Int J Softw Tools Technol Transfer (2011) 13:319–335
DOI 10.1007/s10009-010-0168-4

REGULAR PAPER

Symbolic systems, explicit properties: on hybrid approaches for LTL symbolic model checking

Roberto Sebastiani · Stefano Tonetta · Moshe Y. Vardi

Published online: 16 September 2010
© Springer-Verlag 2010

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 benefit 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.

S. Tonetta was partly supported by the Provincia Autonoma di Trento (project ANACONDA).

R. Sebastiani
DISI, Università di Trento, Trento, Italy
e-mail: rseba@disi.unitn.it

S. Tonetta (✉)
Embedded Systems Unit, Fondazione Bruno Kessler - IRST, Trento, Italy
e-mail: tonettas@fbk.eu

M. Y. Vardi
Dept. of Computer Science, Rice University, Houston, USA
e-mail: vardi@cs.rice.edu

Keywords Linear-time logic · Symbolic model checking · Property-driven partitioning

1 Introduction

Linear-temporal logic (LTL) [30] 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). The key idea of the automata-theoretic approach to MC is that LTL formulas can be compiled into equivalent automata with fairness conditions, i.e. conditions on which infinite words are accepted. Standard techniques consider the formula φ that is the negation of the desired behavior and construct a Generalized Büchi automaton (GBA) A_φ with the same language. Then, they compute the product of this automaton A_φ with the system M 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.

Explicit-state model checking uses highly optimized LTL-to-GBA compilation, cf. [14,15,17,20,21,23,24,32,35], 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 either a nested depth-first search [13,33] or an optimized decomposition into strongly connected components [11,25]. To deal with the state-explosion problem, various state-space reductions are used, e.g. [29,37].

Symbolic model checking (SMC) [2] 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) [16,18,31]. For "weak" property automata, the doubly-nested fixpoint algorithm can be replaced by a singly-nested fixpoint algorithm [6]. 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 [26], or a pure symbolic approach, e.g. in NuSMV[8]. 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 [6] is based on an explicit construction of the property automaton. (See [3,12] for other hybrid approaches.)

In [34], motivated by previous work on *generalized symbolic trajectory evaluation* (GSTE) [40], we proposed a hybrid approach to LTL model checking, referred to as *property-driven partitioning* (PDP). In this approach, the property automaton A_φ 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 [34] that PDP is superior to SMC, in many cases demonstrating exponentially better scalability.

While the results in [34] 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. 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. 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 [22].

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 [6] 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 [15,32], postprocessing state minimization by means of simulations [15,17,21,32], and midprocessing state minimization by means of alternating simulations [20,24]. 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.

This paper extends the work presented in [36] by giving:

- a more detailed description of PDP and its relationship with search algorithms; in particular, the algorithm that reduces liveness to safety is described in details both for the explicit-state case and for property-driven partitioning.
- a deeper analysis of the results; in particular, the comparison of the different search algorithms is shown and discussed.

The outline of the paper is as follows. 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, Sect. 5 contains some concluding remarks.

2 Background

2.1 Properties and systems

We use LTL with its standard syntax and semantics [30] to specify properties. Let $Prop$ be a set of propositions, which refer to important facts of the systems under analysis. A propositional literal (i.e., a proposition p in $Prop$ or its negation $\neg p$) is a LTL formula; if φ_1 and φ_2 are LTL formulas, then $\neg\varphi_1$, $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$, $X\varphi_1$, $\varphi_1 U \varphi_2$, $G\varphi_1$, and $F\varphi_1$ are LTL formulae, where X, U, G, and F are the standard "next", "until", "globally", and "eventually" temporal operators respectively. We refer the reader to [10] for a formal definition of the semantics.

We describe the systems under analysis with Fair Transition Systems (FTSs). We take Σ to be equal to 2^{Prop}. An 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).

2.2 Explicit-state LTL model checking

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, $\delta \subseteq \mathcal{B} \times \Sigma \times \mathcal{B}$ is the transition relation, and $\mathcal{F}^{\mathcal{B}} \subseteq 2^{\mathcal{B}}$ is the set of fairness constraints.

The product between an FTS M and a GBA A 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 φ of a property into a GBA A^{φ} and checking the emptiness of the product P between the FTS M and A^{φ} [38]. In explicit-state model checking, emptiness checking is performed by state enumeration: a Depth-First Search (DFS) can detect if there exists a fair strongly-connected component reachable from the initial states [13].

2.3 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 $\mathcal{S} = 2^V$, 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 every $a \in \Sigma$ 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 $M^1 = \langle \mathcal{S}^1, \mathcal{S}_0^1, T^1, \Sigma, \mathcal{L}^1, \mathcal{F}^1 \rangle$ with $\mathcal{S}^1 = 2^{V^1}$ and $M^2 = \langle \mathcal{S}^2, \mathcal{S}_0^2, T^2, \Sigma, \mathcal{L}^2, \mathcal{F}^2 \rangle$ with $\mathcal{S}^2 = 2^{V^2}$, the synchronous composition of M^1 and M^2 is the FTS $\langle \mathcal{S}^{\mathcal{P}}, \mathcal{S}_0^{\mathcal{P}}, T^{\mathcal{P}}, \Sigma, \mathcal{L}^{\mathcal{P}}, \mathcal{F}^{\mathcal{P}} \rangle$, where

- $\mathcal{S}^{\mathcal{P}} = 2^{V^{\mathcal{P}}}$, $V^{\mathcal{P}} = V^1 \cup V^2$,
- $\mathcal{S}_0^{\mathcal{P}}(v_1, v_2) = \mathcal{S}_0^1(v_1) \wedge \mathcal{S}_0^2(v_2)$,
- $T^{\mathcal{P}}(v_1, v_2, v_1', v_2') = T^1(v_1, v_1') \wedge T^2(v_2, v_2')$
- $\mathcal{L}^{\mathcal{P}}(v_1, v_2) = \mathcal{L}^1(v_1) \wedge \mathcal{L}^2(v_2)$,
- $\mathcal{F}^{\mathcal{P}} = \mathcal{F}^1 \cup \mathcal{F}^2$.

Again, the negation φ 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 $\langle \mathcal{S}^{\varphi}, \mathcal{S}_0^{\varphi}, T^{\varphi}, \Sigma, \mathcal{L}^{\varphi}, \mathcal{F}^{\varphi} \rangle$, where $\mathcal{S}^{\varphi} = 2^{V^{\varphi}}$, $V^{\varphi} = Atoms(\varphi) \cup Extra(\varphi)$, so that $Atoms(\varphi) \subseteq Prop$ are the propositions that occur in φ, $Extra(\varphi) \cap V = \emptyset$ and $Extra(\varphi)$ contains one variable for every temporal connective occurring in φ [2,9,39]. We call this *syntactic compilation*.

To check language containment, a symbolic model checker implements a fixpoint algorithm [2]. 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:

$$\texttt{PreImage}(Q) = \exists V'((Q[V'/V])(V') \wedge T(V, V'))$$
$$\texttt{PostImage}(Q) = (\exists V(Q(V) \wedge T(V, V')))[V/V']$$

The most used representation for predicates on Boolean variables are Binary Decision Diagrams (BDDs) [7]. 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.

2.4 Emerson–Lei algorithm

In a symbolic approach, the application of a DFS, as in explicit-state model checking, is not efficient. In fact, the effectiveness of symbolic algorithms relies on handling sets

```
1  EL(Q)
2  begin
3      Z := Q;
4      repeat
5          Z' := Z;
6          for F ∈ 𝓕 do
7              Y := EU(Z, F ∩ Z);
8              Z := Z ∩ PreImage(Y);
9      until Z' = Z ;
10     return Z
11 end

12 EU(Q1, Q2)
13 begin
14     R := Q2;
15     repeat
16         Z := Q1 ∩ PreImage(R);
17         R := R ∪ Z;
18     until Z = R ;
19     return R
20 end
```

Fig. 1 Emerson–Lei algorithm

of states. The standard approach in SMC is the one of Emerson–Lei (EL) [16].

The algorithm of [16] checks the emptiness of the product by computing the set of states which can be extended to a fair path. These states are called *fair states*. EL computes the set of fair states with a doubly-nested fixpoint computation, as the one shown in Fig. 1. The outer fixpoint keeps an approximation Z of the set. At every iteration it refines the approximation by restricting the set to the states that can be extended to a fair path lying in Z. This is computed with the inner fixpoint.

2.5 Alternatives to EL

Symbolic model checkers typically compute the fair states by means of some variant of EL [18,31]. There are interesting alternatives that try to solve the emptiness problem in a more efficient way.

2.5.1 Weak automata

For "weak" property automata, the doubly-nested fixpoint algorithm can be replaced by a singly-nested fixpoint [6]. This is the case of safety properties, where we can identify a set B of "bad" states, and a violation of the property is just an initial path that reaches one of these states. Then, we can check the language emptiness by searching for a path reaching B. In other cases, the fairness conditions partition the Strongly Connected Components (SCCs) in two groups: for every fairness condition F, either an SCC is contained in F or it does not intersect F. Then, if B is the set of states of the accepting SCCs, we can check the language emptiness by looking for a cycle in B reachable from the initial states. This requires just a single fixpoint computation. The drawback of this approach is that not all LTL formulas can be translated into "weak" automata. We refer to this technique as w/s.

2.5.2 Liveness to safety

An alternative algorithm [1] reduces emptiness checking to reachability checking (which requires a singly-nested fixpoint computation). Intuitively, the *liveness-to-safety* algorithm (l2s) applies a forward search and it non-deterministically saves a visited state: the emptiness is violated if the search reaches a state that was previously saved. To assure that the loop is fair, one has to add another symbolic variable for every fairness constraint. Formally, if the product machine is the FSM $M = \langle \mathcal{S}, \mathcal{S}_0, T, \Sigma, \mathcal{L}, \mathcal{F} \rangle$ with $\mathcal{S} = 2^V$, l2s produces the FSM $M' = \langle \mathcal{S}', \mathcal{S}'_0, T', \Sigma, \mathcal{L}', \mathcal{F}' \rangle$ where

- $\mathcal{S}' = 2^{V'}$ with $V' = V \cup V_c \cup \{save, saved, looped\} \cup V_{\mathcal{F}}$, where
  - V_c contains one (copy) variable v_c for every variable v in V;
  - $save$, $saved$, and $looped$ are new (fresh) variables;
  - $V_{\mathcal{F}}$ contains one variable v_F for every F in $\mathcal{F}$;
- $\mathcal{S}'_0 := \mathcal{S}_0 \wedge \neg saved \wedge \bigwedge_{F \in \mathcal{F}} \neg v_F$
- $T'(v, v_c, save, saved, looped, v_F, v', v'_c, save', saved', looped', v'_F) = T(v, v') \wedge v'_c \leftrightarrow ((save \wedge !saved \wedge v) \vee v_c) \wedge saved' \leftrightarrow (save \vee saved) \wedge v'_F \leftrightarrow (v_F \vee ((save \vee saved) \wedge F)) \wedge looped \rightarrow (saved \wedge \bigwedge_{F \in \mathcal{F}} v_F \wedge \bigwedge_{v \in V} (v \leftrightarrow v_c))$;
- $\mathcal{L}'(v, v_c, save, saved, v_F) = \mathcal{L}'(v)$;
- $\mathcal{F}' = \{F'\}$ and $F' = looped$

In other words, the value of $save$ is always nondeterministic; once $save$ becomes true, we set $saved$ to true and we store the current state of S, i.e. we copy the value of each v in v_c; from that point, we set v_F to true when we visit F; finally, $looped$ is true if, after visiting every fair condition, we loop back to the stored state. Thus, we can reach F' in S' if and only if there exists a reachable fair loop in S (for a formal proof, see [1]).

The drawback of this technique is that it requires to double the number of symbolic variables.

3 Hybrid approaches

3.1 Symbolic systems, explicit-state property automata

Between the explicit-state and the symbolic approach described in Sect. 2, *hybrid* approaches represent the system

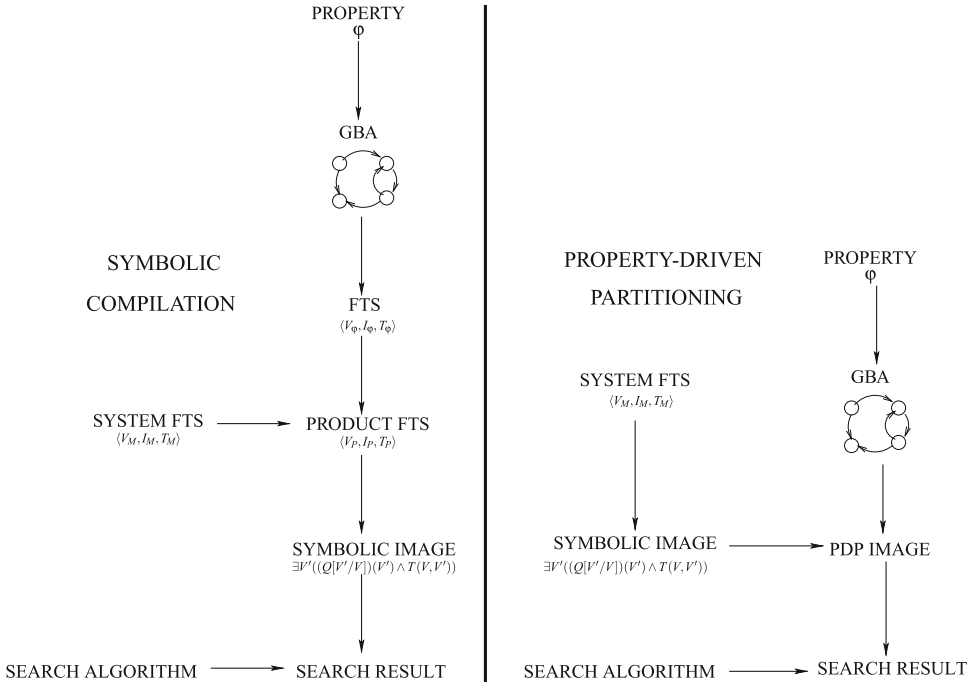

Fig. 2 Hybrid approaches' flow

symbolically and the property automaton explicitly. Thus, they semantically compile the LTL formula into a GBA. We identify two main classes of hybrid approaches. In the first class, the explicit-state automaton is encoded into an FTS, the synchronous product between the resulting FTS with the FTS of the system is built, and finally the product state space is searched for an accepting fair path by applying an algorithm based on a fully symbolic image computation. In the second class, the search is performed on an implicit representation of the product state space, but the algorithm adopts an hybrid image computation that combines the symbolic image computation of the system with the explicit list of successors of the GBA.

The flow of the two approaches is depicted in Fig. 2. 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.

3.2 Symbolic compilation of explicit-state property automaton

Given the GBA $A = \langle \mathcal{B}, b_0, \Sigma, \delta, \mathcal{F} \rangle$ corresponding to the formula φ, we can compile A into the FTS $\langle \mathcal{S}^A, \mathcal{S}_0^A, T^A, \Sigma, \mathcal{L}^A, \mathcal{F}^A \rangle$, where $\mathcal{S}^A = 2^{V^A}$, $V^A = Extra(A) \cup Atoms(\varphi)$, $Extra(A) \cap Prop = \emptyset$ and $|Extra(A)| = log(|\mathcal{B}|)$, $\mathcal{S}_0^A$ represents $\{b_0\}$, $T^A(s, a, s', a')$ is true iff $(s, a, s') \in \delta$, $\mathcal{L}^A(s, a) = a$ and finally every $F^A \in \mathcal{F}^A$ 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*.

3.3 Property-driven partitioning (PDP)

Given an FTS M and a GBA A, 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 M : $\{Q_b\}_{b\in\mathcal{B}}$, 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:

$$\begin{gathered} Q^1 \wedge Q^2 := \{Q_b^1 \wedge Q_b^2\}_{b\in\mathcal{B}} \\ Q^1 \vee Q^2 := \{Q_b^1 \vee Q_b^2\}_{b\in\mathcal{B}} \\ \neg Q := \{\neg Q_b\}_{b\in\mathcal{B}} \\ \texttt{PdpPreImage}(Q) := \{\bigvee_{(b,a,b')\in\delta} \texttt{PreImage}(Q_{b'}) \wedge a\}_{b\in\mathcal{B}} \\ \texttt{PdpPostImage}(Q) := \{\bigvee_{(b',a,b)\in\delta} \texttt{PostImage}(Q_{b'} \wedge a)\}_{b\in\mathcal{B}} \end{gathered} \tag{1}$$

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 [34]).

3.4 PDP version of EL

The PDP version of the EL algorithm is shown in Fig. 3. The difference with the non-partitioned versions is that while `EU` and `EL` operate on a single set of states in the product automaton, `PdpEU` and `PdpEL` operate on an array of sets of states of the system (one set for every vertex of the GBA). Thus, every variable in the algorithms of Fig. 3 can be considered as an array of propositional formulas, implemented as BDDs. The

```
1  begin PdpEL(Q)
2      Z' := Q ;                                   // for all b ∈ B, Z'_b := Q_b
3      repeat
4          Z' := Z ;                               // for all b ∈ B, Z_b := Z'_b
5          for F ∈ F do
6              G :=PdpRestrict(S,F) ;              // for all b ∈ B,
                                                   // if b ∈ F R_b := S, else R_b := ∅
7              Y :=PdpEU(Z,G∩Z);
8              Z := Z∩PdpPreImage(Y) ;              // see Eq. 1
9      until Z' = Z ;
10     return Z
11 end

12 PdpEU(Q1,Q2)
13 begin
14     R := Q2 ;                                   // for all b ∈ B, R_b := Q2b
15     repeat
16         Z := Q1∩PdpPreImage(R) ;                 // see Eq. 1
17         R := R∪Z ;                              // for all b ∈ B, R_b := R_b ∨ Z_b
18     until Z = R ;
19     return R
20 end
```

Fig. 3 Partitioned version of EL

side comments explain the operation performed on the single BDD. The backward image of an array is given by Eq. 1.

3.5 Explicit-state and hybrid versions of l2s

The combination of PDP with l2s is not straightforward. Thus, we dedicate this section to show how we solved it. First, we have to conceive an automata-theoretic version of l2s. We used this technique to convert the property automaton into an equivalent automaton on finite words. Second, we must combine this explicitly-represented automaton with the system, for which we use the standard symbolic version of l2s.

Recall that l2s doubles the number of symbolic variables in order to reduce emptiness to reachability (see Sect. 2.5). 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.

If the property automaton is the GBA $A = \langle \mathcal{B}, b_0, \Sigma, \delta, \{F\}\rangle$, we can find an automaton A' on finite words, which is empty (i.e. has empty language) if and only if A is empty. To this purpose, we apply the l2s idea straighforwardly to the GBA A. As l2s uses a linear number of additional variables to non-deterministically store the current state, we similarly add a copy of A for every state $b \in \mathcal{B}$ (when a run passes to the copy corresponding to b, we virtually store b as the state to reach again to find a loop). Since, we are considering GBAs with only one fair condition F, we can restrict the number of copies to $|F|$ (i.e., we store a state only if it is in F). Formally, we define A' as the following automaton $A' = \langle \mathcal{B} \times (\{0\} \cup F), (b_0, 0), \Sigma, \delta', \{F'\}\rangle$ where δ' and $F' \subseteq \mathcal{B} \times \mathcal{B}$ are defined as follows:

- $\delta' = \{((b,0), a, (b',0)) | (b,a,b') \in \delta\} \cup$
  $\{((b,0), a, (b',b)) | (b,a,b') \in \delta \text{ and } b \in F\} \cup$
  $\{((b,b''), a, (b',b'')) | (b,a,b') \in \delta \text{ and } b'' \in F\}$
- $F' = \bigcup_{b \in F} (b,b)$

In other words, there are $|F| + 1$ copies of A: initially, we start from the copy "0"; from a state b in F, we can pass to the copy "b"; when we reach b in the copy "b", we find an accepting loop of A.

The hybrid version of l2s combines the standard symbolic version of l2s applied to the system (doubling the number of variables) and the above explicit version of l2s applied to the property automaton (considering a linear number of copies). To this purpose, we must modify slightly the construction of A' in order to synchronize the property automaton with the system generated by l2s: the move from the copy "0" to the copy "b" must happen when the system saves the current state in order to find a fair loop (see Sect. 2.5); we must reach F when the system loops back to the saved state. Formally, we build the automaton $A'' = \langle \mathcal{B} \times (\{0\} \cup F) \cup F \times \{1\}, (b_0, 0), \Sigma \cup \{save, saved, looped\}, \delta'', \{F \times \{1\}\}\rangle$ where δ'' is defined as follows:

- $\delta'' = \{((b,0), a, (b',0)) | (b,a,b') \in \delta\} \cup$
  $\{((b,0), a \wedge save \wedge \neg saved, (b',b)) | (b,a,b') \in \delta \text{ and } b \in F\} \cup$
  $\{((b,b''), a \wedge saved, (b',b'')) | (b,a,b') \in \delta \text{ and } b'' \in F\} \cup$
  $\{((b,b''), a \wedge looped, (b',1)) | (b,a,b') \in \delta \text{ and } b'' = b'\}$

This automaton is used to guide the partitioning of the product with the system.

3.6 Hypothesis

Our hypothesis is that hybrid approaches combine the best features of explicit-state and symbolic model checking techniques. On the 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 [15,32]; postprocessing minimization by means of simulations [15,17,21,32], and midprocessing minimization by means of alternating simulation [20,24].

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. [19]. Furthermore, PDP enables traversing the product state space without computing it explicitly. The experiments reported in the next section test our hypothesis.

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 M, the LTL property φ and the model-checking algorithm used.

4.1 Verification models

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`. In this system, customers are symmetric. For this reason, we refer only to the first and second customer in the properties, even if there are actually N customers.

We consider four properties for the `gas` system. `gas.prop1` states that, if the first customer started the pump and has not yet finished when the second customer prepays, then the second customer must be the next served. `gas.prop2` states the same property of `gas.prop1`, but with the assumption that whenever the first customer started the pump, she does not do a second prepayment before receiving the change of the first prepayment. `gas.prop3` states that, if the first customer prepays first, then she will be served first. `gas.prop4` states that whenever the first customer starts the pump, the second customer must wait until the first finished.

The second system is a model of a stack with the standard pop and push functions, a function to process the elements of the stack in a top-down order, and a function to check if the stack is empty. In this case, scalability is given by the maximum size N of the stack.

We consider five properties for the `stack` system. `stack.prop1` states that, if we push some data $d1$ and call the top-down processing function before removing the $d1$, then we will surely process $d1$. `stack.prop2` states that, if we push some data and call the empty function before any pop call, then the function will say that the stack is not empty. `stack.prop3` states that between a push call and a empty call returning that the stack is empty, there must be a pop call. `stack.prop4` states that, if we push $d1$ and later we push $d2$ and later we call the top-down function, and in the meanwhile $d1$ and $d2$ are not popped, then the function will process $d2$ and later $d1$. `stack.prop5` states that, if we push $d1$ and later we push $d2$, and in the meanwhile $d1$ is not popped, then $d2$ must be popped before $d1$.

The properties of these two systems are displayed in Table 1.

4.2 LTL to Büchi automata conversion

In this section, we focus the attention on the compilation of LTL formulas into GBAs. For syntactic compilation, we used `ltl2smv`, distributed together with NuSMV. As for semantic compilation, we used MODELLA, which uses also some techniques described in [15,17,23,32]. In Table 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. Degeneralization 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).

4.3 Model checking algorithms

We investigated three different MC algorithms for emptiness checking (see Sect. 2):

- **EL:** the first model checking algorithm we used is the classic Emerson–Lei algorithm [16], which computes the set of fair states;
- **l2s:** the second algorithm is the reduction of liveness checking to safety checking [1];
- **w/s:** the third technique consists of checking if the automaton is simple enough to apply a single fixpoint compu-

Table 1 The LTL properties of the `gas` and the `stack` systems

Name	Formula	Type
`gas.prop1`	$G((pump\_started1 \wedge ((!pump\_charged1)\ U\ operator\_prepaid\_2)) \rightarrow ((!operator\_activate\_1)\ U\ (operator\_activate\_2 \mid G!operator\_activate\_1)))$	Terminal
`gas.prop2`	$(G(pump\_started1 \rightarrow ((!operator\_prepaid\_1)\ U\ (operator\_change\_1 \mid G!operator\_prepaid\_1)))) \rightarrow (G((pump\_started1 \wedge ((!pump\_charged1)\ U\ operator\_prepaid\_2)) \rightarrow ((!operator\_activate\_1)\ U\ (operator\_activate\_2 \mid G!operator\_activate\_1)))$	Neither
`gas.prop3`	$((!operator\_prepaid\_2)\ U\ operator\_prepaid\_1) \rightarrow (!pump\_started2\ U\ (pump\_started1 \mid G(!pump\_started2)))$	Terminal
`gas.prop4`	$G(pump\_started1 \rightarrow ((!pump\_started2)\ U\ (pump\_charged1 \mid G(!pump\_started2))))$	Terminal
`stack.prop1`	$G(callPushd1 \wedge ((!returnPopd1)\ U\ callTop\_Down) \rightarrow F(callTop\_Down \wedge F(callProcessd1)))$	Weak
`stack.prop2`	$G((callPush \wedge (!returnPop\ U\ callEmpty)) \rightarrow F(callEmpty \wedge F(returnEmptyFalse)))$	Weak
`stack.prop3`	$G((callPushd1 \wedge F(returnEmptyTrue)) \rightarrow (!returnEmptyTrue\ U\ returnPopd1))$	Terminal
`stack.prop4`	$G((callPushd1 \wedge ((!returnPopd1)\ U\ (callPushd2 \wedge ((!returnPopd1 \wedge !returnPopd2)\ U\ callTop\_Down)))) \rightarrow F(callTop\_Down \wedge F(callProcessd2 \wedge FcallProcessd1)))$	Weak
`stack.prop5`	$G((callPushd1 \wedge (!returnPopd1\ U\ callPushd2)) \rightarrow (!returnPopd1\ U\ (!returnPopd2 \mid G!returnPopd1)))$	Terminal

First column shows the name, the second column the LTL formula, and the third column the type of the corresponding automaton

Table 2 Number of states, number of (extra) symbolic variables, and number of fairness conditions of the automata corresponding to the LTL properties of `gas` and the `stack` systems, obtained with `ltl2smv` and MODELLA

Property	ltl2smv		Modella		
	Extra variables	Fairness constraints	States	Extra variables ($\lceil \log(\text{states}) \rceil$)	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

tation in order to find a fair loop [6]; 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.

4.4 Tested 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 (see, for

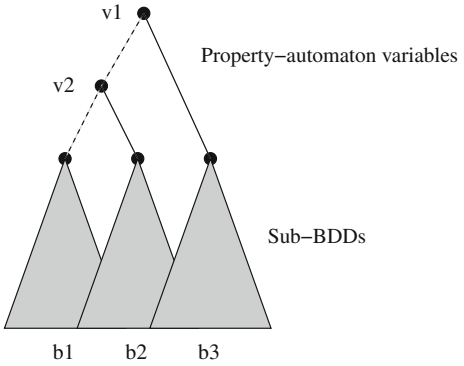

Fig. 4 Example of BDD with the top-ordering of property-automaton variables

example, the data in Table 2 where the semantic compiler MODELLA is compared to the syntactic compiler `ltl2smv`). 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 top-ordering 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 of states of the product of the system with the property automaton. Let us consider an assignment to the symbolic variables of the property automaton corresponding to the state b_1. If you follow this assignment in the structure of d, you find a sub-BDD, which corresponds to the set Q_{b_1} of system's states that are paired with b in the product (see Fig. 4). 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 (note that although this is a partitioning of the product state space, the sub-BDDs may share some states of the system).

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 [34]).

4.5 Results

We used NuSMV as platform to perform out tests. We run NuSMV on the Rice Terascale Cluster (RTC) [1], a TeraFLOP Linux cluster based on Intel Itanium 2 Processors. A timeout has been fixed to 72 h for each run. The execution time (in seconds) has been plotted in log scale against the size of the system (i.e., the number of customers in the `gas` system and the size of the stack in the `stack` system). The results are shown in Figs. 5–30.[2] Every plot corresponds to one system, one property, one model checking algorithm. Figs. 5–13 show the results of EL algorithm. Figs. 14–22 show the results of l2s. Figs. 23–30 show the results of w/s (in these plots, syntactic compilation uses EL).

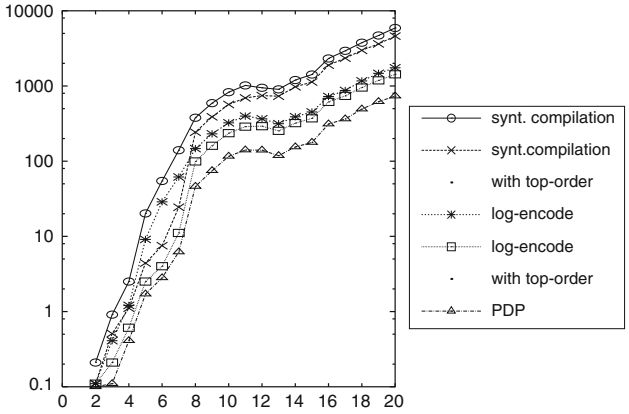

Fig. 5 `gas`, prop. 1, EL

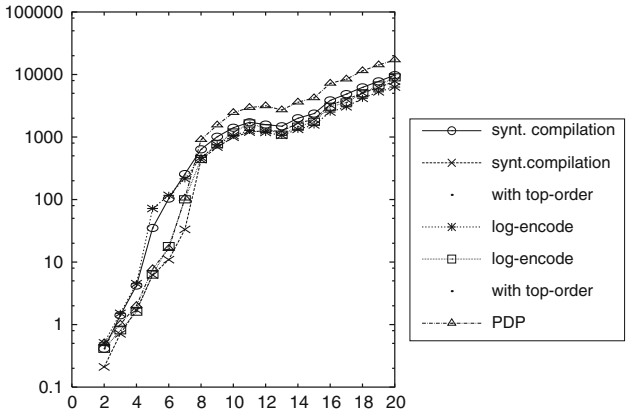

Fig. 6 `gas`, prop. 2, EL

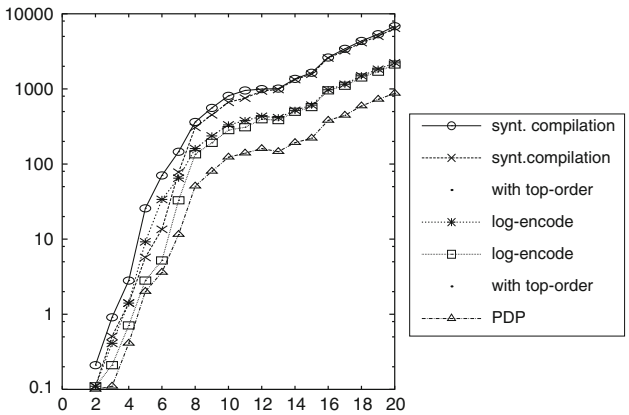

Fig. 7 `gas`, prop. 3, EL

Analyzing the plots, we see that syntactic compilation performs always worse than semantic compilation. This is the case for all experiments but one, the property 2 of the

[1] http://www.citi.rice.edu/rtc/.

[2] All examples, data and tools used in these tests, as well as larger plots, are available at http://www.science.unitn.it/~stonetta/CAV05.

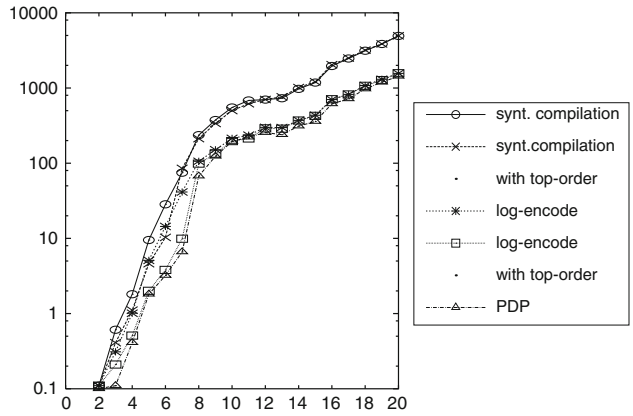

Fig. 8 `gas`, prop. 4, EL

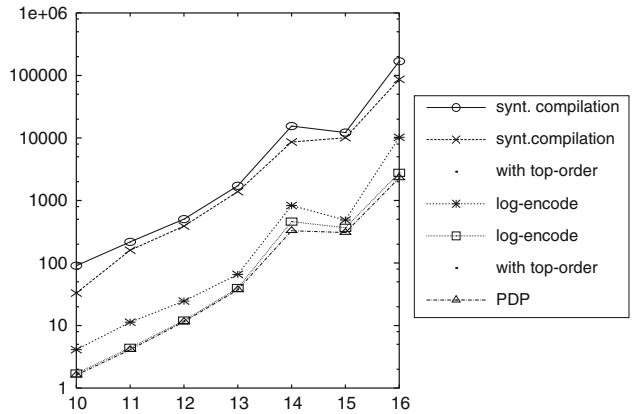

Fig. 9 `stack`, prop. 1, EL

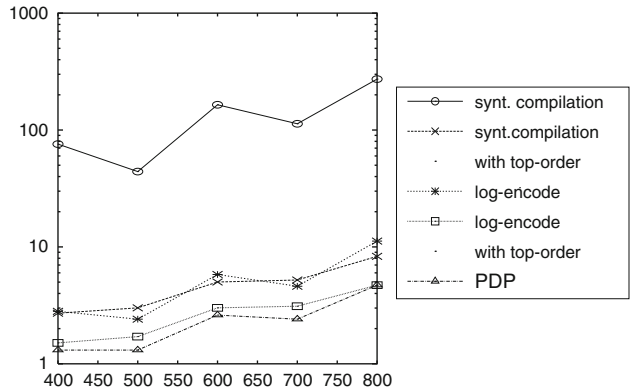

Fig. 10 `stack`, prop. 2, EL

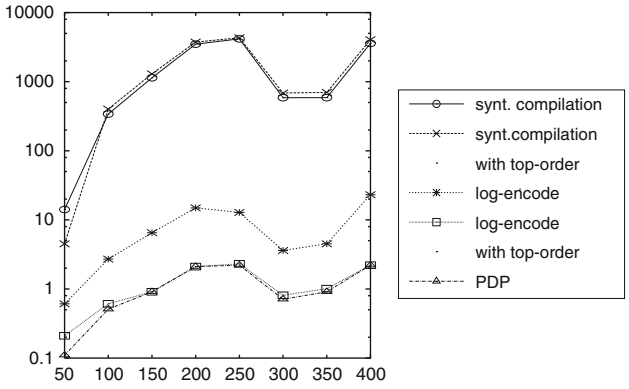

Fig. 11 `stack`, prop. 3, EL

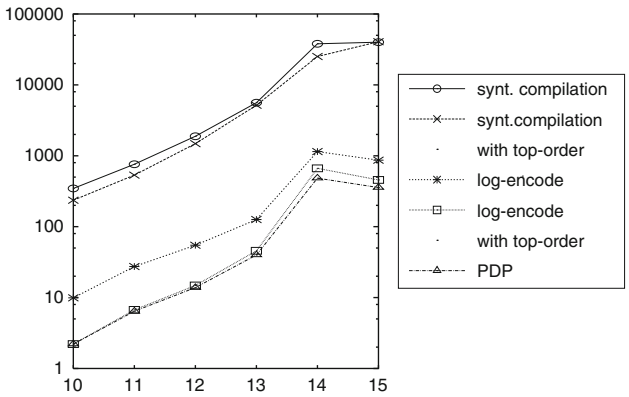

Fig. 12 `stack`, prop. 4, EL

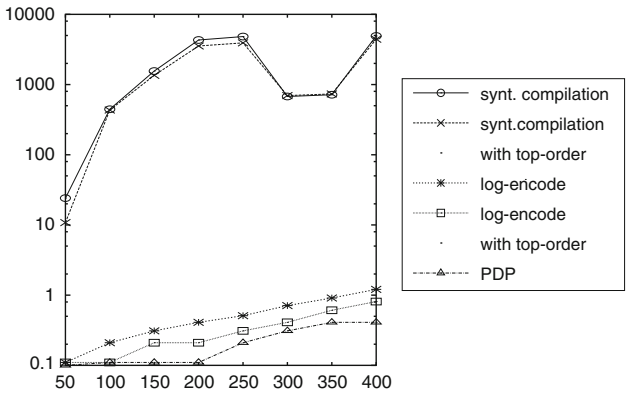

Fig. 13 `stack`, prop. 5, EL

`gas` example with the EL algorithm, where the syntactic compilation is more efficient than PDP, and comparable to `log-encode`. In all other cases, the syntactic compilation is not efficient and is outperformed by the semantic compilation, independently from the system, the property or the search algorithm under consideration. In the case of the `stack` system, the gap is considerable. This suggests that the gain given by the simplification of the property automaton obtained with semantic compilation can be magnified by the complexity of the system state space. For this reason, the improvement can reach three order of magnitude (see Fig. 13).

The `top-order` option typically helps logarithmic encoding, while in the case of syntactic compilation it is less reliable: in some cases (see Figs. 17, 23, 24, and 25), it degrades the performance a lot.

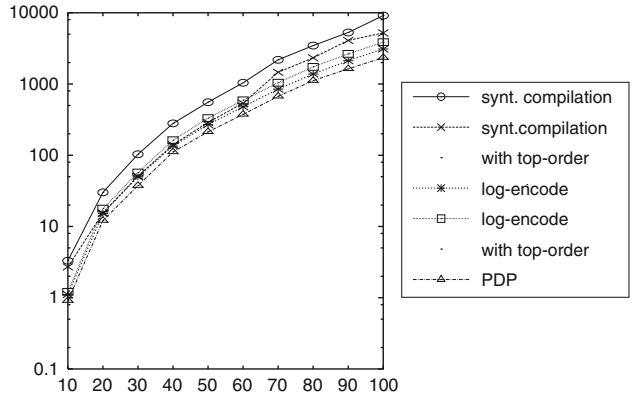

Fig. 14 `gas`, prop. 1, l2s

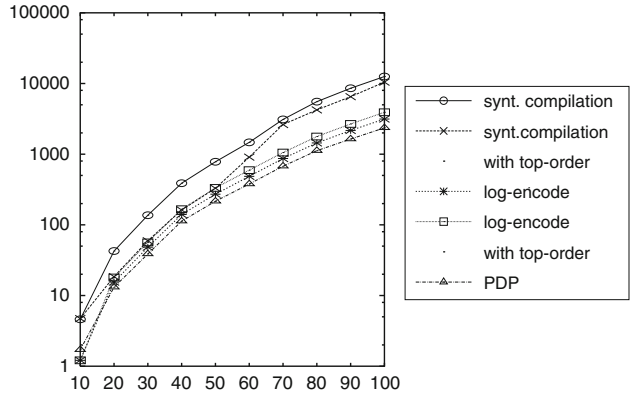

Fig. 15 `gas`, prop. 2, l2s

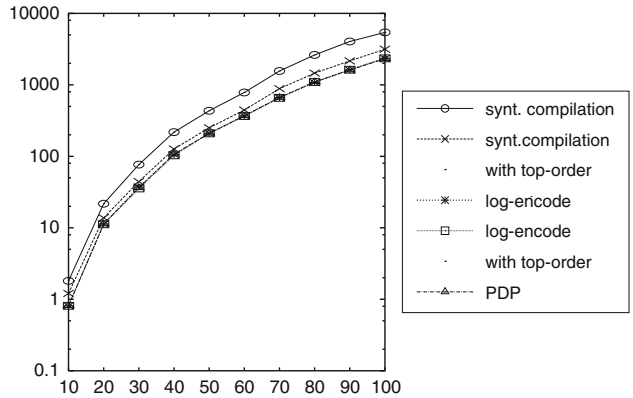

Fig. 16 `gas`, prop. 3, l2s

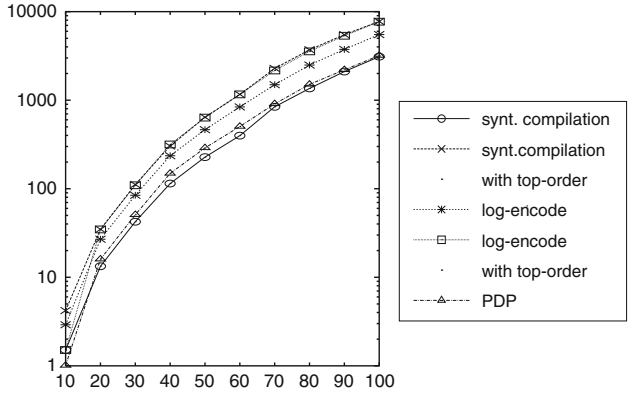

Fig. 17 `gas`, prop. 4, l2s

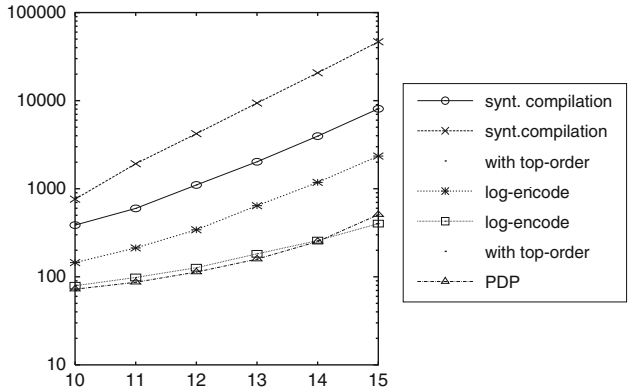

Fig. 18 `stack`, prop. 1, l2s

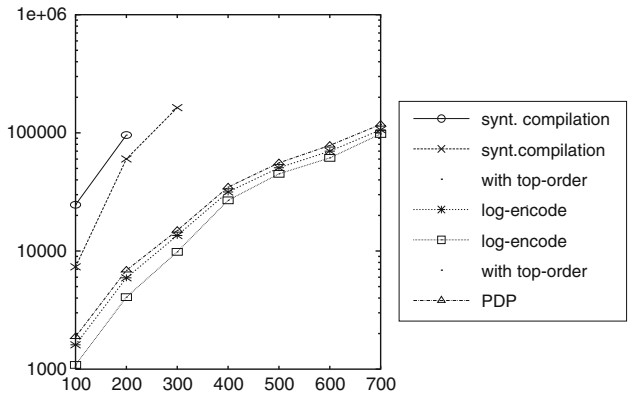

Fig. 19 `stack`, prop. 2, l2s

PDP usually performs better than `log-encode`, even if combined with `top-order`. Therefore, we conclude that 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.

4.6 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 the following

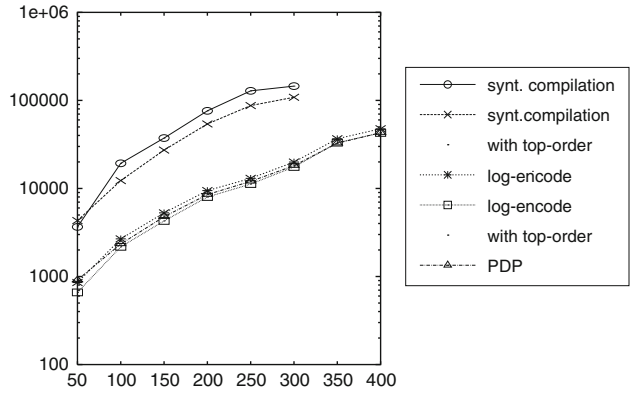

Fig. 20 `stack`, prop. 3, l2s

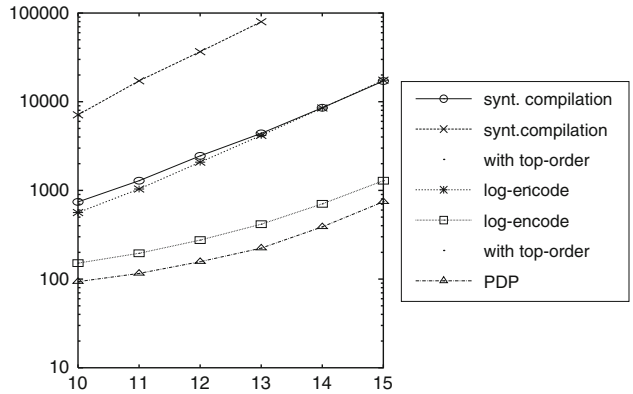

Fig. 21 `stack`, prop. 4, l2s

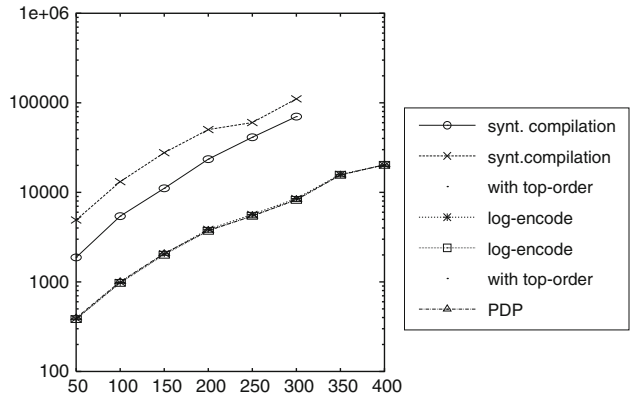

Fig. 22 `stack`, prop. 5, l2s

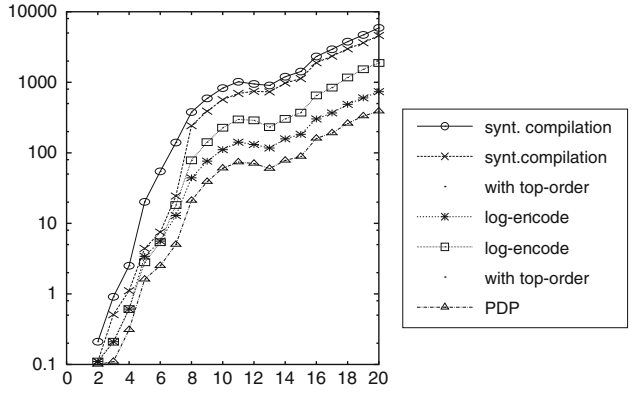

Fig. 23 `gas`, prop. 1, safety

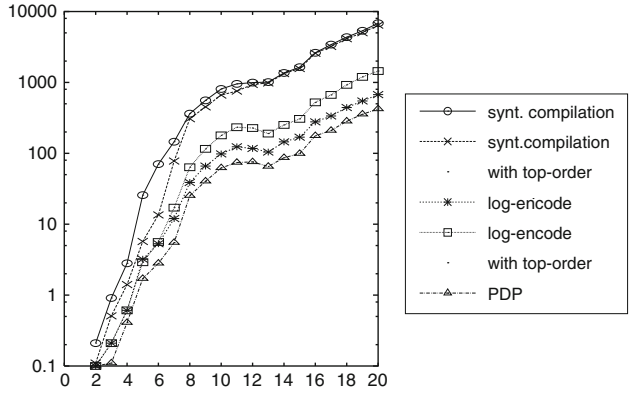

Fig. 24 `gas`, prop. 3, safety

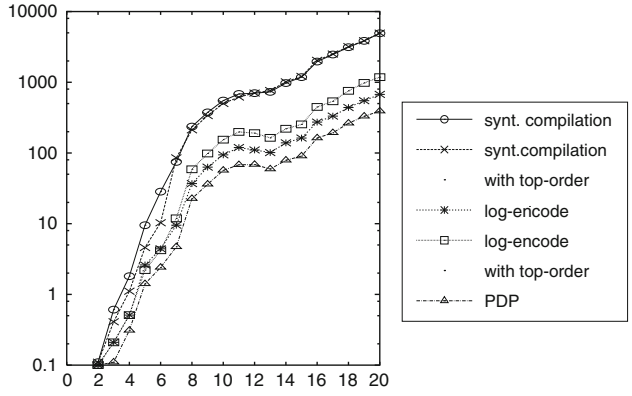

Fig. 25 `gas`, prop. 4, safety

property used in [5]:

$$((GFp0 \rightarrow GFp1) \wedge (GFp2 \rightarrow GFp0)$$
$$(GFp3 \rightarrow GFp2) \wedge (GFp4 \rightarrow GFp2)$$
$$(GFp5 \rightarrow GFp3) \wedge (GFp6 \rightarrow GF(p5 \vee p4))$$
$$(GFp7 \rightarrow GFp6) \wedge (GFp1 \rightarrow GFp7))$$
$$\rightarrow GFp8$$

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.[3] For this reason, we built a new compiler specialized for this kind of properties (Boolean

[3] Actually, the only translator that succeeded was `ltl2tgba` (http://spot.lip6.fr/wiki/). However, we had to disable simulation-based reduction so that the resulting automaton had more than 70,000 states and even parsing such an automaton took more than model checking time.

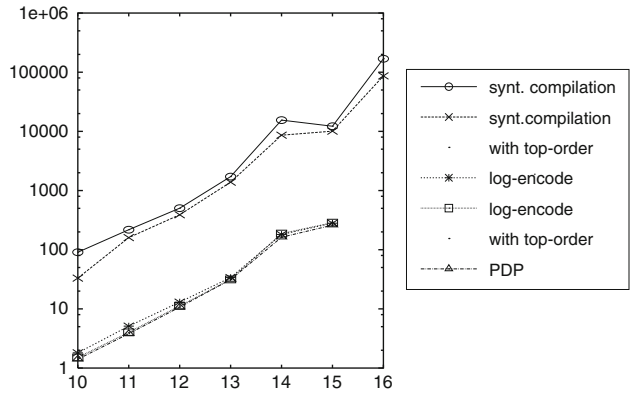

Fig. 26 `stack`, prop. 1, weak

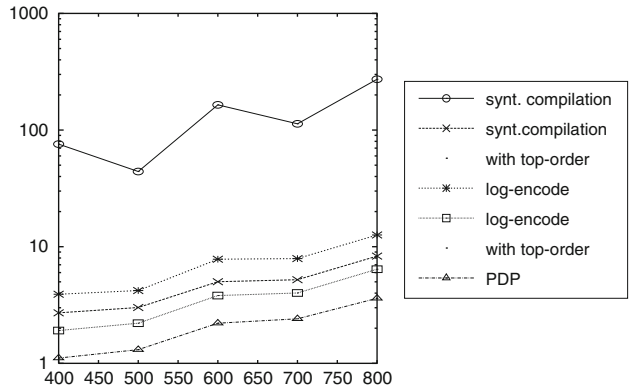

Fig. 27 `stack`, prop. 2, weak

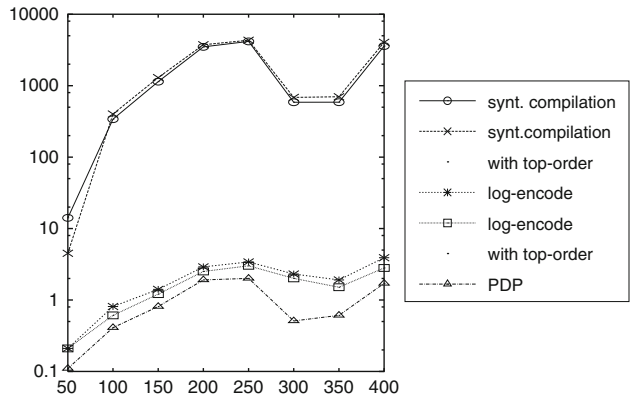

Fig. 28 `stack`, prop. 3, safety

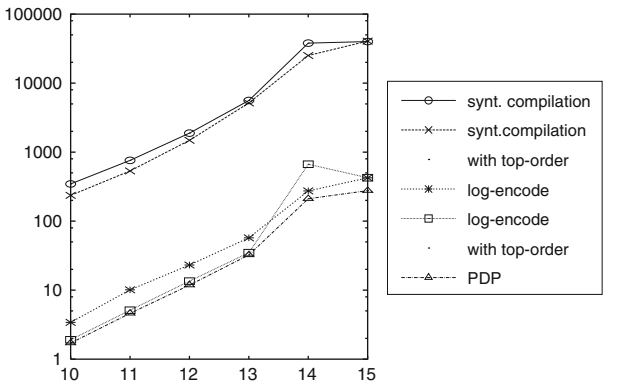

Fig. 29 `stack`, prop. 4, weak

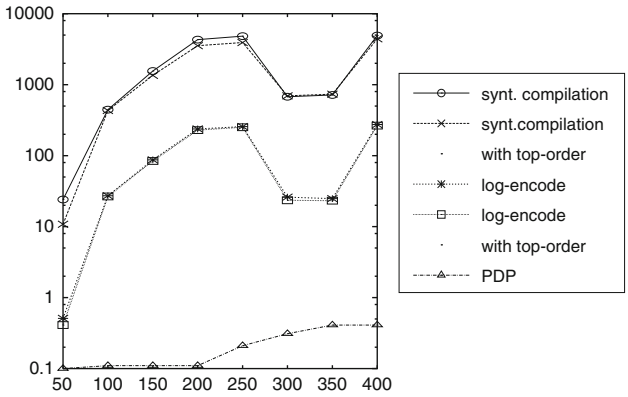

Fig. 30 `stack`, prop. 5, safety

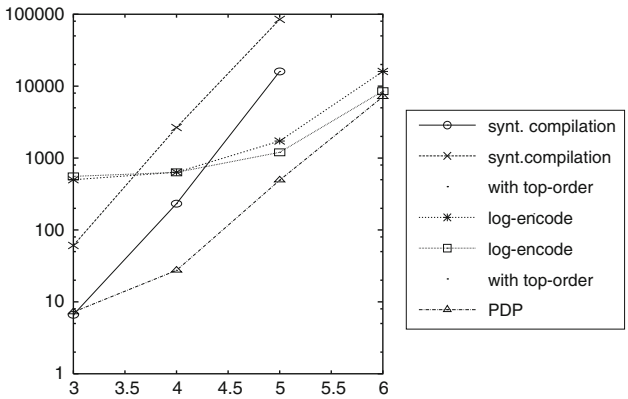

Fig. 31 `LCR`, large prop. (true), EL

combination of GF formulas). The resulting automaton has 1,281 states. We checked this property on the leader election algorithm LCR, cf. [27]. We instantiated the propositions in order to make the property true in one case, false in another. The results are plotted in Figs. 31–32. 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 automaton. However, as the size of the system grows, this technique outperforms syntactic compilation.

4.7 MC-algorithm comparisons

As a side-effect of our investigation, we can compare the algorithms for emptiness checking that we tested. In

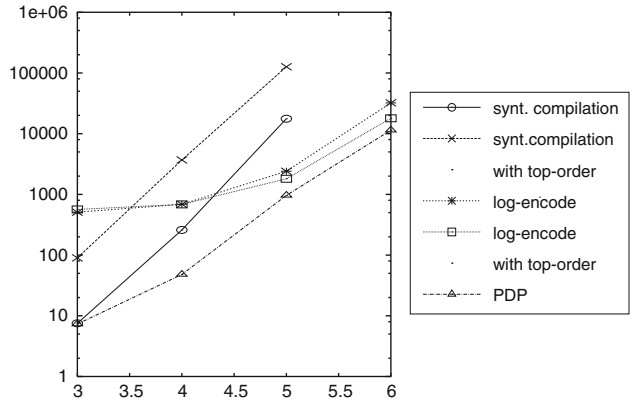

Fig. 32 LCR, large prop. (false), EL

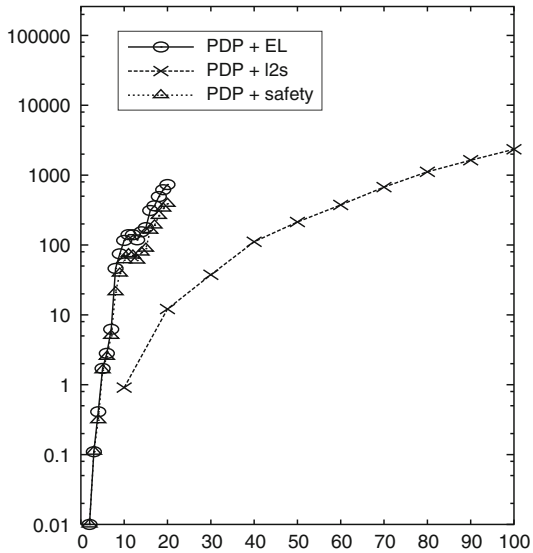

Fig. 33 gas, prop. 1

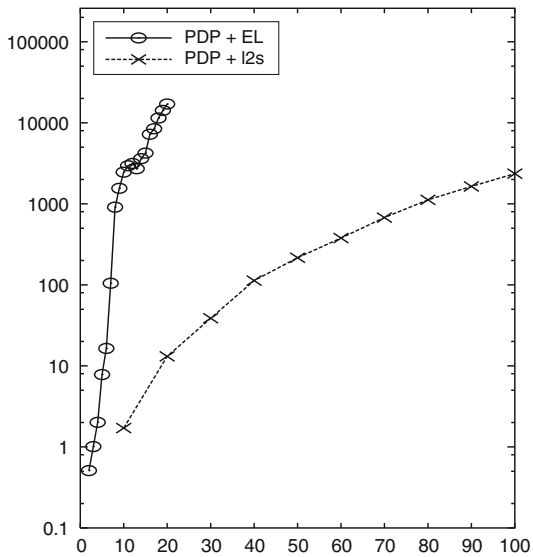

Fig. 34 gas, prop. 2

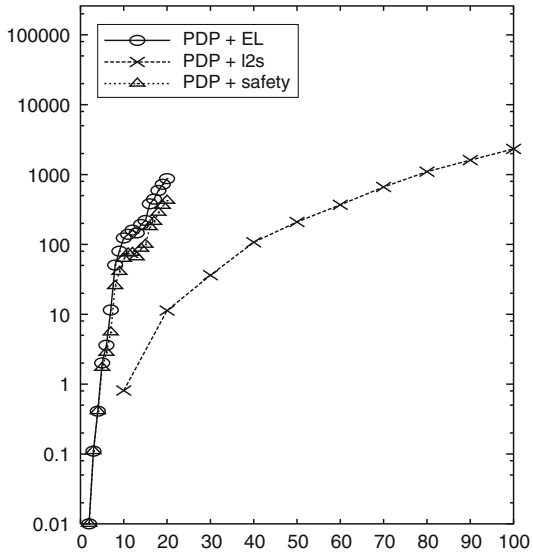

Fig. 35 gas, prop. 3

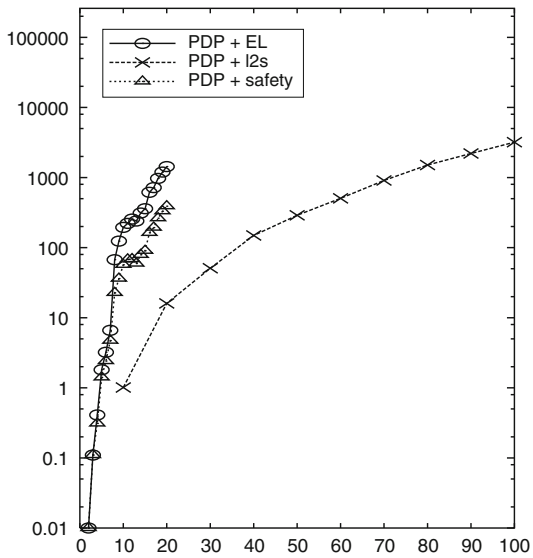

Fig. 36 gas, prop. 4

Figs. 33–41, we plotted the performance of EL, l2s and w/s on the systems and properties we have described above. For the product representation we used PDP (we noticed that the pattern is not different by using the syntactic compilation or the log-encoding of the semantic compilation).

Analyzing the plots, we can see that on one hand the improvement of w/s is not compelling for these examples; on the other hand, the effectiveness of l2s depends enormously on the system under verification (in the gas system, we have a positive exponential gap, while in the stack system the gap is largely negative).

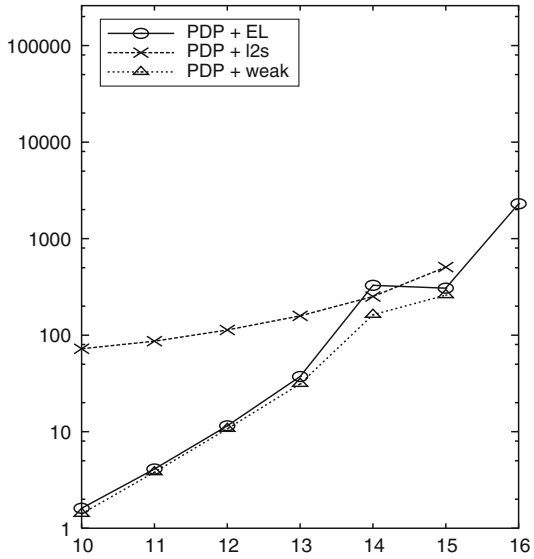

Fig. 37 `stack`, prop. 1

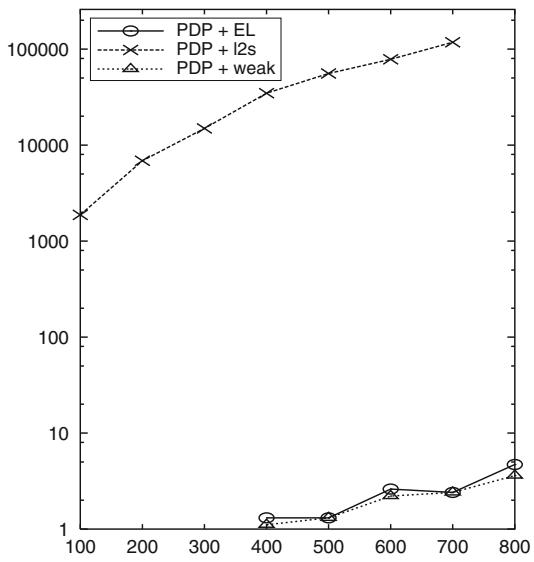

Fig. 38 `stack`, prop. 2

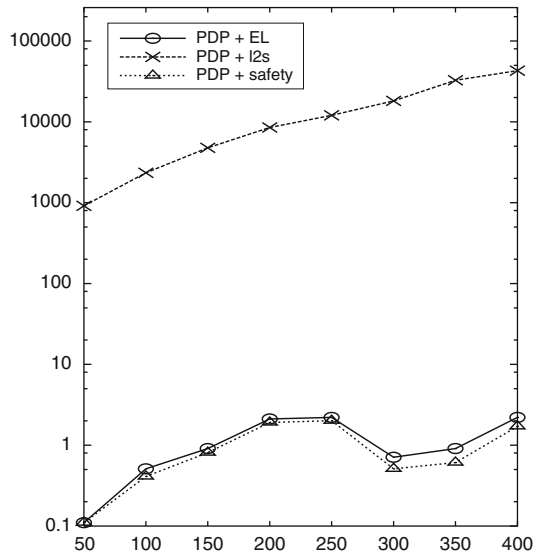

Fig. 39 `stack`, prop. 3

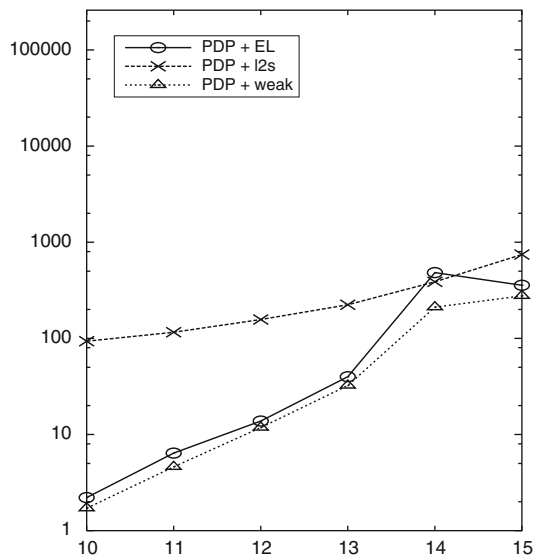

Fig. 40 `stack`, prop. 4

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, on one hand, calls for further research into the algorithmics of LTL symbolic model checking. The main focus of the research in this area has been either on the implementation of BDD operations, cf. [28], or on symbolic algorithms for FTS emptiness, cf. [31], ignoring the distinction between system and property. While ignoring this distinction allows for simpler algorithms, it comes with a significant performance penalty. On the other hand, the paper shows that practitioners of LTL symbolic model checking would better consider to implement or re-use semantic compilation of LTL formulas, rather than a simpler but less efficient symbolic compilation.

A second contribution of the paper is a clarification on the reasons why PDP is superior to standard MC: part of the improvement is due to the optimization applied to the explicit-state automaton corresponding to the LTL formulas; a second factor is the exploitation of the explicit automata transitions in the computation of the image used in the search.

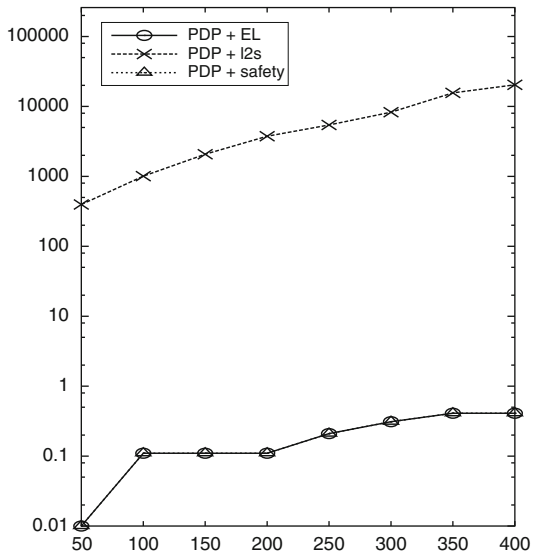

Fig. 41 `stack`, prop. 5

As a final remark, we note that the benchmarks we used in the experimental evaluation cannot be considered sufficiently representative, and we need to confirm our conclusions by more extensive benchmarking.

Acknowledgments We are grateful to R.E. Bryant, A. Goel and F. Somenzi for making interesting observations on the variable ordering of the property automaton encoding. 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.

References

1. Biere, A., Artho, C., Schuppan, V.: Liveness checking as safety checking. Electr. Notes Theor. Comput. Sci. **66**(2) (2002)
2. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. Inf. Comput. **98**(2), 142–170 (1992)
3. Biere, A., Clarke, E.M., Zhu, Y.: Multiple state and single state tableaux for combining local and global model checking. In: Correct System Design, vol. 1710 of LNCS, pp. 163–179. Springer, Berlin (1999)
4. Brayton, R.K., Hachtel, G.D., Sangiovanni-Vincentelli, A., Somenzi, F., Aziz, A., Cheng, S.T., Edwards, S., Khatri, S., Kukimoto, Y., Pardo, A., Qadeer, S., Ranjan, R.K., Sarwary, S., Shiple, T.R., Swamy, G., Villa, T.: Vis: a system for verification and synthesis. In: Alur, R., Henzinger, T.A. (eds.) Proceedings of the 8th International Conference on Computer Aided Verification CAV'96, vol. 1102, pp. 428–432. Springer, Berlin (1996)
5. Baukus, K., Lakhnech, Y., Stahl, K.: Verification of parameterized protocols. J. UCS **7**(2), 141–158 (2001)
6. Bloem, R., Ravi, K., Somenzi, F.: Efficient decision procedures for model checking of linear time logic properties. In: CAV, pp. 222–235 (1999)
7. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Trans. Comput. **C-35**(8), 677–691 (1986)
8. Cimatti, A., Clarke, E.M., Giunchiglia, F., Roveri, M.: NuSMV: a new symbolic model verifier. In: Proceedings of the 11th International Conference on Computer-Aided Verification, vol. 1633 of LNCS, pp. 495 – 499. Springer, Berlin (1999)
9. Clarke, E.M., Grumberg, O., Hamaguchi, K.: Another look at LTL model checking. Formal Methods Syst. Des. **10**(1), 47–71 (1997)
10. Clarke, E.M., Grumberg, O., Peled, D.A.: Model checking. MIT Press, Cambridge (1999)
11. Couvreur, J.-M.: On-the-fly verification of linear temporal logic. In: World Congress on Formal Methods, pp. 253–271 (1999)
12. Cimatti, A., Roveri, M., Bertoli, P.: Searching powerset automata by combining explicit-state and symbolic model checking. In: Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, vol. 2031 of LNCS, pp. 313–327. Springer, Berlin (2001)
13. Courcoubetis, C., Vardi, M.Y., Wolper, P., Yannakakis, M.: Memory-efficient algorithms for the verification of temporal properties. Formal Methods Syst Des **1**(2/3), 275–288 (1992)
14. Daniele, N., Giunchiglia, F., Vardi, M.Y.: Improved automata generation for linear temporal logic. In: Proceedings of the 11th International Conference on Computer-Aided Verification, vol. 1633 of LNCS, pp. 249–260. Springer, Berlin (1999)
15. Etessami, K., Holtzmann, G.: Optimizing Büchi automata. In: Proceedings of CONCUR'2000, vol. 1877 of LNCS, Springer, Berlin (2000)
16. Emerson, E.A., Lei, C.L.: Efficient model checking in fragments of the propositional μ-calculus. In: Proceedings of the Symposium on Logic in Computer Science, pp. 267–278. IEEE Computer Society, New York (1986)
17. Etessami, K., Wilke, T., Schuller, R.: Fair simulation relations, parity games, and state space reduction for büchi automata. In: Orejas, F., Spirakis, P.G., van Leeuwen, J. (eds.) Automata, Languages and Programming, 28th International Colloquium, vol. 2076 of LNCS. Springer, Berlin (2001)
18. Fisler, K., Fraer, R., Kamhi, G., Vardi, M.Y., Yang, Z.: Is there a best symbolic cycle-detection algorithm? In: Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, vol. 2031 of LNCS, pp. 420–434. Springer, Berlin (2001)
19. Fraer, R., Kamhi, G., Ziv, B., Vardi, M.Y., Fix, L.: Prioritized traversal: efficient reachability analysis for verification and falsification. In: Proceedings of the 12th International Conference on Computer-Aided Verification, vol. 1855 of LNCS, pp. 389–402. Springer, Berlin (2000)
20. Fritz, C., Wilke, T.: State space reductions for alternating Büchi automata: quotienting by simulation equivalences. In: Proceedings of 22th Conference on the Foundations of Software Technology and Theoretical Computer Science, vol. 2556 of Lecture Notes in Computer Science, pp. 157–169 (2002)
21. Gurumurty, S., Bloem, R., Somenzi, F.: Fair simulation minimization. In: Proceedings of CAV'02, number 2404 in LNCS. Springer, Berlin (2002)
22. Godefroid, P., Holzmann, G.J.: On the verification of temporal properties. In: PSTV, pp. 109–124 (1993)
23. Giannakopoulou, D., Lerda, F.: From states to transitions: improving translation of LTL formulae to Büchi automata. In: Proceedings of FORTE'02., number 2529 in LNCS. Springer, Berlin (2002)
24. Gastin, P., Oddoux, D.: Fast ltl to büchi automata translation. In: Computer Aided Verification, Proceedings of 13th International Conference, vol. 2102 of Lecture Notes in Computer Science, pp. 53–65. Springer, Berlin (2001)
25. Geldenhuys, J., Valmari, A.: Tarjan's algorithm makes on-the-fly LTL verification more efficient. In: Proceedings of 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science 2988, pp. 205–219. Springer, Berlin (2004)
26. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addison Wesley, Boston (2003)

27. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann Publishers Inc., San Francisco (1996)
28. Ochi, H., Yasuoka, K., Yajima, S.: Breadth-first manipulation of very large binary-decision diagrams. In: Proceedings of the 1993 IEEE/ACM International Conference on Computer-aided Design (ICCAD'93), pp. 48–55. IEEE Computer Society Press (1993)
29. Peled, D.: Combining partial order reductions with on-the-fly model-checking. In: CAV, pp. 377–390 (1994)
30. Pnueli, A.: The temporal logic of programs. In: Proceedings of 18th IEEE Symposium on Foundation of Computer Science, pp. 46–57 (1977)
31. Ravi, K., Bloem, R., Somenzi, F.: A comparative study of symbolic algorithms for the computation of fair cycles. In: Proceedings of the 3rd International Conference on Formal Methods in Computer-Aided Design, vol. 1954 of LNCS, pp. 143–160. Springer, Berlin (2000)
32. Somenzi, F., Bloem, R.: Efficient Büchi automata from LTL formulae. In: Proceedings of the 12th International Conference on Computer-Aided Verification, vol. 1855 of LNCS, pp. 247–263. Springer, Berlin (2000)
33. Schwoon, S., Esparza, J.: A note on on-the-fly verification algorithms. In: Proceedings of 11th Internetional Conference on Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science 3440, pp. 174–190. Springer, Berlin (2005)
34. Sebastiani, R., Singerman, E., Tonetta, S., Vardi, M.Y.: GSTE is partitioned model checking. In: Proceedings of the 15th International Conference on Computer-Aided Verification (CAV), vol. 3114 of LNCS, pp. 229–241. Springer, Berlin (2004)
35. Sebastiani, R., Tonetta, S.: "More Deterministic" vs. "Smaller" Büchi automata for efficient LTL model checking. In: Proceedings of the Conference on Correct Hardware Design and Verification Methods (CHARME), vol. 2860 of LNCS, pp. 126–140. Springer, Berlin (2003)
36. Sebastiani, R., Tonetta, S., Vardi, M.Y.: Symbolic systems, explicit properties: on hybrid approaches for LTL symbolic model checking. In: Proceedings of the 16th International Conference on Computer-Aided Verification (CAV'05), pp. 350–363 (2005)
37. Valmari, A.: Error detection by reduced reachability graph generation. In: ATPN (1988)
38. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification. In: Proceedings of the 1st Symposium on Logic in Computer Science, pp. 332–344. IEEE Computer Society (1986)
39. Vardi, M.Y., Wolper, P.: Reasoning about infinite computations. Inf. Comput. **115**(1), 1–37 (1994)
40. Yang, J., Seger, C.-J.H.: Generalized symbolic trajectory evaluation—abstraction in action. In: Proceedings of the 4th International Conference on Formal Methods in Computer-Aided Design, vol. 2517 of LNCS, pp. 70–87. Springer, Berlin (2002)