# MC-SOG: An LTL Model Checker Based on Symbolic Observation Graphs[⋆]

Kais Klai[1] and Denis Poitrenaud[2]

[1] LIPN, CNRS UMR 7030
Université Paris 13
99 avenue Jean-Baptiste Clément
F-93430 Villetaneuse, France
`kais.klai@lipn.univ-paris13.fr`
[2] LIP6, CNRS UMR 7606
Université P. et M. Curie
104, avenue du Président Kennedy
75016 Paris, France
`Denis.Poitrenaud@lip6.fr`

**Abstract.** Model checking is a powerful and widespread technique for the verification of finite distributed systems. However, the main hindrance for wider application of this technique is the well-known state explosion problem. During the last two decades, numerous techniques have been proposed to cope with the state explosion problem in order to get a manageable state space. Among them, *on-the-fly* model-checking allows for generating only the "interesting" part of the model while *symbolic model-checking* aims at checking the property on a compact representation of the system by using Binary Decision Diagram (BDD) techniques. In this paper, we propose a technique which combines these two approaches to check LTL\X state-based properties over finite systems. During the model checking process, only an abstraction of the state space of the system, namely the *symbolic observation graph*, is (possibly partially) explored. The building of such an abstraction is guided by the property to be checked and is equivalent to the original state space graph of the system w.r.t. LTL\X logic (i.e. the abstraction satisfies a given formula $\varphi$ iff the system satisfies $\varphi$). Our technique was implemented for systems modeled by Petri nets and compared to an explicit model-checker as well as to a symbolic one (NuSMV) and the obtained results are very competitive.

## 1 Introduction

Model checking is a powerful and widespread technique for the verification of finite distributed systems. Given a Linear-time Temporal Logic (LTL) property and a formal model of the system, it is usually based on converting the negation

[⋆] The work presented in this paper is partially supported by the FME3 ANR Project: Enhancing the Evaluation of Error consequences using Formal Methods.

of the property in a Büchi automaton (or tableau), composing the automaton and the model, and finally checking for the emptiness of the synchronized product. The last step is the crucial stage of the verification process because of the state explosion problem. In fact, the number of reachable states of a distributed system grows exponentially with the number of its components. Numerous techniques have been proposed to cope with the state explosion problem during the last two decades. Among them the *symbolic model checking* (e.g. [6,10,12,4]) aims at checking the property on a compact representation of the system using binary decision diagrams (BDD) techniques [1], while the *On-the-fly model checking* (e.g., [13,8]) allows for generating only the "interesting" part of the state space. The exploration of the synchronized product is stopped as soon as the property is proved unsatisfied by the model. An execution scenario of the system illustrating the violation of the property (counter-example) can then be supplied in order to correct the model.

In this paper, we present a hybrid approach for checking linear time temporal logic properties of finite systems combining on-the-fly and symbolic approaches. Instead of composing the whole system with the Büchi automaton representing the negation of the formula to be checked, we propose to make the synchronization of the automaton with an abstraction of the original reachability graph of the system, namely a *state-based symbolic observation graph*. An event-based variant of the symbolic observation graph has already been introduced in [12]. Its construction is guided by the set of events occurring in the formula to be checked. Such events are said to be observed while the other events are unobserved. The event-based symbolic observation graph is then presented as a hybrid structure where nodes are sets of states (reachable by firing unobserved events) encoded symbolically and edges (corresponding to the firings of observed events) are represented explicitly. It supports on-the-fly model-checking and is equivalent to the reachability graph of the system with respect to event-based $LTL$ semantic. Once built, the observation graph can be analyzed by any standard $LTL \setminus X$ model-checker. In [12], the evaluation of the method is only based on the size of the observation graph which is, in general, very moderate due to the small number of visible events occurring in a typical formula.

The event-based and state-based formalisms are interchangeable: an event can be encoded as a change in state variables, and likewise one can equip a state with different events to reflect different values of its internal variables. However, converting from one representation to the other often leads to a significant enlargement of the state space. Typically, event-based semantic is adopted to compare systems according to some equivalence or pre-order relation (e.g. [20,7,14]), while state-based semantic is more suitable to model-checking approaches [10,19].

The main contributions of this paper are, first the formal definition of a generalized state-based symbolic observation graph and the design and evaluation of an on-the-fly $LTL \setminus X$ (LTL minus the next operator) model-checker instantiating the approach for Petri net models.

The paper is structured as follows: In section 2, we introduce some preliminary definitions as well as some useful notations. In section 3, we formally define the state-based symbolic observation graph and establish some preservation results. Then, section 4 describes the on-the-fly model checker tool implementation and gives a brief presentation of the way the *spot library* has been used for that purpose. The different algorithms were implemented in a software tool and experiments comparing our approach to both explicit on-the-fly and symbolic $LTL$ model checkers are discussed in section 5. Finally, Section 6 concludes the paper and gives some perspectives.

## 2 Preliminaries

This section is dedicated to the definition of some relevant concepts and to the presentation of useful notations. The technique we present in this paper applies to different kinds of models, that can map to finite labeled transition systems, e.g. high-level bounded Petri nets. For the sake of simplicity and generality, we chose to present it for *Kripke structures*, since the formalism is rather simple and well adapted to state-based semantics.

**Definition 1 (Kripke structure).** *Let $AP$ be a finite set of atomic propositions. A* Kripke structure *(KS for short) over $AP$ is a 4-tuple $\langle \Gamma, L, \rightarrow, s_0\rangle$ where:*

- *$\Gamma$ is a finite set of* states *;*
- *$L : \Gamma \rightarrow 2^{AP}$ is a labeling (or interpretation) function;*
- *$\rightarrow \subseteq \Gamma \times \Gamma$ is a* transition relation *;*
- *$s_0 \in \Gamma$ is the* initial state*.*

**Notations**

- Let $s, s' \in \Gamma$. We denote by $s \rightarrow s'$ that $(s, s') \in \rightarrow$,
- Let $s \in \Gamma$. $s \not\rightarrow$ denotes that $s$ is a *dead* state (i.e. $\nexists s' \in \Gamma$ such that $s \rightarrow s'$),
- $\pi = s_1 \rightarrow s_2 \rightarrow \cdots$ is used to denote paths of a Kripke structure and $\overline{\pi}$ denotes the set of states occurring in $\pi$,
- A finite path $\pi = s_1 \rightarrow \cdots \rightarrow s_n$ is said to be a *circuit* if $s_n \rightarrow s_1$. If $\overline{\pi}$ is a subset of a set of states $S$ then $\pi$ is said to be a circuit of $S$.
- Let $\pi = s_1 \rightarrow \cdots \rightarrow s_n$ and $\pi' = s_{n+1} \rightarrow \cdots \rightarrow s_{n+m}$ be two paths such that $s_n \rightarrow s_{n+1}$. Then, $\pi\pi'$ denotes the path $s_1 \rightarrow \cdots \rightarrow s_n \rightarrow s_{n+1} \rightarrow \cdots \rightarrow s_{n+m}$.
- $\forall s, s' \in \Gamma$, $s \xrightarrow{*} s'$ denotes that $s'$ is reachable from $s$ (i.e. $\exists s_1, \cdots, s_n \in \Gamma$ such that $s_1 \rightarrow \cdots \rightarrow s_n \wedge s = s_1 \wedge s' = s_n$). $s \xrightarrow{+} s'$ denotes the case where $n > 1$ and $s \xrightarrow{*}_S s'$ (resp. $s \xrightarrow{+}_S s'$) stands when the states $s_1, \cdots, s_n$ belong to some subset of states $S$.

**Definition 2 (maximal paths).** *Let $\mathcal{T}$ be Kripke structure and let $\pi = s_1 \rightarrow s_2 \rightarrow \cdots \rightarrow s_n$ be a path of $\mathcal{T}$. Then, $\pi$ is said to be* a maximal path *if one of the two following properties holds:*

– $s_n \not\rightarrow$,
– $\pi = s_1 \rightarrow \cdots \rightarrow s_m \rightarrow \cdots \rightarrow s_n$ *and* $s_m \rightarrow \cdots \rightarrow s_n$ *is a circuit.*

Since LTL is interpreted on infinite paths, a usual solution in automata theoretic approach to check LTL formulae on a KS is to convert each of its finite maximal paths to an infinite one by adding a loop on its dead states. From now on, the KS obtained by such a transformation is called *extended Kripke structure* (*EKS* for short).

Moreover, it is well known that LTL formulae without the 'next' operator are invariant under the so-called *stuttering equivalence* [5]. We will use this equivalence relation to prove that observation graphs can be used for model checking. Stuttering occurs when the same atomic propositions (label) hold on two or more consecutive states of a given path. We recall the definition of *stuttering equivalence* between two paths.

**Definition 3 (Stuttering equivalence).** *Let $\mathcal{T}$ and $\mathcal{T}'$ be two Kripke structures over an atomic proposition set $AP$ and let $\pi = s_0 \rightarrow s_1 \rightarrow \cdots$ and $\pi' = r_0 \rightarrow r_1 \rightarrow \cdots$ be respectively paths of $\mathcal{T}$ and $\mathcal{T}'$. $\pi$ and $\pi'$ are said to be stuttering equivalent, written as $\pi \sim_{st} \pi'$, if there are two sequences of integers $i_0 = 0 < i_1 < i_2 < \cdots$ and $j_0 = 0 < j_1 < j_2 < \cdots$ s.t. for every $k \geq 0, L(s_{i_k}) = L(s_{i_k+1}) = \cdots = L(s_{i_{k+1}-1}) = L'(r_{j_k}) = L'(r_{j_k+1}) = \cdots = L'(r_{j_{k+1}-1})$.*

## 3 Symbolic Observation Graph

Two motivations are behind the idea of the state-based *Symbolic Observation Graph* (SOG). First, state-based logics are more intuitive and more suitable for model-checking approaches than event-based ones. Second, we wanted to give a flexible definition of the SOG allowing several possible implementation which, as we will see in section 5, could improve the performances. In fact, thanks to the flexibility of the formal definition of the state-based *SOG*, the construction algorithm of [12] can be viewed as a specialization of our technique by observing events altering the truth values of the induced atomic propositions.

From now on, *SOG* will denote the state-based variant.

### 3.1 Definitions

We first define formally what is an *aggregate*.

**Definition 4 (Aggregate).** *Let $\mathcal{T} = \langle \Gamma, L, \rightarrow, s_0 \rangle$ be a KS over an atomic proposition set $AP$. An* aggregate *$a$ of $\mathcal{T}$ is a non empty subset of $\Gamma$ satisfying $\forall s, s' \in a, L(s) = L(s')$.*

We introduce three particular sets of states and two predicates. Let $a$ and $a'$ be two aggregates of $\mathcal{T}$.

– $Out(a) = \{s \in a \mid \exists s' \in \Gamma \setminus a, s \rightarrow s'\}$
– $Ext(a) = \{s' \in \Gamma \setminus a \mid \exists s \in a, s \rightarrow s'\}$

- $In(a, a') = \{s' \in a' \setminus a \mid \exists s \in a, s \to s'\}$ (i.e. $In(a, a') = Ext(a) \cap a'$)
- $Dead(a) = (\exists s \in a \text{ s.t. } s \not\to)$
- $Live(a) = (\exists \pi \text{ a circuit of } a)$

Let us describe informally, for an aggregate $a$, the meaning of the above notations: $Out(a)$ denotes the set of *output states* of $a$ i.e. any state of $a$ having a successor outside of $a$. $Ext(a)$ contains any state, outside of $a$, having a predecessor in $a$. Given an aggregate $a'$, $In(a, a')$ denotes the set of *input states* of $a'$ according to the predecessor $a$, notice that $In(a, a') = Ext(a) \cap a'$. Finally the predicate $Dead(a)$ (resp. $Live(a)$) holds when there exists a dead state (resp. a circuit) in $a$.

We first introduce the *compatibility* relation between an aggregate and a set of states.

**Definition 5 (Compatibility).** *An aggregate $a$ is said* compatible *with a set of states $S$ if and only if:*

- $S \subseteq a$
- $\forall s \in Out(a), \exists s' \in S$ *such that* $s' \xrightarrow{*}_a s$
- $Dead(a) \Rightarrow \exists s' \in S, \exists d \in a$ *such that* $d \not\to \wedge s' \xrightarrow{*}_a d$
- $Live(a) \Rightarrow \exists s' \in S, \exists \pi$ *a circuit of* $a, \exists c \in \overline{\pi}$ *such that* $s' \xrightarrow{*}_a c$

Now, we are able to define the symbolic observation graph structure according to a given $KS$.

**Definition 6 (Symbolic Observation Graph).** *Let $\mathcal{T} = \langle \Gamma, L, \to, s_0 \rangle$ be a $KS$ over an atomic proposition set $AP$. A* symbolic observation graph *of $\mathcal{T}$ is a 4-tuple $\mathcal{G} = \langle \Gamma', L', \to', a_0 \rangle$ where:*

1. $\Gamma' \subseteq 2^{\Gamma}$ *is a finite set of aggregates*
2. $L' : \Gamma' \to 2^{AP}$ *is a labeling function satisfying* $\forall a \in \Gamma'$, *let* $s \in a, L'(a) = L(s)$.
3. $\to' \subseteq \Gamma' \times \Gamma'$ *is a transition relation satisfying:*
   (a) $\forall a, a' \in \Gamma'$ *such that* $a \to' a'$,
      i. $a' \neq a \Rightarrow In(a, a') \neq \emptyset$ *and* $a'$ *is compatible with* $In(a, a')$
      ii. $a' = a \Rightarrow$ *there exists a circuit* $\pi$ *of* $a$ *such that* $a$ *is compatible with* $\overline{\pi}$ *and* $\forall E \in In_{\mathcal{G}}(a)$, *there exists a circuit* $\pi_E$ *of* $a$ *such that* $a$ *is compatible with* $\overline{\pi_E}$ *and* $\exists e \in E, c \in \overline{\pi_E}$ *satisfying* $e \xrightarrow{*}_a c$
      where $In_{\mathcal{G}}(a) = \{E \subseteq a \mid E = \{s_0\} \vee \exists a' \in \Gamma' \setminus \{a\}, a' \to' a \wedge E = In(a', a)\}$
   (b) $\forall a \in \Gamma', Ext(a) = \bigcup_{a' \in \Gamma', a \to' a'} In(a, a')$.
4. $a_0 \in \Gamma'$ *is the initial aggregate and is compatible with* $\{s_0\}$

While points (1), (2) and (4) are trivial, the transition relation (3) of the $SOG$ requires explanation: Given two aggregates $a$ and $a'$ such that $a \to' a'$, then we distinguish two cases:

(3(a)i) stands for $a'$ is different from $a$. In this case, we impose $a'$ to be compatible with $In(a, a')$. This means that entering in $a'$ by following the arc between $a$ and $a'$, all the output states of $a'$, as well as a dead state (if $Dead(a')$ holds) and a circuit (if $Live(a')$ holds) must be reachable.

(3(a)ii) treats the non trivial case of a loop on $a$ (i.e. $a \rightarrow' a$). Notice first that such a condition can be removed without changing the validity of our theoretical results. In this case, no loop will be authorized on a single aggregate $a$ but it does not matter since $Live(a)$ holds and the stuttering implied by the loop will be captured by this predicate. However, this point appears to allow more flexibility for the *SOG* construction. A loop $a \rightarrow' a$ is authorized if for any subset $E$ such that $E = \{s_0\}$ (if $s_0 \in a$) or $E = In(a', a)$ for some predecessor $a'$ of $a$ (in $\mathcal{G}$), there exists a cycle $\pi_E$ reachable from $E$ and $a$ is compatible with $\overline{\pi_E}$. If such a subset $E$ does not exist (i.e. $In_{\mathcal{G}}(a) = \emptyset$) then the aggregate $a$ has no predecessor in $\mathcal{G}$ (and is not the initial aggregate) and we authorize a loop on $a$ if and only if there exists a circuit $\pi$ of $a$ such that $a$ is compatible with $\overline{\pi}$.

Finally, point (3b) implies that all the successors of the states present in an aggregate $a$ are represented in $\Gamma'$.

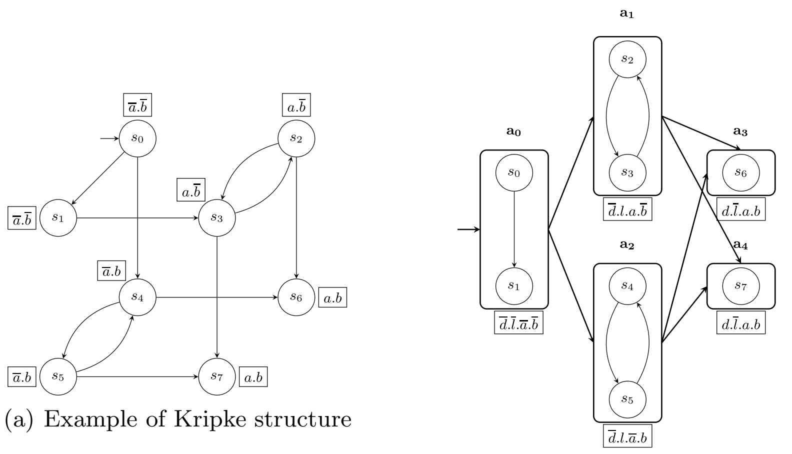
Notice that Definition 6 does not guarantee the uniqueness of a SOG for a given *KS*. In fact, it supplies a certain flexibility for its implementation. In particular, an aggregate can be labeled by the same atomic proposition set than one of its successors and two successors of an aggregate may also have the same labels. We will see in the section 5 that the implementation can take advantage of this flexibility.

**Example:**
Figure 1 illustrates an example of *KS* (Figure 1(a)) and a corresponding SOG (Figure 1(b)). The set of atomic propositions contains two elements $\{a, b\}$ and each state of the *KS* is labeled with the values of these propositions. The presented SOG consists of 5 aggregates $\{a_0, a_1, a_2, a_3, a_4\}$ and 6 edges. Aggregates $a_1$ and $a_2$ contain circuits but no dead states, whereas $a_3$ and $a_4$ have each a dead state but no circuit. Each aggregate $a$ is indexed with a triplet $(Dead(a), Live(a), L'(a))$. The symbol $d$ (resp. $\overline{d}$) is used when $Dead(a)$ holds (resp. does not hold) and the symbol $l$ (resp. $\overline{l}$) is used when $Live(a)$ holds (resp. does not hold). Notice that states of the *KS* are partitioned into aggregates which is not necessary the case in general (i.e. a single state may belong to two different aggregates). Moreover, one can merge $a_3$ and $a_4$ within a single aggregate and still respect Definition 6.

The following definition characterizes the paths of a SOG which must be considered for model checking.

**Definition 7 (maximal paths of a SOG).** *Let $\mathcal{G}$ be a SOG and $\pi = a_1 \rightarrow a_2 \rightarrow \cdots \rightarrow a_n$ be a path of $\mathcal{G}$. Then $\pi$ is said to be* a maximal path *if one of the three following properties holds:*

- *$Dead(a_n)$ holds,*
- *$Live(a_n)$ holds,*
- *$\pi = a_1 \rightarrow \cdots \rightarrow a_m \rightarrow \cdots \rightarrow a_n$ and $a_m \rightarrow \cdots \rightarrow a_n$ is a circuit (i.e. $a_n \rightarrow a_m$).*



(a) Example of Kripke structure

(b) The corresponding SOG with $AP = \{a, b\}$

**Fig. 1.** A Kripke structure and its SOG

Our ultimate goal is to check LTL\X properties on a *SOG* $\mathcal{G}$ associated with a *KS* $\mathcal{T}$. Thus, in order to capture all the maximal paths of $\mathcal{G}$ under the form of infinite sequences, we transform its finite maximal paths into infinite ones. The following definition formalizes such a transformation and by analogy to extended Kripke structure, the obtained transformed graph is called *extended symbolic observation graph* (ESOG for short).

**Definition 8 (Extended SOG).** *Let $\langle \Gamma', L', \rightarrow', a_0\rangle$ be a SOG over an atomic proposition set AP. The associated ESOG is a KS $\langle \Gamma, L, \rightarrow, s_0\rangle$ where:*

1. $\Gamma = \Gamma' \cup \{v \in 2^{AP} | \exists a \in \Gamma', L'(a) = v \wedge (Dead(a) \vee Live(a))\}$
2. $\forall a \in \Gamma', L(a) = L'(a)$ *and* $\forall v \in \Gamma \setminus \Gamma', L(v) = v$
3. $\rightarrow \subseteq \Gamma \times \Gamma$ *is the transition relation satisfying:*
   (a) $\forall a, a' \in \Gamma', a \rightarrow' a' \Rightarrow a \rightarrow a'$
   (b) $\forall a \in \Gamma', Dead(a) \vee Live(a) \Rightarrow a \rightarrow L'(a)$
   (c) $\forall v \in \Gamma \setminus \Gamma', v \rightarrow v$
4. $s_0 = s'_0$

**Example:**
The extended SOG of Figure 1(b) is obtained by adding three nodes $a.\overline{b}$, $\overline{a}.b$ (corresponding to $a_1$ and $a_2$ respectively because $Live(a_1)$ and $Live(a_2)$ hold and $a.b$ (corresponding to both aggregates $a_3$ and $a_4$ because $Dead(a_3)$ and $Dead(a_4)$ hold). These three added states are labeled with $a.\overline{b}$, $\overline{a}.b$ and $a.b$ respectively and have each a looping arc. We also add the following 4 arcs: one from $a_1$ to $a.\overline{b}$, one from $a_2$ to $\overline{a}.b$, one from $a_3$ to $a.b$ and one from $a_4$ to $a.b$.

## 3.2 LTL\X Model Checking and SOG

The equivalence between checking a given $LTL \setminus X$ property on the observation graph and checking it on the original labeled transition system is ensured by the preservation of maximal paths. This corresponds to the CFFD semantic [14] which is exactly the weakest equivalence preserving next time-less linear temporal logic. Thus, the symbolic observation graph preserves the validity of formulae written in classical Manna-Pnueli linear time logic [15] (LTL) from which the "next operator" has been removed (because of the abstraction of the immediate successors) (see for instance [18,11]).

In the following, we give the main result of the paper: checking an $LTL \setminus X$ formula on a Kripke structure can be reduced to check it on a corresponding SOG. Due to the complexity of the proof of this result, it will be deduced from a set of intermediate lemmas and one proposition.

**Theorem 1.** *Let $\mathcal{G}$ be an ESOG over an atomic proposition set $AP$ and corresponding to an extended KS $\mathcal{T}$. Let $\varphi$ be a formula from LTL\X on $AP$. Then the following holds:*

$$\mathcal{T} \models \varphi \Leftrightarrow \mathcal{G} \models \varphi$$

**Proof.** The proof of Theorem 1 is direct from Proposition 1 (ensuring the preservation of maximal paths) and Definition 8. □

Given a *KS* $\mathcal{T} = \langle \Gamma, L, \rightarrow, s_0 \rangle$ over an atomic proposition set $AP$ and $\mathcal{G} = \langle \Gamma', L', \rightarrow', a_0 \rangle$ a SOG associated with $\mathcal{T}$ according to Definition 6, we present four lemmas about the correspondence between paths of $\mathcal{T}$ and those of $\mathcal{G}$. These lemmas are followed by Proposition 1. The first lemma demonstrates that each finite path in $\mathcal{T}$ has (at least) a corresponding path in $\mathcal{G}$.

**Lemma 1.** *Let $\pi = s_1 \rightarrow s_2 \rightarrow \cdots \rightarrow s_n$ be a path of $\mathcal{T}$ and $a_1$ be an aggregate of $\mathcal{G}$ such that $s_1 \in a_1$. Then, there exists a path $a_1 \rightarrow' a_2 \rightarrow' \cdots \rightarrow' a_m$ of $\mathcal{G}$ and a strictly increasing sequence of integers $i_1 = 1 < i_2 < \cdots < i_{m+1} = n+1$ satisfying $\{s_{i_k}, s_{i_k+1}, \cdots, s_{i_{k+1}-1}\} \subseteq a_k$ for all $1 \leq k \leq m$.*

**Proof.** We proceed by induction on the length of $\pi$. If $n = 1$, knowing that $s_1 \in a_1$ concludes the proof. Let $n > 1$ and assume that $a_1 \rightarrow' a_2 \rightarrow' \cdots \rightarrow' a_{m-1}$ and $i_1, \cdots, i_m$ correspond to the terms of the lemma for the path $s_1 \rightarrow s_2 \rightarrow \cdots \rightarrow s_{n-1}$. Then, $s_{n-1} \in a_{m-1}$. Let us distinguish two cases.

(i) If $s_n \in a_{m-1}$ then the path $a_1 \rightarrow' a_2 \rightarrow' \cdots \rightarrow' a_{m-1}$ and the sequence $i_1, \cdots, i_m + 1$ satisfy the proposition.

(ii) If $s_n \notin a_{m-1}$ then $s_n \in Ext(a_{m-1})$ and, by def. 6 (item 3b), there exists an aggregate $a_m$ such that $a_{m-1} \rightarrow' a_m$ and $s_n \in In(a_{m-1}, a_m)$. As a consequence, the path $a_1 \rightarrow' a_2 \rightarrow' \cdots \rightarrow' a_{m-1} \rightarrow' a_m$ and the sequence $i_1, \cdots, i_m, i_m + 1$ satisfy the proposition. □

The next lemma shows that the converse also holds.

**Lemma 2.** *Let $\pi = a_1 \rightarrow' a_2 \rightarrow' \cdots \rightarrow' a_n$ be a path of $\mathcal{G}$. Then, there exists a path $e_1 \rightarrow (b_2 \xrightarrow{*}_{a_2} e_2) \rightarrow \cdots \rightarrow b_n$ of $\mathcal{T}$ satisfying $e_1 \in a_1$ and $b_n \in a_n$.*

**Proof.** We consider $\pi$ in reverse order and proceed by induction on its length. If $n = 1$, it is sufficient to choose a state $s_1 \in a_1$. If $n = 2$, we have to distinguish two cases.

(i) If $a_1 \neq a_2$ then, by def. 6 (item 3(a)i), $\mathit{In}(a_1, a_2) \neq \emptyset$ and, by definition of $\mathit{In}$, there exist $e_1 \in \mathit{Out}(a_1)$ and $b_2 \in \mathit{In}(a_1, a_2)$ such that $e_1 \rightarrow b_2$. This path verifies the proposition.

(ii) If $a_1 = a_2$ then, by def. 6 (item 3(a)ii), we know that there exists a circuit $\sigma$ of $a_1$. Let $b_2 \in \overline{\sigma}$. The path $b_2 \xrightarrow{+}_{a_1} b_2$ satisfies the proposition.

Let $n > 2$ and assume that $e_2 \rightarrow \cdots \rightarrow b_n$ corresponds to the terms of the proposition for the path $a_2 \rightarrow' \cdots \rightarrow' a_n$. We know that $e_2 \in a_2$. Here four cases have to be considered.

(iii) If $a_1 \neq a_2 \wedge e_2 \in \mathit{Out}(a_2)$ then, using def. 6 (item 3(a)i), we know that there exists a state $b_2 \in \mathit{In}(a_1, a_2)$ such that $b_2 \xrightarrow{*}_{a_2} e_2$ and a state $e_1 \in \mathit{Out}(a_1)$ such that $e_1 \rightarrow b_2$. The path $e_1 \rightarrow (b_2 \xrightarrow{*}_{a_2} e_2) \rightarrow \cdots \rightarrow b_n$ verifies the proposition.

(iv) If $a_1 = a_2 \wedge e_2 \in \mathit{Out}(a_2)$ then, by def. 6 (item 3(a)ii), we know that $a_1$ contains a circuit $\sigma$ such that $a_1$ is compatible with $\overline{\sigma}$. Let $e_1, b_2 \in \overline{\sigma}$ such that $e_1 \rightarrow b_2$. Since $a_1$ is compatible with $\overline{\sigma}$ and $e_2 \in \mathit{Out}(a_1)$ then $e_2$ is reachable from $b_2$ in $a_1$. In consequence, the path $e_1 \rightarrow (b_2 \xrightarrow{*}_{a_2} e_2) \rightarrow \cdots \rightarrow b_n$ satisfies the proposition.

(v) $a_1 \neq a_2 \wedge e_2 \notin \mathit{Out}(a_2)$ then $(a_2, a_2) \in \rightarrow'$. Since $\mathit{In}(a_1, a_2) \neq \emptyset$ and due to def. 6 (item 3(a)ii), there exists a circuit $\sigma$ of $a_2$ reachable from some state $b_2 \in \mathit{In}(a_1, a_2)$ and such that $a_2$ is compatible with $\overline{\sigma}$. Moreover, there exists $e_1 \in \mathit{Out}(a_1)$ such that $e_1 \rightarrow b_2$. Let $c \in \overline{\sigma}$. Let us distinguish the two following subcases:

(a) If there exists $i > 2$ such that $e_i \in \mathit{Out}(a_i)$ then, let $j$ be the smallest such an $i$. Then, since $a_j$ is compatible with $\overline{\sigma}$, $e_j$ is reachable in $a_j$ from $c$. Hence, the path $e_1 \rightarrow b_2 \xrightarrow{*}_{a_2} c(\xrightarrow{+}_{a_2} c)^{j-1} \xrightarrow{*}_{a_j} e_j \cdots \rightarrow b_n$ verifies the proposition.

(b) If for all $i > 2$, $e_i \notin \mathit{Out}(a_i)$ then the path $e_1 \rightarrow b_2 \xrightarrow{*}_{a_2} c(\xrightarrow{+}_{a_2} c)^{n-1}$ satisfies the proposition.

(vi) $a_1 = a_2 \wedge e_2 \notin \mathit{Out}(a_2)$ then, by def. 6 (item 3(a)ii), we know that $a_1$ contains a circuit $\sigma$ such that $a_1$ is compatible with $\overline{\sigma}$. We also know that $e_2 \in \overline{\sigma}$ by construction. Let $e_1 \in \overline{\sigma}$ such that $e_1 \rightarrow e_2$. Then the path $e_1 \rightarrow (e_2 \xrightarrow{*}_{a_2} e_2) \rightarrow \cdots \rightarrow b_n$ satisfies the proposition. $\square$

We are now in position to study the correspondence between maximal paths.

**Lemma 3.** *Let $\pi = s_0 \rightarrow \cdots \rightarrow s_n$ be a maximal path of $\mathcal{T}$. Then, there exists a maximal path $\pi' = a_0 \rightarrow' \cdots \rightarrow' a_m$ of $\mathcal{G}$ such that there exists a sequence of integers $i_0 = 0 < i_1 < \cdots < i_{m+1} = n + 1$ satisfying $\{s_{i_k}, s_{i_k+1}, \cdots, s_{i_{k+1}-1}\} \subseteq a_k$ for all $0 \leq k \leq m$.*

**Proof.** If $s_n$ is a dead state then knowing that $s_0 \in a_0$ (Definition 6 (item 4) and using Lemma 1, we can construct a path $\pi' = a_0 \rightarrow a_2 \cdots a_m$ and the associated integer sequence corresponding to $\pi$. Because the last visited state of $\pi$ belongs to $a_m$, $\mathit{Dead}(a_m)$ necessarily holds and $\pi'$ is then a maximal path in the sense of Definition 7.

Now, if $s_n$ is not a dead state then, one can decompose $\pi$ as follows: $\pi = \pi_1\pi_2$ s.t. $\pi_1 = s_0 \rightarrow s_1 \rightarrow \cdots \rightarrow s_{n-1}$ and $\pi_2 = s_n \rightarrow s_{n+1} \rightarrow \cdots \rightarrow s_{n+m}$ (where $\pi_2$ is a circuit). Once again, applying Lemma 1 from $a_0$, one can construct a path $\pi'_1 = a_0 \rightarrow' a_1 \rightarrow' \cdots a_k$ corresponding to $\pi_1$. The corresponding path of $\pi'_2$ can be also constructed applying the same lemma. However, this path must be constructed from $a_k$ if $s_n \in a_k$ or from a successor of $a_k$ containing $s_n$ otherwise (Definition 6 ensures its existence). Let $\pi'_2 = a_{b_1} \rightarrow' a_{b_1+1} \rightarrow' \cdots a_{e_1}$ be this path.

Then, let us distinguish the following four cases:

1. if $\pi'_2$ is reduced to a single aggregate $a$ then $\overline{\pi_2} \subseteq a$ and, because $\pi_2$ is a circuit of $\mathcal{T}$, $\mathit{Live}(a)$ holds. Then, the path $\pi'_1\pi'_2$ is maximal in $\mathcal{G}$.
2. else if $a_{e_1} \rightarrow' a_{b_1} \wedge s_n \in \mathit{In}(a_{e_1}, a_{b_1})$ then $\pi'_2$ is a circuit of $\mathcal{G}$ and $\pi'_1\pi'_2$ is a maximal path of $\mathcal{G}$ satisfying the proposition.
3. else if $s_n \in a_{e_1}$ (i.e $a_{b_1} = a_{e_1}$) then the path $a_{b_1+1} \rightarrow' \cdots a_{e_2}$ is a circuit of $\mathcal{G}$ and $\pi'_1 \rightarrow' a_{b_1} \rightarrow' a_{b_1+1} \rightarrow' \cdots a_{e_2}$ is a maximal path of $\mathcal{G}$ satisfying the proposition.
4. else, by Definition 6, there exists a successor of $a_{e_1}$ containing $s_n$. Applying again Lemma 1 from this aggregate, we can construct a new path in $\mathcal{G}$ corresponding to $\pi_2$. Let $a_{b_2} \rightarrow' a_{b_2+1} \rightarrow' \cdots a_{e_2}$ be this path. If we can deduce a circuit of $\mathcal{G}$ from this path applying one of the three above points, this concludes the proof. Otherwise, it is also possible to construct a circuit of $\mathcal{G}$ by linking $a_{e_2}$ to $a_{b_1}$ similarly to the point 2 and 3 above and deduce a circuit. If this is not the case, we can construct a new path corresponding to $\pi_2$ starting from a successor of $a_{e_2}$. Because the number of aggregates in $\mathcal{G}$ is finite, a circuit will be necessarily obtained.

Notice that for all the previous cases above, a sequence of integers can be easily constructed from the ones produced by Lemma 1. □

**Lemma 4.** *Let $\pi' = a_0 \rightarrow' \cdots \rightarrow' a_n$ be a maximal path of $\mathcal{G}$. Then, there exists a maximal path $\pi = (s_0 \xrightarrow{*}_{a_0} e_0) \rightarrow \cdots \rightarrow (b_n \xrightarrow{*}_{a_n} e_n)$ of $\mathcal{T}$.*

**Proof.** Let $\pi'$ be a maximal path reaching an aggregate $a_n$ such that $\mathit{Dead}(a_n)$ or $\mathit{Live}(a_n)$ hold. First, let us notice that the proof is trivial if the path $\pi'$ is reduced to a single aggregate because of the compatibility of $a_0$ with $\{s_0\}$ which implies that a dead state (resp. a state of a circuit of $a_0$) is reachable from $s_0$.

Otherwise, using Lemma 2, there exists a path $\pi = e_0 \rightarrow (b_1 \xrightarrow{*}_{a_1} e_1) \rightarrow \cdots \rightarrow b_n$ of $\mathcal{T}$ satisfying $e_0 \in a_0$ and $b_n \in a_n$. If $e_0 \in \mathit{Out}(a_0)$, we have $s_0 \xrightarrow{*}_{a_0} e_0$ since $a_0$ is compatible with $\{s_0\}$. Otherwise, $e_0$ belongs to a circuit of $a_0$ and there exists in $\mathcal{G}$ an arc from $a_0$ to itself. Definition 6 (item 3(a)ii) ensures that this circuit can be chosen to be reachable from $s_0$ (and compatible with $a_0$) during the construction of $\pi$. Finally, there exists a state $e_n \in a_n$ such that $b_n \xrightarrow{*}_{a_n} e_n$, where $e_n$ is a dead state (if $Dead(a_n)$ holds) or a state of a circuit of $a_n$ (if $Live(a_n)$ holds), because $a_n$ is compatible with $In(a_{n-1}, a_n)$. Thus, the path $(s_0 \xrightarrow{*}_{a_0} e_0) \rightarrow (b_1 \xrightarrow{*}_{a_1} e_1) \rightarrow \cdots \rightarrow (b_n \xrightarrow{*}_{a_n} e_n)$ satisfies the lemma.

Now, if neither $Dead(a_n)$ nor $Live(a_n)$ hold, then by Definition 7, $\pi' = a_0 \rightarrow \cdots \rightarrow a_m \rightarrow \cdots \rightarrow a_n$ with $a_m \rightarrow \cdots \rightarrow a_n$ a circuit of $\mathcal{G}$. We distinguish the two following cases:

1. If $\forall m \leq i \leq n, a_i = a_m$. Using Lemma 2, we can construct a path of $\mathcal{T}$, namely $\pi = e_0 \rightarrow (b_1 \xrightarrow{*}_{a_1} e_1) \rightarrow \cdots \rightarrow b_m$ corresponding to $a_0 \rightarrow \cdots \rightarrow a_m$ such that $e_0$ is chosen to be reachable from $s_0$ (similarly to the above case). Because $a_m \rightarrow a_m$, $a_m$ contains a circuit and $b_m$ can be chosen such that this circuit is reachable from $b_m$. This leads to the construction of a maximal path of $\mathcal{T}$.
2. Otherwise, $m$ can be chosen such that $a_m \neq a_n$ and $a_n \rightarrow' a_m$. From this decomposition of $\pi'$, Lemma 2 can be used to construct a maximal path of $\mathcal{T}$ satisfying the current lemma.

□

The following proposition is a direct consequence of the two previous lemmas.

**Proposition 1.** *Let $\mathcal{G} = \langle \Gamma', L', \rightarrow', a_0 \rangle$ be a SOG over an atomic proposition set AP and corresponding to a KS $\mathcal{T} = \langle \Gamma, L, \rightarrow, s_0 \rangle$. Then the following holds:*

1. *$\forall \pi = s_0 \rightarrow s_1 \rightarrow \cdots$, a maximal path of $\mathcal{T}$, $\exists \pi' = a_0 \rightarrow' a_1 \rightarrow' \cdots$, a maximal path of $\mathcal{G}$ s.t. $\pi \sim_{st} \pi'$.*
2. *$\forall \pi' = a_0 \rightarrow' a_1 \rightarrow' \cdots$, a maximal path of $\mathcal{G}$, $\exists \pi = s_0 \rightarrow s_1 \rightarrow \cdots$, a maximal path of $\mathcal{T}$ s.t. $\pi \sim_{st} \pi'$.*

**Proof.** The proof of Proposition 1 is direct by considering Lemmas 3 and 4 as well as Definitions 3 and 4. □

## 4 Construction of a SOG Model-Checker for Petri Nets

This section presents the model-checker (named MC-SOG) we have implemented to verify state based LTL\X formulae on bounded Petri nets [17] (i.e. nets having a finite reachability graph) . The implementation of MC-SOG is based on Spot [9], an object-oriented model checking library written in C++. Spot offers a set of building blocks to experiment with and develop your own model checker. It is based on the automata theoretic approach to LTL model checking. In this approach, the checking algorithm visits the synchronized product of the $\omega$-automaton corresponding to the negation of the formula and of the one corresponding to the system. Spot proposes a module for the translation of an LTL formula into an $\omega$-automaton but is not dedicated to a specific formalism for the representation of the system to be checked. Then, many of the algorithms are based on a set of three abstract classes representing $\omega$-automata and which must be specialized depending on your own needs. The first abstract class defines a state, the second allows to iterate on the successors of a given state and the last one represents the whole $\omega$-automaton. In our context, we have derived these classes for implementing ESOG of bounded Petri nets. It is important to notice that the effective construction of the ESOG is driven by the emptiness check algorithm of Spot and therefore, will be managed on-the-fly.

Spot is freely distributed under the terms of the GNU GPL (`spot.lip6.fr`) and has been recently qualified as one of the best explicit LTL model checkers [19].

We assume that the bound of the considered net is known *a priori*. Each set of markings corresponding to an aggregate is represented by a BDD using the encoding presented by Pastor & al in [16]. For a $k$-bounded place, this encoding uses $\lceil \log_2(k) \rceil$ BDD variables. The representation of the transition relation is achieved using two sets of variables. The first one allows to specify the enabling condition on the current state while the second represents the effect of the firing (see [16] for a more precise description).

The atomic propositions that we accept are place names. In a given marking, an atomic proposition associated to a place is satisfied if the place contains at least a token.

The construction of aggregates depends on places appearing as atomic proposition in the checked formula. In the following, we denote by $AP$ this subset of places. The construction starts from a set of initial markings $M$, all of them satisfying exactly the same atomic propositions. To compute the initial aggregate of the ESOG, we state $M = \{m_0\}$. Then, the complete set of markings corresponding to an aggregate is obtained by applying until saturation (i.e. no new states can be reached any more) the transition relation limited to the transitions which do not modify the truth value of atomic propositions. Instead of checking this constraint explicitly, we statically restrict the set of Petri net transitions to be considered to the ones which do not modify the marking of places used as atomic propositions (i.e. the set $\mathit{Local} = \{t \in T \mid \forall p \in AP, \mathit{Pre}(p,t) = \mathit{Post}(p,t)\}$). We denote by $\mathit{Sat}(M)$ the set of markings reachable from at least a marking of $M$ by the firings of transitions in *Local* only. In other terms, $\mathit{Sat}(M)$ is defined by induction as follows:

- $\forall m \in M, m \in \mathit{Sat}(M)$,
- $\forall m \in \mathit{Sat}(M), (\exists t \in \mathit{Local} \text{ s.t. } m[t > m') \Rightarrow m' \in \mathit{Sat}(M)$ (where $m[t > m'$ denotes that $t$ is firable from $m$ and its firing leads to $m'$).

The successors of an aggregate are obtained by considering one by one the transitions excluded during the computation of aggregates (of the set $\mathit{Extern} = T \setminus \mathit{Local}$). For each of these transitions, we compute the set of markings directly reachable by firing it from the markings composing the aggregate. Then, these reached markings are the seed for the construction of the successor aggregate. For a transition $t \in \mathit{Extern}$, we define $\mathit{Out}(M,t) = \{m \in \mathit{Sat}(M) \mid m[t >\}$ and $\mathit{Ext}(M,t) = \{m' \mid \exists m \in \mathit{Sat}(M) \text{ s.t. } m[t > m'\}$. Notice that the definition of the set *Extern* and the fact that the construction is started from $\{m_0\}$ ensure that $\forall t \in \mathit{Extern}, \forall m, m' \in \mathit{Sat}(\mathit{Ext}(M,t))$ and $\forall p \in AP$, we have $m(p) = m'(p)$ and therefore, that all the markings of an aggregate satisfy the same atomic propositions.

We also note $\mathit{Out}(M) = \bigcup_{t \in \mathit{Extern}} \mathit{Out}(M,t)$. Finally, we define two predicates: $\mathit{Dead}(M)$ holds if $\mathit{Sat}(M)$ contains at least a dead marking and $\mathit{Live}(M)$ if $\mathit{Sat}(M)$ contains at least a circuit.

Starting from $M = \{m_0\}$, we construct on-the-fly an ESOG where aggregates are $\mathit{Sat}(M)$ and successors of such a node are $\mathit{Sat}(\mathit{Ext}(M,t))$ for each $t$ of *Extern* (satisfying $\mathit{Ext}(M,t) \neq \emptyset$). However, two aggregates $\mathit{Sat}(M)$ and $\mathit{Sat}(M')$ can be merged (i.e. identify as equal in the hash table of the emptiness check) if $\mathit{Out}(M) = \mathit{Out}(M') \wedge \mathit{Dead}(M) = \mathit{Dead}(M') \wedge \mathit{Live}(M) = \mathit{Live}(M')$ without contradicting the definition 6. This situation is illustrated in fig.2.

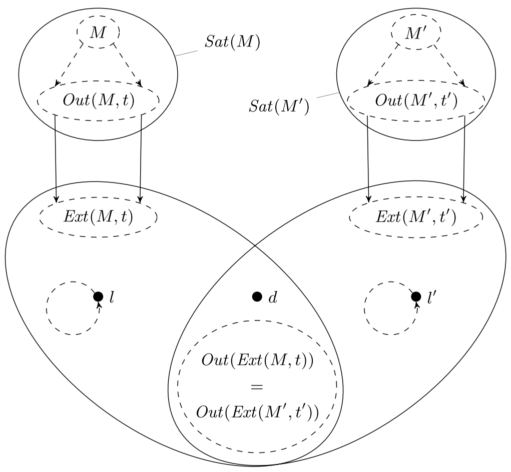


**Fig. 2.** The aggregates $\mathit{Sat}(\mathit{Ext}(M,t))$ and $\mathit{Sat}(\mathit{Ext}(M',t'))$ can be merged

This has an important consequence on the implementation. An aggregate, constructed from a set $M$, is only identified by the markings composing $\mathit{Out}(M)$ (one BDD) associated to the truth value of $\mathit{Dead}(M)$ and $\mathit{Live}(M)$ (two booleans). Then, the hash table is only composed by such triplets. This explains for an important part why the SOG construction obtains good results in terms of memory consumption. Indeed, when *Extern* contains a limited number of transitions, the sets $\mathit{Out}(M)$ are generally small as well as the number of aggregates.

To determine if an aggregate contains a dead marking or a circuit, we use the algorithms presented in [12]. Moreover, when an aggregate contains one or the other, an artificial successor is added to the existing ones. This new aggregate is only characterized by the truth value of the atomic propositions (encoded by a BDD) and has itself as unique successor.

This computation is illustrated on the net of Figure 3 when considering two dining philosophers and the formula $\Box(wl_1 \wedge wr_1 \Rightarrow \Diamond(e_1))$. This formula

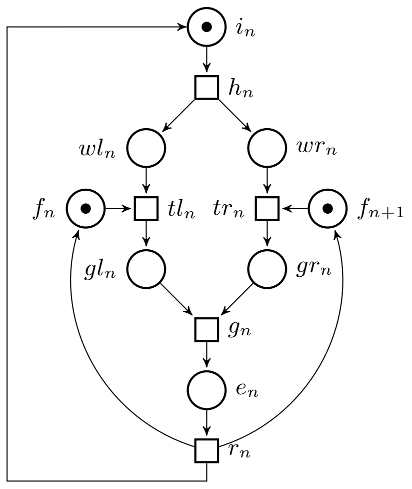


**Fig. 3.** A Petri net modeling the $n^{th}$ dining philosopher

expresses that when the first philosopher wants to eat ($wl_1 \wedge wr_1$) then he must eat ($e_1$) eventually in the future. All the transitions connected to the three places $wl_1$, $wr_1$ and $e_1$ are excluded during the computation of the markings composing an aggregate. Notice that these are all the transitions associated to the first philosopher. For instance, the initial aggregate, presented at the left of Figure 4, groups all the markings reachable from the initial marking when only the second philosopher runs. The transition $h_1$ is enabled from all these markings. Because this transition has been excluded during the computation and no other transition of *Extern* is enabled from any of these markings, the initial aggregate has only one successor whose construction begins with all the markings immediately reached after the firing of $h_1$.

Notice that the initial aggregate contains no deadlock ($d$ is false) but a circuit ($l$ is true). Moreover, no atomic proposition is satisfied ($wl_1$, $wr_1$ and $e_1$ are negated). Notice that the two dead markings of the net are represented in the two aggregates in the middle of the Figure 4. The artificial aggregates corresponding to the presence of dead markings or circuits have not been represented in the figure.

## 5 Evaluation

The performance of three LTL model checkers are compared. The first one, NuSMV [3], allows for the representation of synchronous and asynchronous finite state systems, and for the analysis of specifications expressed in Computation Tree Logic (CTL) and Linear Temporal Logic (LTL), using BDD-based and SAT-based model checking techniques. Only BDD-based LTL components have been used for our experiments. They implement the algorithms presented in [4]. This method consists in: (1) construct the transition relation of the system ;

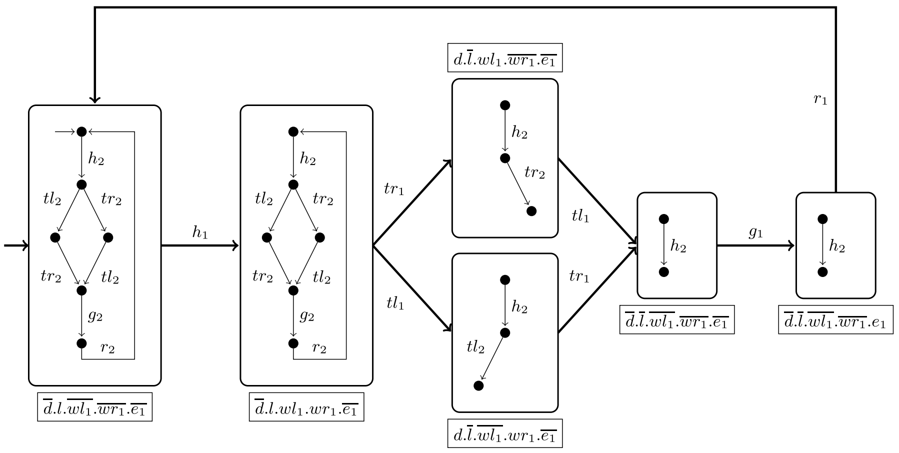

$d.\overline{l}.wl_1.\overline{wr_1}.\overline{e_1}$
$r_1$
$h_2$
$tl_2$
$tr_2$
$g_2$
$r_2$
$h_1$
$tr_1$
$tl_1$
$g_1$
$\overline{d}.l.\overline{wl_1}.\overline{wr_1}.\overline{e_1}$
$\overline{d}.l.wl_1.wr_1.\overline{e_1}$
$d.\overline{l}.\overline{wl_1}.wr_1.\overline{e_1}$
$\overline{d}.\overline{l}.\overline{wl_1}.\overline{wr_1}.\overline{e_1}$
$\overline{d}.\overline{l}.\overline{wl_1}.\overline{wr_1}.e_1$

**Fig. 4.** The constructed SOG for two philosophers and the formula $\Box(wl_1 \wedge wr_1 \Rightarrow \Diamond(e_1))$

(2) translate the LTL formula into an equivalent $\omega$-automaton and construct its transition relation ; (3) construct the synchronized product of the two relations. The decision procedure is then reduced to the verification of a CTL formula with fairness constraints. For our experiments, we have submitted to NuSMV, the encoding of Petri nets by the transition relation as defined by Pastor & al in [16]. On each dead marking, a transition looping on itself has been added to take into account all the maximal sequences.

The second model checker is MC-SOG presented in the previous section and mixing symbolic and explicit algorithms. Notice that the BDD representations of the transition relations used by this tool and NuSMV as well as the order of BDD variables are the same. Notice also that better encodings of considered nets may exist. In particular, the distribution of NuSMV proposes an example for the dining philosophers. This model encodes each philosopher and each fork by an automaton. This allows to significantly decrease the number of BDD variables comparing to the encoding we have chosen. However, a better encoding will be favorable for both tools (MC-SOG and NuSMV). In MC-SOG, the construction of aggregates is realized using this transition relation and the LTL verification is delegated to Spot [9].

The third model checker is also based on Spot (it is distributed with the library as a tutorial under the name CheckPN) but visits the reachability graph instead of a symbolic observation graph. Each state of the reachability graph is encoded by an integer vector. For this model checker also, we have added a transition looping on each dead marking.

The measurements presented in Table 1 concern 4 families of Petri nets. The Petri nets of a given family are obtained by instantiating a parameter (e.g. the number of philosophers). All these examples are taken from [2]. On each of these nets, 100 randomized formulae have been checked. Each formula is of size 8 and takes its atomic propositions randomly among the places of the considered net.

**Table 1.** Experiments on 4 families of Petri nets

| model | | | Symbolic | | Symbolic & Explicit | | | | Explicit | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | (1) | (2) | (3) | (4) | (3) | (5) | (6) | (4) | (5) | (6) | (4) |
| `fms 2` | 3444 | 26 | 801536 | 4.0 | 23207 | 762 | 1249 | 0.9 | 15522 | 50037 | 0.2 |
| `fms 3` | 48590 | 24 | 803790 | 9.2 | 23328 | 1359 | 2897 | 6.8 | 92942 | 346783 | 1.8 |
| `fms 4` | 438600 | 24 | 849616 | 25.7 | 51282 | 2617 | 7074 | 93.2 | 641853 | 2665186 | 15.2 |
| `fms 5` | $2.8 \times 10^6$ | 24 | 863802 | 48.6 | 91254 | 4917 | 15652 | 677.7 | 3391134 | 14956669 | 98.5 |
| `kanban 2` | 4600 | 31 | 772640 | 2.9 | 16715 | 871 | 1450 | 0.4 | 19014 | 108284 | 0.4 |
| `kanban 3` | 58400 | 30 | 770249 | 6.3 | 19185 | 1473 | 3183 | 0.7 | 196986 | 1437346 | 5.8 |
| `kanban 4` | 454475 | 30 | 783286 | 19.1 | 51451 | 2686 | 7121 | 2.9 | 1405537 | 11603892 | 328.0 |
| `kanban 5` | $2.5 \times 10^6$ | 30 | 810446 | 47.0 | 94913 | 4721 | 14317 | 7.5 | | not treated | |
| `kanban 6` | $1.1 \times 10^7$ | 30 | 825585 | 124.4 | 178021 | 7911 | 26247 | 18.9 | | | |
| `philo 4` | 466 | 26 | 735463 | 4.2 | 11946 | 527 | 631 | 0.5 | 4709 | 12736 | 0.1 |
| `philo 6` | 10054 | 25 | 745489 | 11.6 | 18119 | 521 | 683 | 1.1 | 62250 | 293668 | 2.1 |
| `philo 8` | 216994 | 22 | 792613 | 32.3 | 24180 | 552 | 730 | 2.2 | | | |
| `philo 10` | $4.7 \times 10^6$ | 23 | 845180 | 98.7 | 30316 | 546 | 724 | 4.0 | | not treated | |
| `philo 20` | $2.2 \times 10^{13}$ | 23 | 4743606 | 2585.5 | 59689 | 587 | 916 | 41.5 | | | |
| `ring 3` | 504 | 43 | 828719 | 10.8 | 11450 | 1621 | 4388 | 1.3 | 7541 | 24421 | 0.2 |
| `ring 4` | 5136 | 40 | 910641 | 66.6 | 18027 | 1922 | 4066 | 15.2 | 59145 | 295414 | 1.5 |
| `ring 5` | 53856 | 39 | 983840 | 438.0 | 69680 | 9682 | 29638 | 612.8 | 807737 | 4946514 | 181.2 |

| | |
|---|---|
| (1) Number of reachable markings | (4) Verification time in seconds |
| (2) Number of verified formulae | (5) Number of visited states |
| (3) Peak number of live BDD nodes (only an estimation for MC-SOG) | (6) Number of visited transitions |

For each net, we give its number of reachable markings (column numbered (1)) as well as the number of satisfied formulae (2). For each tool, we have measured the time in seconds (4) consumed by the verification of the 100 formulae.

For each tool using the symbolic approach (NuSMV and MC-SOG), we also give the peak number of live BDD nodes (3) (i.e. peak even if aggressive garbage collection is used). Notice that the numbers given for MC-SOG are an estimation. Our implementation is based on the BDD package Buddy which delays the freeing of dead nodes to take advantage of its caches. Then, we have forced frequent garbage collections to estimate the peak. NuSMV is based on the BDD package CUDD which measures the exact memory peak in terms of live BDD nodes if only its immediate freeing function is used. NuSMV exclusively uses this function of CUDD.

For each tool using the explicit approach (MC-SOG and CheckPN) is indicated the number of states (5) and transitions (6) of the 100 $\omega$-automata visited by the LTL emptiness check algorithm. These automata correspond to the synchronized products of the ESOG or reachability graph and the $\omega$-automaton of each formula. Among all the emptiness check algorithms proposed by Spot, we have used the Couvreur's one [8]. The visited states are constructed (and stored) on-the-fly during the LTL verification. Then the number of states corresponds to the memory used and the number of transitions to the time consumed by the emptiness check.

A first remark concerning the obtained results in terms of time is that no model checker has the advantage on the others for all the experiments. NuSMV is the faster for the net `fms` and MC-SOG performs well for the nets `kanban` and `philo` while CheckPN obtains the best results for the net `ring`. The good results of NuSMV on `fms` are interesting. Only the number of tokens in the initial marking depends on the parameter of this net and, in particular, the number of places remains constant. Since the number of BDD variables is logarithmic with respect to the bound of the net when it is linear with respect to the number of places, we can deduce that NuSMV is more sensitive than MC-SOG to a great number of variables. We suspect that the bad results of MC-SOG for the net `fms` are essentially due to the way we construct aggregates and exclude some transitions for the saturation. This technique is particulary efficient when the bound of the places remains reasonable.

If we compare the memory peaks of NuSMV against the ones of MC-SOG, it is clear that NuSMV consumes more. This is due to the fact that MC-SOG only stores a forest of BDD (one by aggregate) corresponding to *Out*$(M)$ while NuSMV has to store the set of all reachable states. However, when the parameters of nets grow, the exponential increase of MC-SOG is more marked than the one of NuSMV (with the exception of the net `philo`).

**Table 2.** Experiments with limited expansion of aggregats

| model | | | Symbolic | | Symbolic & Explicit | | | | Explicit | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | (1) | (2) | (3) | (4) | (3) | (5) | (6) | (4) | (5) | (6) | (4) |
| `fms 5` | $2.8 \times 10^6$ | 24 | 863802 | 48.6 | 91254 | 10882 | 25989 | 52.1 | 3391134 | 14956669 | 98.5 |
| `fms 6` | $1.5 \times 10^7$ | 24 | 885576 | 147.6 | 293518 | 17104 | 45246 | 182.2 | | not treated | |
| `ring 5` | 53856 | 39 | 983840 | 438.0 | 102668 | 56187 | 329075 | 239.3 | 807737 | 4946514 | 181.2 |

Finally, the size of the $\omega$-automata visited by the explicit part (the emptiness checks) of MC-SOG is extremely reduced compared to the one required by CheckPN. It is clear that the computation time (which could be important) of MC-SOG is essentially used for the construction of the aggregates. However, we have seen in the section 3 that the definition of SOG allows some freedom degrees. In particular, the expansion of an aggregate can be stopped at any moment. Doing that, we limit the times required by the computation of aggregate but increase the size of the SOG. In table 2, we present some results for three nets. The expansion of aggregates has been limited by forbidding the firing of arbitrary chosen Petri net transitions during their construction (i.e. these transitions have been added to *Extern*). We can notice that the construction presented in the previous section does not allows the presence of self loop on aggregates. Adding some not needed transitions in *Extern* can lead to such loops but without contradicting the definition of SOG.

In table 2, we can remark that even if the size of visited $\omega$-automata has been increased importantly, the complete checking time has decreased drastically and that, by using this simple heuristic, MC-SOG is now comparable with the two other model checkers on these examples.

## 6 Conclusion

In this paper, we designed and analyzed a new on-the-fly model-checker for $LTL \setminus X$ logic based on state-based symbolic observation graphs. Our approach is hybrid in the sense that the explored state space is represented by a hybrid graph: nodes are encoded symbolically while edges are encoded explicitly. In fact symbolic model-checker is rather common when dealing with computational tree logics (e.g. CTL), however checking LTL properties symbolically is not trivial. To the best of our knowledge the unique symbolic algorithm for LTL logic is the one proposed in [4] (implemented for instance in NuSMV [3]).The advantages of our technique in comparison to this approach is that the computation of a SOG can be done on-the-fly during the emptiness check. Moreover, when the SOG is visited entirely during the model checking (i.e. the property holds), it can be reused for the verification of another formula (at the condition that the set of atomic propositions is included in the one used by the SOG). Experiments show that our approach can outperform both explicit model-checkers and symbolic ones, especially when the freedom degrees related to the SOG building are exploited. However, it would be interesting to try different heuristics to limit more the computation time of aggregates. For instance, one can adapt the BDD variable ordering such that those used to encode the atomic propositions of the formula are consecutive. This could reduce drastically the time used for computing the successors of an aggregate since their values do not change within an aggregate. Other perspectives are to be considered in the near future. For example, it would be interesting to experiment our tool against realistic examples and not only toy ones.

## References

1. Bryant, R.E.: Symbolic boolean manipulation with ordered binary-decision diagrams. ACM Computing Surveys 24(3), 293–318 (1992)
2. Ciardo, G., Lüttgen, G., Siminiceanu, R.: Efficient symbolic state-space construction for asynchronous systems. In: Nielsen, M., Simpson, D. (eds.) ICATPN 2000. LNCS, vol. 1825, pp. 103–122. Springer, Heidelberg (2000)
3. Cimatti, A., Clarke, E.M., Giunchiglia, F., Roveri, M.: NUSMV: A new symbolic model checker. International Journal on Software Tools for Technology Transfer 2(4), 410–425 (2000)
4. Clarke, E.M., Grumberg, O., Hamaguchi, K.: Another look at LTL model checking. Formal Methods in System Design 10(1), 47–71 (1997)
5. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press, Cambridge (2000)
6. Clarke, E.M., McMillan, K.L., Campos, S.V.A., Hartonas-Garmhausen, V.: Symbolic model checking. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 419–427. Springer, Heidelberg (1996)
7. Cleaveland, R., Hennessy, M.: Testing equivalence as a bisimulation equivalence. In: Proceedings of the international workshop on Automatic verification methods for finite state systems, pp. 11–23. Springer, New York (1990)

8. Couvreur, J.-M.: On-the-fly verification of linear temporal logic. In: Woodcock, J.C.P., Davies, J., Wing, J.M. (eds.) FM 1999. LNCS, vol. 1709, pp. 253–271. Springer, Heidelberg (1999)
9. Duret-Lutz, A., Poitrenaud, D.: SPOT: an extensible model checking library using transition-based generalized Büchi automata. In: Proceedings of the 12th IEEE/ACM International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2004), Volendam, The Netherlands, pp. 76–83. IEEE Computer Society Press, Los Alamitos (2004)
10. Geldenhuys, J., Valmari, A.: Techniques for smaller intermediary BDDs. In: Larsen, K.G., Nielsen, M. (eds.) CONCUR 2001. LNCS, vol. 2154, pp. 233–247. Springer, Heidelberg (2001)
11. Goltz, U., Kuiper, R., Penczek, W.: Propositional temporal logics and equivalences. In: CONCUR, pp. 222–236 (1992)
12. Haddad, S., Ilié, J.-M., Klai, K.: Design and evaluation of a symbolic and abstraction-based model checker. In: Wang, F. (ed.) ATVA 2004. LNCS, vol. 3299, pp. 196–210. Springer, Heidelberg (2004)
13. Henzinger, T.A., Kupferman, O., Vardi, M.Y.: A space-efficient on-the-fly algorithm for real-time model checking. In: Sassone, V., Montanari, U. (eds.) CONCUR 1996. LNCS, vol. 1119, pp. 514–529. Springer, Heidelberg (1996)
14. Kaivola, R., Valmari, A.: The weakest compositional semantic equivalence preserving nexttime-less linear temporal logic. In: Cleaveland, W.R. (ed.) CONCUR 1992. LNCS, vol. 630, pp. 207–221. Springer, Heidelberg (1992)
15. Manna, Z., Pnueli, A.: The temporal logic of reactive and concurrent systems. Springer, New York (1992)
16. Pastor, E., Roig, O., Cortadella, J., Badia, R.: Petri net analysis using boolean manipulation. In: Valette, R. (ed.) ICATPN 1994. LNCS, vol. 815, pp. 416–435. Springer, Heidelberg (1994)
17. Peterson, J.L.: Petri Net Theory and the Modeling of Systems. Prentice Hall PTR, Upper Saddle River, NJ, USA (1981)
18. Puhakka, A., Valmari, A.: Weakest-congruence results for livelock-preserving equivalences. In: Baeten, J.C.M., Mauw, S. (eds.) CONCUR 1999. LNCS, vol. 1664, pp. 510–524. Springer, Heidelberg (1999)
19. Rozier, K., Vardi, M.: LTL satisfiability checking. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 149–167. Springer, Heidelberg (2007)
20. Tao, Z.P., von Bochmann, G., Dssouli, R.: Verification and diagnosis of testing equivalence and reduction relation. In: ICNP 1995: Proceedings of the 1995 International Conference on Network Protocols, Washington, DC, USA, p. 14. IEEE Computer Society, Los Alamitos (1995)