Efficient Symbolic Model Checking for Process Algebras*

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. Different approaches have been developed to mitigate the state space explosion of model checking techniques. Among them, symbolic verification techniques use efficient representations such as BDDs to reason over sets of states rather than over individual states. Unfortunately, past experience has shown that these techniques do not work well for loosely-synchronized models. This paper presents a new algorithm and a new tool that combines BDD-based model checking with partial order reduction (POR) to allow the verification of models featuring asynchronous processes, with significant performance improvements over currently available tools. We start from the ImProviso algorithm (Lerda et al.) for computing reachable states, which combines POR and symbolic verification. We merge it with the FwdUntil method (Iwashita et al.) that supports verification of a subset of CTL. Our algorithm has been implemented in a prototype that is applicable to action-based models and logics such as process algebras and ACTL. Experimental results on a model of an industrial application show that our method can verify properties of a large industrial model which cannot be handled by conventional model checkers.

1 Introduction

Model checking is a technique used to verify concurrent systems such as sequential circuit designs and communication protocols, by exhaustively exploring the state space of a finite-space description of the processes involved. The properties to be verified on such systems are typically expressed in (linear or branching) temporal logics such as LTL, CTL or CTL*, with different algorithms and complexities depending on the logic used. In particular, McMillan achieved a breakthrough with the use of symbolic representations based on the use of Ordered Binary Decision Diagrams (BDD) [1] to perform model checking of CTL, making it possible to verify systems with a very large number of states [2].

Unfortunately, the size of the state space to be explored is frequently prohibitive due, among other causes, to the modeling of concurrency by interleaving. The aim of partial order reduction (POR) techniques is to reduce the number of interleaving sequences that must be be considered. When a specification cannot

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

D. Cofer and A. Fantechi (Eds.): FMICS 2008, LNCS 5596, pp. 69–84, 2009.
© Springer-Verlag Berlin Heidelberg 2009

distinguish between two interleaving sequences that differ only by the order in which concurrently executed events are taken, it is sufficient to analyse one of them [3].

This paper presents a new algorithm and tool that combine BDD-based model checking with partial order reduction (POR) to allow the verification of models featuring asynchronous processes, with significant performance improvements over currently available tools. We start from the ImProviso algorithm of Lerda et al. [4] for computing reachable states, which combines POR and symbolic verification. We merge it with the FwdUntil method of Iwashita et al. [5] that supports verification of a subset of CTL. Our algorithm has been implemented in a prototype that is applicable to action-based models and logics such as LOTOS [6] and ACTL [7].

The main contributions of this paper are the FwdUntilPOR algorithm that combines POR and forward CTL model checking, a new symbolic model checker which implements this algorithm and is applicable to action-based models and logics, and an experimental evaluation of this algorithm and tool on a realistic-sized model.

The remainder of the paper is structured as follows. In Section 2, we introduce some background concepts, definitions and notations that are used throughout the paper. In Section 3, we review the Two-Phase algorithm for POR, and its symbolic incarnation in *ImProviso*. In Section 4, we discuss a forward approach to CTL symbolic model-checking, built around the *FwdUntil* operator. In Section 5, we present our own combination of ImProviso and FwdUntil, leading to the *ForwardUntil-POR* algorithm. In Section 6, we present a prototype of a new symbolic model checker implementing the FowardUntilPOR method. In Section 7, we present the results obtained by applying our method on a case study. Section 8 reviews related works. In Section 9, we give conclusions as well as directions for future work.

2 Background

2.1 Transitions Systems

We model the behaviour of a system as a set T of transitions over some state space S, where each transition is a binary relation over states. Formally, let AP be a set of atomic propositions. A *state transition system* is a four tuple $M = (S, T, S_0, L)$ where S is a finite set of states, $S_0 \subseteq S$ is the set of initial states, T is a set of transitions such that for each $\alpha \in T$, $\alpha \subseteq S \times S$ and $L : S \to 2^{AP}$ is a function that labels each state with the set of atomic propositions that are true in that state.

We write $s \xrightarrow{\alpha} s'$ for $(s, s') \in \alpha$. A transition α is *enabled* in a state s iff there is a state s' such that $s \xrightarrow{\alpha} s'$. We write $enabled(s)$ for the set of enabled transitions in s. A transition α is *deterministic* in a state s iff there is at most one s' such that $s \xrightarrow{\alpha} s'$.

We define the classical *pre-* and *post-image* of a set of states X over a set of transitions R, used in backward and forward state-space traversal respectively:

$$pre(R, X) = \{s' \in S | \exists s \in X, \alpha \in R \cdot s' \xrightarrow{\alpha} s\}$$
$$post(R, X) = \{s' \in S | \exists s \in X, \alpha \in R \cdot s \xrightarrow{\alpha} s'\}$$

2.2 Partial-Order Reduction

The goal of the partial-order reduction methods (POR) is to reduce the number of states explored by model-checking, by not exploring different equivalent interleavings of concurrent events. Naturally, these methods are best suited for strongly asynchronous programs. Interleavings which are required to be preserved may depend on the property to be checked.

Partial-order reduction is based on the notions of *independence* between transitions and *invisibility* of a transition with respect to a property. Two transitions are *independent* if they do not disable one another and executing them in either order results in the same state. A transition is *invisible* with respect to a property f when its execution from any state does not change the value of the atomic propositions in f. Intuitively, if two independent transitions α and β are invisible w.r.t. the property f that one wants to verify, then it does not matter whether α is executed before or after β, 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. In effect, POR amounts to exploring a reduced model $M' = (S', T', S_0, L)$ with $S' \subseteq S$ and for each $\alpha' \in T'$ there is an $\alpha \in T$ such that $\alpha' \subseteq \alpha$. In practice, classical POR algorithms [3,8] execute a modified depth-first search (DFS). At each state s, an adequate subset $ample(s)$ of the transitions 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, $ample(s)$ has to respect a set of conditions, based on the independence and invisibility notions previously defined. In some cases, all enabled transitions have to be explored. The following conditions are set forth in [9]:

C1. Only operations in $T - ample(s)$ that are independent from operations in $ample(s)$ can be executed before an operation from $ample(s)$ is executed.[1]
C2. For every cycle in the state graph, there is at least one state s in the cycle that is fully expanded, i.e. $ample(s) = enabled(s)$.
C3. If $ample(s) \neq enabled(s)$, then all transitions in $ample(s)$ are invisible.
C4. If $ample(s) \neq enabled(s)$, then $ample(s)$ contains only one transition that is deterministic in s.[2]

Conditions C1, C2 and C3 are sufficient to guarantee that the reduced model preserves properties expressed in LTL_X, i.e. linear temporal logic without the *next* operator (see e.g. [8]). Condition C4 is significantly more restrictive but is necessary to ensure preservation of branching temporal logics. In that case, [9] shows that there is a so-called *visible bisimulation* between the complete and reduced models, which ensures preservation of both state-based logics such as CTL^*_X (and thus CTL_X). This fits our purpose, since we address action-based models and logics.

Conditions C1 and C2 depend on the whole state graph and are not directly exploitable in a verification algorithm. Instead, one uses sufficient conditions,

[1] or equivalently in [9], No operation in $T - ample(s)$ that is dependent on an operation in $ample(s)$ can be executed before an operation from $ample(s)$ is executed.
[2] This is more general than [9], which assumes that all transitions are deterministic.

typically derived from the structure of the model description, to safely decide where reduction can be performed. In our case, we refine the model in terms of processes, with local and global variables and transitions and base the reduction on the notion of *safe local transition*, as developed in next section.

2.3 Process Models

We assume a process-oriented modeling language, where a concurrent system consists of a finite set of *processes* P accessing a set of *variables* V. Each process $p \in P$ maintains a set of local variables $V(p)$ that only it can access. All the processes also share a set of global variables, given by $V - \bigcup_{p \in P} V(p)$. The *global state* $s \in S$ consists of the values of all the variables; the *local state* of p, written $s(p)$, consists of the values of the local variables $V(p)$ in s. More generally, we write $s(W)$ for the values of a set of variables $W \subseteq V$ in s. The set of all possible states of the system is thus the cartesian product of the range of all variables, which we assume to be finite.

Transitions $\alpha \in T$ are characterized by the variables $V(\alpha)$ that they may modify or depend on: α is reducible to a relation α_0 on the range of $V(\alpha)$ such that $s \xrightarrow{\alpha} s'$ iff $s(V(\alpha)) \xrightarrow{\alpha_0} s'(V(\alpha))$ and $s(V - V(\alpha)) = s'(V - V(\alpha))$.

We consider that processes participate in all the transitions that affect their variables, and define the set of transitions of a process p as $T(p) = \{\alpha \in T | V(p) \cap V(\alpha) \neq \emptyset\}$, which divides into *local* transitions $T_l(p) = \{\alpha \in T(p) | V(\alpha) \subseteq V(p)\}$ and *shared* transitions $T_s(p) = T(p) - T_l(p)$. The *locally offered transitions* of p in s are the transitions of p that are allowed by $s(p)$, that is, $trans(p, s) = \{\alpha \in T(p) | \exists s' \cdot s'(p) = s(p) \wedge \alpha \in enabled(s')\}$. Note that a locally offered transition is not necessarily (globally) enabled, i.e. $trans(p, s) \not\subseteq enabled(p)$. A transition is *safe* iff it is deterministic and invisible (w.r.t. the property f being verified) in all states. We write $safe(p)$ for the set of safe local transitions of p, where $safe(p) \subseteq T_l(p) \subseteq T(p)$. It is easily seen that if $V(\alpha) \cap V(\beta) = \emptyset$, then α an β are independent. In particular, if $\alpha \in T_l(p)$ and $\alpha' \notin T(p)$ then α and α' are independent.

With this in hand, we can define sufficient conditions to apply partial-order reduction to our refined process-based models: essentially, as long as some process offers only safe local transitions, among which only one is enabled, then that transition can be selected as the ample set while meeting conditions C1 to C4. More formally, a process p is defined as *deterministic* in state s if the following conditions are met: (1) only safe local transitions are locally offered by p in s, that is, $trans(p, s) \subseteq safe(p)$, and (2) only one transition of p is enabled in s, that is, $\left|T(p) \cap enabled(s)\right| = 1$.

If p is deterministic in s, then s can be partially expanded by $ample(s) = T(p) \cap enabled(s) = \{\alpha\}$, where α is a single safe local transition of p, while maintaining the validity of verification (cf. Section 5).

3 The Two-Phase Approach to Partial Order Reduction

3.1 The Two-Phase Algorithm

The Two-Phase algorithm (presented in [10]) is a variant of the classical DFS algorithm with POR of [3,8]. It alternates between two distinct phases:

- Phase 1 expands only deterministic states considering each process at a time, in a fixed order. As long as a process is deterministic, the single transition that is enabled for that process is executed. Otherwise, the algorithm moves on to the next process. After expanding all processes, the last reached state is passed on to phase 2.
- Phase 2 is simple. It performs a full expansion of the states resulting from the phase 1, then applies phase 1 to the reached states.

In order not to postpone a transition indefinitely, for each cycle in the reduced state space, at least one state in this cycle must be fully expanded. Such an indefinite postponing can only arise within phase 1, and is handled by detecting cycles within the current phase 1 expansion and switching to a phase 2 expansion when they occur.

As shown in [10], the Two-Phase algorithm produces a reduced state space which is stuttering equivalent to the whole one, and therefore preserves CTL_X properties [9].

3.2 ImProviso

[4] proposes `ImProviso`, a symbolic version for computing the reachable states of the Two-Phase algorithm. It efficiently combines the advantages of POR and symbolic methods. Classical symbolic model checking algorithms use a single transition relation (partitioned or not) to carry out the required computation on the state space. On the other hand, the `ImProviso` method defines $n + 1$ transition relations, where n is the number of processes in the system. One is the full transition relation T used in phase 2, and the others contain only the safe local transitions from deterministic states, denoted as $T_1(p)$ of each process p, used in phase 1.

Contrary to the nested DFS preferred by classical POR methods, the symbolic methods amount to a *breadth-first* search (BFS). It is thus harder to detect cycles within phase 1 in the symbolic case, and that detection is required to maintain the validity of the algorithm. ImProviso adopts a pessimistic approach: at each step during phase 1, it is assumed pessimistically that any previously expanded state that is reached again might close a cycle, although these occurrences might actually be on different execution paths. This over-approximation guarantees that all cycles are correctly identified, but possibly needlessly reduces the number of states where phase 1 can be applied. This is the key justification for basing ImProviso on the Two-Phase algorithm, as this limits the need for cycle detection to each single execution of phase 1, as opposed to the whole exploration for more traditional POR approaches.

The original ImProviso algorithm is not detailed here but is very similar to the `FwdUntilPOR` algorithm of Section 5, which is based on ImProviso.

4 Forward Symbolic Model-Checking of CTL

In [5], Iwashita et al. present a model checking algorithm based on forward state traversal, which is shown to be more effective than backward state traversal in

many situations. Forward traversal is applicable only to a subset of CTL, but can be combined with backward traversal for the rest of the formulæ. In the following sections we combine this algorithm with ImProviso in order to extend the advantages of partial-order reduction in symbolic methods from reachability properties to CTL properties.

The semantics of a CTL formula f is defined as a relation $s \models f$ over states $s \in S$. In this paper we define the *language* of f as $\mathcal{L}(f) = \{s \in S | s \models f\}$. In the sequel we assimilate temporal logic formulæ f to the set of states $\mathcal{L}(f)$ that they denote, for the sake of simplifying the notations. In particular, we denote set-based computations as the formula-based fixpoints that they compute.

Given a formula f and initial conditions h_0, conventional BDD-based symbolic model-checking can be described as evaluating $\mathcal{L}(f)$ over the sub-formulas of f in a bottom-up manner, and checking whether $\mathcal{L}(h_0) \subseteq \mathcal{L}(f)$. This can be expressed as checking whether $h_0 \Rightarrow f$, or equivalently, whether $h_0 \wedge \neg f = false$. The evaluation of (future) CTL operators in f results in a backward state-space traversal of the model.

[5] introduces forward exploration by transforming a property $h \wedge op(f) = false$ into equi-satisfiable one $op'(h) \wedge f = false$, 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. In general, h is then a past-CTL formula. The following (past-temporal) operations over formulæ are defined

$$FwdUntil(h, f) = \mu Z.[h \vee post(Z \wedge f)]$$
$$EH(h) = \nu Z.[h \wedge post(Z)]$$
$$FwdGlobal(h, f) = EH(FwdUntil(h, f) \wedge f)$$

$FwdUntil(h, f)$ computes states s that can be reached from h within f (except for s itself), and $EH(h)$ computes states reachable from a cycle, all within h. On this basis, the following equivalences are established:

$$h \wedge EXf = false \iff post(h) \wedge f = false$$
$$h \wedge E[g\ U f] = false \iff FwdUntil(h, g) \wedge f = false$$
$$h \wedge EGf = false \iff FwdGlobal(h, f) = false$$

The transformation process starts from $h_0 \wedge \neg f = false$, where h_0 is the initial conditions and $\neg f$ is the (suitably re-written) negation of the property to be verified. The equivalences above are applied recurrently until the right part cannot be reduced further, either because all temporal operators have been eliminated or because no rule applies to those remaining. Disjunctions in f can also be handled by case-splitting. Given the final $h \wedge f = false$, $\mathcal{L}(h)$ is evaluated using forward traversal, and the resulting set of states is used as the new initial conditions for a classical, backward model-checking of the remaining f.

By using these equivalences, it is possible to replace an outermost EX, EU or EG operator in f with a forward traversal operator in h. For instance, one can derive the following equivalence:

$$h_0 \Rightarrow AG(req \rightarrow AFack) \iff$$
$$FwdGlobal((FwdUntil(h_0, true) \wedge req), \neg ack) = false$$

Unfortunately, it is not possible to achieve the complete conversion of all CTL properties by applying this method. For instance, the following property is not fully transformable, because the negation in the right term cannot be eliminated:

$$p_0 \Rightarrow AG\ EFa \iff FwdUntil(p_0, true) \wedge \neg EFa = false$$

Informally, reduction is possible for a restricted fragment of *universal CTL*, where temporal operators do not appear in the context of disjunctions nor on the left side of *Until* operators.[3]

5 Forward Model Checking with Partial Order Reduction

In this section we bring together the POR approach of `ImProviso`, presented in Section 3.2, and the forward model checking approach of Section 4. The key element is to define a new algorithm *FwdUntilPOR*, which applies ImProviso's principles to perform POR during the forward exploration of $FwdUntil$. The algorithm for $FwdUntilPOR(h, f)$ is given in Listing 1.1, and is based on ImProviso's algorithm in [5].

Given two visible constraints h and f, the `FwdUntilPOR` algorithm computes the set of states of the reduced state space belonging to a path of the form $s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_{n-1} \rightarrow s_n$ where $s_0 \models h$ and $\forall i \in \{0, \ldots, n-1\} : s_i \models f$. `T` is the global transition relation of the model, and `T1[p]` contains safe local transitions of process p, and only from such states where p is deterministic. The `deadState(T, X)` function computes the states of X that have no enabled transition from T, i.e. $deadStates(T, X) = \{s \in X | \neg \exists s' \in S, \alpha \in T \cdot s \xrightarrow{\alpha} s'\}$.

The `FwdUntilPOR` procedure initializes the global variables and 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 `phase1` procedure performs the first phase, consisting of partial expansion of deterministic transitions. It is composed of two nested loops. The outer one (lines 22 – 33) expands all processes in a given order. The inner one (lines 25 – 31) expands all deterministic transitions of the current process, from states satisfying f, until no more new states can be found.

The `stack` variable contains all the states which have already been reached during the current run of `phase1`. `cycleApprox` over-approximates the set of states closing a cycle. It contains all the states which have been reached twice during the current run of phase 1. The `dead` variable gathers all the states with no outgoing deterministic transition for the current process; those states are added back to the current frontier when moving to the next process (line 32).

The original ImProviso algorithm defines an additional outermost loop in phase 1, which guarantees that a state is passed from phase 1 to phase 2 only if

[3] *Universal CTL* is the fragment of CTL such that negated normal forms contain only universal path quantifiers (AX, AU, AG, AF). As detailed in [5], if the model has a single initial state ($\mathcal{L}(h_0) = \{s_0\}$), then the validity of f can also be phrased as $h_0 \wedge f \neq false$, and existential formulæ can be handled as well.

it has no enabled deterministic transition for any process. This is useful in the case where a local transition from one process can activate another process, as for example posting a message to a channel that can be subsequently received. In our case, this fixpoint calculation is not needed because by construction our notions of local transition and deterministic process do not allow this kind of situation. It would easily be added back if it were to become useful.

The `phase2` procedure performs a full expansion of the states of the current frontier satisfying the constraint f.

```
1  global T        //total transition relation
2  global T1[1..n] // safe local transitions of each process
3
4  global f        // constraints f
5  global frontier // current frontier
6  global visited  // visited states
7
8  procedure FwdUntilPOR(inH, inF)
9    frontier := inH
10   f := inF
11   visited := inF
12   while (frontier != {}) {
13     phase1()
14     phase2()
15   }
16 }
17
18 procedure phase1() {
19   local cycleApprox := {}
20   local stack := frontier
21
22   foreach(p : Processes) {
23     local dead := {}
24     local image := post(T1[p], frontier and f)
25     while ((image - stack) != {}) {
26       dead := dead or deadStates(T1[p], frontier)
27       cycleApprox := cycleApprox or (image and stack)
28       stack := stack or image
29       frontier := image - stack
30       image := post(T1[p],frontier and f)
31     }
32     frontier := frontier or dead
33   }
34   frontier := frontier or cycleApprox
35   visited := visited or stack
36 }
37
38 procedure phase2() {
39   local image := post(T, frontier and f)
40   frontier := image - visited
41   visited :=  visited or image
42 }
```

Listing 1.1. FwdUntilPOR algorithm

Correctness. A full formal proof of correctness of the proposed approach is beyond the scope of this paper. The validity of the overall verification technique depends on a number of components, a number of which are inherited from existing techniques and tools. One important point is the validity of the ample sets used for POR, which we address in more details first, based on the definitions of Section 2. The following lemma will be useful in the main proof:

Lemma 1. *If a process p is deterministic in a state s with* $enabled(s) \cap T(p) = \{\alpha\}$ *and there is a transition* $\alpha' \neq \alpha$ *such that* $s \xrightarrow{\alpha'} s'$ *then* $s'(p) = s(p)$ *and p is deterministic in* s' *with* $enabled(s') \cap T(p) = \{\alpha\}$.

Proof. We have that $\alpha' \in enabled(s)$. Since $enabled(s) \cap T(p) = \{\alpha\}$, $\alpha' \notin T(p)$ and $s(p) = s'(p)$. $trans(p, s)$ only depends on $s(p)$ so $trans(p, s) = trans(p, s')$ and $enabled(s') \cap T(p) \subseteq trans(p, s') \subseteq safe(p)$ so $enabled(s') \cap T(p) = enabled(s) \cap T(p)$. □

Then we come to the main result:

Theorem 1 (Correctness of Ample Sets). *Given a state s, if* $trans(p, s) \subseteq safe(p)$ *and* $\left|T(p) \cap enabled(s)\right| = 1$, *then* $ample(s) = T(p) \cap enabled(s) = \{\alpha\}$ *is a valid ample set for s.*

Proof. This requires checking that $ample(s)$ meets conditions C1 to C4 of Section 2.2.
C1 is proved by contradiction. Suppose that there is a path $s = s_0 \xrightarrow{\alpha_1} s_1 \ldots \xrightarrow{\alpha_m} s_m \xrightarrow{\alpha'} s'$, where all α_i are independent from α and α' is dependent on (but different from) α. By applying lemma 1 inductively in $s, s_1, \ldots, s_{m-1}$, we get that $s(p) = s_1(p) = \cdots = s_m(p)$, and p is deterministic in s_m with $T(p) \cap enabled(s_m) = \{\alpha\}$, and therefore $\alpha' \notin T(p)$. If $\alpha' \notin T(p)$, then since $\alpha \in safe(p)$, α and α' are independent, a contradiction.
C2 is satisfied because in every cycle in the reduced graph, at least one state is fully expanded in phase 2 of the algorithm. If the cycle contains at least one states s where no process is deterministic, then that state will not be expanded in phase 1. If the cycle is composed exclusively of states where a process is deterministic, then the algorithm guarantees conservatively that the loop is detected in phase 1 and one state is deferred to phase 2 (see the `cycleApprox` variable, line 27).
C3 is satisfied because either $ample(s) = enabled(s)$ or $ample(s) \subseteq safe(s)$ by construction.
C4 is satisfied by construction of $ample(s)$. □

The validity of the overall technique follows, based on the following arguments:

1. The equivalence relations between backward and forward operators of section 4 are valid, in the sense that the transformed formulas, introducing forward traversal where feasible, are satisfiable if and only if the original formulas are satisfiable. This result is assumed from [5].
2. Classical backward BDD-based model-checking, and in particular the reduction of CTL to EX, EU and EG operators, is valid. This is a well-established result, see e.g. [8].
3. The single enabled transition of a deterministic process p in a state s, as defined in Section 2.3, is indeed a valid ample set for s (Theorem 1).
4. Assuming § 3, `FwdUntilPOR` performs a valid exploration of a subset of the behaviours explored by $FwdUntil$, reduced through POR. This is verified by checking that `FwdUntilPOR` is a valid symbolic implementation of the Two-Phase algorithm based on deterministic processes as defined in Secton 2.3 in the same way as the original ImProviso, but adapted for adapted for restricting the exploration to paths of the form $s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_{n-1} \rightarrow s_n$ where $s_0 \models h$ and $\forall i \in \{0, \ldots, n-1\} : s_i \models f$.

5. The overall, combined forward and backward exploration is a valid model-checking technique for CTL_X. This combines the validity of the transformation (§ 1), of classic CTL model-checking (§ 2) and of the POR-reduced exploration by `FwdUntilPOR` (§ 4), combined with the observation in Section 2.2 that POR reduction respecting conditions C1 to C4 preserves CTL_X properties.

6 Implementation

We have developed the `FwdUntilPOR` method in a new symbolic model checker. It allows to describe concurrent systems and to verify CTL properties, and action-based extension thereof, on these models.

Our prototype has been implemented with the Scala language [11]. Scala is a multi-paradigm programming language, fully interoperable with Java, designed to integrate features of object-oriented programming and functional programming. Scala is a pure object-oriented language in the sense that every value is an object. Scala is also a functional language in the sense that every function is a value. To obtain better performance, our model checker uses a BDD package, named BuDDy[12], written in C.

The model checker defines a language for describing transitions systems. The design of the language has been influenced on the one hand by process algebras and on the other hand by the NuSMV language [13]. A model of a concurrent system declares a set of global variables, a set of shared actions and a set of processes. A process p declares a set of local variables, a set of local actions and the set of shared actions which p is synchronized on. Each process has a distinguished local program counter variable *pc*. For each value of *pc*, the behavior of a process is defined by means of a list of *action-labelled guarded commands* of the form $[\alpha]$ `c` $\rightarrow$ `u`, where α is an action, `c` is a condition on variables and `u` is an assignment updating some variables. *Shared* actions are used to define synchronization between the processes. A *shared* action occurs simultaneously in all the processes that share it, and only when all enable it.

Properties are expressed in an action-based extension of CTL similar to ACTL [7] [4]. These properties can be checked with three techniques: the backward traversal method, the FwdUntil method (Section 4) and the FwdUntilPOR method (Section 5). If the FwdUntilPOR is applied, some syntactic restrictions are imposed in order to satisfy the conditions allowing the POR. For instance, the propositions allowed in the CTL_X properties can only concern the global variables so as to satisfy the visibility condition. For each p, safe commands are determined at compile time and combined into $T1(p)$. A guarded command is considered as safe if it contains only local variables and actions. For each *pc*, a list of guarded command *gcs* is considered as safe, if all elements of *gcs* are safe and all of them are mutually exclusive .

Ordering of the Variables. BDDs require a fixed ordering among the boolean variables used to represent the system. The size of BDDs, and therefore the

[4] Specifically, we use Action-Restricted CTL (ARCTL) [14], which associates actions to path quantifiers rather than temporal operators.

performance of BDD-based model-checking, strongly depends on this ordering. For instance, the size of the BDD representing a n-bit comparator ($x_1 = x'_1 \wedge \ldots \wedge x_n = x'_n$) can go from $3*n+2$ nodes with the order $x_1 \prec x'_1 \prec \ldots \prec x_n \prec x'_n$ to $3 * 2^n - 1$ nodes with the order $x_1 \prec \ldots \prec x_n \prec x'_1 \prec \ldots \prec x'_n$. In general, finding the best variable ordering is a NP-complete problem. The topic has been intensively studied and several heuristics have been developed for finding a good ordering between variables.

This research has mostly focused on ordering variables within a state, but there is also an opportunity for optimizing the order of variables used for the transitions relation $T(s, \alpha, s')$, which ranges over sets of boolean variables, s, α, s' respectively. If $s \xrightarrow{\alpha} s'$, s is named the *source state* and s' is named the *target state*. In order to represent the relation T as a boolean function $T(s, \alpha, s')$, three sets of boolean variables are used: $s = s_1, s_2, \ldots, s_m$, $\alpha = \alpha_1, \alpha_2, \ldots, \alpha_n$ and $s' = s'_1, s'_2, \ldots, s'_m$. An intuitive approach would be to start with α, followed by s, then s'. In the case of strongly asynchronous systems, this approach leads to an explosion of the BDD size [15]. A better solution is proposed in [15] for asynchronous models such as those obtained from process algebra specifications. The action variables are encoded first, followed by an "interlacing" between the source variables and the target variables: $a_1 \prec a_2 \prec \ldots \prec a_n \prec s_1 \prec s'_1 \prec s_2 \prec s'_2 \prec \ldots \prec s_m \prec s'_m$.

Experimental results show that the resulting BDDs only grow linearly in the number of asynchronous components. Intuitively, the ordering works well due to the fact that, in the case of asynchronous processes, most of the time a small number of processes proceed, so only the variables of those processes change while most variables remains the same (i.e. $s_i = s'_i$). These constraints are more efficiently encoded in the BDD, if s_i and s'_i are next to each other in the ordering, similarly to the n-bit comparator example above.

Table 1 compares the transition relation BDD size and the time between the intuitive and the interlaced ordering, based on the case study of Section 7. The size of the model is driven by the parameter #drill, and the time corresponds to verifying property $p6$. It confirms the much reduced growth rate of the interlaced ordering, allowing a much larger number of components to be added.

Table 1. Size of the transition relation BDD (in # nodes) and verification time (in seconds) for property p6 of the Turntable case study, using interlaced vs. non-interlaced orderings, — correspond to memory exhausted (2 GB)

# drills	# vars	interlaced		non-interlaced	
		size	time	size	time
1	24	1 543	.041	153 056	6.222
2	31	1 913	.070	4 051 081	409.078
3	38	2 307	.114	—	—
20	157	12 184	4.436	—	—
40	297	31 572	30.884	—	—

7 Case Study

In order to assess the effectiveness of our method, we applied it to a turntable model which is described in [16,17]. For initial experiments, we modelled the system in the NuSMV language. We then converted the language of our prototype. We compared performance of verification using three methods: classical backward, FwdUntil and FwdUntil with POR, as well as with the NuSMV tool and with the non-symbolic tool from the CADP toolset: Evaluator [18]. This section presents the system and the results we obtained. All the test have been run on a 2,16 GHz Intel Core 2 Duo with 2 GB of RAM memory.

The turntable system consists of a round turntable, n drills and a testing device, as illustrated in Figure 1. The turntable transports products between the drills, the testing device and input and output positions. The drills bore holes in the products. After being drilled, the products are delivered to the tester, where the depth of the holes is measured, since it is possible that drilling went wrong. 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. Although a turntable with 40 drills is a bit artificial, it gives a model of a fairly large realistic size.

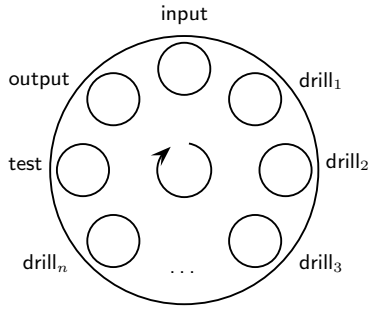

Fig. 1. Turntable System

The original model was described in LOTOS, a formal specification technique based on process algebras [6]. First we translated the LOTOS model into a NuSMV model. The difficult part of this task comes from the fact that LOTOS and NuSMV do not have the same concurrency model. LOTOS has a more expressive synchronization mechanism. The conversion in the prototype language was easier because our language is inspired by languages like LOTOS.

We have verified 13 properties from [17] expressed as a regular alternation-free μ-calculus formulae [19], here labelled $p1$ to $p13$. $p1$ to $p6$ are safety properties and $p7$ to $p13$ are liveness properties. For instance, the safety property $p6$ states that if a piece is well drilled, no alarm will be raised during the next cycle. The liveness property $p11$ states that each piece will be removed from the turntable after it is tested.

```
P6 :[true*.INF !TESTED !TRUE.(not INF !TURNED)*.INF !TURNED.(not INF !TURNED)*.ERR] false
P11:[true*.INF !TESTED.*]inev(not CMD !TURN,CMD !TURN,inev(not CMD !TURN,REQ !REMOVE.*,true))
```

For 11 of the 13 properties, the FwdUntilPOR method outperforms the classical backward CTL algorithm. However for $p1$ and $p2$, the classical method is approximately 30 times faster than the FwdUntilPOR algorithm. Currently, we do not

Table 2. Verification times (in seconds) for properties p6 and p11 of the Turntable model, using NuSMV, CADP and our prototype using standard backward exploration (Bwd), FwdUntil (Fwd) and FwdUntilPOR (Fwd+POR). — indicates that the computation did not end within 5 hours.

# drill	property p6					property p11			
	NuSMV	CADP	Bwd	Fwd	Fwd+POR	CADP	Bwd	Fwd	Fwd+POR
1	2.001	2.770	.041	.022	.055	3.640	.037	.076	.097
2	36.400	5.480	.070	.043	.082	19.350	.062	.132	.139
3	578.500	81.260	.114	.074	.106	335.130	.094	.167	.185
4	6617.000	13393.630	.157	.104	.144	19031.340	.132	.244	.245
10	—	—	.721	.875	.335	—	.663	1.064	.589
20	—	—	4.436	9.698	.838	—	6.112	9.187	1.496
30	—	—	13.842	31.295	1.475	—	19.412	25.520	2.780
40	—	—	30.884	80.519	2.499	—	40.304	67.761	4.355

have an explanation for such a difference which is left for further investigation. On this model, the FwdUntil method is less efficient than the classical method, taking exception from the general observation reported in [5].

Table 2 shows the time for the verification of the properties $p6$ and $p11$. If the turntable comprises 40 drills, $p6$ properties is checked approximately 12 times faster and $p11$ is checked approximately 9 times faster with the FwdUntilPOR method than with the backward method. The causes for the huge increase for CADP between 3 and 4 drills remain to be investigated.

Table 3 compares the time needed for computing the reachable state space between NuSMV and our prototype. We notice that NuSMV cannot handle a model beyond 4 drills, while our prototype can still easily handle up to 40 drills. It is quite interesting to note that while POR increases the number of BDD nodes for the reduced state space (likely due to breaking some symmetry in the full state space), it results in substantial speed improvements. One possible explanation for the huge difference between NuSMV and our prototype is that the modeling language of NuSMV, as opposed to that of our prototype, does not support synchronization through shared actions, and so the translation of the original LOTOS model is more convoluted and less straightforward. This issue deserves further investigation.

Table 3. BDD size (in # nodes), state space size (in # states) and computation time (in seconds) for the reachable state space of the Turntable model in NuSMV vs. our prototype, both with full and POR exploration (i.e. using the ImProviso algorithm). — indicates that the computation did not end within 5 hours.

# drills	# nodes			# states			time		
	NuSMV	proto full	proto POR	NuSMV	proto full	proto POR	NuSMV	proto full	proto POR
1	2660	131	196	10 068	9 084	5572	1.009	.289	.149
2	12888	274	488	170 058	146 784	7948	32.000	.297	.153
3	64616	428	869	$\approx 10^6$	$\approx 10^6$	10324	553.400	.401	.181
4	244967	582	1286	$\approx 10^7$	$\approx 10^7$	12700	4 784.600	.545	.228
10	—	1506	4709	—	$\approx 10^{11}$	26956	—	1.892	.528
20	—	3046	14039	—	$\approx 10^{20}$	50716	—	7.984	1.147
40	—	6126	49649	—	$\approx 10^{37}$	98236	—	63.721	3.434

8 Related Work

In [20], Alur et al. transform an explicit model checking algorithm performing partial order reduction and able to check invariance of local properties. They start from a DFS algorithm to obtain a modified BFS algorithm. Both expand an ample set of transitions in each step. In order to detect the cycles, they assume pessimistically that each previous expanded state might close a cycle. By contrast, `ImProviso` makes a smaller over-approximation of such states because it only needs to consider cycles formed exclusively by deterministic transitions. Consequently it looks for possible cycles only with respect to states visited during phase 1.

In [21], 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. It can be applied either to finite or infinite state spaces. One difference between this approach and ours is the checked properties. This approach deals both with backward and forward reachability analysis, while we are able to check a subset of CTL_X properties using only forward analysis.

In [22], Kurshan et al. introduce a partial order reduction algorithm based on static analysis. They notice that each cycle in the state space is composed of some local cycles. The method performs a static analysis of the checked model so as to discover local cycles and set up all the reductions at compile time. The reduced state space can be handled with symbolic techniques. Their approach differs from ours in that it performs the reduction at compile time. On the contrary our approach performs the reduction at run time. Lerda et al. suggest that the Improviso method is more efficient than the one introduced by Kurshan et al.[4]. However, we think that it still would be interesting to see how both approaches can benefit from each other.

In [23], Fantechi et al. present SAM, a symbolic model checker based on BSP (Boolean symbolic programming), a programming language aimed at defining computations on boolean functions. SAM takes as input an LTS s and a (possibly recursive) μ-ACTL formula p, and transforms both into BSP programs, which are then compiled into a sequence of calls to BDD primitives. Checking that s verifies p reduces to checking whether the boolean function "$tr(s) \Rightarrow tr(p)$" is a tautology. SAM is able to check μ-ACTL formulae which is a richer language that the one of our prototype, but does not address performance optimizations such as partial-order reduction.

9 Conclusion and Perspectives

In this paper, we introduced the FwdUntilPOR algorithm that combines two existing techniques to provide efficient symbolic model checker of CTL on asynchronous models. The first technique is the ImProviso algorithm which efficiently merges POR and symbolic methods. The second technique is the forward symbolic model checking approach applicable to a subset of CTL.

We also implemented the FwdUntilPOR algorithm in a new symbolic model checker. Its input syntax supports actions-based models and logics. We show on a realistic-sized case study that our method achieves a strong improvement in comparison to the classical backward algorithm, in the majority of cases.

Although it is usually considered that symbolic model checking is inadequate for loosely-synchronized models, our results show that with appropriate optimization this approach might in fact be quite effective to tackle the state space explosion problem. On this basis, we plan to develop our approach and our prototype in a number of ways:

- We plan to extend the FwdUntilPOR method for applying POR to a larger subset of CTL. We will investigate how the approach of [21] can be extended for combining the classical backward symbolic model checking algorithms and POR.
- We need to explore how it is possible to compute a better approximation of the deterministic states. There exists a large body of literature on this subject.
- We need to extend our prototype by adding generation of counter-examples for failed properties Another source of improvement can come from applying traditional partitioning techniques to BDDs representing the transition relations. Besides, it would be convenient to accept or translate, as input, a popular language such as LOTOS in order to exploit the numerous case studies available in this formalism.
- As observed in our case study, for some properties the FwdUntilPOR method performs much worse than the standard backward model checking. We will investigate this issue, in order to try to characterize the classes of properties where this happens and to investigate whether our algorithm can be improved to better handle those cases.

References

1. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers 35(8), 677–691 (1986)
2. Burch, J., Clarke Jr., E., McMillan, K., Dill, D., Hwang, L.: Symbolic Model Checking: 10^{20} States and Beyond. In: Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science, Washington, D.C., pp. 1–33. IEEE Computer Society Press, Los Alamitos (1990)
3. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems. LNCS, vol. 1032. Springer, Heidelberg (1996)
4. 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, vol. 89. Elsevier, Amsterdam (2003)
5. Iwashita, H., Nakata, T., Hirose, F.: CTL model checking based on forward state traversal. In: ICCAD 1996: Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design, Washington, DC, USA, pp. 82–87. IEEE Computer Society, Los Alamitos (1996)
6. ISO/IEC: LOTOS — a formal description technique based on the temporal ordering of observational behaviour. International Standard 8807, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève (1988)

7. De Nicola, R., Vaandrager, F.: Action versus state based logics for transition systems. In: Guessarian, I. (ed.) LITP 1990. LNCS, vol. 469, pp. 407–419. Springer, Heidelberg (1990)
8. Clarke Jr., E., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press, Cambridge (1999)
9. Gerth, R., Kuiper, R., Peled, D., Penczek, W.: A partial order approach to branching time logic model checking. Inf. Comput. 150(2), 132–152 (1999)
10. Nalumasu, R., Gopalakrishnan, G.: A new partial order reduction algorithm for concurrent systems. In: Delgado Kloos, C., Cerny, E. (eds.) Hardware Description Languages and their Applications (CHDL 1997), Toledo, Spain. Chapman and Hall, Boca Raton (1997)
11. Odersky, M., Spoon, L., Venners, B.: Programming in Scala, a comprehensive step-by-step guide. PrePrint^{TM} Edition, Version 3 (May 2008)
12. Lind-Nielsen, J.: Buddy - a binary decision diagram package (June 10, 2008), `http://vlsicad.eecs.umich.edu/BK/Slots/cache/www.itu.dk/research/buddy/index.html`
13. Cimatti, A., Clarke, E., Giunchiglia, F., Roveri, M.: NuSMV: a new symbolic model verifier. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 495–499. Springer, Heidelberg (1999)
14. Pecheur, C., Raimondi, F.: Symbolic model checking of logics with actions. In: Edelkamp, S., Lomuscio, A. (eds.) MoChArt IV. LNCS, vol. 4428, pp. 113–128. Springer, Heidelberg (2007)
15. Enders, R., Filkorn, T., Taubner, D.: Generating BDDs for symbolic model checking in CCS. Distrib. Comput. 6(3), 155–164 (1993)
16. Bortnik, E., Trčka, N., Wijs, A.J., Luttik, B., van de Mortel-Fronczak, J., Baeten, J.C.M., Fokkink, W.J., Rooda, J.: Analyzing a χ model of a turntable system using SPIN, CADP and UPPAAL. Journal of Logic and Algebraic Programming 65(2), 51–104 (2005)
17. Mateescu, R.: 5. IC2 treatise. Systèmes temps réel 1 - techniques de description et de vérification. Lavoisier, 151–180 (2006)
18. Garavel, H.: Open/Cæsar: An open software architecture for verification, simulation, and testing. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, pp. 68–84. Springer, Heidelberg (1998)
19. Mateescu, R., Sighireanu, M.: Efficient on-the-fly model-checking for regular alternation-free mu-calculus. Science of Computer Programming 46(3), 255–281 (2003)
20. 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, pp. 340–351 (1997)
21. 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, pp. 379–390 (1998)
22. Kurshan, R.P., Levin, V., Minea, M., Peled, D., Yenigün, H.: Static partial order reduction. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, pp. 345–357. Springer, Heidelberg (1998)
23. Fantechi, A., Gnesi, S., Mazzanti, F., Pugliese, R., Tronci, E.: A symbolic model checker for ACTL. In: Hutter, D., Traverso, P. (eds.) FM-Trends 1998. LNCS, vol. 1641, pp. 228–242. Springer, Heidelberg (1999)