Combining Partial-Order Reduction and Symbolic Model Checking to verify LTL properties *

José Vander Meulen[1] and Charles Pecheur[2]

[1] Université catholique de Louvain, jose.vandermeulen@uclouvain.be
[2] Université catholique de Louvain, charles.pecheur@uclouvain.be

Abstract. BDD-based symbolic techniques and partial-order reduction (POR) are two fruitful approaches to deal with the combinatorial explosion of model checking. Unfortunately, past experience has shown that BDD-based techniques do not work well for loosely-synchronized models, whereas POR methods allow explicit-state model checkers to deal with large concurrent models. This paper presents an algorithm that combines symbolic model checking and POR to verify linear temporal logic properties without the next operator (LTL_X), which performs better on models featuring asynchronous processes. Our algorithm adapts and combines three methods: Clarke et al.'s tableau-based symbolic LTL model checking, Iwashita et al.'s forward symbolic CTL model checking and Lerda et al.'s ImProviso symbolic reachability with POR. We present our approach, outline the proof of its correctness, and present a prototypal implementation and an evaluation on two examples.

1 Introduction

Two common approaches are commonly exploited to fight the combinatorial state-space explosion in model-checking, with different perspectives: partial-order reduction methods (POR) explore a reduced state space in a property-preserving way [1, 2] while symbolic techniques use efficient structures such as binary decision diagrams (BDDs) to concisely encode and compute large state spaces [3]. In their basic form, symbolic approaches tend to perform poorly on asynchronous models where concurrent interleavings are the main source of explosion, and explicit-state model-checkers with POR such as Spin [4] have been the preferred approach for such models.

This paper presents an approach that integrates POR in BDD-based model checking for LTL_X to provide an efficient and scalable symbolic verification solution for models featuring asynchronous processes. Our approach proceeds as follows:

1. We start from the tableau-based reduction of LTL verification to fair-CTL of Clarke et al. [5], which results in looking for fair executions in the product P of the model and a tableau-based encoding of the (negated) property.

* This work is supported by project MoVES under the Interuniversity Attraction Poles Programme — Belgian State — Belgian Science Policy.

2. We construct P_r, a property-preserving partial-order reduction of P, using an adaptation of Lerda et al.'s ImProviso algorithm [6]. We also implemented the algorithm of Alur et al. [7] for comparison purposes.
3. Finally, we check within P_r whether P contains a fair cycle using the forward traversal approach of Iwashita et al. [8]. We also implemented the classical backward as a basis for comparison, though experimental results show the forward approach to be more efficient than the backward approach.

We have implemented this new approach in a prototype and obtained experimental results that show a significant performance gain with respect to symbolic techniques without POR.

The main contributions of this paper are the global symbolic verification algorithm for checking LTL_X properties which adapts and combines tableau-based LTL, fair-cycle detection and partial-order reduction, a proof of correctness of the global algorithm, a prototype implementation, and an experimental evaluation on two models.

The remainder of the paper is structured as follows. Section 2 establishes basic definitions and notations and presents the tableau-based reduction of LTL to fair-CTL and the forward traversal approach. Section 3 presents partial-order reduction and its application to symbolic model checking in ImProviso. In Section 4, we present our new approach for LTL model-checking with POR and detail our adapation of the ImProviso algorithm. Section 5 presents our implementation and reports experimental results. Section 6 reviews related work. Finally, Section 7 gives conclusions as well as directions for future work.

2 Symbolic LTL Model Checking

2.1 Transitions Systems

We represent the behavior of a system as a *transition system*, with labelled transitions and propositions interpreted over states. In the rest of this paper, we assume a set AP of *atomic propositions* and a set A of *actions*[3]. Without loss of generality, the set AP can be restricted to the propositions that appear in the property to be verified on the system. A *fair transition system* is a transition system enriched with a set of *fairness constraints*, each constraint consisting of a set of states.

Definition 1 (Transition System). *Given a set of actions A and a set of atomic propositions AP, a* transition system *(over A and AP) is a structure $M = (S, R, I, L)$ where S is a finite set of states, $I \subseteq S$ are initial states, $R \subseteq S \times A \times S$ is a transition relation, and $L : S \to 2^{AP}$ is an interpretation function over states.*

Definition 2 (Fair Transition System). *A* fair transition system *is a structure $M = (S, R, I, L, F)$ where (S, R, I, L) is a transition system and $F \subseteq 2^S$ is a set of fairness constraints.*

[3] Often called *transitions* in the literature, notably in [9]. For clarity, we only call *transitions* specific transition instances $s \xrightarrow{a} s'$.

We write $s \xrightarrow{a} s'$ for $(s, a, s') \in R$. An action a is *enabled* in a state s iff there is a state s' such that $s \xrightarrow{a} s'$. We write enabled(s, R) for the set of enabled actions of R in s. When the context is clear, we write enabled(s) instead of enabled(s, R). We assume that R is total (i.e. enabled$(s) \neq \emptyset$ for all $s \in S$). The set of all *paths* of M is defined as $\mathrm{tr}(M) = \{s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots \mid s_0 \in I \wedge \forall i \in \mathbb{N} \cdot s_i \xrightarrow{a_i} s_{i+1}\}$. A path π is said to be *fair* if and only if for every $F_i \in F$, $\inf(\pi) \cap F_i \neq \emptyset$, where $\inf(\pi)$ is the set of states that appear infinitely often in π. The set of all fair paths, or fair traces, of M is defined as $\mathrm{ftr}(M) = \{\pi \mid \pi \in \mathrm{tr}(M) \wedge \forall F_i \in F \cdot \inf(\pi) \cap F_i \neq \emptyset\}$.

We write $M \sqsubseteq M'$ iff M is a *sub-transition system* of M', in the following sense:

Definition 3 (Inclusion of fair transition systems). *Let $M = (S, R, I, L, F)$, $M' = (S', R', I', L', F')$ be two fair transition systems. M is a* sub-transition system *of M', denoted $M \sqsubseteq M'$, if and only if $S \subseteq S'$, $R \subseteq R'$, $I \subseteq I'$, $L(s) = L'(s)$ for $s \in S$, and $\forall F'_i \in F' \cdot \exists F_i \in F \cdot F_i \subseteq F'_i$.*

We can see that if $M \sqsubseteq M'$, each fair path of M is a fair path of M'.

Lemma 1. *if $M \sqsubseteq M'$ then $\mathrm{ftr}(M) \subseteq \mathrm{ftr}(M')$.*

2.2 From LTL to Fair-CTL

This section outlines the algorithm, introduced in [5], to verify LTL properties using BDD-based symbolic model checking.

We consider the verification of properties expressed in LTL_X, linear propositional temporal logic without the next operator. LTL formulæ are interpreted over each (infinite) execution path of the model. We denote the classical temporal operators as **F**, **G** and **U**. Informally, let π be a execution path, $\mathbf{G}\, f$ (globally f) says that f will hold in all future states of π, $\mathbf{F} f$ (finally f) says that f will hold in some future state of π, $f\ \mathbf{U}\ g$ (f until g) says that g will hold in some future state of π and, at every preceding state of π, f will hold. We will reason for the most part in terms of *(un)satisfiability* of the negation of the desired property $\neg f$. We write $(M, s) \models \mathbf{E} g$ to express that there exists a path from state s in M that satisfies a formula g.

Given a transition system M and an LTL property f, the *tableau* of $\neg f$ is constructed. The tableau of a formula g is a fair transition system $T = (S_T, R_T, I_T, L_T, F_T)$ over the singleton alphabet $A = \{\bot\}$ and the set AP of propositions which appear in g. Each state of the tableau is a set of formulae derived from g, which characterizes the sub-formulae of g that are satisfied on fair traces from that state. Initial states are those that entail g, and the fairness constraints ensure that all eventualities occurring in g are fulfilled. The fair traces of the tableau correspond to the traces that satisfy g. See [5] for details.

The tableau of $\neg f$ is then composed with the initial system M to produce a new fair transition system P. If P contains deadlocks, we remove from S_P all the states which lead necessarily to deadlocks and restrict R_p to the remaining states.

Definition 4 (Product of M and T). *Given a system $M = (S, R, I, L)$ and a tableau $T = (S_T, R_T, I_T, L_T, F_T)$, the product of M and T, denoted $M \times T$, is a fair transition system $P = (S_P, R_P, I_P, L_P, F_P)$ where:*

- $S_P = \{(s_t, s) \in S_T \times S \,|\, L_T(s_t) = L(s)\}$
- $R_P = \{((s_t, s), a, (s'_t, s')) \,|\, R_T(s_t, \bot, s'_t) \wedge R(s, a, s')\}$
- $I_P = S_P \cap (I_T \times I)$
- $L_P((s_t, s)) = L_T(s_t) = L(s)$
- $F_P = \left\{\{(s_t, s) \in S_P \,|\, s_t \in F_T^i\} \,|\, F_T^i \in F_T\right\}$

It is shown in [5] that M contains a path which satisfies $\neg f$ iff there is an infinite fair path in P that starts from an initial state (i_t, i). Furthermore, the existence of fair traces is captured by the fair CTL formula $\mathbf{E}_F\mathbf{G}\, true$, to be read as "there exists a fair path such that globally true". The interest is that fair-CTL formulae can be verified with BDD-based symbolic model checking.

Theorem 1. *Let T be the tableau of $\neg f$ and P be the product of M and T. Given a state $i \in I$, $(M, i) \models \mathbf{E}\neg f$ if and only if there is a state (i_t, i) in I_P such that $(P, (i_t, i)) \models \mathbf{E}_F\mathbf{G}\, true$.*

2.3 Forward Symbolic Model-Checking

In [8], Iwashita et al. present a model-checking algorithm for a fragment of fair-CTL based on forward state traversal. In the following sections, we enrich this algorithm with partial-order reduction to efficiently check the unsatisfiability of the $\mathbf{E}_F\mathbf{G}\, true$ formula derived from tableau-based LTL model-checking.

The semantic of a CTL formula f is defined as a relation $s \models f$ over states $s \in S$. We define the *language* of f as $\mathcal{L}(f) = \{s \in S \,|\, s \models f\}$. In the sequel we assimilate a temporal logic formula f to the set of states $\mathcal{L}(f)$ that it denotes, for the sake of simplifying the notations.

Given a model M, a formula f and initial conditions i, conventional BDD-based symbolic model-checking can be described as evaluating $\mathcal{L}(f)$ over the sub-formulæ of f in a bottom-up manner, and checking whether $\mathcal{L}(i) \subseteq \mathcal{L}(f)$. The evaluation of (future) CTL operators in f results in a backward state-space traversal of the model. $\mathcal{L}(i) \subseteq \mathcal{L}(f)$ can be expressed as checking whether $i \implies f$, or equivalently, checking unsatisfiability of $i \wedge \neg f$ in M.

The forward exploration from [8] works by transforming a property $h \wedge op(g)$ into $op'(h) \wedge g$, where a future, backward-traversal CTL operator op in the right term is transformed into a past, forward-traversal operator op' in the left term. It is shown in [8] that these formulae are *equisatisfiable in M*, in the sense that there exists a state in M which satisfies the transformed formula iff there exists a state in M which satisfies the original formula.

In general, h is then a past-CTL formula. The following (past-temporal) operations over formulæ are defined: [4]

$$\begin{aligned} \mathrm{FwdUntil}(h, g) &= \mu Z.[h \vee \mathrm{post}(Z \wedge g)] \\ \mathrm{FairEH}(h) &= \nu Z.[h \wedge \mathrm{post}(\textstyle\bigwedge_{F_i \in F} \mathrm{FwdUntil}(F_i, Z) \wedge Z)] \end{aligned}$$

where $\mathrm{post}(X) = \{s' \in S \,|\, \exists s \in X, a \in A \cdot s \xrightarrow{a} s'\}$ is the post-image of X. $\mathrm{FwdUntil}(h, g)$ computes states s that can be reached from h within g (except

[4] The notation $\mu Z.\tau(Z)$ (resp. $\nu Z.\tau(Z)$) denotes the *least fixed point* (resp. *greater fixed point*) of the *predicate transformer* τ. For more details, we refer the reader to [5].

for s itself), and FairEH(h) computes states reachable from a fair cycle all within h[5]. On this basis, it is established that $h \wedge \mathbf{E}_F\mathbf{G}\, g$ is equisatisfiable in M to FairEH(FwdUntil(h, g) $\wedge\, g$).

In particular, for $h = i$ and $g = \mathit{true}$ this reduces to FairEH(FwdUntil(i, *true*)), where FwdUntil(i, *true*) exactly computes the reachable state space of M, which we denote $\mathit{Reachable}(M)$. We thus obtain the following fact.

Theorem 2.

$$\exists i \in I \cdot (M, i) \models \mathbf{E}_F\mathbf{G}\, \mathit{true} \quad \mathit{iff} \quad \exists s \in S \cdot (M, s) \models \text{FairEH}(\text{Reachable}(M))$$

In essence, this theorem captures the fact that the fair-CTL model-checking problem resulting from the tableau-based reduction of LTL can be decomposed into two distinct parts, the computation of the reachable state space and the search for a fair cycle. Besides, the POR theory shows that only a subset of the reachable state space needs to be computed to see whether a property is satisfied or not. The following sections will demonstrate that different methods can be used to compute the (reduced) reachable state space, and also that different methods can be used to perform the fair-cycle detection.

3 Partial-Order Reduction

The goal of partial-order reduction methods (POR) is to reduce the number of states explored by model-checking, by avoiding the exploration of different equivalent interleavings of concurrent transitions [10, 2, 9].

Partial-order reduction is based on the notions of *visibility* of actions and *independence* between actions. An action a is *invisible* if and only if it does not affect atomic propositions, i.e. if $L(s) = L(s')$ for any $s \xrightarrow{a} s'$ (and *visible* otherwise). Two actions are *independent* if they do not disable one another and executing them in either order results in the same state. Intuitively, if two independent actions a and b are invisible with respect to the property f that one wants to verify, then it does not matter whether a is executed before or after b, because they lead to the same state and do not affect the truth of f. Partial-order reduction consists in identifying such situations and restricting the exploration to either of these two alternatives. Given a transition system $M = (S, R, I, L)$, POR amounts to exploring a reduced model $M_R = (S_R, R_R, I, L_R)$ with $S_R \subseteq S$, $R_R \subseteq R$, and $L_R = \{(s_r, A) \in L \mid s_r \in S_R\}$. In practice, classical POR algorithms [2, 9] execute a modified depth-first search (DFS). At each state s, an adequate subset ample(s) of the actions enabled in s are explored. To ensure that this reduction is adequate, that is, that verification results on the reduced model hold for the full model, ample(s) must respect the following set of conditions as set forth in [9, 10]:

C_0 ample(s) $= \emptyset$ if and only if enabled(s) $= \emptyset$.

[5] Both FwdUntil and FairEH can be expressed in the past version of fair-CTL: FairEH(h) corresponds to $\mathbf{E}_F\mathbf{G}\, h$ and FwdUntil(h, f) corresponds to $h \vee \mathbf{EX}\,\mathbf{E}[f\ \mathbf{U}\ (h \wedge f)]$, where the direction of temporal operators is reversed.

C_1 Along every path in the full state graph that starts at s, an action $a \notin \text{ample}(s)$ that is dependent on an action in $\text{ample}(s)$ cannot be executed without an action in $\text{ample}(s)$ occurring first.
C_2 If $\text{ample}(s) \neq \text{enabled}(s)$, then all actions in $\text{ample}(s)$ are invisible.
C_3 A cycle is not allowed if it contains a state in which some action is enabled, but is never included in $\text{ample}(s)$ on the cycle.

Conditions C_0, C_1, C_2 and C_3 are sufficient to guarantee that the reduced model preserves properties expressed in LTL_X, but does not preserve properties expressed in LTL [9]:

Theorem 3. *Given M a transition system, f a LTL_X property, if M_R is a POR reduction of M using an* $\text{ample}(s)$ *that satisfies conditions C_0–C_3, then $(M, i) \models \mathbf{E}f$ iff $(M_R, i) \models \mathbf{E}f$.*

Conditions C_1 and C_3 depend on the whole state graph. C_1 is not directly exploitable in a verification algorithm. Instead, one uses sufficient conditions, typically derived from the structure of the model description, to safely decide where reduction can be performed. Contrary to C_1, C_3 can be checked on the reduced graph, though in a nontrivial way. However, a stronger condition can be used. A sufficient condition for C_3 is that at least one state along each cycle is fully expanded.

3.1 Process Model

In the sequel, we assume a process-oriented modeling language. We define a *safe process model* as an extension of a transition system which distinguishes disjoint subsets of local actions A_i, that are suitable candidates for partial-order reduction. Typically, such actions will correspond to local transitions of different processes p_i in a concurrent program.

Definition 5 (Safe Process Model). *Given a transition system $M = (S, R, I, L)$, a* process model *for M consists of a finite set of disjoint sets of local actions $A_0, A_1, \ldots, A_{m-1}$ with $A_i \subseteq A$. The local transitions are defined as $R_i = R \cap (S \times A_i \times S)$. A process model is* safe *with respect to M iff all its local transitions are safe, that is, for all $a \in A_i$, a is invisible, and for all $s \in S$,* $\text{ample}(s) = \text{enabled}(s, R_i)$ *satisfies condition C_1.*

Note that this definition guarantees that $\text{ample}(s) = \text{enabled}(s, R_i)$ respects conditions C_1 and C_2, but not C_3, which is ensured dynamically by detecting cycles within the reduction algorithm.

3.2 Partial-Order Reduction with BDDs

In this section we discuss two algorithms which implement a symbolic version of the POR method presented in Section 3. Both approaches can be used to compute a reduced reachable state space.

In [6], Lerda et al. propose ImProviso, a BDD-based symbolic version of the Two-Phase POR algorithm for computing a reduced state space. The Two-Phase algorithm was first presented by Nalumasu and Gopalakrishnan in [11]. ImProviso

alternates between two distinct phases: Phase-1 and Phase-2. Phase-1 expands only safe transitions considering each process at a time, in a fixed order. As long as a process offers safe transitions, those transitions alone are executed, otherwise the algorithm moves on to the next process. Phase-2 performs a full expansion of the final states reached in Phase-1, then Phase-1 is recursively applied to the reached states.

In [7], Alur et al. propose another approach based on a modified *breadth-first search* (BFS) algorithm which respects conditions C_0–C_3, using BDD techniques. It produces a reduced graph by expanding at each step a subset of the transition relation.

Both approaches perform a BFS instead of a DFS. Hence, it is much harder to detect cycles. To tackle this problem, both algorithms over-approximate the cycles. The over-approximation guarantees that all cycles are correctly identified, but possibly needlessly decreases the number of states where the reduction can be applied.

Although Alur's method and ImProviso are similar, they differ in the following ways:

- In Alur's method, a single subset of the whole transition relation is computed at each step. In ImProviso, for each process a transition relation which contains only safe actions is precomputed. These transition relations are used during Phase-1. We contend that this leads to better performance because each Phase-1 step is computed with much smaller BDDs.
- The Two-Phase approach reduces the over-approximation by limiting cycle detection to the current execution of Phase-1.

4 LTL Model checking with Partial-Order Reduction

In this section we bring together the computation of the reachable state space by means of POR and the fair-cycle detection. Given a transition system $M = (S, R, I, L)$ with a safe process model $A_1, \ldots, A_n$ and a LTL_X property f, our algorithm verifies whether M satisfies f by building a tableau T for $\neg f$ and checking the absence of accepting traces in $P = (S_P, R_P, I_P, L_P, F_P)$, the product of M and T.

This check is performed symbolically. We first compute a reduced state space of P, and then we check whether P contains a fair cycle within the reduced state space. In this section, we use a variant of the ImProviso to compute the reduced state space. We also use the forward model checking to perform the fair-cycle detection. In other words, this check is performed symbolically, by checking the emptiness of the following formula using BDDs: FairEH(ReachablePOR(P)). In Section 5, we compare different methods to compute the reduced graph, as well as to look for fair cycles.

4.1 Computation of the reachable states

A key new element is the algorithm ReachablePOR which constructs a reduced reachable state space of P. It is given in Figure 1, and is based on the ImProviso algorithm of [6]. In order to apply partial-order reduction on the product system

P, we lift the process model from M to P and pre-compute, for each safe action set A_i, the BDD of the partial transition relation $R_{P,i} = R_P \cap (S_P \times A_i \times S_P)$.

```
1  global R_P
2  global R_{P,i}[0..m-1]
3
4  global frontier // current frontier
5  global visited  // visited states
6
7  procedure ReachablePOR(P_I)
8    frontier,visited := I_P, I_P
9    while (frontier ≠ {}) {
10     phase1()
11     phase2()
12   }
13 }
14
15 function deadStates(R, X) {
16   return X \ dom R
17 }
18
19 procedure phase2() {
20   local image := post(R_P, frontier)
21   frontier := image\visited
22   visited :=  visited ∪ image
23 }
24
25
26 procedure phase1() {
27   local cycleApprox := {}
28   local stack := frontier
29
30   foreach (i in 0,··· ,m − 1) {
31     local image :=
32       post(R_{P,i}[i], frontier)
33     local dead :=
34       deadStates(R_{P,i}[i], frontier)
35
36     while ((image\stack) ≠ {}) {
37       stack := stack ∪ image
38       cycleApprox := cycleApprox ∪
39         (image ∩ stack)
40       frontier := image\stack
41       image := post(R_{P,i}[i], frontier)
42       dead := dead ∪
43         deadStates(R_{P,i}[i], frontier)
44     }
45
46     frontier := frontier ∪ dead
47   }
48   frontier := frontier ∪ cycleApprox
49   visited := visited ∪ stack
50 }
```

Fig. 1. ReachablePOR algorithm

ReachablePOR performs the two phases alternatively until no states to visit remain. The global variable `frontier` contains the current frontier, that is, the set of states which have been reached but not expanded yet. The global variable `visited` contains all the reached states. The first phase (`phase1`) performs partial expansion of the safe transitions of each process. The outer loop (lines 30–47) iterates over the processes. The inner loop (lines 36–44) expands all safe transitions of the current process, until no more new states can be found. The following invariants hold at line 36:

- The `stack` variable contains all the states which have already been reached during the current run of `phase1`.
- The `cycleApprox` contains all the states already in `stack` which have been reached again in a consecutive iteration. Those states over-approximate the set of states closing a cycle; they are added back to the current frontier when moving to Phase-2 (line 48).
- The `dead` variable contains all the reached states with no enabled transitions for the current process, as computed by `deadStates`. Those states are added back to the frontier when moving to the next process (line 46).

The second phase (`phase2`) performs a single-step full expansion of the states of the current frontier.

ReachablePOR differs from ImProviso in the following ways:

1. ReachablePOR explores a product system P. The ample sets, captured in $R_{P,i}$, depend only on the model M, while the cycle condition is checked on the product P. In ImProviso, there is no tableau and everything is computed on the original model M.
2. When a presumed cycle is detected on state s in Phase-1, ImProviso will expand s during the expansion of the next process, whereas ReachablePOR will postpone expansion of s to the next Phase-2. When a product P is reduced, we have noticed that this modification tends to improve both the number of visited states and the verification time.
3. ReachablePOR keeps track of states that have no transition with the current process (lines 34 and 43) and passes them to the next processes. If this computation was not done, we could have missed some states during the BFS. So, we could have violated the condition $C0$. The need for this computation was apparently not addressed in [6].
4. ImProviso performs an additional outermost loop in Phase-1, to expand any additional safe actions that have been enabled by the previous round over all processes, such as receiving a message on a channel where it has been previously sent. This is not needed in ReachablePOR because by construction our notion of safe action does not allow this kind of situation. It would easily be added back if it were to become useful.

ReachablePOR(P) explores a reduced transition system $P_R = (S_R, R_R, I_P, L_R, F_R)$, where $L_R = \{(s_r, A) \in L_P \mid s_r \in S_R\}$, and F_R is the restriction of F_P to S_R, i.e. $F_R = \{F_i \cap S_R \mid F_i \in F_P\}$. It returns the explored states $S_R = \text{ReachablePOR}(P)$ as the final value of `visited`. By construction, $P_R \sqsubseteq P$.

4.2 Fair-cycle Detection

The reduced state space S_R is used to search for infinite fair paths by computing FairEH(S_R). From the definition of FairEH, it is clear that FairEH(S_R) only explores states within S_R. Note, however, that FairEH uses the full transition relation R_P rather than the reduced transition relation R_R implicitly explored by ReachablePOR. FairEH(S_R) thus explores an *induced* fair transition system $P_I = (S_R, R_P \cap (S_R \times A \times S_R), I_P, L_P, F_P)$. By construction $P_R \sqsubseteq P_I \sqsubseteq P$ and thus, by Lemma 1, $\text{ftr}(P_R) \subseteq \text{ftr}(P_I) \subseteq \text{ftr}(P)$. Note that S_R is evidently equal to Reachable(P_I). Hence evaluating FairEH(ReachablePOR(P)) in P amounts to evaluating FairEH(Reachable(P_I)) in P_I and we have the following lemma:

Lemma 2. *Given* $P = (S_P, R_P, I_P, L_P, F_P)$, $S_R = \text{ReachablePOR}(P)$, $P_I = (S_R, R_P \cap (S_R \times A \times S_R), I_P, L_P, F_P)$ *and* $s_P \in S_P$, $(P, s_P) \models \text{FairEH}(\text{ReachablePOR}(P))$ *if and only if* $(P_I, s_P) \models \text{FairEH}(\text{Reachable}(P_I))$. *When* $(P_I, s_P) \models \text{FairEH}(\text{Reachable}(P_I))$, $s_P \in S_R$.

4.3 Correctness

To demonstrate the correctness of our approach, we have to prove that, given a property f and a model M, f holds in M iff FairEH(ReachablePOR(P)) returns an empty set, where P is the product of M and the tableau for $\neg f$. Conversely, we

will prove that there is a path from an initial state i in M on which $\neg f$ holds, written $(M, i) \models \mathbf{E}\neg f$, iff there is a state in P satisfying FairEH(ReachablePOR(P)).

Before getting to this main result, we need to address two technical issues. First, the following two lemmas establish that the preservation of properties when reducing M to M_R is carried over when reducing P to P_R based on the transitions of M, as performed in ReachablePOR.

Lemma 3. *Given a product system $P = M \times T$ and P_R the reduced transition system explored by* ReachablePOR(P)*, there exists a reduced transition system M_R such that $P_R = M_R \times T$ and M_R is a property-preserving reduction of M, i.e. $(M, i) \models \mathbf{E}\neg f$ iff $(M_R, i) \models \mathbf{E}\neg f$.*

Proof. We follow the same reasoning as Theorem 4.2 in [1], which we only outline here. In [1], given a transition system G and a LTL property f, a Büchi automaton B which accepts the language $\mathcal{L}(\neg f)$ is constructed.[6] It is shown that $G \models \mathbf{A}f$ if and only if the intersection (i.e. product) A of G and B is empty, or equivalently if A does not contain any cycle, reachable from some initial state, that contains some accepting state. A reduced version A' of A is constructed by choosing at each step of the DFS a valid ample set. The conditions C_1 and C_2 are checked on G alone, while C_0 and C_3 are checked on the whole product. It is shown that A' corresponds to a product of a reduced system G_R and B such that G_R is a property-preserving reduction of G, i.e. $(G, i) \models \mathbf{E}\neg f$ iff $(G_R, i) \models \mathbf{E}\neg f$.

The ReachablePOR procedure follows the same process. It constructs a reduced version P_R of P by choosing at each step a valid ample set, i.e ample$((s_t, s))$ $= \{a \mid (s_t, s) \xrightarrow{a} (s'_t, s') \wedge a \in \text{ample}(s)\}$. By following the same reasoning as in [1] we can conclude that P_R is the product of a reduced system M_R and T such that M_R is a property-preserving reduction of M, i.e. $(M, i) \models \mathbf{E}\neg f$ iff $(M_R, i) \models \mathbf{E}\neg f$. □

Together with Theorem 1, the following lemma follows directly.

Lemma 4. $(P, i_P) \models \mathbf{E}_F\mathbf{G}$ *true if and only if* $(P_R, i_P) \models \mathbf{E}_F\mathbf{G}$ *true.*

Secondly, the following lemma establishes that P_I, which corresponds to the system explored by FairEH, preserves the properties of P_R, which corresponds to the system explored by ReachablePOR.

Lemma 5. $(P_R, i_P) \models \mathbf{E}_F\mathbf{G}$ *true if and only if* $(P_I, i_P) \models \mathbf{E}_F\mathbf{G}$ *true.*

Proof. We know that $\text{ftr}(P_R) \subseteq \text{ftr}(P_I) \subseteq \text{ftr}(P)$. Therefore, any fair path of P_R is also a fair path of P_I. Conversely, any fair path of P_I is also a fair path of P and therefore there exists a corresponding fair path in P_R by Lemma 4. □

We now get to the main result.

Theorem 4. *Given a model M, a property f and the product P of M and the tableau of $\neg f$, there exists a state $i \in I$ such that $(M, i) \models \mathbf{E}\neg f$ iff there exists a state $s_P \in S_P$ such that $(P, s_P) \models$* FairEH(ReachablePOR(P)).

[6] Although Theorem 4.2 in [1] considers only deterministic transition systems, both the theorem and its proof remain valid with non-deterministic transition systems. The proof remains exactly the same.

Proof. Let P_R and P_I be defined as previously. We have successively:

$$
\begin{aligned}
& \exists i \in I \cdot (M, i) \models \mathbf{E}\neg f \\
\Leftrightarrow\ & \exists i_P \in I_P \cdot (P, i_P) \models \mathbf{E}_F\mathbf{G}\, \mathit{true} && \text{(Theorem 1)} \\
\Leftrightarrow\ & \exists i'_P \in I_P \cdot (P_R, i'_P) \models \mathbf{E}_F\mathbf{G}\, \mathit{true} && \text{(Lemma 4)} \\
\Leftrightarrow\ & \exists i''_P \in I_P \cdot (P_I, i''_P) \models \mathbf{E}_F\mathbf{G}\, \mathit{true} && \text{(Lemma 5)} \\
\Leftrightarrow\ & \exists s_P \in S_R \cdot (P_I, s_P) \models \text{FairEH}(\text{Reachable}(P_I)) && \text{(Theorem 2)} \\
\Leftrightarrow\ & \exists s_P \in S_P \cdot (P, s_P) \models \text{FairEH}(\text{ReachablePOR}(P)) && \text{(Lemma 2)}
\end{aligned}
$$

□

Given any algorithm which constructs a valid POR-reduced reachable state set Reduced(M) of a transition system M, we can use that algorithm instead of ReachablePOR in our approach, checking the emptiness of FairEH(Reduced(P)). In the same way, other algorithms can be used to detect fair cycles, for instance the classical backward CTL model-checking algorithm can be used. Actually, these approaches are valid, and the demonstration of Section 4.3 remains the same.

5 Evaluation

We extended the Milestones model checker presented in [12] to support the method presented in this paper. Milestones is available under the GNU General Public License at http://lvl.info.ucl.ac.be/Tools/Milestones. Milestones allows us to describe concurrent systems and to verify LTL_X properties. It defines a language for describing transition systems. The design of the language has been influenced by the NuSMV language [13] and by the synchronization by rendezvous mechanism. Milestones detects fair cycles either with the classical backward fair CTL model-checking algorithm [9] (hereafter denoted as *bwd*), or with the forward approach described in Section 2.3 (denoted as *fwd*). The reachable state space can be generated using the ReachablePOR approach, as well as Alur's method mentioned in Section 3.2, or without any POR reduction. Together these offer $2 \times 3 = 6$ different modes of operation.

In order to assess the effectiveness and scalability of the approach proposed in this paper, we discuss two models which were translated both into the language of Milestones, NuSMV, and Spin. This section presents the models and the results we obtained. All the tests have been run on a 2.16 GHz Intel Core 2 Duo with 2 GB of RAM. We compare the verification performance between all six different modes. We also compare to NuSMV, which performs bwd without POR, and Spin which performs explicit model checking [4].

The first model is a variant of a producer-consumer system where all producers and consumers contribute on the production of every single item. The model is composed of $2 \times m$ processes: m producers and m consumers. Each producer and each consumer has two local transitions. The producers and consumers communicate together via a bounded buffer composed of eight slots. Each producer works locally on a piece p, then it waits until all producers terminate their task. Then, p is added to the buffer, and the producers start processing the next piece. When the consumers remove p from the buffer, they work locally on it. When all the consumers have terminated their local work, another piece can be removed from the buffer. The size of the reachable state space grows exponentially by a factor of 40 at each step of m.

Five properties have been analyzed on this model. For instance, P_3 states that at any time the producers will eventually add a piece into the buffer (satisfied), and P_4 states that the buffer will never overflow (unsatisfied). Figure 2 compares the times for the verification of the property P_3. Similar results have been obtained for the other four properties.

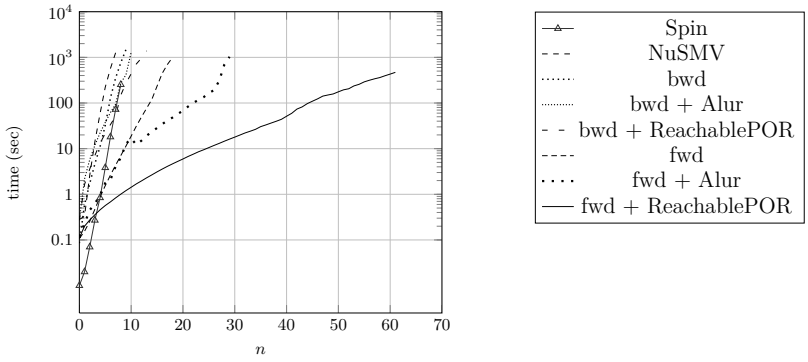

Fig. 2. Verification times for the Producer-Consumer property P3

The second model is a turntable model, described in [14]. The turntable system consists of a round turntable, an input place, an output place, n drills and a testing device. The turntable transports products in sequence between the different tools, where they are drilled and tested. The turntable has $n + 3$ slots that each can hold a single product. The original model had only one drill; we extended it to represent an arbitrary number of drills. The size of the reachable state space grows exponentially by a factor of 7 at each step of n.

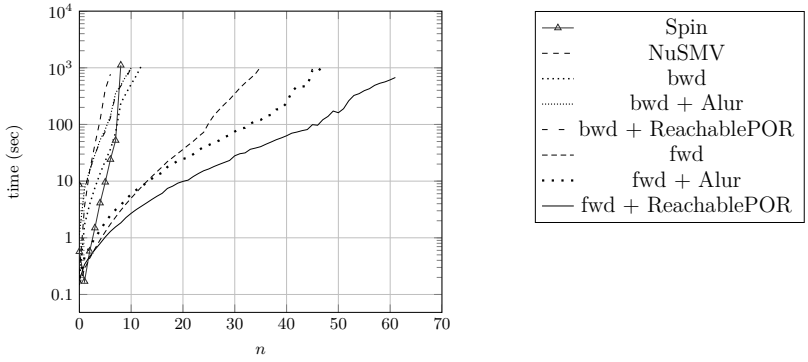

Fig. 3. Verification times for the Turntable property T_3

We have verified six properties on this system: four properties that the system satisfies, and two properties which are not fulfilled. For instance, the property T_3 states that if in the future there will be a piece which is not well drilled, the alarm will necessarily resonate. Here is the translation of this property in LTL: $\mathbf{G}\,[\mathbf{F}\ \texttt{a piece is not well drilled} \implies \mathbf{F}\ \texttt{an alarm is raised}]$. Figure 3 compares the times for the verification of the property T_3.

Table 1 compares the state space computed by the three forward methods (without POR, with Alur's method and with ReachablePOR), in terms of number of BDD nodes, number of states and computation time. It is quite interesting to note that while POR substantially decreases the number of reached states, the number of BDD nodes is increased (likely due to breaking some symmetry in the full state space). However, it still results in substantial speed improvements. We also notice that the state spaces produced by the Alur's method and the ReachablePOR method have approximately the same size.

Table 1. BDD size (in # nodes), state space size (in # states) and computation time (in seconds) for the P reachable state space computed either by the forward method without POR, or the forward method and Alur's approach, or the ReachablePOR method. "–" indicates that the computation did not end within 1000 seconds.

# drills	# nodes			# states			time (sec)		
	Fwd	Fwd + Alur	ReachPOR	Fwd	Fwd + Alur	ReachPOR	Fwd	Fwd + Alur	ReachPOR
1	197	318	282	86488	27408	26668	.11	.28	.17
2	442	880	744	521944	39188	38044	.17	.45	.23
4	975	2444	2021	$2.49 \times 10^{+7}$	62804	60796	.35	.90	.34
8	2040	5832	4717	$5.98 \times 10^{+10}$	111012	106300	1.08	2.59	.65
16	4168	17165	14126	$3.45 \times 10^{+17}$	207120	197308	5.13	9.34	1.49
32	8421	57467	47218	$1.15 \times 10^{+31}$	394208	379324	37.09	57.68	4.35
40	–	91214	75104	–	495668	470332	–	173.84	6.37
47	–	125194	103300	–	580484	549964	–	971.60	8.59
50	–	–	105844	–	–	584092	–	–	9.76
61	–	–	146912	–	–	709228	–	–	14.9

6 Related Work

Besides the approaches of Alur [7] and Improviso [6] on which this work is based (as presented in Section 3.2), several other approaches have been proposed that combine symbolic model checking and POR to verify different classes of properties.

This paper builds on our previous work combining POR and the forward state traversal approach to verify CTL_X properties[7] [12]. It remains to evaluate the compared merits of the two approaches for properties that can be expressed in both LTL_X and CTL_X. For conventional BDD-based model checking, experiments in [5] have found that, in the absence of POR, CTL verification tends to be faster.

In [15], we present another LTL_X model-checking algorithm which combines the Two-Phase algorithm and SAT-based bounded model checking (BMC). On

[7] CTL_X is the subset of the CTL logic without the next operator.

the property P_2 of the producer-consumer system of Section 5, the BMC algorithm of [15] takes approximately 68 minutes to find a counter-example of length 1,017, while the algorithm presented here takes only 314 milliseconds to show the violation.

In [16], Abdulla et al. present a general method for combining POR and symbolic model checking. Their method can check safety properties either by backward or forward reachability analysis. So as to perform the reduction, they employ the notion of commutativity in one direction, a weakening of the dependency relation which is usually used to perform POR. This approach deals both with backward and forward analysis but for reachability only, while we are able to check LTL_X properties but using only forward analysis.

In [17], Kurshan et al. perform partial-order reduction at compile time. The method applies static analysis techniques to discover local cycles and produce a reduced model, which can be verified using standard symbolic model checking. It could be interesting to investigate whether this kind of analysis could help in ensuring the cycle condition C_3 in our approach.

In [18], Holzmann performs a reduction of individual processes by *merging local transitions*. Then, the processes are put in parallel to be explicitly verified. Merging local transitions can be seen as a special application of partial reduction method. It avoids to create intermediate states between local transitions. Actually, all the transitions which are merged will be considered as safe transitions by our approach. Those transitions will be explored dynamically during `phase1`, i.e. when POR is applied. By contrast, the Holzmann algorithm removes them statically at compile time. We notice that our approach might visit more than once a state which will be removed by the Holzmann algorithm.

7 Conclusion

In this paper, we presented an improved BDD-based model-checking algorithm for verifying LTL_X properties on asynchronous models. Our approach combines the tableau-based reduction of LTL model-checking to fair-CTL from [5], forward state-traversal of fair-CTL formulæ from [8] used to detect fair cycles, and a symbolic partial-order reduction based on ImProviso [6] to reduce the forward state traversal.

We implemented the new algorithm in our existing model checker and observed on two case studies that our approach achieves a significant improvement in comparison to the tableau-based approach of [5] without POR, in both its backward and forward versions. It remains to confirm those results on a larger range of case studies and to compare with other methods and tools.

The reduced state set computed by ReachablePOR could as well be used in other BDD-based model-checking circumstances: as a filter during fixpoint computations in classical backward model-checking, or even to restrict the BDD of the transition relation before standard, non-POR techniques are applied. It would be interesting to compare the benefits of the reduction in the different approaches. For the latter case, however, the size of the BDD representing the transition relation of M_R could become unmanageable due to the loss of some symmetry.

References

1. Peled, D.: Combining partial order reductions with on-the-fly model-checking. Formal Methods in System Design **8**(1) (1996) 39–64
2. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem. Volume 1032 of Lecture Notes in Computer Science. Springer-Verlag (1996)
3. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, J.: Symbolic model checking: 10^{20} states and beyond. Information and Computation **98**(2) (1992) 142–170
4. Holzmann, G.J.: The model checker SPIN. IEEE Transactions on Software Engineering **23**(5) (1997)
5. Clarke, E.M., Grumberg, O., Hamaguchi, K.: Another look at LTL model checking. Form. Methods Syst. Des. **10**(1) (1997) 47–71
6. Lerda, F., Sinha, N., Theobald, M.: Symbolic model checking of software. In Cook, B., Stoller, S., Visser, W., eds.: Electronic Notes in Theoretical Computer Science. Volume 89., Elsevier (2003)
7. Alur, R., Brayton, R.K., Henzinger, T.A., Qadeer, S., Rajamani, S.K.: Partial-order reduction in symbolic state space exploration. In: Computer Aided Verification. (1997) 340–351
8. Iwashita, H., Nakata, T., Hirose, F.: CTL model checking based on forward state traversal. In: ICCAD '96: Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design, Washington, DC, USA, IEEE Computer Society (1996) 82–87
9. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. Mit Press (1999)
10. Gerth, R., Kuiper, R., Peled, D., Penczek, W.: A partial order approach to branching time logic model checking. Information and Computation **150**(2) (1999) 132–152
11. Nalumasu, R., Gopalakrishnan, G.: A new partial order reduction algorithm for concurrent system verification. In: CHDL'97: Proceedings of the IFIP TC10 WG10.5 international conference on Hardware description languages and their applications : specification, modelling, verification and synthesis of microelectronic systems, London, UK, UK, Chapman & Hall, Ltd. (1997) 305–314
12. Vander Meulen, J., Pecheur, C.: Efficient symbolic model checking for process algebras. In: 13th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2008). Volume 5596., LNCS (2008) 69–84
13. Cimatti, A., Clarke, E., Giunchiglia, F., Roveri, M.: NuSMV: a new symbolic model verifier. In: Proc. of International Conference on Computer-Aided Verification. (1999)
14. Bortnik, E.M., Trčka, N., Wijs, A., Luttik, B., van de Mortel-Fronczak, J.M., Baeten, J.C.M., Fokkink, W., Rooda, J.E.: Analyzing a χ model of a turntable system using spin, cadp and uppaal. J. Log. Algebr. Program. **65**(2) (2005) 51–104
15. Vander Meulen, J., Pecheur, C.: Combining partial order reduction with bounded model checking. In: Communicating Process Architectures 2009 - WoTUG-32. Volume 67 of Concurrent Systems Engineering Series., IOS Press (2009) 29 – 48
16. Abdulla, P.A., Jonsson, B., Kindahl, M., Peled, D.: A general approach to partial order reductions in symbolic verification (extended abstract). In: Computer Aided Verification. Volume 1427/1998., Springer Berlin / Heidelberg (1998) 379–390
17. Kurshan, R.P., Levin, V., Minea, M., Peled, D., Yenigün, H.: Static partial order reduction. In: TACAS '98: Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems, London, UK, Springer-Verlag (1998) 345–357
18. Holzmann, G.J.: The engineering of a model checker: the gnu i-protocol case study revisited. In: SPIN. (1999) 232–244