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-
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 typical formula. Thus, the time complexity of the verification process on the SOG is negligible w.r.t. 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:

- \( \Gamma \) 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 \( \Gamma \), \( S, S' \) subsets of \( \Gamma \) 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 \( \exists a \in Act, \exists s' \in \Gamma \) s.t. \( s \xrightarrow{a} s' \);
- \( S \xrightarrow{a} S' \) denotes that \( (\forall s \in S, \exists s' \in S' \) s.t. \( s \xrightarrow{a} s' \) \) and \( \exists s' \in S, s' \in S \) s.t. \( s \xrightarrow{a} s' \);
- \( s \not\xrightarrow{a} \), means that \( \exists s' \in \Gamma : s \not\xrightarrow{a} s' \);
- \( Enable(s) \) 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 \( UnObs = Act \setminus Obs \), are said to be unobserved.

**Definition 2 (Meta-state):** Let \( T = \langle \Gamma, Act, \rightarrow, I \rangle \) 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 \( \Gamma \) where:
   a. \( \forall s \in S, \exists i \in I \) and \( \exists \sigma \in Act^* \) s.t. \( l \xrightarrow{\sigma} s \);
   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): 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 $M'$.
- $3(a)ii$: 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 $C = s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \ldots \xrightarrow{a_n} s_1$ 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 $metastate(S)$ 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 $\Gamma'$ if it is a new one. Function $arc(M,a,M')$ adds an arc, labeled by $a$, from the meta-state $M$ to the meta-state $M'$, to the transition relation $\rightarrow'$. Function $succ(S,a)$ returns the set of successors of states in $S$ by action $a$ (i.e. $succ(S,a) = \{s' | \exists s \in S.t. s \xrightarrow{a} s'\}$).

**Algorithm 1: Symbolic Observation Graph**

**Require:** a LTS $(\Gamma,Obs \cup UnObs,\rightarrow,I)$

**Ensure:** SOG $(\Gamma',Obs,\rightarrow',I')$

1. $M_0 \leftarrow metastate(I)$;
2. $\Gamma' \leftarrow \{M_0\}$;
3. $\Gamma' \leftarrow \{M_0\}$;
4. $Waiting.Pop(M_0)$;
5. **while** Waiting $\neq \emptyset$ **do**
6. $Waiting.Pop(M = (S,l,d))$;
7. **for all** $a \in Obs$ **do**
8. **if** enabled $(M.S,a)$ **then**
9. $s' \leftarrow succ(M.S,a)$;
10. $M' = metastate(S')$;
11. **if** $\exists M'' \in \Gamma'$ s.t. $M'' = M'$ **then**
12. $arc(M,a,M''')$;
13. **else**
14. $arc(M,a,M''')$;
15. $\Gamma' \leftarrow \Gamma' \cup \{M''\}$;
16. $Waiting.Pop(M'')$;
17. **end if**
18. **end if**
19. **end for**
20. **end while**

**Example:** Figure 1 illustrates an example of LTS (Figure 1(a)) and the corresponding SOG (Figure 1(b)) for $Obs = \{Sync, obs\}$. 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
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 $\varphi$ be a formula from LTL\(\setminus X\). Let $G$ be the symbolic observation graph associated with a labeled transition system $T$, with $Obs$ containing all the actions in $\varphi$. Then: $T \models \varphi \Leftrightarrow G \models \varphi$.

### 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 LTSs 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 $T_i = (\Gamma_i, Act_i, \rightarrow_i, I_i), i = 1, 2$ be two LTS. The synchronized product of $T_1$ and $T_2$ over their common (synchronized) actions $Act_1 \cap Act_2$, is the labeled transition system $T_1 \times T_2 = \langle \Gamma, Act, \rightarrow, I \rangle$ such that:

1. $\Gamma = \Gamma_1 \times \Gamma_2$;
2. $Act = Act_1 \cup Act_2$;
3. $\rightarrow$ is the transition relation, defined by:
   \[
   \forall (s_1, s_2) \in \Gamma : (s_1, s_2) \xrightarrow{a} (s'_1, s'_2) \iff
   \begin{cases}
   s_1 \xrightarrow{a} s'_1 \land s_2 \xrightarrow{a} s'_2 & \text{if } a \in Act_1 \cap Act_2 \\
   s_1 \xrightarrow{a} s'_1 \land s_2 = s'_2 & \text{if } a \in Act_1 \setminus Act_2 \\
   s_1 = s'_1 \land s_2 \xrightarrow{a} s'_2 & \text{if } a \in Act_2 \setminus Act_1
   \end{cases}
   \]
4. $I = I_1 \times I_2$.

Point 3 of Definition 4 explicits the synchronization operation. An action in both LTS $T_1$ and $T_2$ 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: $Act = \{a \in Act_1 \cup Act_2 \mid \exists s, s' \in \Gamma, s_1 \xrightarrow{a} (s, s')\}$. The algorithm constructing the synchronized product of two LTS is rather simple. Starting from the initial states in $I$, it constructs the synchronized product 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 labeled transition system of figure 1(a).

![Fig. 1. A LTS and its SOG](image)

![Fig. 2. Two modules synchronizing on Sync](image)
Algorithm 2: Computing the deadlock attribute of a product meta-state

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>if $d_1 \land d_2$ then</td>
</tr>
<tr>
<td>2.</td>
<td>return true;</td>
</tr>
<tr>
<td>3.</td>
<td>end if</td>
</tr>
<tr>
<td>4.</td>
<td>for all $i, j \in {1, 2}$ s.t. $i \neq j$ do</td>
</tr>
<tr>
<td>5.</td>
<td>if $(d_i = false) \land (\exists s_i \in S_i \text{ s.t.} \ Enable(s_i) \subseteq (Sync \setminus Enable(S_j)))$ then</td>
</tr>
<tr>
<td>6.</td>
<td>$d_i \leftarrow$ true</td>
</tr>
<tr>
<td>7.</td>
<td>end if</td>
</tr>
<tr>
<td>8.</td>
<td>end for</td>
</tr>
<tr>
<td>9.</td>
<td>if $d_1 \land d_2$ then</td>
</tr>
<tr>
<td>10.</td>
<td>return true;</td>
</tr>
<tr>
<td>11.</td>
<td>end if</td>
</tr>
<tr>
<td>12.</td>
<td>for all $i, j \in {1, 2}$ s.t. $i \neq j$ do</td>
</tr>
<tr>
<td>13.</td>
<td>if $(d_i \land \neg d_j) \land (\exists s_j \in S_j \text{ s.t.} \ Enable(s_j) \subseteq Sync_{j})$ then</td>
</tr>
<tr>
<td>14.</td>
<td>return true;</td>
</tr>
<tr>
<td>15.</td>
<td>end if</td>
</tr>
<tr>
<td>16.</td>
<td>end for</td>
</tr>
<tr>
<td>17.</td>
<td>if $\exists sync_1, sync_2 \subseteq (Enable(S_i) \cap Enable(S_j))$ s.t. $(sync_1 \cap sync_2 = \emptyset)$ and, for $i \in {1, 2}$ $((Enable^{-1}(sync_1) \setminus Enable^{-1}(Act \setminus Sync)) \cap S_i \neq \emptyset$ then</td>
</tr>
<tr>
<td>18.</td>
<td>return true;</td>
</tr>
<tr>
<td>19.</td>
<td>end if</td>
</tr>
<tr>
<td>20.</td>
<td>return false;</td>
</tr>
</tbody>
</table>

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 $2|Enable(S_i)\cap Enable(S_j)|$. 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 $sync_1 \cap sync_2 = \emptyset$. 

SOGs associated with two LTSs $T_i$. Let $M_i = (S_i, \ell_i, d_i)$ be a meta-state of $G_i$. The product meta-state $M = (S, \ell, d) = M_1 \times M_2$ is defined with respect to $Obs_1 \cap Obs_2$ as follows: $S = S_1 \times S_2$, $\ell = \ell_1 \lor \ell_2$ and $d$ is true iff $\exists (s_1, s_2) \in S$ s.t. $(s_1, s_2)$ is a dead state of $T_1 \times T_2$.

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 $(s_1, s_2)$
We distinguish the following three cases:

1) sync₁ = sync₂ = ∅: this case is handled in lines 1–3 of Algorithm 2 and the value of d is true.

2) sync₁ = ∅ ⊕ sync₂ = ∅: 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) sync₁ ≠ ∅ and sync₂ ≠ ∅: 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 $G_i = (\Gamma_i, Ob_i, \rightarrow_i, I_i^i), i = 1, 2$ be the two symbolic observation graphs associated with $T_i = (\Gamma_i, Ob_i \cup \text{UnOb}_i, \rightarrow_i, I_i)$. Then, the synchronized product $G = G_1 \times G_2 = (\Gamma', \text{Act}', \rightarrow', I')$ is such that:

1) $\Gamma' = \Gamma_1' \times \Gamma_2'$
2) $\text{Act}' = Ob_1 \cup Ob_2$
3) $\forall M = M_1 \times M_2 \in \Gamma': M \rightarrow M' = M_1 \times M_2'$
   - $M_1 \rightarrow_{M_1} M_3' \land M_2 \rightarrow_{M_2} M_2'$ if $a \in Ob_1 \cap Ob_2$
   - $M_1 \rightarrow_{M_1} M_3' \land M_2 = M_2'$ if $a \in Ob_1 \setminus Ob_2$
   - $M_1 = M_1' \land M_2 \rightarrow_{M_2} M_2'$ if $a \in Ob_2 \setminus Ob_1$
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., Act₁ \cap Act₂ ⊆ Obᵢ 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.
of the modules SOGs.

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

Property 1: Let \( T_i, i = 1, 2 \) be two LTS, with \( T_i = (I_i, Obs_i \cup \text{UnObs}_i, -i, I_i) \), and let \( \text{Sync} \subseteq \text{Obs}_i \) be a subset of synchronization actions. Then, \( \text{SOG}(T_1 \times T_2, \text{Obs}_1 \cup \text{Obs}_2) \) and \( \text{SOG}(\text{Sync}, \text{Obs}_1) \times \text{SOG}(T_2, \text{Obs}_2) \) 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 \( T = T_1 \times T_2 \) be a labelled transition system obtained by synchronizing two LTS \( T_1 \) and \( T_2 \). Let \( G \) be the symbolic observation graph obtained by synchronizing the symbolic observation graphs associated with \( T_1 \) and \( T_2 \) respectively. Let \( \varphi \) be a formula from \( \text{LTL} \backslash X \). Then: \( T \models \varphi \iff G \models \varphi \).

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

![Fig. 4. The reduced SOG of two synchronized modules](image)

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.

Example: Let us now suppose we want to check the \( \text{LTL} \backslash X \) formula: \( \varphi = \text{G}\text{obs} \). 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, \( \text{Obs} = \{\text{Sync, obs}\} \). 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 \( \varphi \) 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
### TABLE I. Occurrence graphs

<table>
<thead>
<tr>
<th>Model</th>
<th>param</th>
<th>(N_{OG})</th>
<th>(A_{OG})</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 AGVs</td>
<td></td>
<td>30,965,760</td>
<td>345,784,320</td>
</tr>
<tr>
<td>Database</td>
<td>(n)</td>
<td>(n \times 3^{n-1} + 1)</td>
<td>(2(n - 1) \times 3^{n-2} + 2n)</td>
</tr>
<tr>
<td>Philosophers</td>
<td>(n)</td>
<td>(N_{OG}(n - 1) + N_{OG}(n - 2))</td>
<td>(2n \times F_n, F_2 = F_3 = 1)</td>
</tr>
<tr>
<td>Poisoned</td>
<td>(n)</td>
<td>(4N_{OG}(n - 1) + 3N_{OG}(n - 2) + 6)</td>
<td>(38)</td>
</tr>
<tr>
<td>Railway</td>
<td>(n)</td>
<td>(4(n^2 + n + 1))</td>
<td>(4n^3 + 22n^2 + 16n + 11)</td>
</tr>
</tbody>
</table>

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
TABLE II. Modular State Spaces

<table>
<thead>
<tr>
<th>Model</th>
<th>param</th>
<th>$N_{MSS}$</th>
<th>$A_{MSS}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 AGVs</td>
<td></td>
<td>900</td>
<td>2,687</td>
</tr>
<tr>
<td>Database</td>
<td>$n$</td>
<td>$6n + 3$</td>
<td>$4n$</td>
</tr>
<tr>
<td>Philosophers</td>
<td>2</td>
<td>11</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>16</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>$n$</td>
<td>$N_{OG}(n) + 4n$</td>
<td>$A_{OG}(n)$</td>
</tr>
<tr>
<td>Poisoned</td>
<td>2</td>
<td>33</td>
<td>30</td>
</tr>
<tr>
<td>philosophers</td>
<td>3</td>
<td>99</td>
<td>171</td>
</tr>
<tr>
<td>Railway</td>
<td>$n$</td>
<td>$\frac{n(\alpha + 1)}{2} + 5n + 10$</td>
<td>$n^2 + 8n + 10$</td>
</tr>
</tbody>
</table>

TABLE III. Modular and flat Symbolic Observation Graphs

<table>
<thead>
<tr>
<th>Model</th>
<th>param</th>
<th>$N_{(m)}_{sog}$</th>
<th>$A_{(m)}_{sog}$</th>
<th>$\text{bdd}_{sog}$</th>
<th>$\text{bdd}_{msog}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 AGVs</td>
<td></td>
<td>12</td>
<td>18</td>
<td>344</td>
<td>78</td>
</tr>
<tr>
<td>Database</td>
<td>$n$</td>
<td>$n + 1$</td>
<td>$2n$</td>
<td>$\frac{3}{2}n^2 + \frac{3}{2}n - 4$</td>
<td>14</td>
</tr>
<tr>
<td>Philosophers</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>12</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>4</td>
<td>6</td>
<td>42</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>/</td>
<td>16</td>
<td>86</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>11</td>
<td>30</td>
<td>150</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>18</td>
<td>60</td>
<td>258</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>$n$</td>
<td>$N_{OG}(n)$</td>
<td>$A_{OG}(n)$</td>
<td>$2\text{bdd}_{sog}(n-1)$</td>
<td>$-\text{bdd}_{sog}(n-3)$</td>
</tr>
<tr>
<td>Poisoned philosophers</td>
<td>2</td>
<td>17</td>
<td>24</td>
<td>138</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>75</td>
<td>162</td>
<td>686</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>$n$</td>
<td>$4N_{(m)}<em>{sog}(n-1)$ + $N</em>{(m)}_{sog}(n-2) + 4$</td>
<td>$4\text{bdd}<em>{sog}(n-1)$ + $\text{bdd}</em>{sog}(n-2) + 126$</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>Railway</td>
<td>$n$</td>
<td>$\frac{n(n+1)}{2} + 2n + 3$</td>
<td>$n^2 + 4n + 1$</td>
<td>$\frac{1}{2}n^3 + \frac{3}{2}n^2 + 16n + 17$</td>
<td>8</td>
</tr>
</tbody>
</table>

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\(\neg X\) 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\(\neg X\) 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 synchronization 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