http://www-lipn.univ-paris13.fr/~petrucci/PAPERS
In Proc. 8th Int. Conf. on Application of Concurrency to System Design (ACSD'08), Xi'an, China, June 2008.
IEEE Comp. Soc. Press, pages 88-97, 2008.

Modular Construction of the Symbolic Observation Graph

Kais Klai and Laure Petrucci

LIPN, CNRS UMR 7030
Université Paris 13
99 avenue Jean-Baptiste Clément
F-93430 Villetaneuse, France
Email: {kais.klai,laure.petrucci}@lipn.univ-paris13.fr

Abstract

Model checking for Linear Time Logic (LTL) is usually based on converting the (negation of a) property into a Büchi automaton, composing the automaton and the model, and finally checking for emptiness of the language of the composed system. The last step is the crucial stage of the verification process because of the state explosion problem. In this work, we present a solution which builds, in a modular *way, an observation graph represented in a non-symbolic manner but where the nodes are essentially* symbolic *sets of states and the edges either labeled by events occurring in the formula or by synchronization actions between the system components. Due to the small number of events to be observed in a typical formula, this graph has a very moderate size and thus the time complexity for verification is negligible w.r.t. the time to build the observation graph. Experimental results show that our method outperforms both a non-modular generation of the symbolic graph and existing non-symbolic approaches (modular or not).*

I. 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 of the property into a Büchi automaton, 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 i.e. the exponential increase of the number of states w.r.t. the number of system components. Numerous techniques have been proposed to cope with the state explosion problem during the last two decades. Among them, *symbolic model checking* (e.g. [5], [8], [10], [7]) aims at checking the property on a compact representation of the system using binary decision diagrams (BDD) techniques [1], while *modular verification* (e.g. [20], [4], [14], [13]) takes advantage of the modular design of concurrent and distributed systems in order to downsize the verification of the global system to the analysis of its individual components.

In this paper, we present a hybrid framework for checking linear time temporal logic properties of concurrent and distributed systems. Actually, we propose to separately build a symbolic abstraction of each component of the system. Ensure that the abstraction preserves LTL\X properties, and deduce the global abstraction of the global system by synchronization. Finally, we ensure the preservation of LTL\X properties at the global level as well. The symbolic observation graph [10] (*SOG* for short) is a reduced deterministic graph where nodes are symbolic sets and where edges are exclusively labeled by actions occurring in the formula to be checked. It represents an abstraction of the system on which the verification of a LTL\X property is equivalent to the veri-

[0]This work is supported by the University Paris 13 BQR project PROVISO.

fication on the original reachability graph. The main contribution of this paper is to allow the preservation of this property by composition, i.e. the SOGs of the components are built in isolation and a synchronized product is then built in such a way that the graph obtained is equivalent to the classical reachability graph with respect to LTL\X properties. This graph can thus be used by a standard LTL model-checker to check a family of properties: the set of properties involving a subset of the observed actions, i.e. the actions of the formula to be checked.

The paper is structured as follows. In section II, we formalize the symbolic observation graph technique, which was elaborated in a flat setting (i.e. non modular). Then, section III shows how this construction can be achieved in a modular way. It details the different algorithms at stake, and proves the validity of the approach. These algorithms were implemented in a software tool and experiments comparing this approach to both the modular state space technique and the standard reachability graph technique are discussed in section IV. Section V is devoted to discussing our technique w.r.t. related works while Section VI concludes the paper and gives some perspectives.

II. The symbolic observation graph

In [10], the authors have introduced the symbolic observation graphs (*SOG* for short) as an abstraction of the *reachability state graph* of concurrent systems. They have also shown that the verification of an event-based formula of $LTL\backslash X$ (LTL minus the next operator) on the *SOG* is equivalent to the verification of the formula on the classical reachability graph. The construction of the *SOG* is guided by the set of actions occurring in the formula to be checked. Such actions are said to be observed while the other actions of the system are unobserved. Then, the *SOG* is defined as a graph where each node is a set of states linked by unobserved actions and each arc is labeled by an observed action. Nodes of the *SOG* are called *meta-states* and may be represented and managed efficiently using decision diagram techniques (BDDs for instance). Even though the number of meta-states of a SOG is exponential w.r.t. the number of states of the original system, the SOG has a very moderate size in practice. This is due to the small number of actions in a typical formula. Thus, the time complexity of the verification process on the SOG is negligible w.r.t. to its building time.

The technique presented in this paper applies to different kinds of models, that can map to labeled transition systems, e.g. high-level Petri nets. For the sake of simplicity and generality, we chose to present it for labeled transition systems, since the formalism is rather simple.

Definition 1 (Labeled Transition System): A *labeled transition system* (LTS for short) is a 4-tuple $\langle \Gamma, Act, \rightarrow, I \rangle$ where:

- Γ is a finite set of *states* ;
- Act is a finite set of *actions* ;
- $\rightarrow \subseteq \Gamma \times Act \times \Gamma$ is a *transition relation* ;
- $I \subseteq \Gamma$ is a set of *initial states*.

Notations: Let $\langle \Gamma, Act, \rightarrow, I \rangle$ be a labeled transition system and let s, s' be states in Γ, S, S' subsets of Γ and a an action in Act.

- $s \xrightarrow{a} s'$ denotes that $(s, a, s') \in \rightarrow$;
- $s \not\rightarrow$ denotes dead states i.e. states $s \in \Gamma$ satisfying $\nexists a \in Act, \nexists s' \in \Gamma$ s.t. $s \xrightarrow{a} s'$;
- $S \xrightarrow{a} S'$ denotes that ($\forall s \in S, \exists s' \in S'$ s.t $s \xrightarrow{a} s' \wedge \forall s' \in S', \exists s \in S$ s.t $s \xrightarrow{a} s'$) ;
- $s \xrightarrow{a}$, means that $\exists s' \in \Gamma : s \xrightarrow{a} s'$;
- *Enable(s)* denotes the set of actions a such that $s \xrightarrow{a}$.

In [10], the authors presented the algorithm to construct the SOG but did not give a formal definition for it. Here we formally define *meta-states* before giving a formal definition of the SOG associated with a LTS, given a set Obs of *observed* actions. We shall see later how this set is chosen. The other actions, in $UnObs = Act \backslash Obs$, are said to be *unobserved*.

Definition 2 (Meta-state): Let $\mathcal{T} = \langle \Gamma, Act, \rightarrow, I \rangle$ be a labeled transition system. Let $Act = Obs \cup UnObs$ be partitioned into observed and unobserved actions. Then, a *meta-state* is a triple $M = \langle S, l, d \rangle$ defined by:

1) S is a non-empty subset of Γ where:
   a) $\forall s \in S$, $\exists i \in I$ and $\exists \sigma \in Act^*$ s.t. $i \xrightarrow{\sigma} s$;
   b) $\forall s \in S, \forall s' \in \Gamma, \forall \sigma \in UnObs^*$: $s \xrightarrow{\sigma} s' \Rightarrow s' \in S$;
2) $l = true$ iff the subgraph induced by S contains a cycle ;
3) $d = true$ iff S contains a dead state.

In the following $M.S$, $M.l$ and $M.d$ denote the attributes of a meta-state M.

Explanation:
(1a): A *meta-state* contains only reachable states.
(1b): Given a meta-state M, for each state $s \in M.S$, all states s' reachable from s by a sequence of unobserved actions are also in $M.S$.

(2): l indicates that there exists a cycle with unobserved actions only, within meta-state M.
(3): d indicates that the set of states S contains (at least) one dead state f.

Definition 3 (Symbolic Observation Graph): A *symbolic observation graph* is a LTS $\mathcal{G} = \langle \Gamma', Act', \rightarrow', I' \rangle$ associated with a LTS $\mathcal{T} = \langle \Gamma, Obs \cup UnObs, \rightarrow, I \rangle$ such that:

1) Γ' is a finite set of meta-states;
2) $Act' = Obs$;
3) $\rightarrow' \subseteq \Gamma' \times Act' \times \Gamma'$ is a transition relation, such that:
   a) $\forall M, M' \in \Gamma'$ s.t. $M \xrightarrow{a}{}' M'$ for some $a \in Act'$ if one of the following conditions holds:
      i) $M \neq M' \Rightarrow \forall s \in M.S, \forall s' \in \Gamma, s \xrightarrow{a} s' \Rightarrow s' \in M'.S$;
      ii) $M = M' \Rightarrow$ Let $Pred = \{S \subseteq M.S \mid S = I \vee \exists M' \in \Gamma', \exists S' \subseteq M'.S$ and $\exists a' \in Act'$ s.t. $M' \xrightarrow{a'} M \wedge S' \xrightarrow{a'} S\}$, then $\forall S \in Pred, \exists s \in S, \exists s_1 \in M.S$ s.t. $s \xrightarrow{\sigma_1} s_1 \xrightarrow{\sigma_2} s_1$ with $\sigma_1 \in UnObs^*$, $\sigma_2 \in (UnObs \cup \{a\})^+$.
   b) $\forall s, s' \in \Gamma, \forall a \in Obs\ s \xrightarrow{a} s' \Rightarrow \exists M, M' \in \Gamma'$ s.t. $s \in M.S$, $s' \in M'.S$ and $M \xrightarrow{a} M'$.
4) $I' = \{M_0\}$, where the meta-state M_0 satisfies $I \subseteq M_0.S$ and $\nexists M \in \Gamma' \setminus M_0$ s.t. $M_0.S \subseteq M.S$.

Explanation:
(1): The *nodes* of the symbolic observation graph are meta-states.
(2): An arc of the SOG is labeled by an observed action.
(3a): There exists an arc from a meta-state M to M' labeled by a if one of the two following conditions holds:
(3(a)i) M and M' are different, then each state of M enabling a has its successor in M'.
(3(a)ii) stands for a loop on a given meta-state (i.e. $M \xrightarrow{a}{}' M$), such a loop is permitted if we guarantee that it is possible to execute a possibly infinitely often each time M is reached (from any path of the SOG). $Pred(M)$ is the set of subsets of states in $M.S$ that can be reached from an external meta-state by some event a', then the loop is permitted if there exists a circuit $C = s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \cdots \xrightarrow{a_n} s_1$ inside $M.S$ involving no observed actions except a that is reachable from a state $s \in S$ for each $S \in Pred(M)$.
(3b): All the "observed" arcs in the original LTS are preserved in the SOG.
(4): All the initial states of the original labeled transition system are in the *initial* meta-state of the symbolic observation graph.

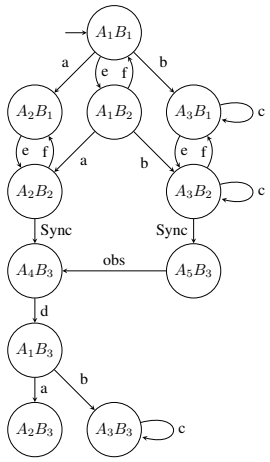
A simplified algorithm to generate the SOG is described in Algorithm 1. It uses a stack $Waiting$ containing meta-states to be processed. Function $\texttt{metastate}(S)$ constructs the meta-state associated with the states in S. It first adds all the states obtained by firing sequences of unobserved actions only (according to definition 2.1b). Then d and l are easily set using the algorithms presented in [10]. The algorithm also adds the meta-state to $Waiting$ and Γ' if it is a new one. Function $\texttt{arc}(M, a, M')$ adds an arc, labeled by a, from the meta-state M to the meta-state M', to the transition relation $\rightarrow'$.
Function $\texttt{succ}\ (S, a)$ returns the set of successors of states in S by action a (i.e. $\texttt{succ}\ (S, a) = \{s' \mid \exists s \in S$ s.t. $s \xrightarrow{a} s'\}$).

Algorithm 1: Symbolic Observation Graph

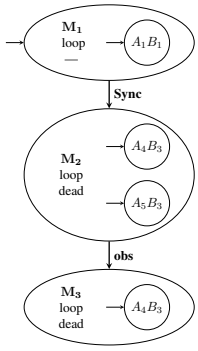
Require: a LTS $\langle \Gamma, Obs \cup UnObs, \rightarrow, I \rangle$
Ensure: SOG $\langle \Gamma', Obs, \rightarrow', I' \rangle$
{Initial meta-state};
$M_0 \leftarrow \texttt{metastate}(I)$;
$I' \leftarrow \{M_0\}$;
$\Gamma' \leftarrow \{M_0\}$;
$Waiting$.**Push**(M_0);
{Process states in $Waiting$};
while $Waiting \neq \emptyset$ **do**
  $Waiting$.**Pop**($M = \langle S, l, d \rangle$);
  for all $a \in Obs$ **do**
    if $\texttt{enabled}\ (M.S, a)$ **then**
      $S' \leftarrow \texttt{succ}(M.S, a)$;
      $M' = \texttt{metastate}(S')$;
      if $\exists M'' \in \Gamma'$ s.t. $M'' = M'$ **then**
        $\texttt{arc}(M, a, M'')$;
      else
        $\texttt{arc}(M, a, M')$;
        $\Gamma \leftarrow \Gamma \cup \{M'\}$;
        $Waiting$.**Push**(M')
      end if
    end if
  end for
end while

Example: Figure 1 illustrates an example of LTS (Figure 1(a)) and the corresponding SOG (Figure 1(b)) for $Obs = \{\text{Sync}, \text{obs}\}$. The obtained SOG consists of 3 meta-states M_1, M_2 and M_3, an arc from M_1 to M_2 labeled by Sync and one from M_2 to M_3 labeled by obs. Meta-state M_1 contains loops but no dead state, whereas M_2 and M_3 have both a loop and a dead state. Note that state A_4B_3 is stored in both M_2 and M_3.

The equivalence between checking a given property on the observation graph and checking



(a) Example of LTS



(b) SOG of LTS in figure 1(a) with $Obs = \{\text{Sync}, \text{obs}\}$

Fig. 1. A LTS and its SOG

it on the original labeled transition system is ensured by the preservation of three kinds of sequences: the infinite observed sequences, the finite maximal sequences and the infinite divergent sequences. Thus, the observation graph obtained preserves the validity of formulae written in classical Manna-Pnueli linear time logic [15] (LTL) without the "next operator" (because of the abstraction of the immediate successors) (see for instance [17], [9], [11]).

Proposition 1: [10] Let φ be a formula from LTL$\setminus$X. Let $\mathcal{G}$ be the symbolic observation graph associated with a labeled transition system $\mathcal{T}$, with Obs containing all the actions in φ. Then: $\mathcal{T} \models \varphi \Leftrightarrow \mathcal{G} \models \varphi$.

III. A modular construction of the observation graph

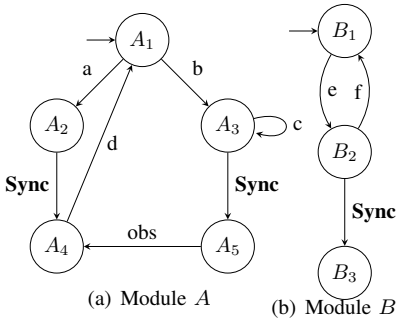
Large systems are often designed in a modular way, thus adopting a software engineering approach and allowing for component reuse. This section constitutes the core of the paper. Starting from several LTS which synchronize over a common set of actions, it shows how to check LTL$\setminus$X properties using the symbolic observation graph of the whole system. The construction proposed here also follows a modular approach, and thus avoids complete construction of the synchronized product of the system modules.

Definition 4 (LTS synchronized product): Let $\mathcal{T}_i = \langle \Gamma_i, Act_i, \rightarrow_i, I_i \rangle, i = 1, 2$ be two LTS. The *synchronized product* of $\mathcal{T}_1$ and $\mathcal{T}_2$ over their common (synchronized) actions $Act_1 \cap Act_2$, is the labelled transition system $\mathcal{T}_1 \times \mathcal{T}_2 = \langle \Gamma, Act, \rightarrow, I \rangle$ such that:

1) $\Gamma = \Gamma_1 \times \Gamma_2$;
2) $Act = Act_1 \cup Act_2$;
3) $\rightarrow$ is the transition relation, defined by:
   $\forall (s_1, s_2) \in \Gamma : (s_1, s_2) \xrightarrow{a} (s'_1, s'_2) \Leftrightarrow$
   - $s_1 \xrightarrow{a}_1 s'_1 \wedge s_2 \xrightarrow{a}_2 s'_2$ if $a \in Act_1 \cap Act_2$
   - $s_1 \xrightarrow{a}_1 s'_1 \wedge s_2 = s'_2$ if $a \in Act_1 \setminus Act_2$
   - $s_1 = s'_1 \wedge s_2 \xrightarrow{a}_2 s'_2$ if $a \in Act_2 \setminus Act_1$
4) $I = I_1 \times I_2$.

Point 3 of Definition 4 explicits the synchronization operation. An action in both LTS $\mathcal{T}_1$ and $\mathcal{T}_2$ is synchronized, i.e. it must be enabled and occurs simultaneously in both LTS. Otherwise, if the action belongs to only one of the LTS, its transition relation is preserved. The set of states is reduced to reachable states only i.e. $\Gamma = \{(s_1, s_2) \in \Gamma_1 \times \Gamma_2 \mid \exists (i_1, i_2) \in I_1 \times I_2, \exists \sigma \in Act^* : (i_1, i_2) \xrightarrow{\sigma} (s_1, s_2)\}$. Similarly, the set of actions is reduced to those that can effectively take place in the synchronized product: $Act = \{a \in Act_1 \cup Act_2 \mid \exists s, s' \in \Gamma, \xrightarrow{a}(s, s')\}$. The algorithm constructing the synchronized product of two LTS is rather simple. Starting from the initial states in I, it constructs the successors according to the transition relation of definition 4.

Example: Let us consider the two modules A and B in figure 2. Their synchronization on action Sync leads to the labelled transition system of figure 1(a).



(a) Module A (b) Module B

Fig. 2. Two modules synchronizing on Sync

Our approach consists in synchronizing symbolic observation graphs. Therefore, we define the product of two meta-states.

Definition 5 (Meta-states product): Let $\mathcal{G}_i = \langle \Gamma'_i, Obs_i, \rightarrow'_i, I'_i \rangle, i = 1, 2$ be two

SOGs associated with two LTSs $\mathcal{T}_i$. Let $M_i = \langle S_i, l_i, d_i \rangle$ be a meta-state of $\mathcal{G}_i$. The *product meta-state* $M = \langle S, l, d \rangle = M_1 \times M_2$ is defined with respect to $Obs_1 \cap Obs_2$ as follows: $S = S_1 \times S_2$, $l = l_1 \vee l_2$ and d =true iff $\exists (s_1, s_2) \in S$ s.t. (s_1, s_2) is a dead state of $\mathcal{T}_1 \times \mathcal{T}_2$.

While the computation of the loop attribute of the product meta-state is straightforward, the computation of the deadlock attribute is rather complex. In fact, it is well-known that deadlock-freeness is not preserved by synchronization. Figure 3 illustrates such a situation, where two modules (Figure 3(a) and Figure 3(b)) without deadlocks lead, by synchronization over {**c, d, e**}, to a synchronized product (Figure 3(c)) containing two dead states. Here, we take advantage of the local deadlock properties computed on the meta-states separately. Algorithm 2 computes the deadlock attribute of a product meta-state $M = M_1 \times M_2$. A straightforward case is detected when both attributes d_1 and d_2 are true (lines 1–3). In this case, there exists a dead state in both M_1 and M_2, e.g. s_1 and s_2 respectively. Then the product (s_1, s_2) is a dead state of S (hence d=true). Now, assume that $d_i = false$ (for a meta-state M_i). First, we detect states in S_i that enable only synchronization transitions which are not enabled in M_j (for $j \neq i$). Such states are dead states and d_i is thus changed to true. This is achieved in the first loop of Algorithm 2 (lines 4–8) as a first stage before the computation of the deadlock attribute of the product meta-state. If d_1 and d_2 are both true S contains a dead state (lines 9–11). In the second loop (lines 12–16), a second case is handled: assume there exists a dead state $s_1 \in S_1$. Then, if there exists a state $s_2 \in S_2$ enabling only synchronization actions, the product state (s_1, s_2) is a dead state.
Finally, the case of absence of deadlock in both meta-states is taken into account (lines 17–20). In this case, the computation of the deadlock attribute of the composed meta-state is performed as follows: for $i = 1, 2$, we detect the existence of subsets, namely $sync_i$, of synchronization actions enabled in M_i that satisfy two properties: (1) $sync_i$ is enabled in M_j (for $j \neq i$) ; (2) There exists a subset of states in S_i enabling actions of $sync_i$ only.

As soon as two such subsets $sync_1$ and $sync_2$ are detected (from the study of M_1 and M_2 respectively), if $sync_1$ and $sync_2$ are disjoint, then one can deduce that the composed meta-state contains a dead state. In fact, the state (s_1, s_2) where s_1 and s_2 are states from S_1 and S_2 that enable only $sync_1$ and $sync_2$ respectively is dead because no synchronization is possible ($sync_1 \cap sync_2 = \emptyset$).

Algorithm 2: Computing the deadlock attribute of a product meta-state

Require: a product meta-state $M = M_1 \times M_2$
Ensure: Computes d
1: **if** $d_1 \wedge d_2$ **then**
2: **return true**;
3: **end if**
4: **for all** $i, j \in \{1, 2\}\, s.t.\, i \neq j$ **do**
5: **if** $(d_i = false) \wedge (\exists s_i \in S_i$ s.t. $Enable(s_i) \subseteq (Sync \setminus Enable(S_j)))$ **then**
6: $d_i \leftarrow true$
7: **end if**
8: **end for**
9: **if** $d_1 \wedge d_2$ **then**
10: **return true**;
11: **end if**
12: **for all** $i, j \in \{1, 2\}\, s.t.\, i \neq j$ **do**
13: **if** $(d_i \wedge \neg d_j) \wedge (\exists s_j \in S_j$ s.t. $Enable(s_j) \subseteq Sync_j)$ **then**
14: **return true**;
15: **end if**
16: **end for**
17: **if** $\exists sync_1, sync_2 \subseteq$(*Enable*$(S_i) \cap$ *Enable*(S_j)) s.t. $(sync_1 \cap sync_2 = \emptyset)$ and, for $i \in \{1, 2\}$ $((Enable^{-1}(sync_i) \setminus Enable^{-1}(Act \setminus Sync)) \cap S_i \neq \emptyset$ **then**
18: **return true**;
19: **end if**
20: **return false**;

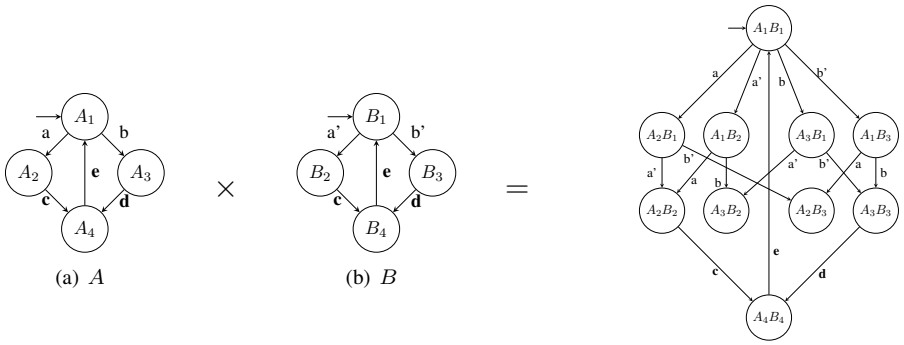
Finally, the case of absence of deadlock in both meta-states is taken into account (lines 17–20). In this case, the computation of the deadlock attribute of the composed meta-state is performed as follows: for $i = 1, 2$, we detect the existence of subsets, namely $sync_i$, of synchronization actions enabled in M_i that satisfy two properties: (1) $sync_i$ is enabled in M_j (for $j \neq i$) ; (2) There exists a subset of states in S_i enabling actions of $sync_i$ only.

As soon as two such subsets $sync_1$ and $sync_2$ are detected (from the study of M_1 and M_2 respectively), if $sync_1$ and $sync_2$ are disjoint, then one can deduce that the composed meta-state contains a dead state. In fact, the state (s_1, s_2) where s_1 and s_2 are states from S_1 and S_2 that enable only $sync_1$ and $sync_2$ respectively is dead because no synchronization is possible ($sync_1 \cap sync_2 = \emptyset$).

The most expensive operation of Algorithm 2 is the test at line 17. Its worst case complexity is $(2^{|Enable(S_1) \cap Sync| + |Enable(S_2) \cap Sync|})$. However, in practice the number of outgoing observed actions of a given meta-state is very small. Moreover, all the operations of Algorithm 2 can be done symbolically by using decision diagram techniques (e.g. BDDs).
In the following, we prove the correctness of Algorithm 2.

Theorem 1: Let M_1 and M_2 be two meta-states and let $M = M_1 \times M_2$ be the product meta-state as defined in Definition 1. Then the value d computed by Algorithm 2 is true iff $S = S_1 \times S_2$ contains a dead state.

Proof: $\Rightarrow$ Straightforward (see the description of Algorithm 2).
$\Leftarrow$ Let $(s_1, s_2) \in S_1 \times S_2$ be a dead state. Then, for $i \in \{1, 2\}$ no local action (i.e. in $Act_i \setminus (Act_1 \cap Act_2)$) is enabled in s_i. Let $sync_i$ denote the set (possibly empty) of synchronization actions enabled in s_i. Then $sync_1 \cap sync_2 = \emptyset$.



(a) A (b) B (c) $A \times B$

Fig. 3. Non-preservation of deadlock-freeness by synchronisation

We distinguish the following three cases:

1) $sync_1 = sync_2 = \emptyset$: this case is handled in lines 1–3 of Algorithm 2 and the value of d is $true$;
2) $sync_1 = \emptyset \ \oplus \ sync_2 = \emptyset$: this case is handled in the first loop of Algorithm 2 (lines 4–8) and the value of d is set to $true$ (lines 9–11) ;
3) $sync_1 \neq \emptyset \wedge sync_2 \neq \emptyset$: this case is treated in lines 17–18 of Algorithm 2 and the value of d is $true$.

■

The construction of the symbolic observation graph of a synchronized product of modules consists in first building the SOGs of the individual modules and then synchronizing them. Therefore, we now define the synchronization of symbolic observation graphs.

Definition 6 (SOG synchronized product): Let $\mathcal{G}_i = \langle \Gamma'_i, Obs_i, \rightarrow'_i, I'_i \rangle, i = 1, 2$ be the two symbolic observation graphs associated with $\mathcal{T}_i = \langle \Gamma_i, Obs_i \cup UnObs_i, \rightarrow_i, I_i \rangle$. Then, the synchronized product $\mathcal{G} = \mathcal{G}_1 \times \mathcal{G}_2 = \langle \Gamma', Act', \rightarrow', I' \rangle$ is such that:

1) $\Gamma' = \Gamma'_1 \times \Gamma'_2$
2) $Act' = Obs_1 \cup Obs_2$
3) $\forall M = M_1 \times M_2 \in \Gamma' : M \xrightarrow{a}{}' M' = M'_1 \times M'_2 \Leftrightarrow$
   - $M_1 \xrightarrow{a}{}'_1 M'_1 \ \wedge \ M_2 \xrightarrow{a}{}'_2 M'_2$ if $a \in Obs_1 \cap Obs_2$
   - $M_1 \xrightarrow{a}{}'_1 M'_1 \wedge M_2 = M'_2$ if $a \in Obs_1 \setminus Obs_2$
   - $M_1 = M'_1 \wedge M_2 \xrightarrow{a}{}'_2 M'_2$ if $a \in Obs_2 \setminus Obs_1$
4) $I' = I'_1 \times I'_2$

Once again, only reachable meta-states are kept. The actions are those observed in the two SOGs (point 2). Note that the synchronization actions have to be a part of the observed action in each SOG (i.e. $Act_1 \cap Act_2 \subseteq Obs_i$ for $i = 1, 2$). The synchronization is similar to that for the product of LTS (point 3). The initial meta-state is obtained by composition of the initial meta-states

Algorithm 3: Synchronous product of 2 SOG

Require: $\mathcal{G}_i = \langle \Gamma'_i, Obs_i, \rightarrow'_i, I'_i \rangle$ for $i = 1, 2$
Ensure: $\mathcal{G}_1 \times \mathcal{G}_2 = \langle \Gamma', Act', \rightarrow', I' \rangle$
{Initial meta-state};
1: $Waiting$.**Push**($I'_1 \times I'_2$);
{Process states in $Waiting$};
2: **while** $Waiting \neq \emptyset$ **do**
3: $Waiting$.**Pop**($M = M_1 \times M_2$);
4: **for all** $a \in Act'$ **do**
5: **if** $a \in Obs_1 \cap Obs_2$ **then**
6: **if** $M_1 \xrightarrow{a}{}'_1 M'_1 \wedge M_2 \xrightarrow{a}{}'_2 M'_2$ **then**
7: $M' = M'_1 \times M'_2$;
8: **end if**
9: **else**
10: **if** $a \in Obs_1 \setminus Obs_2$ **then**
11: **if** $M_1 \xrightarrow{a}{}'_1 M'_1$ **then**
12: $M' = M'_1 \times M_2$;
13: **end if**
14: **else**
15: **if** $M_2 \xrightarrow{a}{}'_2 M'_2$ **then**
16: $M_1 \times M'_2$;
17: **end if**
18: **end if**
19: **end if**
20: **if** $\exists M'' \in \Gamma'$ s.t. $M'' = M'$ **then**
21: $\texttt{arc}(M, a, M'')$;
22: **else**
23: $\texttt{arc}(M, a, M'')$;
24: $\Gamma' \leftarrow \Gamma \cup \{M'\}$;
25: $Waiting$.**Push**(M')
26: **end if**
27: **end for**
28: $Waiting \leftarrow Waiting \setminus \{M\}$;
29: **end while**

of the modules SOGs.

Algorithm 3 implements the synchronized product of two symbolic observation graphs. This algorithm is very similar to the construction of LTSs synchronized product.

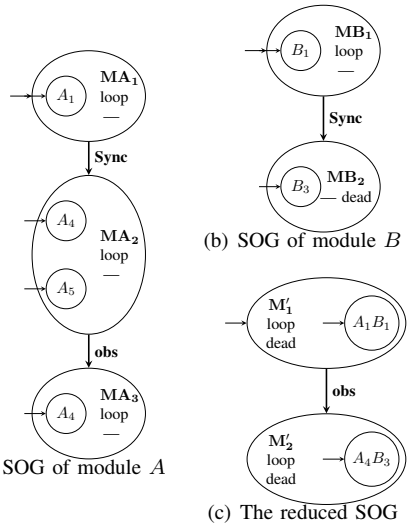
Property 1: Let $\mathcal{T}_i, i = 1,2$ be two LTS, with $\mathcal{T}_i = \langle \Gamma_i, Obs_i \cup UnObs_i, \rightarrow_i, I_i \rangle$, and let $Sync \subseteq Obs_i$ be a subset of synchronization actions. Then, $SOG(\mathcal{T}_1 \times \mathcal{T}_2, Obs_1 \cup Obs_2)$ and $SOG(\mathcal{T}_1, Obs_1) \times SOG(\mathcal{T}_2, Obs_2)$ are isomorphic.

Proof: The property is a consequence of definitions 4, 3, 5, 6 and theorem 1. ■

Using Proposition 1 and Property 1, one can easily deduce the following result.

Corollary 1: Let $\mathcal{T} = \mathcal{T}_1 \times \mathcal{T}_2$ be a labelled transition system obtained by synchronizing two LTS $\mathcal{T}_1$ and $\mathcal{T}_2$. Let $\mathcal{G}$ be the symbolic observation graph obtained by synchronizing the symbolic observation graphs associated with $\mathcal{T}_1$ and $\mathcal{T}_2$ respectively. Let φ be a formula from LTL\X. Then: $\mathcal{T} \models \varphi \Leftrightarrow \mathcal{G} \models \varphi$.

Example: Figure 4 shows the symbolic observation graphs of modules A and B from figure 2, when observing $Obs = \{\text{Sync}, \text{obs}\}$. The synchronized product of these two SOGs is the SOG in figure 1(b).



(a) SOG of module A

(b) SOG of module B

(c) The reduced SOG

Fig. 4. The reduced SOG of two synchronized modules

The verification process is as follows: first, we apply the previous algorithms to obtain the product SOG, observing both the synchronized actions and those appearing in the formula (the observed actions). Then, we reduce it further so as to keep only observed actions. This is sketched in algorithm 4. The formula to verify is checked on this reduced symbolic observation graph by using any standard (on-the-fly) model checker tool.

Algorithm 4: Checking a LTL\X formula

Require: 2 LTS: $\mathcal{T}_1$ and $\mathcal{T}_2$ **input** : LTL\X formula φ
Ensure: satisfaction of φ {SOG of $\mathcal{T}_1$ and $\mathcal{T}_2$};
1: Obs =action in φ;
2: $SOG_1 \leftarrow SOG(\mathcal{T}_1, Obs \cup (Act_1 \cap Act_2))$;
3: $SOG_2 \leftarrow SOG(\mathcal{T}_2, Obs \cup (Act_1 \cap Act_2))$;
{Product of SOGs};
4: $SOG \leftarrow SOG_1 \times SOG_2$;
{Reduction of SOG};
5: **for all** $M = \langle S, l, d \rangle \in SOG$ **do**
6: **for all** $a \in Act_1 \cap Act_2$: $M \xrightarrow{a} M' = \langle S', l', d' \rangle$ **do**
7: $M \leftarrow \langle S \cup S', l \vee l', d \vee d' \rangle$;
8: $SOG \leftarrow SOG \setminus \{M'\}$;
9: **end for**
10: **end for**

Example: Let us now suppose we want to check the LTL\X formula: $\varphi = \mathsf{GF}\text{obs}$. This formula expresses that, always, obs will be eventually executed in the future. The first step of the algorithm constructs the symbolic observation graph with the synchronized actions plus those of the formula as observed actions. Hence, in this example, $Obs = \{\text{Sync}, \text{obs}\}$. This leads to the SOG in figure 1(b). Then, the second step reduces it even further so as to retain only the actions in the formula, i.e. obs, leading to the graph in figure 4(c). It consists of two meta-states in both of which a loop and a deadlock exist, and an arc labeled by obs. Since both meta-states contain a deadlock, we can easily conclude that formula φ does not hold.

IV. Experimental results

A symbolic observation graph software tool has been implemented, in which the user can choose between flat or modular construction. The Buddy BDD package (http://sourceforge.net/projects/buddy) is exploited in order to represent meta-states compactly, to implement the transition relation and to symbolically detect deadlocks and loops within meta-states. Five case studies have been selected for experimenting the modular construction of the symbolic observation graph. Because of lack of space, we refer the reader to [16] for a detailed description of these examples and the corresponding Petri net models. We think that

Model	param	Occurrence Graph	
		N_{OG}	A_{OG}
5 AGVs		$30,965,760$	$345,784,320$
Database	n	$n \times 3^{n-1}+1$	$2(n-1) \times 3^{n-2}+2n$
Philosophers	2	3	4
	3	4	6
	n	$N_{OG}(n-1)$ $+N_{OG}(n-2)$	$2n \times F_n$, $F_2=F_3=1$ $F_n=F_{n-1}+F_{n-2}$
Poisoned philosophers	2	21	38
	3	99	264
	n	$4N_{OG}(n-1)+3N_{OG}(n-2)+6$	
Railway	n	$4(n^2+n+1)$	$4n^3+22n^2+16n+11$

TABLE I. Occurrence graphs

the composition schemes of the chosen examples, with different coupling degrees between modules, are rich enough to claim the significance of the results obtained.

The modular observation graph construction is compared to the flat occurrence graph (the explicit reachability graph), the modular state space presented in [14] and the non-modular version of the symbolic observation graph. The comparison criterion is the size of the obtained graph in terms of number of nodes and arcs. When possible, we give these numbers as a function of the parameter n which represents the number of the synchronized modules. Moreover, the number of bdd nodes is given for the symbolic observation graph techniques in order to measure the size of the generated bdds. For these two techniques (Table III) the size of the generated graph, in terms of number of nodes ($N_{(m)sog}$) and number of arcs ($A_{(m)sog}$), is the same as long as the set of the observed actions is the same. However, the number of bdd nodes grows exponentially (in some cases) w.r.t. n for the SOG technique (column bdd_{sog}) but remains constant in the MSOG (column bdd_{msog}). This is explained by the fact that the same bdd variable can be reused for each processed module. This is the main improvement w.r.t. the SOG technique beside the reusability of the SOG generated separately for each module.

Note that, even if the theoretical complexity of the SOG techniques is exponential w.r.t. the occurrence graph size, this limit is almost never reached in practice. The obtained graphs have a negligible size w.r.t. the occurrence graph size (Table I). In addition, the MSOGs are smaller than the graphs generated by the modular state space approach (Table II) especially when the modules are loosely coupled (e.g. the AGV example). In such cases the number of synchronization actions is relatively small and one can expect a spectacular reduction in the size of the MSOG obtained. Finally, one can note that when there is no local behavior (all actions of the system are observed) the obtained (M)SOGs are isomorphic to the occurrence graphs (e.g. the traditional dining philosophers example).

V. Discussion and related work

During the last 20 years, many researchers have worked on the use of abstraction and compositionality to tackle the explosion problem of model-checking temporal properties on concurrent systems. In [11], the authors established that the CFFD-equivalence is exactly the weakest equivalence preserving next time-less linear temporal logic. The SOG technique is based on this equivalence and we have presented in this paper a modular construction that preserves the CFFD semantics. The computation of the deadlock attribute of a product meta-state can be avoided when its loop attribute is proved to be true. In fact, in this case, we switch to the NDFD equivalence which does not distinguish livelock from deadlock while maintaining the correctness of the analysis result.

The symbolic observation graph presents a new and improved variant of acceptance graphs [6], [2] (or dually refusal graphs [19]) that are used to test equivalence and reduction (traces languages inclusion). In addition, each state of the acceptance graph is labeled with a minimized acceptance set (or dually minimal refusal set). Each element of the minimal acceptance set is a set of observed actions enabled from some state of the node (possibly after executing some sequence of unobserved actions). Here, only one single bit is needed to detect a deadlock within a given meta-state and can be computed symbolically in only one BDD operation for each observed action. The

Model	param	Modular State Space	
		N_{MSS}	A_{MSS}
5 AGVs		900	2,687
Database	n	$6n+3$	$4n$
Philosophers	2	11	4
	3	16	6
	n	$N_{OG}(n)+4n$	$A_{OG}(n)$
Poisoned philosophers	2	33	30
	3	99	171
	n	$4N_{MSS}(n-1)+N_{MSS}(n-2)+8n+4$	
Railway	n	$\frac{n(n+1)}{2}+5n+10$	$n^2+8n+10$

TABLE II. Modular State Spaces

Model	param	MSOG/SOG			
		$N_{(m)sog}$	$A_{(m)sog}$	bdd_{sog}	bdd_{msog}
5 AGVs		12	18	344	78
Database	n	$n+1$	$2n$	$\frac{5}{2}n^2+\frac{25}{2}n-4$	14
Philosophers	2	3	4	22	4
	3	4	6	42	4
	4	7	16	86	4
	5	11	30	150	4
	6	18	60	258	4
	n	$N_{OG}(n)$	$A_{OG}(n)$	$2bdd_{sog}(n-1)$ $-bdd_{sog}(n-3)$	4
Poisoned philosophers	2	17	24	138	24
	3	75	162	686	24
	n	$4N_{(m)sog}(n-1)$ $+N_{(m)sog}(n-2)+4$		$4bdd_{sog}(n-1)$ $+bdd_{sog}(n-2)+126$	24
Railway	n	$\frac{n(n+1)}{2}+2n+3$	n^2+4n+1	$\frac{1}{2}n^3+\frac{9}{2}n^2+16n+17$	8

TABLE III. Modular and flat Symbolic Observation Graphs

acceptance graphs cannot be represented symbolically since, to compute the acceptance sets associated with nodes, one needs to deal with the individual states.

Other approaches based on abstraction refinement and deadlock detection have also been widely studied in various contexts (e.g. [18], [12], [3], ...). However, on the one hand, the conservative abstractions require in general an iterated abstraction refinement mechanism in order to establish specification satisfaction. On the other hand, to the best of our knowledge, none of the deadlock detection approaches involve symbolic abstraction or modularity in an automated form as our present work does.

Modular state spaces were studied in [14] in the case of systems composed of semi-autonomous subsystems. The idea there was to start from a system designed in a modular way and construct the state space of the complete system in a similar fashion: one local state space per module and a synchronization graph showing their interactions. The technique was applied to a problem of controller design, where some of the actions could be controlled and others not. This is very similar to the notion of observable actions, and the approach advocated was also to lift these actions to the global (i.e. synchronization) level, so that both synchronized and controllable actions are visible in the synchronization graph and only there. One can then wonder whether there is a difference between the synchronization graph and the symbolic observation graph. Indeed, if we construct the SOG for the example of [14], the result is similar to the synchronization graph. However, applying the modular state space technique to the example from this paper leads to a synchronization graph larger than the SOG. This is due to the grouping of states within meta-states which is intrinsically different in these two approaches. Another difference lies in properties verification. With modular state spaces, the graph construction does not depend on the formula to check and the verification is done *a posteriori*. Here, the construction of the graph depends on the property, and some characteristics of meta-states are stored on-the-fly.

VI. Conclusion

The Symbolic Observation Graph technique aims at checking LTL\X properties while avoiding the state space explosion problem. Earlier work [10] has defined symbolic observation graphs and has shown that they include sufficient information to check LTL\X properties. However, systems are often so large that they are designed in a modular fashion.

Hence, this paper has addressed the construction of symbolic observation graphs in a modular way, consistent with the design approach. First, actions to be observed are deduced from the formula to check, and are added to the actions performing synchronisation with other modules. A SOG can then be built for each module. These are smaller than the original models since they do not include all actions and all states. Therefore, they are much more manageable. We defined the synchronization of the SOGs, so that it is isomorphic to the SOG that would be obtained from the complete model (where modules are synchronized). In order to check the property, the synchronization transitions are not necessary anymore. Thus the SOG is reduced before the model-checking phase.

The SOG contains meta-states that carry both some states and some properties (loops and deadlocks). Composing meta-states hence requires composing these properties, which is not always a straightforward issue. We have shown how to optimize this calculus in many cases. The algorithms presented here have been implemented in a prototype tool. It has proven efficient for standard examples, and should now be applied to larger case studies. The next step would be to build a model checker based on the (Modular) SOG.

References

[1] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[2] Ufuk Celikkan and Rance Cleaveland. Computing diagnostic test for incorrect processes. In *Proceedings of the IFIP TC6/WG6.1 Twelth International Symposium on Protocol Specification, Testing and Verification XII*, pages 263–277, Amsterdam, The Netherlands, The Netherlands, 1992. North-Holland Publishing Co.

[3] Sagar Chaki, Edmund M. Clarke, Joël Ouaknine, Natasha Sharygina, and Nishant Sinha. Concurrent software verification with states, events, and deadlocks. *Formal Asp. Comput.*, 17(4):461–483, 2005.

[4] Søren Christensen and Laure Petrucci. Modular analysis of Petri nets. *Computer Journal*, 43(3):224–242, 2000.

[5] Edmund M. Clarke, Kenneth L. McMillan, Sérgio Vale Aguiar Campos, and Vassili Hartonas-Garmhausen. Symbolic model checking. In *Int. Conf. on Computer Aided Verification (CAV)*, volume 1102 of *Lecture Notes in Computer Science*, pages 419–427. Springer, 1996.

[6] R. Cleaveland and M. Hennessy. Testing equivalence as a bisimulation equivalence. In *Proceedings of the international workshop on Automatic verification methods for finite state systems*, pages 11–23, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[7] Jean-Michel Couvreur. A BDD-like implementation of an automata package. In *Int. Conf. on Implementation and Application of Automata (CIAA)*, volume 3317 of *Lecture Notes in Computer Science*, pages 310–311. Springer, 2004.

[8] Jaco Geldenhuys and Antti Valmari. Techniques for smaller intermediary BDDs. In *Int. Conf. on Concurrency Theory (CONCUR)*, volume 2154 of *Lecture Notes in Computer Science*, pages 233–247. Springer, 2001.

[9] Ursula Goltz, Ruurd Kuiper, and Wojciech Penczek. Propositional temporal logics and equivalences. In *CONCUR*, pages 222–236, 1992.

[10] Serge Haddad, Jean-Michel Ilié, and Kais Klai. Design and evaluation of a symbolic and abstraction-based model checker. In *Int. Conf. on Automated Technology for Verification and Analysis (ATVA)*, volume 3299 of *Lecture Notes in Computer Science*. Springer, 2004.

[11] Roope Kaivola and Antti Valmari. The weakest compositional semantic equivalence preserving nexttime-less linear temporal logic. In *CONCUR*, pages 207–221, 1992.

[12] Ferhat Khendek and Gregor von Bochmann. Merging behavior specifications. *Formal Methods in System Design*, 6(3):259–293, 1995.

[13] Kais Klai, Serge Haddad, and Jean-Michel Ilié. Modular verification of Petri nets properties: A structure-based approach. In *Int. Conf. on Formal Techniques for Networked and Distributed Systems (FORTE)*, volume 3731 of *Lecture Notes in Computer Science*, pages 189–203. Springer, 2005.

[14] Charles Lakos and Laure Petrucci. Modular analysis of systems composed of semiautonomous subsystems. In *Int. Conf. on Application of Concurrency to System Design (ACSD)*, pages 185–194. IEEE Comp. Soc. Press, 2004.

[15] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992.

[16] Laure Petrucci. Cover picture story: Experiments with modular state spaces. In *Petri Net newsletter*, pages 5–10, 2005.

[17] Antti Puhakka and Antti Valmari. Weakest-congruence results for livelock-preserving equivalences. In *CONCUR*, pages 510–524, 1999.

[18] A. W. Roscoe, Paul H. B. Gardiner, Michael Goldsmith, J. R. Hulance, D. M. Jackson, and J. B. Scattergood. Hierarchical compression for model-checking csp or how to check 10^{20} dining philosophers for deadlock. In *TACAS*, pages 133–152, 1995.

[19] Z. P. Tao, G. von Bochmann, and R. Dssouli. Verification and diagnosis of testing equivalence and reduction relation. In *ICNP '95: Proceedings of the 1995 International Conference on Network Protocols*, page 14, Washington, DC, USA, 1995. IEEE Computer Society.

[20] Antti Valmari. Composition and abstraction. In *MOVEP*, volume 2067 of *Lecture Notes in Computer Science*, pages 58–98. Springer, 2000.