiable precisely when the model checker finds a counterexample.

3 Tools Tested

In total, we tested eleven LTL compilation algorithms from nine research tools. To offer a broad, objective picture of the current state-of-the-art, we tested the algorithms against several different sequences of benchmarks, comparing, where appropriate, the size of generated automata in terms of numbers of states and transitions, translation time, model-analysis time, and correctness of the output.

3.1 Explicit Tools

The explicit LTL model checker SPIN [28] accepts either LTL properties, which are translated internally into Büchi automata, or Büchi automata for complemented properties ("never claims"). We tested SPIN with Promela (PROcess MEta LAnguage) never-claims produced by several LTL translation algorithms. (As SPIN's built-in translator is dominated by TMP, we do not show results for this translator.) The algorithms studied here represent all tools publicly available in 2006, as described in the following table:

Explicit Automata Construction Tools

LTL2AUT	(Daniele–Guinchiglia–Vardi)
Implementations (Java, Perl)	LTL2Buchi, Wring
LTL2BA (C)	(Oddoux–Gastin)
LTL2Buchi (Java)	(Giannakopoulou–Lerda)
LTL $\rightarrow$ NBA (Python)	(Fritz–Teegen)
Modella (C)	(Sebastiani–Tonetta)
SPOT (C++)	(Duret-Lutz–Poitrenaud–Rebiha–Baarir–Martinez)
TMP (SML of NJ)	(Etessami)
Wring (Perl)	(Somenzi–Bloem)

We provide here short descriptions of the tools and their algorithms, detailing aspects which may account for our results. We also note that aspects of implementation including programming language, memory management, and attention to efficiency, seem to have significant effects on tool performance.

Classical Algorithms Following [46], the first optimized LTL translation algorithm was described in [24]. The basic optimization ideas were: (1) generate states by demand only, (2) use node labels rather than edge labels to simplify translation to Promela, and (3) use a *generalized Büchi* acceptance condition so eventualities can be handled one at a time. The resulting generalized Büchi automaton (GBA) is then "degeneralized" or translated to a BA. **LTL2AUT** improved further on this approach by using lightweight propositional reasoning to generate fewer states [14]. We tested two implementations of LTL2AUT, one included in the Java-based LTL2Buchi tool and one included in the Perl-based Wring tool.

TMP[4] [18] and **Wring**[5] [40] each extend LTL2AUT with three kinds of additional optimizations. First, in the *pre-translation optimization*, the input formula is simplified

[4] `www.bell-labs.com/project/TMP/`

[5] `www.ist.tugraz.at/staff/bloem/wring.html`

using Negation Normal Form (NNF) and extensive sets of rewrite rules. Second, *mid-translation optimizations* tighten the LTL-to-GBA-to-BA translation algorithms. Third, the resulting automata are minimized further during *post-translation optimization*. In the end, TMP produces a BA whereas Wring halts translation with a GBA, which we had to degeneralize.

LTL2Buchi[6] [25] optimizes the LTL2AUT algorithm by initially generating transition-based generalized Büchi automata (TGBA) rather than node-labeled BA to allow for more compaction based on equivalence classes, contradictions, and redundancies in the state space. Special attention to efficiency is given during the ensuing translation to node-labeled BA. The algorithm incorporates the formula rewriting and BA-reduction optimizations of TMP and Wring.

Modella[7] focuses on minimizing the *nondeterminism* of the property automaton in an effort to minimize the size of the product of the property and system model automata during verification [38]. If the property automaton is deterministic, then the number of states in the product automaton will be at most the number of states in the system model. Thus, reducing nondeterminism is a desirable goal. This is accomplished using *semantic branching*, or branching on truth assignments, rather than the *syntactic branching* of LTL2AUT. Modella also postpones branching when possible.

Alternating Automata Tools Instead of the direct translation approach of [46], an alternative approach, based on *alternating automata*, was proposed in [43]. In this approach, the LTL formula is first translated into an alternating Büchi automaton, which is then translated to a nondeterministic Büchi automaton.

LTL2BA[8] [21] first translates the input formula into a *very weak* alternating automaton (VWAA). It then uses various heuristics to minimize the VWAA, before translating it to GBA. The GBA in turn is minimized before being translated into a BA, and finally the BA is minimized further. Thus, the algorithm's central focus is on optimization of intermediate representations through iterative simplifications and on-the-fly constructions.

LTL→NBA[9] follows a similar approach to that of LTL2BA [19]. Unlike the heuristic minimization of VWAA used in LTL2BA, LTL→NBA uses a game-theoretic minimization based on utilizing a delayed simulation relation for on-the-fly simplifications.

Back to Classics **SPOT**[10] is the most recently developed LTL-to-Büchi optimized translation tool [15]. It does not use alternating automata, but borrows ideas from all the tools described above. It adds two important optimizations: (1) unlike all other tools, it uses pre-branching states, rather than post-branching states (as introduced in [13]), and (2) it uses BDDs ([6]) for propositional reasoning.

[6] `http://ase.arc.nasa.gov/people/dimitra/LTL2Buchi.php`
[7] `http://www.science.unitn.it/~stonetta/modella.html`
[8] `http://www.liafa.jussieu.fr/~oddoux/ltl2ba/`
[9] `http://www.ti.informatik.uni-kiel.de/ABA-Simulation/ltl.cgi`
[10] `http://spot.lip6.fr/wiki/SpotWiki`

3.2 Symbolic Tools

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 [7]. The model checker uses a BDD-based fix-point algorithm to find a *fair path* in the model-automaton product [17]. CadenceSMV[11] [32] and NuSMV[12] [9] both evolved from the original Symbolic Model Verifier developed at CMU [33]. Both tools support LTL model checking via the symbolic translation of LTL to automata described in [10]. CadenceSMV additionally implements heuristics that attempt to reduce LTL model checking to CTL model checking in some cases [4].

4 Experimental Methods

4.1 Performance Evaluation

We ran all tests on Ada, a Rice University Cray XD1 cluster.[13] Ada is comprised of 158 nodes with 4 processors (cores) per node for a total of 632 CPUs in pairs of dual core 2.2 GHz AMD Opteron processors with 1 MB L2 cache. There are 2 GB of memory per core or a total of 8 GB of RAM per node. The operating system is SuSE Linux 9.0 with the 2.6.5 kernel. Each of our tests was run with exclusive access to one node and was considered to time out after 4 hours of run time. We measured all timing data using the Unix `time` command.

Explicit Tools Each test was performed in two steps. First, we applied the translation tools to the negation of the input LTL formula and ran them with the standard flags recommended by the tools' authors, plus any additional flag needed to specify that the output automaton should be in Promela. Second, each output automaton, in the form of a Promela *never-claim*, was checked by SPIN. In this role, SPIN serves as a search engine for each of the LTL translation tools; it takes a never-claim and checks it for non-emptiness in conjunction with an input model.[14]

In all tests, the model was a *universal* Promela program, enumerating all possible traces over *Prop*. For example, when $Prop = \{A, B\}$, the Promela model is:

```
bool A,B;
/* define an active procedure to generate values for A and B */
active proctype generateValues()
{ do
   :: atomic{ A = 0; B = 0; }
   :: atomic{ A = 0; B = 1; }
   :: atomic{ A = 1; B = 0; }
   :: atomic{ A = 1; B = 1; }
  od }
```

[11] http://www.cadence.com/company/cadence_labs_research.html

[12] http://nusmv.irst.itc.it/

[13] http://rcsg.rice.edu/ada/

[14] It would be interesting to use SPOT's SCC-based search algorithm [23] as the underlying search engine, rather than SPIN's nested depth-first search algorithm [12].

We use the `atomic{}` construct to ensure that the Boolean variables change value in one unbreakable step. Note that the size of this model is exponential in the number of atomic propositions.

Symbolic Tools We compare the explicit tools with CadenceSMV and NuSMV. To check whether a LTL formula φ is satisfiable, we model check $\neg\varphi$ against a universal SMV model. For example, if $\varphi = (X(a))$, we provide the following input to NuSMV:

```
MODULE main
  VAR
    a : boolean;
    b : boolean;
    c : boolean;
  LTLSPEC !(X(a=1 ))
  FAIRNESS
    1
```

SMV negates the specification, $\neg\varphi$, symbolically compiles φ into A_φ, and conjoins A_φ with the universal model. If the automaton is not empty, then SMV finds a fair path, which satisfies the formula φ. In this way, SMV acts as both a symbolic compiler and a search engine.

4.2 Input Formulas

We benchmarked the tools against three types of scalable formulas: random formulas, counter formulas, and pattern formulas. Scalability played an important role in our experiment, since the goal was to challenge the tools with large formulas and state spaces. All tools were applied to the same formulas and the results (satisfiable or unsatisfiable) were compared. The symbolic tools, which were always in agreement, were considered as reference tools for checking correctness.

Random Formulas In order to cover as much of the problem space as possible, we tested sets of 250 randomly-generated formulas varying the formula length and number of variables as in [14]. We randomly generated sets of 250 formulas varying the number of variables, N, from 1 to 3, and the length of the formula, L, from 5 up to 65. We set the probability of choosing a temporal operator $P = 0.5$ to create formulas with both a nontrivial temporal structure and a nontrivial Boolean structure. Other choices were decided uniformly. We report median running times as the distribution of run times has a high variance and contains many outliers. All formulas were generated prior to testing, so each tool was run on the *same* formulas. While we made sure that, when generating a set of length L, every formula was exactly of length L and not *up to L*, we did find that the formulas were frequently reducible. Tools with better initial formula reduction algorithms performed well in these tests.

Counter Formulas Pre-translation rewriting is highly effective for random formulas, but ineffective for structured formulas [18, 40]. To measure performance on scalable, non-random formulas we tested the tools on formulas that describe n-bit binary counters with increasing values of n. These formulas are irreducible by pre-translation rewriting, uniquely satisfiable, and represent a predictably-sized state space. Whereas our measure of correctness for random formulas is a conservative check that the tools find satisfiable formulas to be satisfiable, we check for precisely the unique counterexample for each counter formula. 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 unnested $\mathcal{X}$-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 $b_0, b_1, \ldots, b_{n-1}$ 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 $i = 0 \bmod n$.

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 -> ((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) ->
       ( X(X(X(X(b)))) &&
        X ( ( (!m) &&
           (b -> X(X(X(X(b))))) &&
           (!b -> X(X(X(X(!b))))) ) U m ) ) )
4. [] ( (m && b) ->
        (X(X(X(X(!b)))) &&
          (X ( (b && !m && X(X(X(X(!b))))) U
               (m || (!m && !b && X(X(X(X(b)))) &&
                       X( ( !m && (b -> X(X(X(X(b))))) &&
                            (!b -> X(X(X(X(!b))))) ) U m ) ) ) ) ) )
```

Note that this encoding creates formulas of length $O(n^2)$. A more compact encoding results in formulas of length $O(n)$. For example, we can replace formula (2) above with:

```
2. ((!b) && X((!b) && X((!b) && X(!b))))
```

We can eliminate the use of $\mathcal{U}$-connectives in the formula by adding an atomic proposition c representing the carry bit. The required properties of an n-bit counter with carry are as follows:

```
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 m is 1 and b is 0 then c is 0 and n steps later b is 1.
4) If m is 1 and b is 1 then c is 1 and n steps later b is 0.
5) If there is no carry, then the next bit stays the same n steps later.
6) If there is a carry, flip the next bit n steps later and adjust the carry.
```

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

```
1. (m) && ( [](m -> ((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) -> (!c && X(X(X(X(b))))) )
4. [] ( (m && b) -> (c && X(X(X(X(!b))))) )
5. [] (!c && X(!m)) ->
      ( X(!c) && (X(b) -> X(X(X(X(X(b)))))) &&
                 (X(!b) -> X(X(X(X(X(!b)))))) )
6. [] (c -> ( ( X(!b) -> ( X(!c) && X(X(X(X(X(!b))))) ) ) &&
                         ( X(c) && X(X(X(X(X(b))))) ) ))
```

Pattern Formulas We further investigated the problem space by testing the tools on the eight classes of scalable formulas defined by [22] to evaluate the performance of explicit state algorithms on temporally-complex formulas.

$$E(n) = \bigwedge_{i=1}^{n} \Diamond p_i,\ U(n) = (\dots(p_1\ \mathcal{U}\ p_2)\ \mathcal{U}\ \dots)\ \mathcal{U}\ p_n,\ R(n) = \bigwedge_{i=1}^{n} (\Box\Diamond p_i \vee \Diamond\Box p_{i+1}).$$

$$U_2(n) = p_1\ \mathcal{U}\ (p_2\ \mathcal{U}\ (\dots p_{n-1}\ \mathcal{U}\ p_n)\dots),\ C_1(n) = \bigvee_{i=1}^{n} \Box\Diamond p_i,\ C_2(n) = \bigwedge_{i=1}^{n} \Box\Diamond p_i.$$

$$Q(n) = \bigwedge (\Diamond p_i \vee \Box p_{i+1}),\ S(n) = \bigwedge_{i=1}^{n} \Box p_i.$$

5 Experimental Results

Our experiments resulted in two major findings. First, most LTL translation tools are research prototypes, not industrial quality tools. Second, the symbolic approach is clearly superior to the explicit approach for LTL satisfiability checking.

5.1 The Scalability Challenge

When checking the satisfiability of specifications we need to consider large LTL formulas. Our experiments focus on challenging the tools with scalable formulas. Unfortunately, most explicit tools do not rise to the challenge. In general, the performance of explicit tools degrades substantially as the automata they generate grow beyond 1,000 states. This degradation is manifested in both timeouts (our timeout bound was 4 hours per formula) and errors due to memory management. This should be contrasted with BDD tools, which routinely handle hundreds of thousands and even millions of nodes.

We illustrate this first with run-time results for counter formulas. We display each tool's total run time, which is a combination of the tool's automaton generation time and SPIN's model analysis time. We include only data points for which the tools provide correct answers; we know all counter formulas are uniquely satisfiable. As is shown in Figures 1 and 2,[15] SPOT is the only explicit tool that is somewhat competitive with the symbolic tools. Generally, the explicit tools time out or die before scaling to $n = 10$, when the automata have only a few thousands states; only a few tools passed $n = 8$.

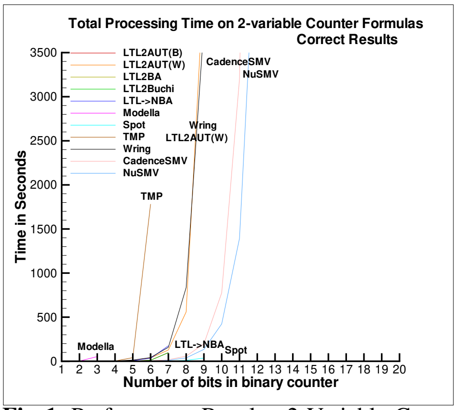

Fig. 1. Performance Results: 2-Variable Counters

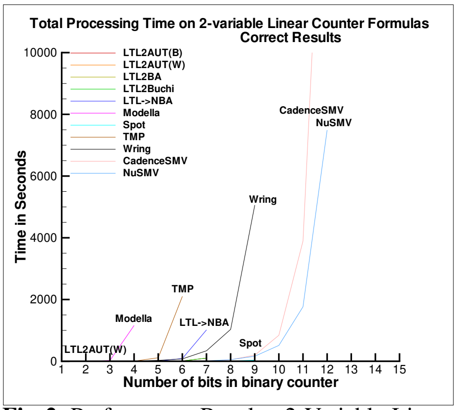

Fig. 2. Performance Results: 2-Variable Linear Counters

Figures 3 and 4 show median automata generation and model analysis times for random formulas. Most tools, with the exception of SPOT and LTL2BA, timeout or die before scaling to formulas of length 60. The difference in performance between SPOT and LTL2BA, on one hand, and the rest of the explicit tools is quite dramatic. Note that up to length 60, model-analysis time is negligible. SPOT and LTL2BA can routinely handle formulas of up to length 150, while the symbolic tools scale past length 200, with run times of a few seconds.

Figure 5 shows performance on the E-class formulas. Recall that $\neg E(n)$ is the formula $\bigvee_{i=1}^{n} \Box \neg p_i$. Since each formula $\Box \neg p_i$ can be translated into an automaton with a fixed number of states, $\neg E(n)$ can be translated into an automaton with $O(n)$ states. Nevertheless, most tools show an unnecessary exponential blow-up. CadenceSMV is the only tool whose performance seems to scale linearly. (The evidence for NuSMV is inconclusive.)

Graceless Degradation Most explicit tools do not behave robustly and die gracelessly. When LTL2Buchi has difficulty processing a formula, it produces over 1,000 lines of `java.lang.StackOverflowError` exceptions. LTL2BA periodically exits with "Command exited with non-zero status 1" and prints into the Promela file, "ltl2ba: releasing a

[15] We recommend viewing all figures online, in color, and magnified.

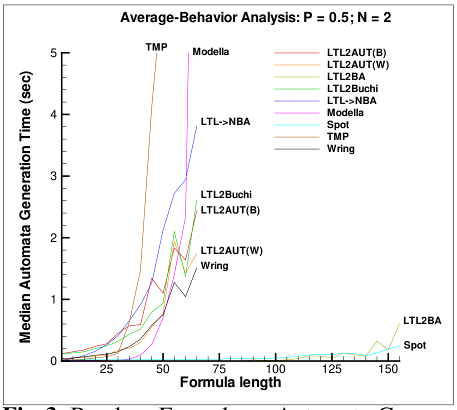

Fig. 3. Random Formulas – Automata Generation Times

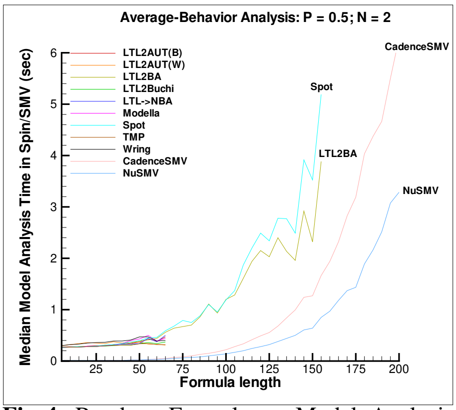

Fig. 4. Random Formulas – Model Analysis Times

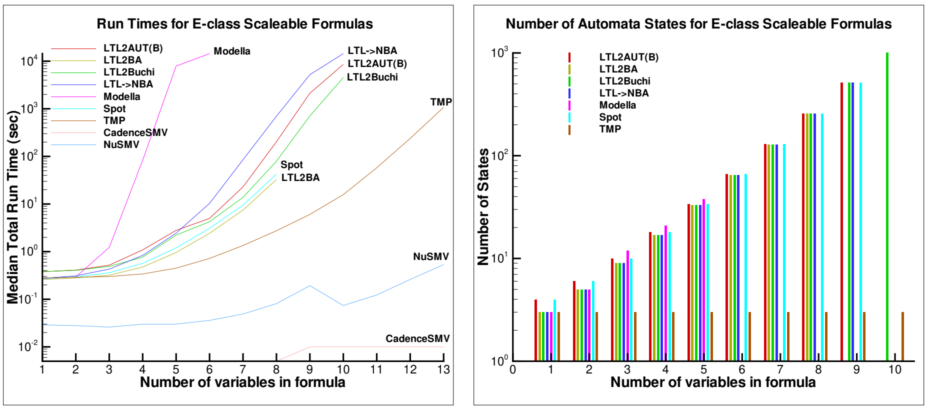

Fig. 5. E-class Formula Data

free block, saw 'end of formula'." Python traceback errors hinder LTL→NBA. Modella suffers from a variety of memory errors including `*** glibc detected *** double free or corruption (out): 0x55ff4008 ***`. Sometimes Modella causes a segmentation fault and other times Modella dies gracefully, reporting "full memory" before exiting. When used purely as a LTL-to-automata translator, SPIN often runs for thousands of seconds and then exits with non-zero status 1. TMP behaves similarly. Wring often triggers Perl ""Use of freed value in iteration" errors. When the translation results in large Promela models, SPIN frequently yields segmentation faults during its own compilation. For example, SPOT translates the formula $E(8)$ to an automaton with 258 states and 6,817 transitions in 0.88 seconds. SPIN analyzes the resulting Promela model in 41.75 seconds. SPOT translates the $E(9)$ formula to an automaton with 514 states and 20,195 transitions in 2.88 seconds, but SPIN segmentation faults when trying to compile this model. SPOT and the SMV tools are the only tools that consistently degrade gracefully; they either timeout or terminate with a succinct, descriptive message.

A more serious problem is that of incorrect results, i.e., reporting "satisfiable" for an unsatisfiable formula or vice versa. Note, for example, in Figure 5, the size of the automaton generated by TMP is independent of n, which is an obvious error. The problem is particularly acute when the returned automaton A_φ is empty (no state). On one hand, an empty automaton accepts the empty language. On the other hand, SPIN conjoins the Promela model for the never-claim with the model under verification, so an empty automaton, when conjoined with a universal model, actually acts as a universal model. The tools are not consistent in their handling of empty automata. Some, such as LTL2Buchi and SPOT return an explicit indication of an empty automaton, while Modella and TMP just return an empty Promela model. We have taken an empty automaton to mean "unsatisfiable." In Figure 6 we show an analysis of correctness for random formulas. Here we counted "correct" as any verdict, either "satisfiable" or "unsatisfiable," that matched the verdict found by the two SMVs for the same formula as the two SMVs always agree. We excluded data for any formulas that timed out or triggered error messages. Many of the tools show degraded correctness as the formulas scale in size.

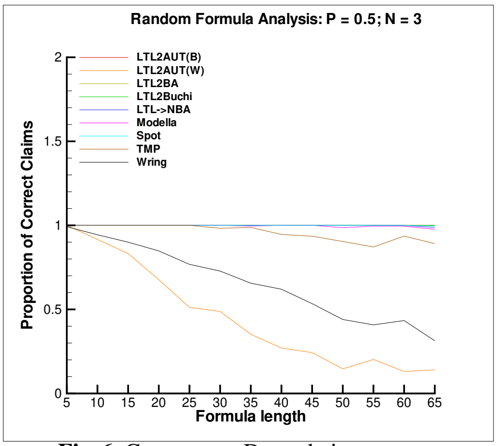

Fig. 6. Correctness Degradation

Does Size Matter? The focus of almost all LTL translation papers, starting with [24], has been on minimizing automata size. It has already been noted that automata minimization may not result in model checking performance improvement [18] and specific attention has been given to minimizing the size of the product with the model [38, 22]. Our results show that size, in terms of both number of automaton states and transitions is not a reliable indicator of satisfiability checking run-time. Intuitively, the smaller the automaton, the easier it is to check for nonemptiness. This simplistic view, however, ignores the effort required to minimize the automaton. It is often the case that tools spend more time constructing the formula automaton than constructing and analyzing the product automaton. As an example, consider the performance of the tools on counter formulas. We see in Figures 1 and 2 dramatic differences in the performance of the tools on such formulas. In contrast, we see in Figures 7 and 8 that the tools do not differ significantly in terms of the size of generated automata. Similarly, Figure 5, shows little correlation between automata size and run time for E-class formulas.

Consider also the performance of the tools on random formulas. In Figure 9 we see the performance in terms of size of generated automata. Performance in terms of run

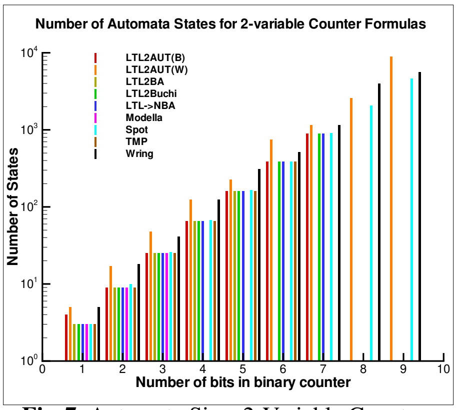

Fig. 7. Automata Size: 2-Variable Counters

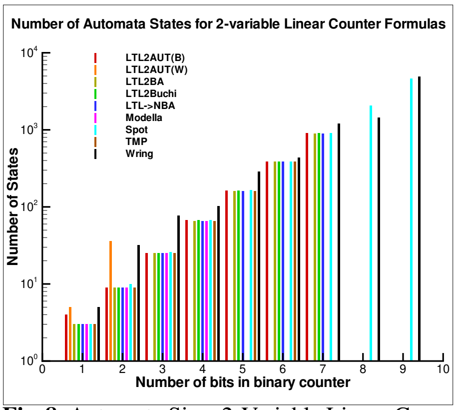

Fig. 8. Automata Size: 2-Variable Linear Counters

time is plotted in Figure 11, where each tool was run until it timed out or reported an error for more than 10% of the sampled formulas. SPOT and LTL2BA consistently have the best performance in terms of run time, but they are average performers in terms of automata size. LTL2Buchi consistently produces significantly more compact automata, in terms of both states and transitions. It also incurs lower SPIN model analysis times than SPOT and LTL2BA. Yet LTL2Buchi spends so much time generating the automata that it does not scale nearly as well as SPOT and LTL2BA.

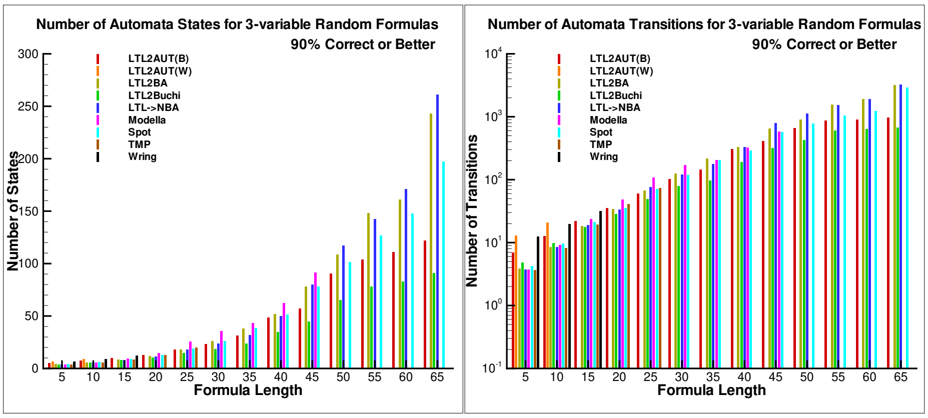

Fig. 9. State and Transition Counts for 3-Variable Random Formulas

5.2 Symbolic Approaches Outperform Explicit Approaches

Across the various classes of formulas, the symbolic tools outperformed the explicit tools, demonstrating faster performance and increased scalability. (We measured only combined automata-generation and model-analysis time for the symbolic tools. The translation to automata is symbolic and is very fast; it is linear in the size of the formula [10].) We see this dominance with respect to counter formulas in Figures 1 and 2, for random formulas in Figures 3, 4, and 11, and for E-class formulas in Figure 5. For U-class formulas, no explicit tools could handle $n = 10$, while the symbolic tools scale up to $n = 20$; see Figure 10. The only exception to the dominance of the symbolic tools occurs with 3-variable linear counter formulas, where SPOT outperforms both symbolic tools. We ran the tools on many thousands of formulas and did not find a single case in which either symbolic tool yielded an incorrect answer yet every explicit tool gave at least one incorrect answer during our tests.

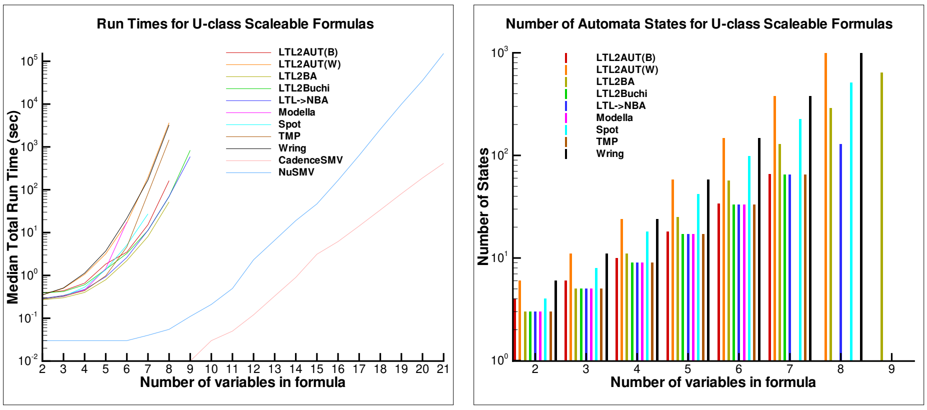

Fig. 10. U-class Formula Data

The dominance of the symbolic approach is consistent with the findings in [35, 36], which reported on the superiority of a symbolic approach with respect to an explicit approach for satisfiability checking for the modal logic K. In contrast, [39] compared explicit and symbolic translations of LTL to automata in the context of symbolic model checking and found that explicit translation performs better in that context. Consequently, they advocate a *hybrid* approach, combining symbolic systems and explicit automata. Note, however, that not only is the context in [39] different than here (model checking rather than satisfiability checking), but also the formulas studied there are generally small and translation time is negligible, in sharp contrast to the study we present here. We return to the topic of model checking in the concluding discussion.

Figures 3, 4, and 11 reveal why the explicit tools generally perform poorly. We see in the figures that for most explicit tools automata-generation times by far dominate model-analysis times, which calls into question the focus in the literature on minimizing automata size. Among the explicit tools, only SPOT and LTL2BA seem to have been designed with execution speed in mind. Note that, other than Modella, SPOT and LTL2BA are the only tools implemented in C/C++.

6 Discussion

Too little attention has been given in the formal-verification literature to the issue of debugging specifications. We argued here for the adoption of a basic sanity check: satisfiability checking for both the specification and the complemented specification. We showed that LTL satisfiability checking can be done via a reduction to checking univer-

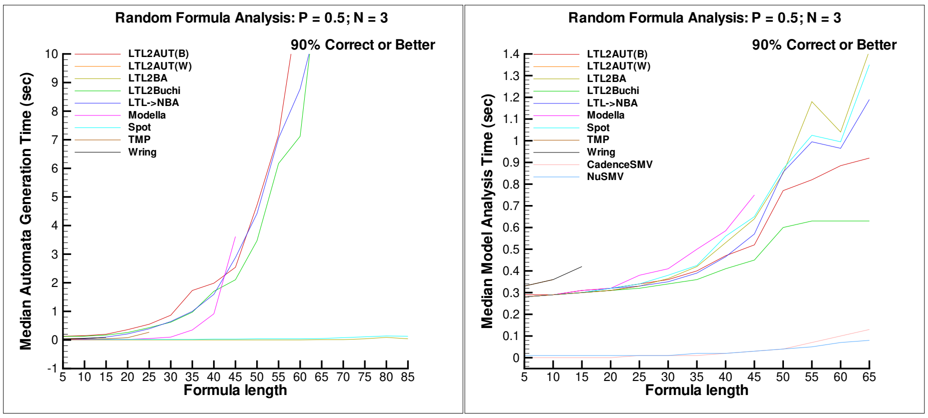

Fig. 11. Automata generation and SPIN Analysis Times for 3-Variable Random Formulas

sal models and benchmarked a large array of tools with respect to satisfiability checking of scalable LTL formulas.

We found that the existing literature on LTL to automata translation provides little information on actual tool performance. We showed that most LTL translation tools, with the exception of SPOT, are research prototypes, which cannot be considered industrial-quality tools. The focus in the literature has been on minimizing automata size, rather than evaluating overall performance. Focusing on overall performance reveals a large difference between LTL translation tools. In particular, we showed that symbolic tools have a clear edge over explicit tools with respect to LTL satisfiability checking.

While the focus of our study was on LTL satisfiability checking, there are a couple of conclusions that apply to model checking in general. First, LTL translation tools need to be fast and robust. In our judgment, this rules out implementations in languages such as Perl or Python and favors C or C++ implementations. Furthermore, attention needs to be given to graceful degradation. In our experience, tool errors are invariably the result of graceless degradation due to poor memory management. Second, tool developers should focus on overall performance instead of output size. It has already been noted that automata minimization may not result in model checking performance improvement [18] and specific attention has been given to minimizing the size of the product with the model [38]. Still, no previous study of LTL translation has focused on model checking performance, leaving a glaring gap in our understanding of LTL model checking.

References

[1] G. Ammons, D. Mandelin, R. Bodik, and J.R. Larus. Debugging temporal specifications with concept analysis. In *PLDI, Proc. ACM Conf.*, pages 182–195, 2003.

[2] R. Armoni, L. Fix, A. Flaisher, O. Grumberg, N. Piterman, A. Tiemeyer, and M.Y. Vardi. Enhanced vacuity detection for linear temporal logic. In *CAV, Proc 15th Int'l Conf.* Springer, 2003.

[3] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. Efficient detection of vacuity in ACTL formulas. *Formal Methods in System Design*, 18(2):141–162, 2001.

[4] R. Bloem, K. Ravi, and F. Somenzi. Efficient decision procedures for model checking of linear time logic properties. In *CAV, Proc 11th Int'l Conf*, volume 1633 of *Lecture Notes in Computer Science*, pages 222–235. Springer, 1999.

[5] 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*, volume 1102 of *Lecture Notes in Computer Science*, pages 428–432. Springer, 1996.

[6] R.E. Bryant. Graph-based algorithms for boolean-function manipulation. *IEEE Trans. on Computers*, C-35(8), 1986.

[7] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, Jun 1992.

[8] D. Bustan, A. Flaisher, O. Grumberg, O. Kupferman, and M.Y. Vardi. Regular vacuity. In *CHARME*, volume 3725 of *LNCS*, pages 191–206. Springer, 2005.

[9] A. Cimatti, E.M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new symbolic model checker. *It'l J. on Software Tools for Tech. Transfer*, 2(4):410–425, 2000.

[10] E. M. Clarke, O. Grumberg, and K. Hamaguchi. Another look at LTL model checking. *Formal Methods in System Design*, 10(1):47–71, 1997.

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

[12] C. Courcoubetis, M.Y. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1:275–288, 1992.

[13] J-M. Couvreur. On-the-fly verification of linear temporal logic. In *Proc. FM*, pages 253–271, 1999.

[14] N. Daniele, F. Guinchiglia, and M.Y. Vardi. Improved automata generation for linear temporal logic. In *CAV, Proc. 11th Int'l Conf*, volume 1633 of *LNCS*, pages 249–260. Springer, 1999.

[15] A. Duret-Lutz and D. Poitrenaud. SPOT: An extensible model checking library using transition-based generalized büchi automata. In *MASCOTS, Proc. 12th Int'l Workshop*, pages 76–83. IEEE Computer Society, 2004.

[16] E.A. Emerson. Temporal and modal logic. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 16, pages 997–1072. Elsevier, MIT Press, 1990.

[17] 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.

[18] K. Etessami and G.J. Holzmann. Optimizing Büchi automata. In *CONCUR, Proc. 11th Int'l Conf.*, Lecture Notes in CS 1877, pages 153–167. Springer, 2000.

[19] C. Fritz. Constructing Büchi automata from linear temporal logic using simulation relations for alternating büchi automata. In *Proc. 8th Intl. CIAA*, number 2759 in Lecture Notes in Computer Science, pages 35–48. Springer, 2003.

[20] C. Fritz. Concepts of automata construction from LTL. In *LPAR, Proc. 12th Int'l Conf.*, Lecture Notes in Computer Science 3835, pages 728–742. Springer, 2005.

[21] P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In *CAV, Proc. 13th Int'l Conf*, volume 2102 of *LNCS*, pages 53–65. Springer, 2001.

[22] J. Geldenhuys and H. Hansen. Larger automata and less work for LTL model checking. In *Model Checking Software, 13th Int'l SPIN Workshop*, volume 3925 of *LNCS*, pages 53–70. Springer, 2006.

[23] J. Geldenhuys and A. Valmari. Tarjan's algorithm makes on-the-fly LTL verification more efficient. In *Proc. 10th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science 2988, pages 205–219. Springer, 2004.

[24] R. Gerth, D. Peled, M.Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In P. Dembiski and M. Sredniawa, editors, *Protocol Specification, Testing, and Verification*, pages 3–18. Chapman & Hall, Aug 1995.

[25] 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.

[26] A. Gurfinkel and M. Chechik. Extending extended vacuity. In *FMCAD, 5th Int'l Conf*, volume 3312 of *Lecture Notes in Comp Sci*, pages 306–321. Springer, 2004.

[27] A. Gurfinkel and M. Chechik. How vacuous is vacuous. In *TACAS, 10th Int'l Conf*, volume 2988 of *Lecture Notes in Computer Science*, pages 451–466. Springer, 2004.

[28] G.J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997. Special issue on Formal Methods in Software Practice.

[29] O. Kupferman. Sanity checks in formal verification. In *CONCUR, Proc. 17th Int'l Conf.*, volume 4137 of *Lecture Notes in Comp Sci*, pages 37–51. Springer, 2006.

[30] O. Kupferman and M.Y. Vardi. Vacuity detection in temporal model checking. *J. on Software Tools For Technology Transfer*, 4(2):224–233, Feb 2003.

[31] R.P. Kurshan. *FormalCheck User's Manual*. Cadence Design, Inc., 1998.

[32] K. McMillan. The SMV language. Technical report, Cadence Berkeley Lab, 1999.

[33] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[34] K.S. Namjoshi. An efficiently checkable, proof-based formulation of vacuity in model checking. In *16th CAV*, volume 3114 of *LNCS*, pages 57–69. Springer, 04.

[35] 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.

[36] N. Piterman and M.Y. Vardi. From bidirectionality to alternation. *Theoretical Computer Science*, 295(1–3):295–321, Feb 2003.

[37] M. Purandare and F. Somenzi. Vacuum cleaning CTL formulae. In *CAV, Proc. 14th Conf*, Lecture Notes in Computer Science, pages 485–499. Springer, Jul 2002.

[38] R. Sebastiani and S. Tonetta. "more deterministic" vs. "smaller" büchi automata for efficient LTL model checking. In *CHARME*, pages 126–140. Springer, 2003.

[39] R. Sebastiani, S. Tonetta, and M.Y. Vardi. Symbolic systems, explicit properties: on hybrid approaches for LTL symbolic model checking. In *CAV, Proc. 17th Int'l Conf.*, Lecture Notes in Computer Science 3576, pages 350–373. Springer, 2005.

[40] F. Somenzi and R. Bloem. Efficient Büchi automata from LTL formulae. In *CAV, Proc. 12th Int'l Conf*, volume 1855 of *LNCS*, pages 248–263. Springer, 2000.

[41] H. Tauriainen and K. Heljanko. Testing LTL formula translation into Büchi automata. *STTT - Int'l J. on Software Tools for Tech. Transfer*, 4(1):57–70, 2002.

[42] X. Thirioux. Simple and efficient translation from LTL formulas to Büchi automata. *Electr. Notes Theor. Comput. Sci.*, 66(2), 2002.

[43] M.Y. Vardi. Nontraditional applications of automata theory. In *STACS, Proc. Int'l*, volume 789, pages 575–597. LNCS, Springer-Verlag, 1994.

[44] M.Y. Vardi. Automata-theoretic model checking revisited. In *Proc. 7th Int'l Conf. on Verification, Model Checking, and Abstract Interpretation*, volume 4349 of *LNCS*, pages 137–150. Springer, 2007.

[45] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. 1st LICS*, pages 332–344, Cambridge, Jun 1986.

[46] M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, Nov 1994.