Modular Verification of Inter-enterprise Business Processes

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

Hanen Ochi
LIPN, CNRS UMR 7030
Université Paris 13
99 avenue Jean-Baptiste Clément
F-93430 Villetaneuse, France
Email: hanen.ochi@lipn.univ-paris13.fr

***Abstract*— In this paper, we propose to adapt the Symbolic Observation Graphs (SOG) based approach in order to abstract, to compose and to check Inter-Enterprise Business Processes (IEBP). Each component (local process) is represented by a SOG where only the collaboration actions of the process are visible while its local behavior and its private structure are hidden. The entire IEBP is then abstracted by the composition of the components' abstractions (i.e., their SOGs). The main result of this paper is to demonstrate that the composition of the SOGs is deadlock free if and only if the original IEBP is deadlock free. We implemented our adaptation of the SOG construction and compared our abstraction and modular verification approach with the Operating Guidelines technique. The obtained results strengthen our belief that the SOGs are suitable to abstract and compose business processes especially when these are loosely coupled.**

***Keywords*-Process composition; abstraction; verification; deadlock-freeness.**

I. INTRODUCTION

Recently, the trend in software architecture is to build Inter-Enterprise Business Processes (IEBP) modularly: Each process is designed separately and then the whole IEBP is obtained by composition. Even if such a modular approach is intuitive and facilitates the design problem, it poses two main problems: First, it is necessary to find an abstraction of the process that respect the privacy of the underlying enterprise (by hiring its internal organisation) and, at the same time, that supply enough information allowing to decide whether the collaboration with some partner is possible or not (safe or not). The second problem is that the correct behavior of each business process of the IEBP taken alone does not guarantee a correct behavior of the composed IEBP (i.e., properties are not preserved by composition). Thus, based on the abstraction of two (or more) processes, we shoud be able to say whether the composed process has the desired behavior or not (in our case, is deadlock free or not).

Proving correctness of the (unknown) composed process is strongly related to the model checking problem of a system model. Among others, the *symbolic observation graph* [5] based approach has proven to be very helpful for efficient model checking in general. Since it is heavily based on abstraction techniques and thus hides detailed information about system components that are not relevant for the correctness decision, it is promising to transfer this concept to the problem rised in this paper.

A SOG is a graph whose construction is guided by a subset of *observed* actions. The nodes of a SOG are *aggregates* hiding a set of states which are connected with non observed actions. The arcs of a SOG are exclusively labeled with observed actions.

The work presented in this paper is in line with those presented in [1] and [2]: How to adapt the SOG's structure in order to abstract and to compose business processes. Such an adaptation is achieved by attaching to each aggregate a (locally computed) sufficient and necessary information for detecting deadlocks that are possibly caused by the composition. The main contribution of this paper is to design a symbolic algorithm (based on sets operations) allowing an efficient computation of this information. This allowed to strengthen the conviction that SOGs represent a suitable abstraction since it respects the consraints mentioned above. Indeed, by observing only the collaborative activities of a process, publishing the corresponding SOG allows to hide its internal strcuture. The analysis power of the SOGs allows, in addition, to check the correctness of a composite process. The composition of SOGs is immediately suitable for synchronously composed processes. However, we can consider asynchronous composition as described in [1], [2]. The key idea is, when combining two processes, to involve a third process, representing the interface, into the composition stage. Such a component consists of buffers and the corresponding sending and receiving actions. Taken separately, this component is an infinite state system in the general case. However, since the whole IEBP is supposed to have finitely many states, only its reachable part is visited during an on-the-fly composition with both processes.

This paper is organized as follows: Section II presents some preliminary notions on WF-nets and labeled transition systems. Section III recalls the symbolic observation graphs and their application on business processes. Composition of SOGs and checking the deadlock freeness on the obtained synchronised product is the issue of Section IV. Section V is devoted to the implementation of our approach in addition to experimental results. In Section VI, we discuss some related works and compare our technique to existing ones. Finally, Section VII concludes the paper and presents some aspects of the future work.

II. Preliminaries

The technique presented in this paper applies to various kinds of process models that can map to labeled transition systems, e.g., Petri nets and, in particular, WorkFlow Petri nets (WF-nets) [11]. Since other modeling languages, which are more frequently used in practice, map to Petri nets, our approach is relevant for a very broad class of modeling languages. Applying our technique does not mean to construct labeled transition systems explicitly. Instead, abstractions of labeled transition systems are directly constructed from the original process models.

Definition 1 (Labeled Transition System): $A\ Labeled\ Transition\ System\ (LTS\ for\ short\ )\ is\ a\ 5-tuple < \Gamma, Act, \rightarrow, I, F > where:$

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

In this paper, we restrain the set of states Γ to those that are reachable from the initial state. Moreover, we assume that a final state f is terminal (it has no successor) and that the set of actions Act is partionned into two disjoint subsets Obs (observed actions) and $UnObs$ (unobserved actions). Below, we present some useful notations:

- For $s, s' \in \Gamma$ and $a \in Act$, we denote by $s \xrightarrow{a} s'$ that $(s, a, s') \in \rightarrow$.
- If $\sigma = a_1 a_2 \cdots a_n$ is a sequence of actions, $\overline{\sigma}$ denotes the set of actions occurring in σ, while $|\sigma|$ denotes the length of σ. $s \xrightarrow{\sigma} s'$ denotes that $\exists s_1, s_2, \cdots s_{n-1} \in \Gamma$: $s \xrightarrow{a_1} s_1 \xrightarrow{a_2} \cdots s_{n-1} \xrightarrow{a_n} s'$.
- The set *Enable(s)* denotes the set of actions a such that $s \xrightarrow{a} s'$ for some state s'. For a set of states S, *Enable(S)* denotes $\bigcup_{s \in S} Enable(s)$.
- $\pi = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \cdots$ is used to denote a path of a LTS.
- $s \not\rightarrow$, for $s \in (\Gamma \setminus F)$, denotes that s is a dead state, i.e., $Enable(s) = \emptyset$.
- $Sat(s) = \{s' \mid s \xrightarrow{\sigma} s' \wedge \overline{\sigma} \subseteq UnObs\}$ is the set of states that are reachable from a state s by unobserved actions only. For $S \subseteq \Gamma$, $Sat(S) = \bigcup_{s \in S} Sat(s)$.
- $s \not\Rightarrow$, for $s \in \Gamma$, denotes that either no final state in F is reachable from s, or no state of $Sat(s)$ enables an observed action, i.e., $Enable(Sat(s)) \cap Obs = \emptyset$. Conversely, $s \Rightarrow$ denotes $\neg(s \not\Rightarrow)$.
- A finite path $C = s_1 \xrightarrow{\sigma} s_n$ is said to be a *cycle* if $s_n = s_1$ and $|\sigma| \geq 1$.
  If $\overline{\sigma} \subseteq UnObs$ then C is said to be a *livelock*. If, in addition, $s_1 \not\Rightarrow$ then C is called a *strong livelock* (a terminal cycle). Otherwise it is called a *weak livelock*.

If $s \not\Rightarrow$ for $s \in (\Gamma \setminus F)$, only a dead state or a *strong livelock* are reachable from s. In this paper we assume that a strong livelock behavior is equivalent to a deadlock. These two behaviors are not distinguished and both are called deadlock.

Consequently, if we want to check whether a given state s is a dead state or not, we need to check wether the predicate $s \not\Rightarrow$ holds or not. We say that we are interested in the *observed behavior* of s: (1) could s lead to the firing of some observed transitions in the future? (2) could s lead to a final state in the future? For this purpose, a *virtual* observed action, called $term$, is added to the observed actions Obs, it mentions that the system terminates proprely. The *Observed behavior*, namely λ, is then defined as a particular mapping applied to the set of states of a LTS as follows:

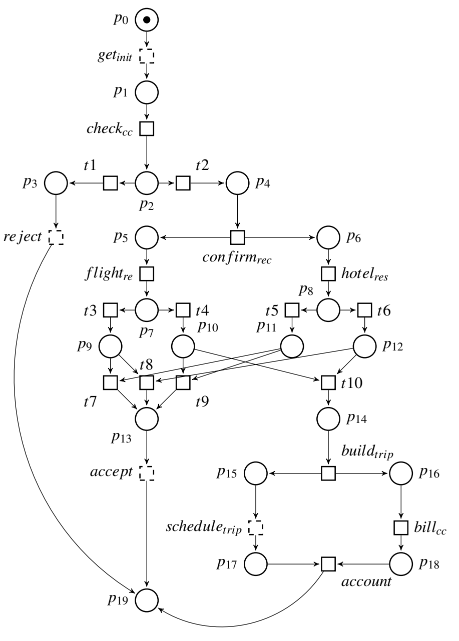
Definition 2 (Observed behavior mapping):
Let $\mathcal{T} = \langle \Gamma, Obs \cup UnObs, \rightarrow, I, F \rangle$ be a LTS. We define:

1) $\lambda_{\mathcal{T}} : \Gamma \rightarrow 2^{Obs}$
$$\lambda_{\mathcal{T}}(s) = \begin{cases} (Enable(Sat(s)) \cap Obs) \cup \{term\} \\ \text{if } F \cap Sat(s) \neq \emptyset \\ (Enable(Sat(s)) \cap Obs) \ \text{otherwise} \end{cases}$$
2) $\lambda_{\mathcal{T}}^{\cap} : 2^{\Gamma} \rightarrow 2^{Obs}$
$\lambda_{\mathcal{T}}^{\cap}(S) = \bigcap_{s \in S} \lambda_{\mathcal{T}}(s)$
3) $\lambda_{\mathcal{T}}^{\subseteq} : 2^{\Gamma} \rightarrow 2^{2^{Obs}}$
$\lambda_{\mathcal{T}}^{\subseteq}(S) = \{\lambda_{\mathcal{T}}^{\cap}(Q) \mid \emptyset \subset Q \subseteq S\}$
4) $\lambda_{\mathcal{T}}^{\min} : 2^{\Gamma} \rightarrow 2^{2^{Obs}}$
$\lambda_{\mathcal{T}}^{\min}(S) = \{X \in \lambda_{\mathcal{T}}^{\subseteq}(S) \mid \not\exists Y \in \lambda_{\mathcal{T}}^{\subseteq}(S) : \ Y \subset (X \setminus \{term\})\}$

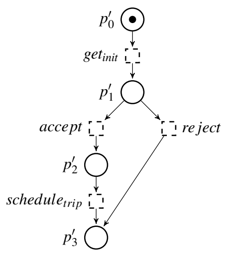
Informally, for each state s of a LTS $\mathcal{T}$, (1) the observed behavior of s, $\lambda_{\mathcal{T}}(s)$, stands for the set of observed actions which can be executed from s, possibly via a sequence of unobserved actions. In addition, *term* is a member of $\lambda_{\mathcal{T}}(s)$ if and only if a final state is reachable from s using unobserved actions only. (2) The observed behavior $\lambda_{\mathcal{T}}^{\cap}$ associated with a set of states S is the intersection of the observed behaviors of its elements. It contains the set of observed actions that are possible from each state of S. (3) $\lambda_{\mathcal{T}}^{\subseteq}(S)$ is a set of sets of observed actions such that each is the result of $\lambda_{\mathcal{T}}^{\cap}$ applied to a nonempty subset Q of S. (4) Finally, $\lambda_{\mathcal{T}}^{\min}(S)$ contains the minimal subsets of $\lambda_{\mathcal{T}}^{\subseteq}(S)$ w.r.t. the inclusion relation not concerning the *term* action. For instance, if there exist two states $s, s' \in S$ such that $\lambda(s) = \emptyset$ and $\lambda(s') = \{term\}$, then both subsets would appear as elements of $\lambda_{\mathcal{T}}^{\min}(S)$. This allows to distinguish whether a dead state or a final state is reached in S (in this case both kinds of state are reachable).

Thanks to the observed behavior, the deadlock freeness of a LTS can be reduced to check whether there exists a state s such that $\lambda_{\mathcal{T}}(s)$ contains the empty set. Similarly, a final state is reachable from a given state s iff $term$ belongs to its observed behavior.

Running example :
We use an example of two business processes (taken from [9]), the trip reservation and the costumer, to illustrate the problem raised in this work. Figure 1(a) illustrates the WF-net associated with the trip reservation's process while Figure1(b) illustrates the WF-net associated with a costumer's process. The corresponding LTSs contain 13 nodes and 36 edges, and 4 nodes and 4 edges, respectively. We choosed such two examples in order to illustrate how SOG-based approach depends on the number of observed actions. In fact, the first contains a big proportion of unboserved actions (17/21), while, in the second all actions are observed. The two processes can collaborate (using dashed transitions) in order to form an IEBP.



(a) WF-net of trip reservation



(b) WF-net of customer

Figure 1. The WF-nets of a trip reservation and a costumer

III. SOG : Symbolic Observation Graph

In this section, we recall the formal definition of a SOG associated with a LTS. We first define what is an *aggregate*: a node of the SOG. Compared to the first definition of SOGs (see [5]), the aggregates are here completed with the *observed behavior* of the hidden states. We will establish that this is the sufficient and necessary information allowing to detect possible deadlock states that can appear by composition. Recall that the deadlock freeness property is not preserved by composition: two deadlock free processes could lead, after composition, to a composite process with a dead state.

Definition 3 (aggregate):
Let $\mathcal{T} = \langle \Gamma, Act, \rightarrow, s_0, F \rangle$ be a labeled transition system with $Act = Obs \cup UnObs$. An *aggregate* is a couple $a = \langle S, \lambda \rangle$ defined as follows:

1) S is a nonempty subset of Γ s.t. $s \in S \Rightarrow Sat(s) \subseteq S$;
2) $\lambda = \lambda^{\min}_{\mathcal{T}}(S)$.

Informally, an aggregate a is defined as a couple (S, λ) where $a.S$ is its set of states (connected with unobserved actions) and $a.\lambda$ its observed behavior. The observed behavior associated with an aggregate a can help to know whether a contains a dead state ($\emptyset \in a.\lambda$) as well as whether a final state belongs to a ($\exists S' \subseteq a.\lambda$ s.t. $term \in S'$). In Section V, we propose a symbolic (set-based) algorithm allowing to efficiently compute the observed behavior of an aggregate.

Definition 4 (Symbolic Observation Graph):
A *symbolic observation graph SOG($\mathcal{T}$)* associated with a LTS $\mathcal{T} = \langle \Gamma, Obs \cup UnObs, \rightarrow, I, F \rangle$ is a LTS $\langle \mathcal{A}, Act', \rightarrow', I', F' \rangle$ such that:

1) $\mathcal{A}$ is a finite set of aggregates s.t.:
   a) There is an aggregate $a_0 \in \mathcal{A}$ s.t. $a_0.S = Sat(I)$;
   b) For each $a \in \mathcal{A}$ and for each $o \in Obs$ the set $\{s' \notin a.S \mid \exists s \in a.S, s \xrightarrow{o} s'\}$ is not empty if and only if it is a pairwise disjoint union of nonempty sets $S_1 \ldots S_k$ and for $i = 1 \ldots k$, there is an aggregate $a_i \in \mathcal{A}$ s.t. $a_i.S = Sat(S_i)$ and $(a, o, a_i) \in \rightarrow'$;
   c) For each aggregate $a \in \mathcal{A}$, the $a.\lambda$ attribute is computed following Definition 3;
2) $Act' = Obs$;
3) $\rightarrow' \subseteq \Gamma' \times Act' \times \Gamma'$ is the transition relation, obtained by applying 1b;
4) $I' = \{a_0\}$ (s.t. $a_0.S = Sat(\{I\})$;
5) $F' = \{a \in \Gamma' \mid \exists Q \in a.\lambda; term \in Q\}$.

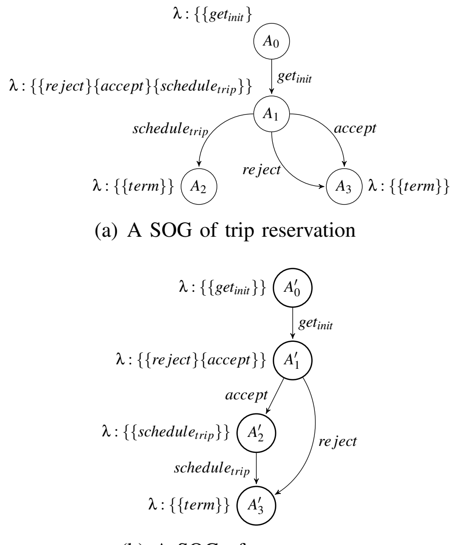
Point 1b of Definition 4 deserves explanation: Given an aggregate a and an observed action o, the set of successors obtained by firing o from states of $a.S$ is partioned in disjoint subsets. For each of these subsets, there exists an aggregate in the SOG obtained by saturation on the states of the subset. For each such an aggregate a' there exists an arc from a to a' labeled with o. Thus, the SOG is non deterministic: an aggregate could have two successors by the same observed transition.

The construction of a SOG following the definition can be started by the initial aggregate a_0, then the SOG will be updated iteratively by adding new aggregates as long as the condition (1.b) is satisfied. Clearly, this construction is not unique. One can take advantage of such a flexibility in order to obtain smaller aggregates (in terms of number of states). Even if the obtained SOG would have more aggregates in this case, it would consume less time and memory. This definition generalises the one given in [1], while the construction algorithm given in [5] is an example of implementation where the obtained graph is deterministic.

Notice that, once the SOG is built, the set of states of each aggregate has not to still be stored in memory any more. The unique useful information is the observed behavior annoting each node. A SOG is said to be deadlock free if none of its nodes admits the empty set as a member of its observed behavior.

The following proposition establishes that checking deadlock freeness of a SOG is equivalent to check deadlock freeness on the associated process (represented by its LTS)

Proposition 1: Let $\mathcal{W}$ be a business process, let $\mathcal{T} = \langle \Gamma, Act = Obs \cup UnObs, \rightarrow, I, F \rangle$ be the labeled transition system of $\mathcal{W}$ and let $\mathcal{G}$ be a SOG of $\mathcal{T}$. Then, $\mathcal{W}$ is deadlock free if and only if $\mathcal{G}$ is deadlock free.



(a) A SOG of trip reservation

(b) A SOG of customer

Figure 2. Two SOGs of the running example models

Figure 2 shows the two SOGs associated with the WF-nets of Figure 1. Figure 2(a) illustrates the SOG of the reservation trip model while Figure 2(b) shows the SOG of the customer model. We note that the two SOGs are deadlock-free: None of the aggregates of each SOG contains a deadlock state. We recall that the reachabilty graphs of the trip reservation and the customer models contain 13 nodes and 36 edges, and 4 nodes and 4 edges, respectively. It is clear, through this example, that bigger is the number of observed actions, smaller is the size of the obtained SOG. Especially, when all the actions of the service are observed, the SOG is isomorphic to the reachabilty graph.

IV. Composition of SOGs

In this section, we tackle the main idea of this paper: Given two (ore more) business processes (each ignoring internal details about the other), how to check that their composition is deadlock free? We showed in the previous section that the SOG can represent a good abstraction of business processes on which the deadlock freeness property can be checked. Here, we prove that they can also be used in a compositional way: The composition of two SOGs can be useful to check the correctness of the composition of the underlying processes.

We propose to build the synchronized product of two (or more) SOGs so that the obtained graph remains a SOG. Now, the difficulty is to compute the observed behavior of the synchronized product SOG. In fact, the states abstracted by an aggregate are hidden (actually, they do not exist in memory any more, once the SOG is built) and directly computing their observed behavior is not possible. Thus, given two aggregates a_1 and a_2 belonging to two different SOGs, we propose to deduce the observed behavior of the product aggregate, $a = a_1 \times a_2$, from those of a_1 and a_2.

Definition 5 (aggregate product):
Let $\mathcal{T}_i = \langle \Gamma_i, Obs_i \cup UnObs_i, \rightarrow_i, I_i, F_i \rangle, i = 1,2$ be two LTSs. Let $a_i = \langle S_i, \lambda_i \rangle$ be two aggregates of two associated SOGs. The *product aggregate* $a = \langle S, \lambda \rangle$, denoted by $= a_1 \times a_2$, is defined by:

- $a.S = a_1.S \times a_2.S$;
- $a.\lambda = \{(x \cap y) \cup (x \cap (Obs_1 \setminus Obs_2)) \cup (y \cap (Obs_2 \setminus Obs_1)) \mid x \in a_1.\lambda,\ y \in a_2.\lambda\}$.

Notice that the *term* action is supposed to be shared by both LTSs. Intuitively, an observed action is possible from a state $s = (s_1, s_2)$ in $a = a_1 \times a_2$ if it is observed in $\mathcal{T}_1$ and $\mathcal{T}_2$ and possible from both states s_1 and s_2, or it is observed only in $\mathcal{T}_1$ (resp. $\mathcal{T}_2$) and possible from s_1 (resp. s_2).

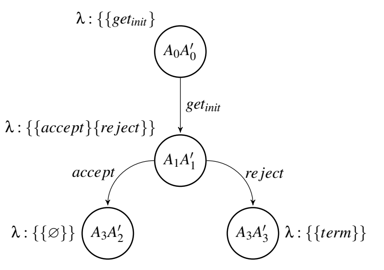
Definition 6 (SOG synchronized product):
Let $\mathcal{T}_i = \langle \Gamma_i, Obs_i, \rightarrow_i, I_i, F_i \rangle, i = 1,2$ be two SOGs. The *synchronized product* of $\mathcal{T}_1$ and $\mathcal{T}_2$, denoted by $\mathcal{T}_1 \times \mathcal{T}_2$ is the SOG $\langle \Gamma, Obs, \rightarrow, I, F \rangle$ where:

1) $\Gamma = \Gamma_1 \times \Gamma_2$;
2) $Obs = Obs_1 \cup Obs_2$;
3) $\rightarrow$ is the transition relation, defined by:
$\forall (a_1, a_2) \in \Gamma' : (a_1, a_2) \xrightarrow{o} (a'_1, a'_2) \Leftrightarrow$
$$\begin{cases} a_1 \xrightarrow{o}_1 a'_1 \wedge a_2 \xrightarrow{o}_2 a'_2 & \text{if } o \in Obs_1 \cap Obs_2 \\ a_1 \xrightarrow{o}_1 a'_1 \wedge a_2 = a'_2 & \text{if } o \in Obs_1 \setminus Obs_2 \\ a_1 = a'_1 \wedge a_2 \xrightarrow{o}_2 a'_2 & \text{if } o \in Obs_2 \setminus Obs_1 \end{cases}$$
4) $I = I_1 \times I_2$;
5) $F = F_1 \times F_2$.

Note: The set of aggregates Γ' is reduced to the states that are reachable from the initial aggregate.

The following proposition establishes that the syncronized product of two SOGs is a SOG. This result combined with Proposition 1 allows to reduce the deadlock freeness verification of an IEBP to the verification of the synchronized product of the SOGs derived from its components.

Proposition 2: Let $\mathcal{W}_i$, for $i \in \{1,2\}$, be two business processes, whose IEBP is $\mathcal{W}$, and let $\mathcal{T}_i$ be their corresponding LTSs. Let $\mathcal{G}_i$ be a SOG associated with T_i with respect to the set of observed actions Obs_i, and let $\mathcal{G}$ be the synchronized product of $\mathcal{G}_i$. Then $\mathcal{G}$ is a SOG of the $\mathcal{W}$'s LTS with respect to $Obs_1 \cup Obs_2$.


$\lambda : \{\{get_{init}\}$
$A_0A'_0$
get_{init}
$\lambda : \{\{accept\}\{reject\}\}$
$A_1A'_1$
$accept$
$reject$
$\lambda : \{\{\varnothing\}\}$
$A_3A'_2$
$A_3A'_3$
$\lambda : \{\{term\}\}$

(a) the SOG synchronized product

Figure 3. the SOG synchronized product

Figure 3 illustrates the SOG obtained by synchronizing the SOGs of Figure 2. We note that it contains a deadlock aggregate $A_3A'_2$ although A_3 and A'_2 are deadlock-free. In fact, $\{term\} \cap \{schedule_{trip}\} = \emptyset$.

V. Implementation and experimental result

A. Implementation

The construction of the original version of SOG has been already implemented in [5] and a model checker of Linear Temporal Logic formulae based on SOGs was proposed in [6]. In this work, we adapted the existing tool to the context of composition of business processes. Thus, the main task was to adapt the construction of the SOG so that the observed behavior [2] is computed for each new aggregate. A direct implementation of the observed behavior of a given aggregate (following Definition 3) implies to consider each state belonging to the aggregate separately. This would considerably decrease the efficiency of the approach. In fact, each aggregate is encoded with a BDD and all the operations manipulating the aggregates should be based on set operations. Therefore, we have implemented an algorithm (see Algorithm 1) for the computation of the observed behavior that is exclusively based on set operations applied to the states of a given aggregate.

The input of Algorithm 1 are an aggregate A, the set of observed transitions Obs, the set of unobserved transitions $UnObs$ and the final set of states F. It computes the observed behavior associated with the aggregate A (i.e., $A.\lambda$).

We use a map (called R) whose elements are couples of sets of events and sets of states (line 1). Each element (O,S) satisfies the following: each state of S enables each transition of O. This map is progressively updated so that, at the end of the algorithm, the set of its keys (the first element of

Algorithm 1 Computing the Observed Behavior

```
Require: $Aggregate A, Obs, UnObs, Set of states F$
Ensure: $A.\lambda$
 1: $Map < Set of events, Set of states > R$
 2: if $F \cap A.S \neq \emptyset$ then
 3:     $insert\ (\{term\}, Pred(F, A.S, UnObs))\ in R$
 4: end if
 5: for $o \in Obs$ do
 6:     if $Enable(A.S, o) \neq \emptyset$ then
 7:         $insert\ (\{o\}, Enable(A.S, o))\ in\ R$
 8:     end if
 9: end for
10: for $(O,S) \in R$ do
11:     for $(O',S') \in R$ do
12:         if $S = S'$ then
13:             $(O,S) \leftarrow (O \cup O', S)$
14:             $remove\ (O',S')\ from\ R$
15:         end if
16:     end for
17: end for
18: $\lambda \leftarrow Set\ of\ keys\ of\ R$
19: $Set of states E \leftarrow \emptyset$
20: for $t \in (Obs \cup UnObs)$ do
21:     $E \leftarrow E \cup Enable(S,t)$
22: end for
23: if $E \neq S$ then
24:     $\lambda \leftarrow \lambda \cup \{\emptyset\}$
25: else
26:     if $(PreIm^*(Enable(A.S, Obs) \cup (F \cap A.S), UnObs) \neq$
        $A.S)$ then
27:         $\lambda \leftarrow \lambda \cup \{\emptyset\}$
28:     end if
29: end if
30: return $\lambda$
```

the couples) form the observed behavior of the aggregagte A (line 18). The first step of the algorithm (lines $2-4$) consists in: (1) checking whether a final state belongs to $A.S$, (2) if it is the case creating a new couple $(\{term\}, S)$ where S is the set of the immediate predecessors of the final states present in A. The latter task is performed by using the $PreIm()$ function. The second step of the algorithm (lines $5-9$) allows to fill the map R with couples of the form $(\{o\}, S)$ where o is an observed action and S the subset of states of A enabeling o. Once the map R is filled, it is analysed in the third part of the algorithm (lines $10-17$). The idea is to look between elements of R those having the same enabling sets of states (the second component of each couple). For each pair (O,S) and (O',S) in R the first couple is updated by adding O' to O while the second is removed from the map. Indeed, states in S enable each action in O or in O' and should be associated with the set $O \cup O'$.

The final part of the algorithm (lines $19-29$) is dedicated

Model	Places	Trans	Obs	RG		OG			SOG		
				States	Edges	States	Edges	time(s)	States	Edges	time(s)
C	18	11	4	26	66	12	20	<1	5	4	<1
SC	15	9	4	11	11	9	11	<1	7	7	<1
OS	15	8	8	10	10	12	17	<1	10	10	<1
R	38	33	17	28	33	369	14 e^2	<1	17	17	<1
Ph5	36	16	10	417	10 e^2	14 e^2	34 e^2	16	297	721	8
Ph6	43	19	12	14 e^2	46 e^2	61 e^2	17 e^3	245	991	28 e^2	42
Ph7	50	22	14	52 e^2	19 e^3	26 e^2	88 e^3	42 e^2	33 e^2	11 e^3	162
Ph10	71	31	20	23 e^5	23 e^4	-	-	-	12 e^4	58 e^4	15 e^2
Ph10	71	31	4	23 e^5	23 e^4	-	-	-	21	50	15

Table I
EXPERIMENTAL RESULTS: OG VS. SOG

to the analysis of the deadlock states inside the aggregate A. If a deadlock state is found in $A.S$ then the empty set is added to λ. A terminal state is detected (lines $19-24$) when the set of states enabling some transition (observed or not) is not equal to the whole set $A.S$. In order to detect strong livelocks (terminal cycles), we iterate on the $PreIm()$ function in order to compute all the states in $A.S$ that possibly lead either to a state in $Enable(A.S, Obs)$ (i.e., a state enabling some observed action), or to a final state. If the result is not equal to $A.S$ then there is a terminal cycle in A and the empty set should belong to $A.\lambda$ (line $26-27$).

In addition to the implementation of the observed behaviour algorithm, we integrated new functionnalities to allow the abstraction and the composition of business processes: Given a WF-net description of one or more business processes, it is possible to check the deadlock freeness property on the fly by building the correspondig SOG. The user can choose to stop the construction of the SOG as soon as a deadlock state is reached, or not. In the last case a textual description of the whole SOG is supplied.

B. Experimental results

We used our implementation in order to build the SOG associated with several business processes from different domains. We do not describe these models here because of lack of place but we give their WF-net models' size (in terms of number of places and transitions) as well as the size of their reachability state graphe RG (in terms of number of nodes and arcs) in Table I. These models were also supplied to Wendy ([10]), a tool to analyse interacting open nets. One of the fonctionalities of Wendy is to build the *Operating Guideline* [3], an annotated automata, in order to abstract a model and to check compatibility between two models (i.e., whether two models can collaborate safely). The corresponding results are illustrate in Table I (column OG)

The obtained results show clearly that SOGs-based approach outperforms than the operating guidelines-based one. The SOG is always (at least for the tested examples) smaller than the operating guideline graph and its construction faster. It is interesting to notice that the size of the operating guideline can be greater than the size of the reachability graph. For instance, this is the case of the online shop model (OS). The corresponding SOG is isomorphic to the reachability graph since all the transitions are observed. The SOG-based approach is especially efficient for loosely coupled models (with a few number of observed actions). This can be easily noticed if we look to the two last lines of Table I: at line 8 the 10 philosophers model is obtained by composition of 10 models (each represents one philosopher). Each philosopher contains two observed transitions (those allowing to pick up the forks) and the total number of observed transitions is 20 over 31. In the last line, however, this model is obtained by composing only two models, each representing five philosophers and contains 2 observed transitions (thus 4 observed transitions over 31). In this case the size of the SOG is negligible comparing to the first case. In both cases Wendy is not able to supply the result because of the explosion of the corresponding state space.

VI. RELATED WORK

The importance of dealing with business processes on one hand and business process composition on the other hand is reflected in the literature by several publications. Below, we discuss some related approaches.

The public-to-private approach introduced by W. van der Aalst in [12] consists of three steps. Firstly, the organizations involved agree on a common and sound public workflow, which serves as a contract between these organizations. Secondly, each task of the public workflow is mapped onto one of the domains (*i.e.*, organization). Each domain is responsible for a part of the public workflow, referred to its public part. Thirdly, each domain can now make use of its autonomy to create a private workflow. To satisfy the correctness of the overall inter-organizational workflow, however, each domain may only choose a private workflow which is a subclass of its public part [13]. The public-to-private approach allows to the local processes to be decoupled as much as possible and to have some degree of understanding about the nature of the interaction between the processes of the different business partners. A problem to be encountered by this approach is confidentiality that prevents a complete view of local workflow. Indeed, to check the deadlock property, one needs the model of the global workflow. This model however is often not available for inter-organizational workflows since organizations are not willing to disclose their workflows (e.g., for privacy reasons). Therefore, our technique that abstracts local workflows using SOGs is well suited to verify properties and preserve organization privacy.

In [7], a formal model for services called service automota is defined by P. Massuthe and K. Schmidt. Based on this representation, the authors combine all the well-interactions between a service and its determinitic partners on an annotated automaton called Operating Guideline. This automaton characterizes all services wich interact properly with the corresponding service. This approach of abstraction was extended to composition of web services by P. Massuthe and K. Wolf in [8], allowing publishers on the web to maintain privacy of the services and to present only the essential behavior information for matching. Authors give a matching algorithm that can be applied between an operating guideline and a web service model and check whether the matching is possible or not.

Another approach for workflow matchmaking was proposed by A. Martens in [9]. It assumes that two workflows match if they are equivalent. To reach this end, the author introduces the notion of communication graph *c-graph* and usability graph (*u-graph*). If the *u-graph* of a workflow is isomorphic to the *c-graph* of another workflow, then the two workflows are considered equivalent.

In conclusion, to the best of our knowledge, none of the existing approaches combine symbolic (using BDDs) abstraction and modular verification to check the correctness of inter-organizational processes. They always deal with an explicite representation of the system's behavior, which accentuate the state space explosion problem.

VII. Conclusion

The main emphasis of this paper is on composition of business processes using symbolic observation graphs: How can we compose extended SOGs such that the resulting SOG is still small but represents the behavior of the IEBP in an appropriate way? We addressed the problem of checking correctness of IEBPs compositionally. We established that and how symbolic observation graphs can be extended and efficiently used for that purpose. We implemented the presented approach and compared the obtained results against the operating guidelines approach. The obtained results confirms our belief that the SOG is a suitable abstraction of business processes that offers, in addition, interesting analysis capabilities.

Our future work will be on studying other correction criteria (e.g., soundness) by depicting the necessar local information (like we did wit the observed behavior) to be stored (within each aggregate) so that the desired property can be checked on the composition of the obtained SOGs. We also plan to extend our approach to deal with resources.

References

[1] K. Klai, S. Tata and J. Desel *Symbolic Abstraction and Deadlock Freeness Verification of Inter-Enterprise Processes*, In Proceedings of the 29th International Conference On Business Process Management, Ulm, Germany, September 2009, 294–309, Springer-Verlag.

[2] K. Klai, S. Tata and J. Desel *Symbolic Abstraction and Deadlock-Freeness Verification of Inter-Enterprise Processes*, In Journal of Data & Knowledge Engeneering (DKE), 2011.

[3] N. Lohmann, P. Massuthe and K. Wolf *Operating Guidelines for Finite-State Services*, In Proceedings of Petri nets'07, Siedlce, Poland, June 25-29, 2007, 321-341, Springer-Verlag, Berlin, Heidelberg

[4] J. Koehlerand B. Srivastava *Web Service Composition: Current Solutions and Open Problems*, ICAPS 2003 Workshop on Planning for Web Services, pages 28 - 35.

[5] S. Haddad, JM. Ilié and K. Klai *Design and Evaluation of a Symbolic and Abstraction-based Model Checker*, In Proceedings of Automated Technology for Verification and Analysis: Second International Conference, ATVA 2004, Taipei, Taiwan, October 31-November 3, 2004,198–210, Springer LNCS 3299

[6] K. Klai and D. Poitrenaud *MC-SOG: An LTL Model Checker Based on Symbolic Observation Graphs*, In Proceedings of Petri Nets'08, Xian, China 2008, 288–306, Springer LNCS, 5062

[7] P. Massuthe and K. Schmidt *Operating Guidelines for Services*, In Proceedings of 12. Workshop "Algorithmen und Werkzeuge fr Petrinetze" September, 2005, 78–83

[8] P. Massuthe and K. Wolf *An Algorithm for Matching Nondeterministic Services with Operating Guidelines*, International Journal of Business Process Integration and Management 2007, Vol. 2, 81 - 90

[9] A. Martens *Usability of web services*, In Proceedings of the Fourth international conference on Web information systems engineering workshops, 2003, Roma, Italy, 182–190, IEEE Computer Society

[10] N. Lohmann and D. Weinberg *Wendy: A Tool to Synthesize Partners for Services*, In Proceedings of Petri Nets' 10, Braga, Portugal, June, 2010, 297-307

[11] W. Van der Aalst *The Application of Petri Nets to Workflow Management*, In Journal of Circuits, Systems, and Computers 1998, 21-66

[12] W. van der Aalst *Loosely Coupled Interorganizational Workflows: Modeling and Analyzing Workflows Crossing Organizational Boundaries*, In Journal of Information and Management, 2000, 67-75

[13] W. van der Aalst and M. Weske,*The P2P Approach to Interorganizational Workflows*, In Proceedings of the 13th International Conference on Advanced Information Systems Engineering, 2001, 140-156