# Symbolic Systems, Explicit Properties: on Hybrid Approaches for LTL Symbolic Model Checking

Roberto Sebastiani[1⋆], Stefano Tonetta[1⋆], and Moshe Y. Vardi[2⋆⋆]

[1] DIT, Università di Trento, {rseba,stonetta}@dit.unitn.it
[2] Dept. of Computer Science, Rice University, vardi@cs.rice.edu

**Abstract.** In this work we study *hybrid* approaches to LTL symbolic model checking; that is, approaches that use explicit representations of the property automaton, whose state space is often quite manageable, and symbolic representations of the system, whose state space is typically exceedingly large. We compare the effects of using, respectively, (i) a purely symbolic representation of the property automaton, (ii) a symbolic representation, using logarithmic encoding, of explicitly compiled property automaton, and (iii) a partitioning of the symbolic state space according to an explicitly compiled property automaton. We apply this comparison to three model-checking algorithms: the doubly-nested fixpoint algorithm of Emerson and Lei, the reduction of emptiness to reachability of Biere et al., and the singly-nested fixpoint algorithm of Bloem et al. for weak automata. The emerging picture from our study is quite clear, hybrid approaches outperform pure symbolic model checking, while partitioning generally performs better than logarithmic encoding. The conclusion is that the hybrid approaches benefits from state-of-the-art techniques in semantic compilation of LTL properties. Partitioning gains further from the fact that the image computation is applied to smaller sets of states.

## 1 Introduction

Linear-temporal logic (LTL) [26] is a widely used logic to describe infinite behaviors of discrete systems. Verifying whether an LTL property is satisfied by a finite transition system is a core problem in Model Checking (MC). Standard automata-theoretic MC techniques consider the formula $\varphi$ that is the negation of the desired behavior and construct a generalized Büchi automaton (GBA) $B_\varphi$ with the same language. Then, they compute the product of this automaton $B_\varphi$ with the system $S$ and check for emptiness. To check emptiness, one has to compute the set of *fair states*, i.e. those states of the product automaton that are extensible to a fair path. The main obstacle to model checking is the *state-space explosion*; that is, the product is often too large to be handled.

⋆ Supported in part by ORCHID, a project sponsored by Provincia Autonoma di Trento, and by a grant from Intel Corporation.
⋆⋆ Supported in part by NSF grants CCR-9988322, CCR-0124077, CCR-0311326, IIS-9908435, IIS-9978135, EIA-0086264, and ANI-0216467, by BSF grant 9800096, by Texas ATP grant 003604-0058-2003, and by a grant from the Intel Corporation.

Explicit-state model checking uses highly optimized LTL-to-GBA compilation, cf. [11, 13, 14, 17–19, 21, 29, 30], which we refer to as *semantic compilation*. Such compilation may involve an exponential blow-up in the worst case, though such blow-up is rarely seen in practice. Emptiness checking is performed using nested depth-first search [10]. To deal with the state-explosion problem, various state-space reductions are used, e.g., [25, 31].

Symbolic model checking (SMC) [6] tackles the state-explosion problem by representing the product automaton symbolically, usually by means of (ordered) BDDs. The compilation of the property to symbolically represented GBA is purely *syntactic*, and its blow-up is linear (which induces an exponential blow-up in the size of the state space), cf. [9]. Symbolic model checkers typically compute the fair states by means of some variant of the doubly-nested-fixpoint Emerson-Lei algorithm (EL) [12, 15, 27]. For "weak" property automata, the doubly-nested fixpoint algorithm can be replaced by a singly-nested fixpoint algorithm [3]. An alternative algorithm [1] reduces emptiness checking to reachability checking (which requires a singly-nested fixpoint computation) by doubling the number of symbolic variables.

Extant model checkers use either a pure explicit-state approach, e.g., in SPIN [22], or a pure symbolic approach, e.g., in NuSMV[7]. Between these two approaches, one can find *hybrid* approaches, in which the property automaton, whose state space is often quite manageable, is represented explicitly, while the system, whose state space is typically exceedingly large, is represented symbolically. For example, the singly-nested fixpoint algorithm of [3] is based on an explicit construction of the property automaton. (See [2, 8] for other hybrid approaches.)

In [28], motivated by previous work on *generalized symbolic trajectory evaluation* (GSTE) [34], we proposed a hybrid approach to LTL model checking, referred to as *property-driven partitioning* (PDP). In this approach, the property automaton $A_\varphi$ is constructed explicitly, but its product with the system is represented in a partitioned fashion. If the state space of the system is $\mathcal{S}$ and that of the property automaton is $\mathcal{B}$, then we maintain a subset $Q \subseteq \mathcal{S} \times \mathcal{B}$ of the product space as a collection $\{Q_b \, : \, b \in \mathcal{B}\}$ of sets, where each $Q_b = \{s \in \mathcal{S} \; : \; (s,b) \in Q\}$ is represented symbolically. Thus, in PDP, we maintain an array of BDDs instead of a single BDD to represent a subset of the product space. Based on extensive experimentation, we argued in [28] that PDP is superior to SMC, in many cases demonstrating exponentially better scalability.

While the results in [28] are quite compelling, it is not clear why PDP is superior to SMC. On one hand, one could try to implement PDP in a purely symbolic manner by ensuring that the symbolic variables that represent the property-automaton state space precede the variables that represent the system state space in the BDD variable order. This technique, which we refer to as *top ordering*, would, in effect, generate a separate BDD for each block in the partitioned product space, but without generating an explicit array of BDDs, thus avoiding the algorithmic complexity of PDP. It is possible that, under such variable order, SMC would perform comparably (or even better) than PDP[3]. On the other hand, it is possible that the reason underlying the good performance of PDP is not the partitioning of the state space, but, rather, the explicit compilation of the property automaton, which yields a reduced state space for the property automaton.

[3] We are grateful to R.E. Bryant and F. Somenzi for making this observation.

So far, however, no detailed comparison of hybrid approaches to the pure symbolic approach has been published. (VIS [4] currently implements a hybrid approach to LTL model checking. The property automaton is compiled explicitly, but then represented symbolically, using the so-called *logarithmic encoding*, so SMC can be used. No comparison of this approach to SMC, however, has been published). Interestingly, another example of property-based partitioning can be found in the context of explicit-state model checking [20].

In this paper we undertake a systematic study of this spectrum of representation approaches: purely symbolic representation (with or without top ordering), symbolic representation of semantically compiled automata (with or without top ordering), and partitioning with respect to semantically compiled automata (PDP). An important observation here is that PDP is orthogonal to the choice of the fixpoint algorithm. Thus, we can study the impact of the representation on different algorithms; we use here EL, the reduction of emptiness to reachability of [1], and the singly-nested fixpoint algorithm of [3] for weak property automata. The focus of our experiments is on measuring *scalability*. We study scalable systems and measure how running time scales as a function of the system size. We are looking for a multiplicative or exponential advantage of one algorithm over another one.

The emerging picture from our study is quite clear, hybrid approaches outperform pure SMC. Top ordering generally helps, but not as much as semantic compilation. PDP generally performs better than symbolic representation of semantically compiled automata (even with top ordering). The conclusion is that the hybrid approaches benefit from state-of-the-art techniques in semantic compilation of LTL properties. Such techniques includes preprocessing simplification by means of rewriting [13, 30], postprocessing state minimization by means of simulations [13, 14, 21, 30], and midprocessing state minimization by means of alternating simulations [17, 18]. In addition, empty-language states of the automata can be discarded. PDP gains further from the fact that the image computation is applied on smaller sets of states. The comparison to SMC with top ordering shows that managing partitioning symbolically is not as efficient as managing it explicitly.

Section 2 contains required background on explicit-state and symbolic model checking. Section 3 describes hybrid approaches to symbolic model checking. Section 4 contains experimental results. Finally, Section 5 contains some concluding remarks.

## 2 Background

### 2.1 Explicit-state LTL model checking

Let *Prop* be a set of propositions. We take $\Sigma$ to be equal to $2^{Prop}$. A fair transition system (FTS) is a tuple $\langle \mathcal{S}, \mathcal{S}_0, T, \Sigma, \mathcal{L}, \mathcal{F}^{\mathcal{S}} \rangle$, where $\mathcal{S}$ is a set of states, $\mathcal{S}_0 \subseteq \mathcal{S}$ are the initial states, $T \subseteq \mathcal{S} \times \mathcal{S}$ is the transition relation, $\mathcal{L} : \mathcal{S} \to \Sigma$ is a labeling function, and $\mathcal{F}^{\mathcal{S}} \subseteq 2^{\mathcal{S}}$ is the set of fairness constraints (each set in $\mathcal{F}^{\mathcal{S}}$ should be visited infinitely often). A generalized Büchi automaton (GBA) is a tuple $\langle \mathcal{B}, b_0, \Sigma, \delta, \mathcal{F}^{\mathcal{B}} \rangle$, where $\mathcal{B}$ is a set of states, $b_0 \in \mathcal{B}$ is the initial state, $\delta \in \mathcal{B} \times \Sigma \times \mathcal{B}$ is the transition relation, and $\mathcal{F}^{\mathcal{B}} \subseteq 2^{\mathcal{B}}$ is the set of fairness constraints. The product between an FTS $S$ and a GBA

$B$ is the FTS $\langle \mathcal{P}, \mathcal{P}_0, T^{\mathcal{P}}, \Sigma, \mathcal{L}^{\mathcal{P}}, \mathcal{F}^{\mathcal{P}} \rangle$, where $\mathcal{P} = \mathcal{S} \times \mathcal{B}$; $\mathcal{P}_0 = \mathcal{S}_0 \times \{b_0\}$; $(p_1, p_2) \in T^{\mathcal{P}}$ iff $p_1 = (s_1, b_1)$, $p_2 = (s_2, b_2)$, $(s_1, s_2) \in T$, $\mathcal{L}(s_1) = a$, and $(b_1, a, b_2) \in \delta$; $\mathcal{L}^{\mathcal{P}}(p) = a$ iff $p = (s, b)$ and $\mathcal{L}(s) = a$; $\mathcal{F}^{\mathcal{P}} = \{F^{\mathcal{S}} \times \mathcal{B}\}_{F^{\mathcal{S}} \in \mathcal{F}^{\mathcal{S}}} \cup \{\mathcal{S} \times F^{\mathcal{B}}\}_{F^{\mathcal{B}} \in \mathcal{F}^{\mathcal{B}}}$.

LTL model checking is solved by compiling the negation $\varphi$ of a property into a GBA $B^{\varphi}$ and checking the emptiness of the product $P$ between the FTS $S$ and $B^{\varphi}$ [32]. In explicit-state model checking, emptiness checking is performed by state enumeration: a depth-first search can detect if there exists a fair strongly-connected component reachable from the initial states [10].

### 2.2 Symbolic LTL model checking

Suppose that for an FTS $\langle \mathcal{S}, \mathcal{S}_0, T, \Sigma, \mathcal{L}, \mathcal{F} \rangle$ there exists a set of symbolic (Boolean) variables $V$ such that $\mathcal{S} \subseteq 2^V$, i.e. a state $s$ of $\mathcal{S}$ is an assignment to the variables of $V$. We can think of a subset $Q$ of $\mathcal{S}$ as a predicate on the variables $V$. Since $a \in \Sigma$ can be associated with the set $\mathcal{L}^{-1}(a) \subseteq \mathcal{S}$, $a$ can be thought of as a predicate on $V$ too. Similarly, the transition relation $T$ is represented by a predicate on the variables $V \cup V'$, where $V'$ contains one variable $v'$ for every $v \in V$ ($v'$ represents the next value of $v$). In the following, we will identify a set of states or a transition relation with the predicate that represents it.

Given two FTS $S^1 = \langle \mathcal{S}^1, \mathcal{S}^1_0, T^1, \Sigma, \mathcal{L}^1, \mathcal{F}^1 \rangle$ with $\mathcal{S}^1 \subseteq 2^{V^1}$ and $S^2 = \langle \mathcal{S}^2, \mathcal{S}^2_0, T^2, \Sigma, \mathcal{L}^2, \mathcal{F}^2 \rangle$ with $\mathcal{S}^2 \subseteq 2^{V^2}$, the composition of $S^1$ and $S^2$ is the FTS $\langle \mathcal{S}^{\mathcal{P}}, \mathcal{S}^{\mathcal{P}}_0, T^{\mathcal{P}}, \Sigma, \mathcal{L}^{\mathcal{P}}, \mathcal{F}^{\mathcal{P}} \rangle$, where $\mathcal{S}^{\mathcal{P}} \subseteq 2^{V^{\mathcal{P}}}$, $V^{\mathcal{P}} = V^1 \cup V^2$, $\mathcal{S}^{\mathcal{P}}(v_1, v_2) = \mathcal{S}^1(v_1) \wedge \mathcal{S}^2(v_2) \wedge (L^1(v_1) \leftrightarrow L^2(v_2))$, $\mathcal{S}^{\mathcal{P}}_0(v_1, v_2) = \mathcal{S}^1_0(v_1) \wedge \mathcal{S}^2_0(v_2)$, $T^{\mathcal{P}}(v_1, v_2, v_1', v_2') = T^1(v_1, v_1') \wedge T^2(v_2, v_2')$, $\mathcal{L}^{\mathcal{P}}(v_1, v_2) = L^1(v_1)$, $\mathcal{F}^{\mathcal{P}} = \mathcal{F}^1 \cup \mathcal{F}^2$.

Again, the negation $\varphi$ of an LTL property is compiled into an FTS, such that the product with the system contains a fair path iff there is a system's violation of the property. The standard compilation produces an FTS $\langle \mathcal{S}^{\varphi}, \mathcal{S}^{\varphi}_0, T^{\varphi}, \Sigma, \mathcal{L}^{\varphi}, \mathcal{F}^{\varphi} \rangle$, where $\mathcal{S}^{\varphi} = 2^{V^{\varphi}}$, $V^{\varphi} = Atoms(\varphi) \cup Extra(\varphi)$, so that $Atoms(\varphi) \subseteq Prop$ are the atoms of $\varphi$, $Extra(\varphi) \cap V = \emptyset$ and $Extra(\varphi)$ contains one variable for every temporal connective occurring in $\varphi$ [6, 9, 33]. We call this *syntactic compilation*.

To check language containment, a symbolic model checker implements a fixpoint algorithm [6]. Sets of states are manipulated by using basic set operations such as intersection, union, complementation, and the preimage and postimage operations. Since sets are represented by predicates on Boolean variables, intersection, union and complementation are translated into resp. $\wedge$, $\vee$ and $\neg$. The preimage and postimage operations are translated into the following formulas:

$$preimage(Q) = \exists v'((Q[v'/v])(v') \wedge T(v, v'))$$
$$postimage(Q) = (\exists v(Q(v) \wedge T(v, v')))[v/v']$$

The most used representation for predicates on Boolean variables are Binary Decision Diagrams (BDDs) [5]. For a given variable order, BDDs are canonical representations. However, the order may affect considerably the size of the BDDs. By means of BDDs, set operations can be performed efficiently.

## 3 Hybrid approaches

Between the two approaches described in Section 2, *hybrid* approaches represent the system symbolically and the property automaton explicitly. Thus, they semantically compile the LTL formula into a GBA. This can be either encoded into an FTS or used to drive PDP. Notice that the choice between the two does not affect the set of states visited. Indeed, the product representation is completely orthogonal to the model-checking algorithm.

**Logarithmic encoding** Given the GBA $B = \langle \mathcal{B}, b_0, \Sigma, \delta, \mathcal{F} \rangle$ corresponding to the formula $\varphi$, we can compile $B$ into the FTS $\langle \mathcal{S}^B, \mathcal{S}_0^B, T^B, \Sigma, \mathcal{L}^B, \mathcal{F}^B \rangle$, where $\mathcal{S}^B = 2^{V^B}$, $V^B = Extra(B) \cup Atoms(\varphi)$, $Extra(B) \cap Prop = \emptyset$ and $|Extra(B)| = log(|\mathcal{B}|)$, $\mathcal{S}_0^B$ represents $\{b_0\}$, $T^B(s, a, s', a')$ is true iff $(s, a, s') \in \delta$, $\mathcal{L}^B(s, a) = a$ and finally every $F^B \in \mathcal{F}^B$ represents the correspondent set $F \in \mathcal{F}$. Intuitively, we number the states of the GBA and then use binary notation to refer to the states symbolically. This is referred to as *logarithmic encoding*.

**Property-driven partitioning** Given an FTS $S$ and a GBA $B$, we can consider the partitioning of the product state space: $\{\mathcal{P}_b\}_{b \in \mathcal{B}}$, where $\mathcal{P}_b = \{p \in \mathcal{P} : p = (s, b)\}$. Thus, a subset $Q$ of $\mathcal{P}$ can be represented by the following set of states of $S$: $\{Q_b\}_{b \in \mathcal{B}}$, where $Q_b = \{s : (s, b) \in Q\}$. If $Q^1 = \{Q_b^1\}_{b \in \mathcal{B}}$ and $Q^2 = \{Q_b^2\}_{b \in \mathcal{B}}$, we translate the set operations used in symbolic algorithms into:

$$Q^1 \wedge Q^2 := \{Q_b^1 \wedge Q_b^2\}_{b \in \mathcal{B}} \quad Q^1 \vee Q^2 := \{Q_b^1 \vee Q_b^2\}_{b \in \mathcal{B}} \quad \neg Q := \{\neg Q_b\}_{b \in \mathcal{B}}$$
$$preimage(Q) := \{\textstyle\bigvee_{(b,a,b') \in \delta} preimage(Q_{b'}) \wedge a\}_{b \in \mathcal{B}}$$
$$postimage(Q) := \{\textstyle\bigvee_{(b',a,b) \in \delta} postimage(Q_{b'} \wedge a)\}_{b \in \mathcal{B}}$$

All symbolic model-checking algorithms that operate on the product FTS can be partitioned according to the property automaton, operating on a BDD array rather than on a single BDD (see [28]).

**Hypothesis** Our hypothesis is that hybrid approaches combine the best features of explicit-state and symbolic model checking techniques. On one hand, they use a symbolic representation for the system and a symbolic algorithm, which may benefit from the compact representation of BDDs. On the other hand, they may benefit from state-of-the-art techniques in LTL-to-Büchi compilation, which aim at optimizing the state space of the property automaton, and prune away redundant and empty-language parts. Optimizations include preprocessing simplification by means of rewriting [13, 30]; post-processing minimization by means of simulations [13, 14, 21, 30], and midprocessing minimization by means of alternating simulation [17, 18].

In addition, PDP has the advantage of using a partitioned version of the product state space. Partitioned methods usually gain from the fact that the image operation is applied to smaller sets of states, cf. [16]. Furthermore, PDP enables traversing the product state space without computing it explicitly. The experiments reported in the next section test our hypothesis.

| | |
|---|---|
| `gas.prop1` | $G((pump\_started1 \ \&\ ((!pump\_charged1)\ U\ operator\_prepaid\_2)) \rightarrow ((!operator\_activate\_1)\ U\ (operator\_activate\_2 \mid G!operator\_activate\_1)))$ |
| `gas.prop2` | $(G(pump\_started1 \rightarrow ((!operator\_prepaid\_1)\ U\ (operator\_change\_1 \mid G!operator\_prepaid\_1)))) \rightarrow (G((pump\_started1 \ \&\ ((!pump\_charged1)\ U\ operator\_prepaid\_2)) \rightarrow ((!operator\_activate\_1)\ U\ (operator\_activate\_2 \mid G!operator\_activate\_1)))$ |
| `gas.prop3` | $((!operator\_prepaid\_2)\ U\ operator\_prepaid\_1) \rightarrow (!pump\_started2\ U\ (pump\_started1 \mid G(!pump\_started2)))$ |
| `gas.prop4` | $G(pump\_started1 \rightarrow ((!pump\_started2)\ U\ (pump\_charged1 \mid G(!pump\_started2))))$ |
| `stack.prop1` | $G(callPushd1 \ \&\ ((!returnPopd1)\ U\ callTop\_Down) \rightarrow F(callTop\_Down \ \&\ F(callProcessd1)))$ |
| `stack.prop2` | $G((callPush \ \&\ (!returnPop\ U\ callEmpty)) \rightarrow F(callEmpty \ \&\ F(returnEmptyFalse)))$ |
| `stack.prop3` | $G((callPushd1 \ \&\ F(returnEmptyTrue)) \rightarrow (!returnEmptyTrue\ U\ returnPopd1))$ |
| `stack.prop4` | $G((callPushd1 \ \&\ ((!returnPopd1)\ U\ (callPushd2 \ \&\ ((!returnPopd1 \ \&\ !returnPopd2)\ U\ callTop\_Down)))) \rightarrow F(callTop\_Down \ \&\ F(callProcessd2 \ \&\ FcallProcessd2)))$ |
| `stack.prop5` | $G((callPushd1 \ \&\ (!returnPopd1\ U\ callPushd2)) \rightarrow (!returnPopd1\ U\ (!returnPopd2 \mid G!returnPopd1)))$ |

**Table 1.** LTL properties

# 4 Experimental results

We tested the different product representations on two scalable systems with their LTL properties, by using three different model-checking algorithms. Every plot we show in this paper is characterized by three elements: the system $S$, the LTL property $\varphi$ and the model-checking algorithm used.

## 4.1 Systems, properties and model checking algorithms

The two systems and their properties are inspired by case studies of the Bandera Project (`http://bandera.projects.cis.ksu.edu`). The first system is a gas-station model. There are $N$ customers who want to use one pump. They have to prepay an operator who then activates the pump. When the pump has charged, the operator give the change to the customer. We will refer to this system as `gas`. The second system is a model of a stack with the standard pop and push functions. In this case, scalability is given by the maximum size $N$ of the stack. The properties of the two systems are displayed in Tab. 1.

The first model checking algorithm we used is the classic Emerson-Lei (EL) algorithm [12], which computes the set of fair states. There are many variants of this algorithm [15, 27], but all are based on a doubly-nested fixpoint computation: an outer loop updates an approximation of the set of fair states until a fixpoint is reached; at every iteration of the inner loop, a number of inner fixpoints is computed, one for every fairness constraint; every inner fixpoint prunes away those states that cannot reach the corresponding fairness constraint inside the approximation.

The second algorithm is a reduction of liveness checking to safety checking (l2s) [1]. The reduction is performed by doubling the number of symbolic variables. This

| property | ltl2smv | | modella | | |
|---|---|---|---|---|---|
| | extra variables | fairness constraints | states | extra variables ($\lceil$log(states)$\rceil$) | fairness constraints |
| gas.prop1 | 4 | 4 | 6 | 3 | 1 |
| gas.prop2 | 7 | 7 | 32 | 5 | 1 |
| gas.prop3 | 3 | 3 | 4 | 2 | 1 |
| gas.prop4 | 3 | 3 | 6 | 3 | 1 |
| stack.prop1 | 4 | 4 | 4 | 2 | 1 |
| stack.prop2 | 4 | 4 | 4 | 2 | 1 |
| stack.prop3 | 3 | 3 | 6 | 3 | 1 |
| stack.prop4 | 6 | 6 | 9 | 4 | 1 |
| stack.prop5 | 4 | 4 | 5 | 3 | 1 |

**Table 2.** Automata details.

way, it is possible to choose non-deterministically a state from which we look for a fair loop. We can check the presence of such a loop by a reachability analysis. To assure that the loop is fair, one has to add a further symbolic variable for every fairness constraint.

The third technique (weak-safety) consists of checking if the automaton is simple enough to apply a single fixpoint computation in order to find a fair loop [3]. If the automaton is weak, we can define a set $\mathcal{B}$ of states such that there exists a fair path if and only if there exists a loop inside $\mathcal{B}$ reachable from an initial state. If the automaton is terminal (the property is safety), we can define a set $\mathcal{B}$ of states such that there exists a fair path if and only if $\mathcal{B}$ is reachable from an initial state.

### 4.2 LTL to Büchi automata conversion

In this section, we focus the attention on the compilation of LTL formulas into (generalized) Büchi automata. For syntactic compilation, we used `ltl2smv`, distributed together with NuSMV. As for semantic compilation, we used MODELLA, which uses also some techniques described in [13, 14, 19, 30]. In Tab. 2, we reported the size of the automata used in the tests.

Note that the automata created by MODELLA are degeneralized, i.e. they have only one fairness constraint. Degenerilization involves a blow-up in the number of states that is linear in the number of fairness constraints (without degeneralization, the same linear factor shows up in the complexity of emptiness testing).

Recall that l2s doubles the number of symbolic variables in order to reduce emptiness to reachability. This is equivalent to squaring the size of the state space. Since in PDP we work directly with the state space of the property automaton, we need to square the explicit state space, while doubling the number of symbolic variables that describe the system. We provide details in the full version of the paper. In order to apply the third technique, we checked which automata were weak or terminal: we found that automata corresponding to stack.prop1, stack.prop2 and stack.prop4 were weak, and that the automata corresponding to gas.prop1, gas.prop3, gas.prop4, stack.prop3 and stack.prop5 were terminal.

### 4.3 Hybrid approaches

Hereafter, `log-encode` stands for the logarithmic encoding of the explicit representation of the automata. Note that an explicit LTL-to-Büchi compiler usually uses fewer symbolic variables than standard syntactic compilation (Tab. 2). Nevertheless, one may think that the syntactic compilation, whose transition constraints are typically simpler, is more suitable for symbolic model checking. As we see below, this is not the case.

We use `top-order` to denote the option of putting the symbolic variables of the property automaton at the top of the variable ordering. Consider a BDD $d$ that represents a set $Q$ of states of the product and a state $b$ of the property automaton. Let $b$ correspond to an assignment to the symbolic variables of the property automaton. If you follow this assignment in the structure of $d$, you find a sub-BDD, which corresponds to the set $Q_b$. Thus, by traversing the product state space with the option `top-order`, every BDD will contain an implicit partitioning of the set of states it represents.

Finally, we consider the PDP representation of the product state space. PDP uses the same automaton encoded by `log-encode` to partition the state space. Unlike `top-order`, the partitioning is handled explicitly (see [28]).

### 4.4 Results

We used NuSMV as platform to perform out tests. We run NuSMV on the Rice Terascale Cluster (RTC)[4], a TeraFLOP Linux cluster based on Intel Itanium 2 Processors. A timeout has been fixed to 72 hours for each run. The execution time (in seconds) has been plotted in log scale against the size of the system. The results are shown in Figs. 1-26.[5] Every plot corresponds to one system, one property, one model checking algorithm. Figs. 1-9 show the results of EL algorithm. Figs. 10-18 show the results of l2s[6] . Figs. 19-26 show the results of BRS (in these plots, syntactic compilation uses EL).

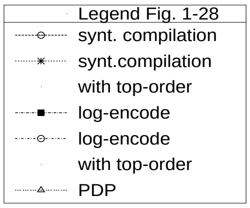
Analyzing the plots, we see that syntactic compilation performs always worse than semantic compilation. In the case of the `stack` system, the gap is considerable. The `top-order` option typically helps logarithmic encoding, while in the case of syntactic compilation it is less reliable: in some cases (see Figs. 14,17,18), it degrades the performance a lot. PDP usually performs better than `log-encode`, even if combined with `top-order`.

In conclusion, the results confirm our hypothesis: hybrid approaches perform better than standard techniques, independently of the model checking algorithm adopted. Moreover, they usually benefit from partitioning. Finally, by handling the partitioning explicitly, we get a further gain. This last point shows that accessing an adjacency list of successors may perform better than existentially quantifying the variables of a BDD.

[4] `http://www.citi.rice.edu/rtc/`

[5] All examples, data and tools used in these tests, as well as larger plots and the full version of the paper, are available at `http://www.science.unitn.it/~stonetta/CAV05`.

[6] We are grateful to Armin Biere and Viktor Schuppan for providing us with their tools in order to test the combination of "liveness to safety" with automata-theoretic approaches.



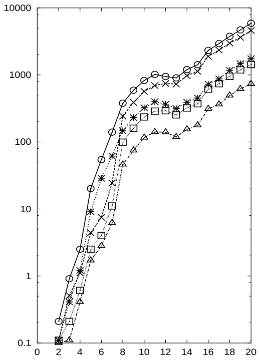



**Fig. 1.** `gas`, prop. 1, EL

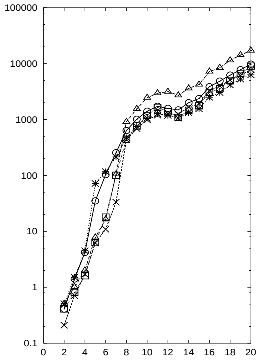


**Fig. 2.** `gas`, prop. 2, EL

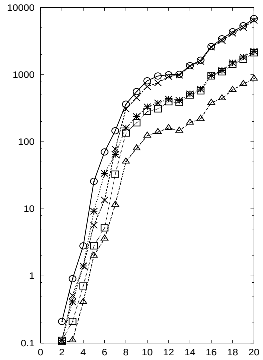


**Fig. 3.** `gas`, prop. 3, EL

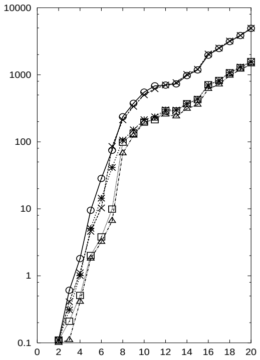


**Fig. 4.** `gas`, prop. 4, EL

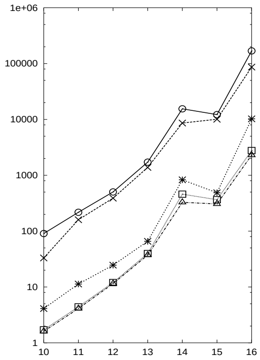


**Fig. 5.** `stack`, prop. 1, EL

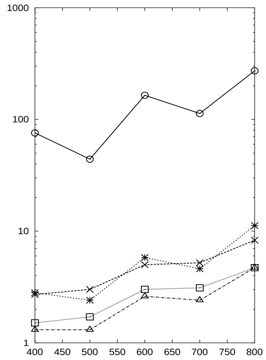


**Fig. 6.** `stack`, prop. 2, EL

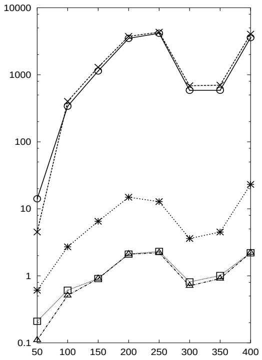


**Fig. 7.** `stack`, prop. 3, EL

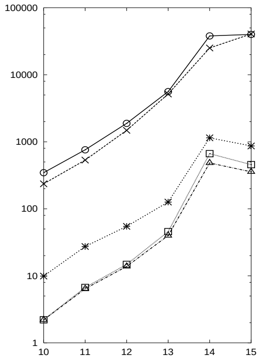


**Fig. 8.** `stack`, prop. 4, EL

### 4.5 Scaling up the number of partitions

In the previous section, we have seen that PDP has the best performance among the techniques we tested. However, a doubt may arise about the feasibility of the partitioning when the number of partitions grows. For this reason, we looked for some LTL properties whose corresponding automaton has a large number of states. We took as example a property used in the PAX Project (`http://www.informatik.uni-kiel.de/˜kba/pax`): $((GF\,p0 \rightarrow GF\,p1)\&(GF\,p2 \rightarrow GF\,p0)\&(GF\,p3 \rightarrow GF\,p2)\&(GF\,p4 \rightarrow GF\,p2)\&(GF\,p5 \rightarrow$

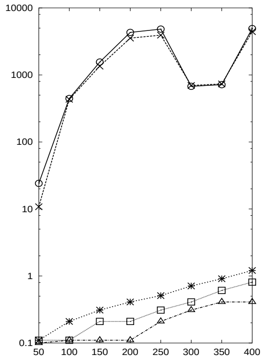


**Fig. 9.** `stack`, prop. 5, EL

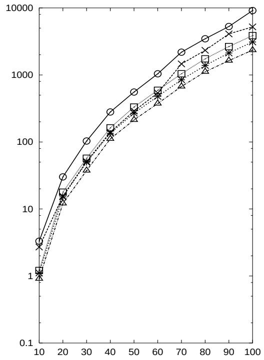


**Fig. 10.** `gas`, prop. 1, l2s

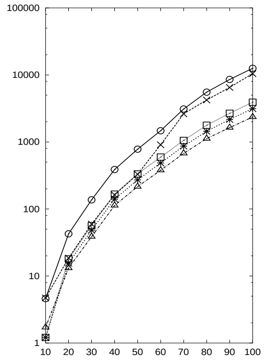


**Fig. 11.** `gas`, prop. 2, l2s

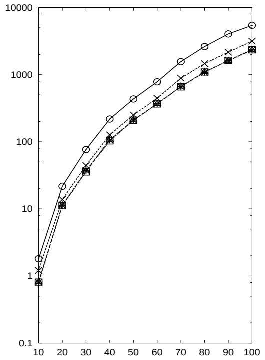


**Fig. 12.** `gas`, prop. 3, l2s

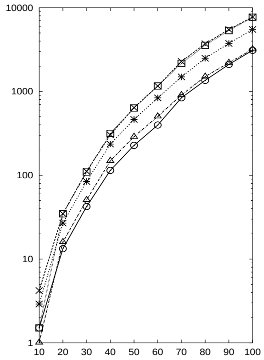


**Fig. 13.** `gas`, prop. 4, l2s

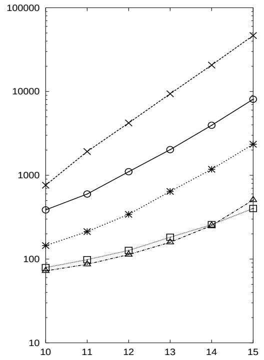


**Fig. 14.** `stack`, prop. 1, l2s

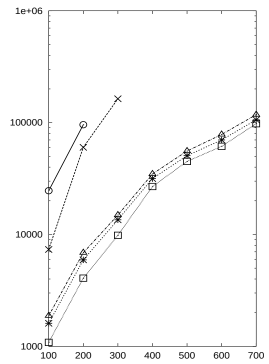


**Fig. 15.** `stack`, prop. 2, l2s

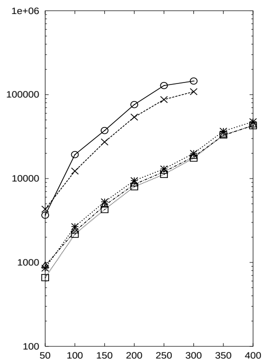


**Fig. 16.** `stack`, prop. 3, l2s

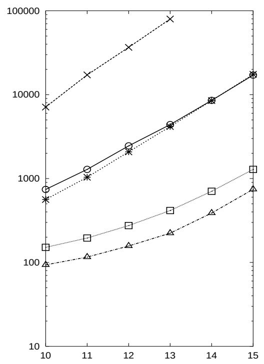


**Fig. 17.** `stack`, prop. 4, l2s

$GF\,p3)\&(GF\,p6 \rightarrow GF(p5|p4))\&(GF\,p7 \rightarrow GF\,p6)\&(GF\,p1 \rightarrow GF\,p7)) \rightarrow GF\,p8$. Trying to compile this property into a GBA, we faced an interesting problem: no compiler we tried managed to translate this property in reasonable time[7]. For this reason,

[7] Actually, the only translator that succeeded was `ltl2tgba` (`http://spot.lip6.fr`). However, we had to disable simulation-based reduction so that the resulting automaton had more than 70000 states and even parsing such an automaton took more than model checking time.

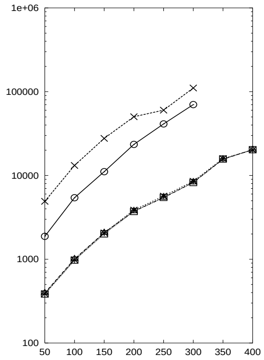


**Fig. 18.** `stack`, prop. 5, l2s

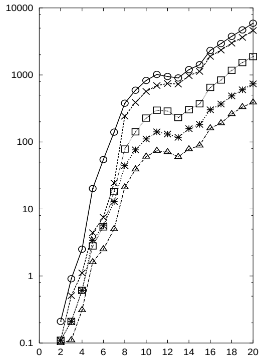


**Fig. 19.** `gas`, prop. 1, saf.

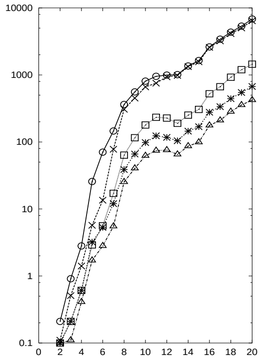


**Fig. 20.** `gas`, prop. 3, saf.

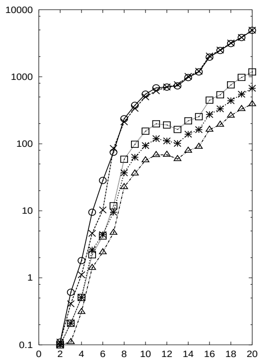


**Fig. 21.** `gas`, prop. 4, saf.

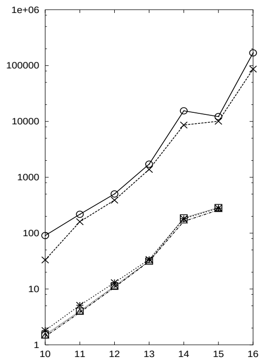


**Fig. 22.** `stack`, prop. 1, weak

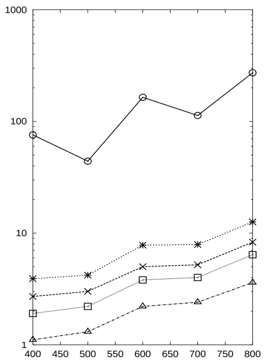


**Fig. 23.** `stack`, prop. 2, weak

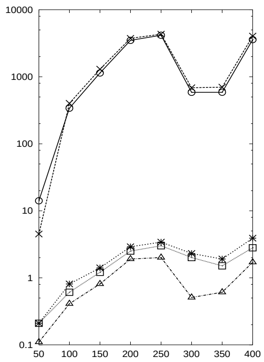


**Fig. 24.** `stack`, prop. 3, saf.

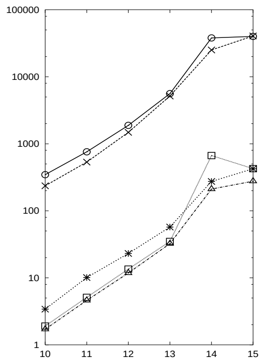


**Fig. 25.** `stack`, prop. 4, weak

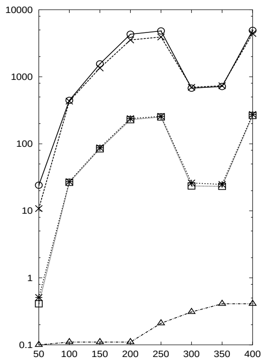


**Fig. 26.** `stack`, prop. 5, saf.

we built a new compiler specialized for this kind of properties (Boolean combination of *GF* formulas). The resulting automaton has 1281 states. We checked this property on the leader election algorithm LCR, cf. [23]. We instantiated the propositions in order to make the property true in one case, false in another. The results are plotted in Figs. 27-28. Note that the pattern is the same as in the previous results. More importantly, partitioning does not seem to be affected by the number of partitions. Notice that the logarithmic encoding pays an initial overhead for encoding symbolically the automa-

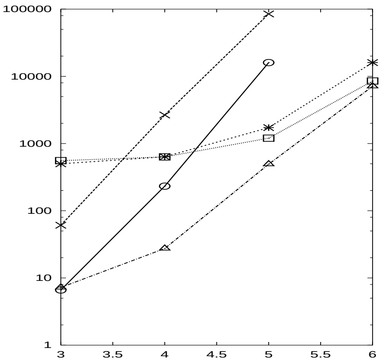



**Fig. 27.** LCR, large prop. (true), EL

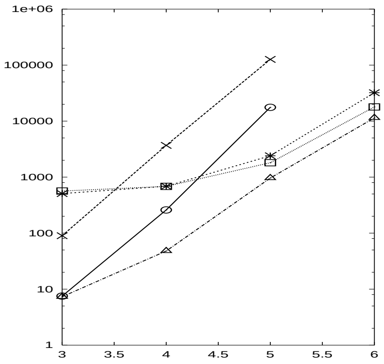



**Fig. 28.** LCR, large prop. (false), EL

ton. However, as the size of the system grows, this technique outperforms syntactic compilation.

## 5 Conclusions

The main finding of this work is that hybrid approaches to LTL symbolic model checking outperform pure symbolic model checking. Thus, a uniform treatment of the system under verification and the property to be verified is not desirable. We believe that this finding calls for further research into the algorithmics of LTL symbolic model checking. The main focus of research in this area has been either on the implementation of BDD operations, cf. [24], or on symbolic algorithms for FTS emptiness, cf. [27], ignoring the distinction between system and property. While ignoring this distinction allows for simpler algorithms, it comes with a significant performance penalty.

## References

1. A. Biere, C. Artho, and V. Schuppan. Liveness Checking as Safety Checking. *Electr. Notes Theor. Comput. Sci.*, 66(2), 2002.
2. A. Biere, E. M. Clarke, and Y. Zhu. Multiple State and Single State Tableaux for Combining Local and Global Model Checking. In *Correct System Design*, pp 163–179, 1999.
3. R. Bloem, K. Ravi, and F. Somenzi. Efficient Decision Procedures for Model Checking of Linear Time Logic Properties. In *CAV*, pp 222–235, 1999.
4. R. K. Brayton, G. D. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S. T. Cheng, S. Edwards, S. Khatri, Y. 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*, pp 428–432, 1996.
5. R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.
6. 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, 1992.
7. A. Cimatti, E.M. Clarke, F. Giunchiglia, and M. Roveri. NUSMV: a new Symbolic Model Verifier. In *CAV*, pp 495 – 499, 1999.

8. A. Cimatti, M. Roveri, and P. Bertoli. Searching Powerset Automata by Combining Explicit-State and Symbolic Model Checking. In *TACAS*, pp 313–327, 2001.
9. E. M. Clarke, O. Grumberg, and K. Hamaguchi. Another Look at LTL Model Checking. *Formal Methods in System Design*, 10(1):47–71, 1997.
10. 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(2/3):275–288, 1992.
11. N. Daniele, F. Guinchiglia, and M.Y. Vardi. Improved automata generation for linear temporal logic. In *CAV*, pp 249–260, 1999.
12. E.A. Emerson and C.L. Lei. Efficient Model Checking in Fragments of the Propositional $\mu$–Calculus. In *LICS*, pp 267–278, 1986.
13. K. Etessami and G. J. Holzmann. Optimizing Büchi Automata. In *CONCUR*, pp 153–167, 2000.
14. K. Etessami, T. Wilke, and R. A. Schuller. Fair Simulation Relations, Parity Games, and State Space Reduction for Büchi Automata. In *ICALP*, pp 694–707, 2001.
15. K. Fisler, R. Fraer, G. Kamhi, M.Y. Vardi, and Z. Yang. Is there a best symbolic cycle-detection algorithm? In *TACAS*, pp 420–434, 2001.
16. R. Fraer, G. Kamhi, B. Ziv, M. Y. Vardi, and L. Fix. Prioritized Traversal: Efficient Reachability Analysis for Verification and Falsification. In *CAV*, pp 389–402, 2000.
17. C. Fritz and T. Wilke. State Space Reductions for Alternating Büchi Automata. In *FSTTCS*, pp 157–168, 2002.
18. P. Gastin and D. Oddoux. Fast LTL to Büchi Automata Translation. In *CAV*, pp 53–65, 2001.
19. D. Giannakopoulou and F. Lerda. From States to Transitions: Improving Translation of LTL Formulae to Büchi Automata. In *FORTE*, pp 308–326, 2002.
20. P. Godefroid and G. J. Holzmann. On the Verification of Temporal Properties. In *PSTV*, pp 109–124, 1993.
21. S. Gurumurthy, R. Bloem, and F. Somenzi. Fair Simulation Minimization. In *CAV*, pp 610–624, 2002.
22. G.J. Holzmann. *The SPIN model checker: Primer and reference manual*. Addison Wesley, 2003.
23. N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., 1996.
24. H. Ochi, K. Yasuoka, and S. Yajima. Breadth-first manipulation of very large binary-decision diagrams. In *ICCAD*, pp 48–55, 1993.
25. Doron Peled. Combining Partial Order Reductions with On-the-fly Model-Checking. In *CAV*, pp 377–390, 1994.
26. A. Pnueli. The temporal logic of programs. In *FOCS*, pp 46–57, 1977.
27. K. Ravi, R. Bloem, and F. Somenzi. A Comparative Study of Symbolic Algorithms for the Computation of Fair Cycles. In *FMCAD*, pp 143–160, 2000.
28. R. Sebastiani, E. Singerman, S. Tonetta, and M. Y. Vardi. GSTE is Partitioned Model Checking. In *CAV*, pp 229–241, 2004.
29. R. Sebastiani and S. Tonetta. "More Deterministic" vs. "Smaller" Büchi Automata for Efficient LTL Model Checking. In *CHARME*, pp 126–140, 2003.
30. F. Somenzi and R. Bloem. Efficient Büchi Automata from LTL Formulae. In *CAV*, pp 247–263, 2000.
31. A. Valmari. Error Detection by Reduced Reachability Graph Generation. In *ATPN*, 1988.
32. M.Y. Vardi and P. Wolper. An Automata-Theoretic Approach to Automatic Program Verification. In *LICS*, pp 332–344, 1986.
33. M.Y. Vardi and P. Wolper. Reasoning about Infinite Computations. *Information and Computation*, 115(1):1–37, 1994.
34. J. Yang and C.-J.H. Seger. Generalized Symbolic Trajectory Evaluation - Abstraction in Action. In *FMCAD*, pp 70–87, 2002.