Symbolic Abstraction and Deadlock-Freeness Verification of Inter-Enterprise Processes

Kais Klai[1], Samir Tata[2], and Jörg Desel[3]

[1] LIPN, CNRS UMR 7030, Université Paris 13
99 avenue Jean-Baptiste Clément, F-93430 Villetaneuse, France
kais.klai@lipn.univ-paris13.fr
[2] Institut TELECOM, CNRS UMR Samovar
9 rue Charles Fourier 91011 Evry, France
Samir.Tata@int-edu.eu
[3] Department of Applied Computer Science
Catholic University of Eichstätt-Ingolstadt, 85071 Eichstätt, Germany
joerg.desel@ku-eichstaett.de

Abstract. The design of complex inter-enterprise business processes (IEBP) is generally performed in a modular way. Each process is designed separately from the others and then the whole IEBP is obtained by composition. Even if such a modular approach is intuitive and facilitates the design problem, it poses the problem that 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). 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* 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: How can the symbolic observation graph technique be adapted and employed for process composition? Answering this question is the aim of this paper.

1 Introduction

Business process composition and cooperation are two important research fields in the business process domain. The questions, what properties of a process has to be public so that potential partners can collaborate with the process without risking to have an ill-designed composed process, and what is the minimum necessary to be published, is a hot topic in the literature since many years (e.g. [2,14,13,9]). Also, one has to make sure that the composition of the processes has the desired behaviour. The importance of dealing with such inter-enterprise business processes (IEBP for short) on one hand and business process composition on the other hand is reflected in the literature by numerous publications [18,4,19,15].

In general, an IEBP can be considered as the cooperation of several local processes designed separately. The activities of each process are formally of two kinds: internal activities and cooperative activities (interface activities). IEBP are often too large for formal analysis, and the details of the components are hidden to the public so that no party knows the entire process definition. Therefore, we defend the idea that the analysis should be on the local business process and, if necessary, on an abstraction of the composition partner or of the IEBP. In this paper, we propose a two steps abstraction technique: In the first step, an abstraction of each local process is built locally using a new variant of *symbolic observation graphs* (SOG for short) [6]. This abstraction has two advantages: the analysis of the corresponding process can be reduced to the analysis of its abstraction, and such an abstraction hides the internal structure and organization of the process, which is a desired requirement in the IEBP context. In the second step, the abstraction of the IEBP is obtained by composing the local abstractions (SOGs), leading to a global abstraction on which the analysis can be performed efficiently.

One of the most important properties an IEBP should enjoy is deadlock-freeness. In other words, assuming that the components itself are deadlock-free, it is undesirable that these components block each other. Taking the view of a single component, we want to identify situations where the other component is waiting for some message or action from this component while this component is waiting for some message or action from the other component. In such a situation, the other component does not do any visible action. However, since only the interface behaviour is visible, it is possible that internal actions of the other component do occur. So this behaviour, usually known as livelock, is as bad as deadlock behaviour. Hence, in this paper, we extend the deadlock notion by considering a deadlock state every state from which no cooperative action is possible in the future. One can check the deadlock-freeness on each SOG using efficient symbolic algorithms [6] (i.e. algorithms based on set operations). Since the deadlock freeness property is not preserved by composition, we supply a new algorithm for checking deadlock freeness of the synchronized product of the local SOGs. This algorithm is based on local information that can be made available once before the composition process. The deadlock freeness of the product guarantees correct cooperation between the underlying processes (i.e. a deadlock-free cooperation).

The composition of SOGs is immediately suitable for synchronous interorganizational processes. It can moreover be used for checking whether a cooperation between two processes, that communicate asynchronously, is deadlock-free. To this end, one can define an additional component that represents the asynchronous channel and has two observed actions: *receive* and *send*. Now the *send* action of the first component synchronizes with the *receive* action of the channel whereas the *receive* action of the second component synchronizes with the *send* action of the channel.

This paper is organized as follows. Section 2 adapts the structure of the *symbolic observation graph* in order to abstract business processes. Section 3

constitutes the core of the paper and shows how to build the symbolic observation graph of an IEBP and how to establish whether processes can be composed (or can collaborate) safely by checking the deadlock-freeness of the obtained composition of SOGs. A case study is used throughout these sections in order to illustrate our approach. Section 4 relates our work to other approaches. Finally, Section 5 summarizes the results and mentions some aspects of future work.

2 Process Abstraction

In this section, we show how the structure of the *symbolic observation graph* [6] (*SOG*) is used to abstract business processes. In [6], the authors have introduced the *SOG* as an abstraction of the *reachability graph* of concurrent systems and showed that the verification of an event-based formula of $LTL \setminus X$ (Linear-time Temporal Logic minus the next operator) on the *SOG* is equivalent to the verification on the original 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. The *SOG* is defined as a graph where each node is a set of states linked by unobserved actions and each arc is labeled with 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 [1]). In practice, due to the small number of actions in a typical formula, the SOG has a very moderate size and thus the time complexity of the verification process is negligible w.r.t. the building time of the SOG (see [6,8,7] for experimental results).

We propose to use a SOG to abstract a business process. The collaboration actions are observed while the internal ones are not. We will establish that such an abstraction is especially efficient for loosely coupled IEBPs.

2.1 Notations and preliminary results

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

Definition 1 (Labeled Transition System).
A labeled transition system *(LTS for short) is a* 5*-tuple* $\langle \Gamma, Act, \rightarrow, I, F \rangle$ *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*;*
- $F \subseteq \Gamma$ *is a set of* final states*.*

In this paper, we distinguish LTS observed actions, denoted by a subset Obs, from unobserved actions, denoted by the subset $UnObs$ (with $Obs \cup UnObs = Act$

and $Obs \cap UnObs = \emptyset$). Observed actions can represent cooperative (or interface) actions, while unobserved actions represent internal actions.
The following notations are used in this paper:

- For $s, s' \in \Gamma$ and $a \in Act$, we denote by $s \xrightarrow{a} s'$ that $(s, a, s') \in \rightarrow$.
- $s \xrightarrow{a}$ means that $\exists s' \in \Gamma$ s.t. $s \xrightarrow{a} s'$. 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 its length. Moreover, $s \xrightarrow{\sigma} s'$ denotes that $\exists s_1, s_2, \cdots s_{n-1} \in \Gamma$: $s \xrightarrow{a_1} s_1 \longrightarrow \cdots s_{n-1} \xrightarrow{a_n} s'$. $s \xrightarrow{*} s'$ denotes that s' is reachable from s (i.e. $\exists \sigma \in Act^*$ s.t. $s \xrightarrow{\sigma} s'$) and $s \xrightarrow{*}_T s'$ holds if $\overline{\sigma}$ is included in some subset of actions T.
- The set *Enable(s)* denotes the set of actions a such that $s \xrightarrow{a}$. 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 an LTS. $\pi = s_0 \xrightarrow{a_1} \cdots \xrightarrow{a_n} s_n$ is said to be a *run* if $s_n \in F$ (i.e. s_n is a final state).
- A finite path $C = s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \cdots \xrightarrow{a_{n-1}} s_n$ is said to be a *cycle* if $s_n = s_1$. If $\{a_1, \ldots a_{n-1}\} \subseteq UnObs$ then C is said to be a *livelock*.
- $s \not\rightarrow$, for $s \in (\Gamma \setminus F)$, denotes that s is a dead state i.e. $\nexists a \in Act$: $s \xrightarrow{a}$.
- $s \not\Rightarrow$, for $s \in (\Gamma \setminus F)$, denotes that no observed action can be enabled in the future starting from s, i.e., $\nexists o \in Obs, \tau \in UnObs^*$: $s \xrightarrow{\tau\ o}$.

If $s \not\Rightarrow$, for $s \in (\Gamma \setminus F)$, one can either reach a dead state using unobserved actions only, or a livelock. Such a livelock is said to be a *strong livelock*. In this paper we assume that a strong livelock behaviour is equivalent to a deadlock. In contrast, a cycle with states from which one can execute an observed action (possibly via an unobserved sequence) is said to be a *weak livelock*.

For checking LTL properties, livelock and deadlock behaviours have exactly the same interpretation. However, in the context of inter-organizational processes, we claim that only a strong livelock should be viewed as a deadlock, but not a weak livelock.

The set of states *Dead* contains the states from which no action is enabled or from which no observed action is enabled in the future, i.e., $Dead := \{s \in (\Gamma \setminus F) \mid s \not\Rightarrow\}$ (we distinguish "dead" and "Dead"). The following definition characterizes deadlocks and strong livelocks in an homogenous way. We define a particular mapping applied to states of an LTS called *Observed behaviour*.

Definition 2 (Observed behaviour mapping).
Let $\mathcal{T} = \langle \Gamma, Obs \cup UnObs, \rightarrow, I, F \rangle$ be an LTS. The mapping $\lambda_{\mathcal{T}} : (\Gamma \setminus F) \rightarrow 2^{Obs}$ is defined by: $\lambda_{\mathcal{T}}(s) = \{o \in Obs \mid \exists s' \in \Gamma$ s.t. $s \xrightarrow{*}_{UnObs} s' \wedge s' \xrightarrow{o}\}$. $\mathcal{T}$ is Deadlock free iff $\lambda_{\mathcal{T}}(s) \neq \emptyset$ for each state s in $(\Gamma \setminus F)$

Informally, for each (non final) state s of an LTS $\mathcal{T}$, the observed behaviour 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. This set is empty for a state s if and only if s is a *Dead* state.

The observed behaviour mapping can be extended to sets of states: Given a set of states γ and a set of observed actions ψ, $\lambda(\gamma) = \psi$ iff $\forall s \in \gamma$, $\lambda(s) = \psi$.

The observed behaviour of a given state can be computed by the following two steps. First, compute $Sat(s)$, i.e., all the states reachable from s by executing unobserved actions only. Once such a set is saturated (no new state can be reached), the observed behaviour of s is $Enable(Sat(s)) \cap Obs$. One can improve the computation of $Sat(s)$ by storing the observed behaviours of already computed states.

2.2 Running example

The example used in this paper is an adaptation of the one given in [18] which is inspired by electronic bookstores. In [18], local processes are modeled by workflow nets. Here, we use the "private" workflows of the involved models. Moreover, we modify these models, by removing some internal behaviours, in order to get manageable LTSs. There are four processes, modeling a customer, a bookstore, a publisher and a shipper. $c1$ (resp. $b1$, $p1$, $s1$) is the initial state of the customer's (resp. bookstore's, publisher's, shipper's) LTS.

The customer (Figure 1(a)) behaves as follows: First, he sends an order to a bookstore (c_order). Then the customer may receive a negative answer (c_reject) or be informed that his order is going to be handled (c_accept). After order handling, either the customer receives from a shipper the ordered book (*ship*) and from the bookstore a bill (c_bill), or he receives the bill first and then the book. After receiving the book and the bill the customer makes a payment (c_pay). Finally, the customer returns (c_init) to his initial state to order other books.

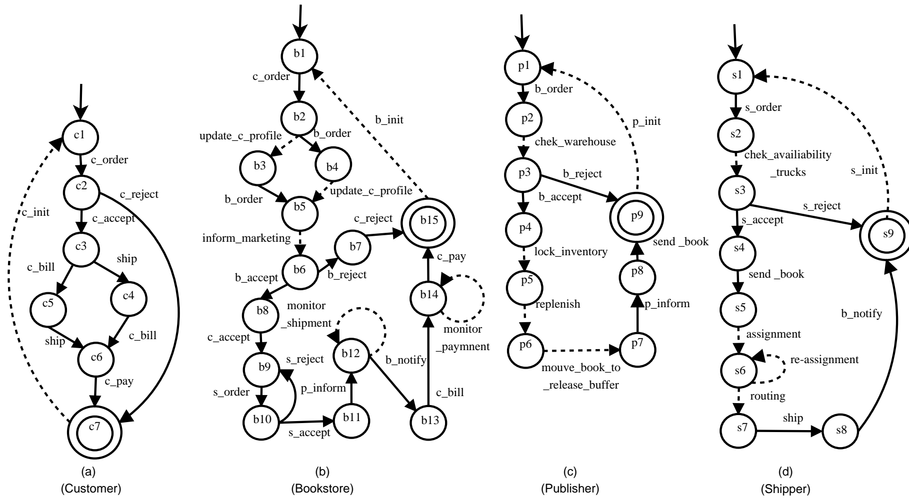

Fig. 1. LTS of a customer, a bookstore, a publisher and a shipper.

Figure 1(b) illustrates the bookstore's LTS that has no books in stock. Therefore, when the bookstore receives an order for a book, it transfers it to a publisher (b_order) and updates the customer profile ($update_c_profile$). Then it informs the marketing department. If the bookstore receives a negative answer (b_reject), i.e. its order was rejected, then it sends a negative response to the customer (c_reject). Otherwise, i.e. the bookstore receives a positive answer (b_accept), the customer is informed (c_accept) and the bookstore sends a request to a shipper (s_order). If the bookstore receives a negative answer (s_reject), it searches another shipper. This process is repeated until a shipper accepts (s_accept). When this happens, the bookstore informs the publisher (p_inform). After that, the bookstore waits for the shipper's notification (b_notify) and sends the bill to the customer (c_bill). Hence, the bookstore processes the payment (c_pay). Finally, after the payment or after an order reject the bookstore returns to its initial state (b_init).

Figure 1(c) presents the publisher's LTS. When receiving an order from a bookstore, the publisher evaluates the order and can either accept it (b_accept) or reject it (b_reject). After that, when the publisher is informed (p_inform) that a shipper was found, he sends the book to the shipper ($send_book$). Finally, after shipment or a request reject, the publisher returns to its initial state (p_init).

Figure 1(d) presents the shipper's LTS. Notice that the original LTS contains 19 nodes and 31 arcs; here we present a reduced version of the graph. When receiving a request from a bookstore (s_order), the shipper evaluates the request and either accepts (s_accept) or rejects (s_reject) the shipping request. In case the shipper receives a book from the publisher ($send_book$), he ships the book to the customer ($ship$) and then notifies the bookstore (b_notify). After shipment or request reject, the shipper returns to its initial state (s_init).

For each LTS of Figure 1, initial states are those having (no source) input arcs while final states are represented with double circles. The observed actions represent, for each component, the collaborative ones and are those labeling dotted arcs. None of these LTSs contains a *Deadlock* state.

2.3 The symbolic observation graph

In this subsection, we first define formally what a *meta-state* is, before providing a formal definition of a SOG associated with an LTS and a set of observed actions. Our definitions are different from those given in [7] because, first, we do not distinguish deadlocks from strong livelocks (we do not pay attention to weak livelocks). Then, we distinguish *final* meta-states from others and, finally, the observed behaviour of the states belonging to a meta-state is stored in this meta-state (as a set of sets of observed actions). Meta-states have associated boolean attributes d and f which indicate whether a meta-state is *Dead* or not and whether it is final or not.

Definition 3 (Meta-state).
Let $\mathcal{T} = \langle \Gamma, Act, \rightarrow, I, F \rangle$ *be a labeled transition system with* $Act = Obs \cup UnObs$. *A* meta-state *is a tuple* $M = \langle S, d, f, \lambda \rangle$ *defined as follows:*

1. *S is a nonempty subset of Γ satisfying:*
   (a) $\forall s \in S\ \exists i \in I, \exists \sigma \in Act^*$ *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. $d \in \{true, false\}$. $d = true$ *iff* $\exists s \in S \setminus F$ *s.t.*$\lambda_{\mathcal{T}}(s) = \emptyset$;
3. $f \in \{true, false\}$. $f = true$ *iff* $S \cap F \neq \emptyset$;
4. $\lambda = \{\psi \subseteq Obs\}$ *s.t.* $\psi \in \lambda$ *iff* $\exists \gamma \subseteq S$ *s.t.* $\lambda(\gamma) = \psi$.

From now on, $M.S$, $M.d$, $M.f$ and $M.\lambda$ denote the corresponding attributes of a given meta-state M. Moreover, we introduce the following set of output states of M: *Out(M)*=$\{s \in M.S \mid \exists o \in Obs\colon s \xrightarrow{o}\}$. Notice that if the set *Out(M)* is empty, then M necessarily contains a *Dead* state.

Definition 4 (Symbolic Observation Graph).
The symbolic observation graph *(SOG($\mathcal{T}$)) associated with an LTS* $\mathcal{T} = \langle \Gamma, Obs \cup UnObs, \rightarrow, I, F \rangle$ *is a 4-tuple* $\langle \Gamma', Act', \rightarrow', I' \rangle$ *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) *For $M, M' \in \Gamma'$ and $a \in Act'$: $M \xrightarrow{a}{}' M'$ if and only if:*
      i. $\forall s \in M.S, s' \in \Gamma\colon s \xrightarrow{a} s' \Rightarrow s' \in M'.S$,
      ii. $\forall o \in Out(M') \exists s \in M.S$, $\exists s' \in M'.S$ *s.t.* $s \xrightarrow{a} s' \wedge s' \xrightarrow{*}_{UnObs} o$,
      iii. $M'.d = true \Rightarrow$
      $(\exists l \in M'.S$ *s.t.* $\lambda_{\mathcal{T}}(l) = \emptyset) \wedge (\exists s \in M.S, \exists s' \in M'.S$ *s.t.* $s \xrightarrow{a} s' \wedge s' \xrightarrow{*}_{UnObs} l)$.
      iv. $M'.f = true \Rightarrow$
      $(\exists f \in M'.S \cap F$ *s.t.* $\forall s \in M.S, \forall s' \in M'.S : s \xrightarrow{a} s' \Rightarrow s' \xrightarrow{*}_{UnObs} f)$.
   (b) $\forall s, s' \in \Gamma\ \forall a \in Obs$
   $(s \xrightarrow{a} s' \Rightarrow \exists M, M' \in \Gamma'\colon s \in M.S, s' \in M'.S \wedge M \xrightarrow{a} M')$,
4. *$I' = \{M_0\}$, where the meta-state M_0 satsifies $I \subseteq M_0.S$.*

Point 3a of the above definition requires explanation. An edge, labeled a, in the SOG is allowed between two meta-states M and M' iff: (3(a)i) each state $s' \in \Gamma$ reachable from some state $s \in M.S$, by action a, belongs to $M'.S$. If $S' = \{s' \in M'.S \mid \exists s \in M.S \wedge s \xrightarrow{a} s'\}$, then (3(a)ii) implies that each output state of M' is reachable from at least one state of S' (using unobserved actions only), while (3(a)iii) implies that when the Deadlock attribute of M' is true then one state l satisfying $\lambda_{\mathcal{T}}(l) = \emptyset$ in $M'.S$ is reachable from at least one state of S' using unobserved actions only. Finally, (3(a)iv) implies that if M' is a final meta-state, then some final *state* $s \in M'.S$ is reachable from each state of S' (defined below).

Figure 2 illustrates the SOGs associated with the LTSs of Figure 1. Final meta-state are represented by dotted circles. The SOG of the *customer* is isomorphic to its corresponding LTS (since all its actions are observed) while the SOG of the *bookstore* contains 12 nodes and 14 arcs (versus 15 nodes and 21 arcs in its corresponding LTS), the SOG of the *publisher* contains 5 nodes and 6 arcs (versus 9 nodes and 10 arcs in its corresponding LTS) and the SOG of the *shipper*

contains 6 nodes and 7 arcs (versus 19 nodes and 31 arcs in its corresponding LTS). All of these SOGs are Deadlock-free. We give below the composition of some meta-states:
- $C1.S = \{c1\}$, $C2.S = \{c2\}$, $C3.S = \{c3\}$, $C4.S = \{c4\}$,
- $B2.S = \{b2, b3\}$, $B3.S = \{b4, b5, b6\}$, $B4.S = \{b8\}$, $B6.S = \{b9\}$,
- $P1.S = \{p1\}$, $P2.S = \{p1, p3\}$, $P3.S = \{p4, p5, p6, p7\}$, $P5.S = \{p9, p1\}$,
- $S1.S = \{s1\}$, $S2.S = \{s2, s3\}$, $S3.S = \{s4\}$, $S4.S = \{s5, s6, s7\}$.

Definition 5 (Deadlock-freeness property of a SOG).
An SOG $\langle \Gamma, Act, \rightarrow, I \rangle$ *is said to be Deadlock-free iff* $\nexists M \in \Gamma$ *s.t.* $M.d = true$.

The following result establishes that the Deadlock-freeness of a SOG is equivalent to the Deadlock-freeness of the corresponding LTS.

Proposition 1. *Let* $\mathcal{T} = \langle \Gamma, Act = Obs \cup UnObs, \rightarrow, I, F \rangle$ *be a labeled transition system and let* $SOG(\mathcal{T})$ *be the corresponding SOG. Then* $\mathcal{T}$ *is Deadlock-free if and only if* $SOG(\mathcal{T})$ *is Deadlock-free.*

Proof. The proof follows from Definition 3 and Definition 4: For each state s of $\mathcal{T}$ there exists a meta-state M of $SOG(\mathcal{T})$ containing s. Conversely, for each meta-state M, all states s in $M.S$ are reachable from some initial state of I in the LTS $\mathcal{T}$.

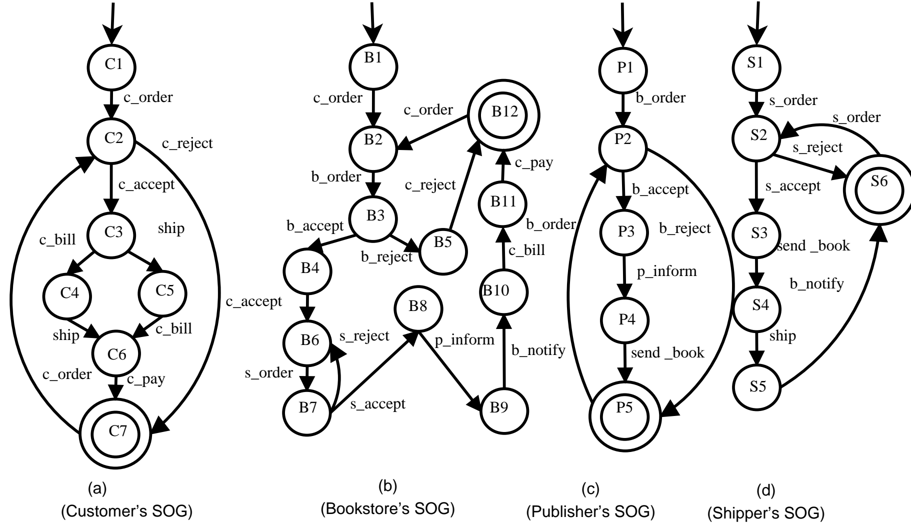

Fig. 2. A Symbolic Observation Graph

We claim that the *SOG* technique is suitable for abstracting processes for several reasons: First, the *SOG* allows to represent the language of the process

projected on the cooperative transitions (i.e. the local behaviors are hidden) in addition to some particular internal behavior which can be relevant for the environment (Deadlock existence). It is a valid abstraction of a given process W because it preserves its privacy while supplying sufficient and necessary information to be known by a potential partner of W. The second reason is that this abstraction is suitable for checking whether two process represented by their *SOGs* can be interconnected (see Section 3). Moreover, given a process, its *SOG* is built once and might be reused as long as local changes do not change its structure. Finally, the reduced size of the *SOG* (in most cases) makes the building and verification of the synchronized product of SOGs much cheaper than the building of the synchronized product of the original LTSs, especially when the involved models are loosely coupled.

3 Composition and Deadlock-Freeness Verification

This section constitutes the core of the paper. Starting from several LTSs which synchronize over a common set of actions, it shows how to synchronize the corresponding SOGs so that the obtained graph is Deadlock-free if and only if the synchronized product of the original LTSs is Deadlock-free.

We start with the standard method for synchronizing two LTSs, namely building their synchronized product. Each state of the resulting transition system is a pair of states, the first component indicating the respective state of the first LTS, the second component indicating the respective state of the second LTS. Each LTS can still do its private activities autonomously, i.e., only one component of the pair representing a state of the composed LTS is changed by such an action. For common activities, however, both components of the state are changed synchronously. Figure 3 shows a simple example of two LTSs (Module A and Module B) and their synchronization $A \times B$.

3.1 Synchronization of LTSs

In the following, we define the synchronized product of two LTSs. The synchronized product of n LTSs (for $n > 2$) can be built by iterative multiplication.

Definition 6 (LTS synchronized product).
Let $\mathcal{T}_i = \langle \Gamma_i, Act_i, \rightarrow_i, I_i, F_i \rangle, i = 1, 2$ *be two LTSs. The* synchronized product *of* $\mathcal{T}_1$ *and* $\mathcal{T}_2$ *is the LTS* $\mathcal{T}_1 \times \mathcal{T}_2 = \langle \Gamma, Act, \rightarrow, I, F \rangle$ *given by:*

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

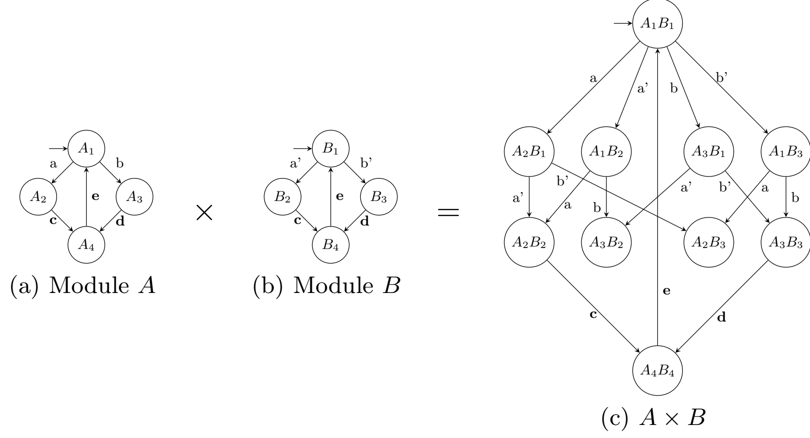

Fig. 3. Synchronized product of two LTSs

5. $F = F_1 \times F_2$.

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: $Act = \{a \in Act_1 \cup Act_2 \mid \exists s, s' \in \Gamma, (s, s') \xrightarrow{a}\}$.

It is well known that the Deadlock-freeness property is not preserved by composition. Given two Deadlock-free LTSs $\mathcal{T}_1$ and $\mathcal{T}_2$, their synchronized product is not guaranteed to be Deadlock-free. Figure 3 illustrates such a situation, where two modules (Figure 3(a) and Figure 3(b)) without Dead states lead, by synchronization over the set of observed actions $\{\mathbf{c},\ \mathbf{d},\ \mathbf{e}\}$, to a synchronized product (Figure 3(c)) containing two Dead states ((A_2, B_3) and (A_3, B_2)).

We characterized Deadlock-freeness of an LTS by considering the *observed behaviour* of its states. In the following proposition, given a synchronized product $\mathcal{T}$ of two LTSs $\mathcal{T}_1$ and $\mathcal{T}_2$, we show how one can deduce the observed behaviour mapping $\lambda_{\mathcal{T}}$ from $\lambda_{\mathcal{T}_1}$ and $\lambda_{\mathcal{T}_2}$. This avoids the analysis of the paths of the synchronized product when such an analysis was already done locally in each involved LTS.

Proposition 2 (Observed behaviour mapping of an LTS synchronized product).
Let $\mathcal{T} = \langle \Gamma, Act = Obs \cup UnObs, \rightarrow, I, F \rangle$ be the synchronized product of two LTSs, $\mathcal{T}_i = \langle \Gamma_i, Act_i = Obs_i \cup UnObs_i, \rightarrow_i, I_i, F_i \rangle, i = 1, 2$. Assume that $Act_1 \cap Act_2 = Obs_1 = Obs_2$. Then the observed behaviour mapping, named $\lambda_{\mathcal{T}}$, satisfies $\forall (s_1, s_2) \in \Gamma$: $\lambda_{\mathcal{T}}(s_1, s_2) = \lambda_{\mathcal{T}_1}(s_1) \cap \lambda_{\mathcal{T}_2}(s_2)$.

Proof. Let (s_1, s_2) be a state of $\mathcal{T}$.

Let us demonstrate that $\forall o \in Obs, o \in \lambda_{\mathcal{T}}(s_1, s_2) \Leftrightarrow o \in \lambda_{\mathcal{T}_1}(s_1) \cap \lambda_{\mathcal{T}_2}(s_2)$. Let $o \in \lambda_{\mathcal{T}}(s_1, s_2)$. Then there exists $\sigma \in UnObs^*$ s.t. $(s_1, s_2) \xrightarrow{\sigma o} (s'_1, s'_2)$. Let σ_1

and σ_2 be the projection of σ on $UnObs_1$ and $UnObs_2$, respectively. Knowing that $UnObs_1 \cap UnObs_2 = \emptyset$, we get that $(s_1, s_2) \xrightarrow{\sigma o} (s'_1, s'_2)$ means that in $\mathcal{T}_1$ and $\mathcal{T}_2$, $s_1 \xrightarrow{\sigma_1 o} s'_1$ and $s_2 \xrightarrow{\sigma_2 o} s'_2$ hold respectively. Thus, $o \in \lambda_{\mathcal{T}_1}(s_1) \cap \lambda_{\mathcal{T}_2}(s_2)$.

The synchronized product of the LTSs of Figure 1 contains 111 nodes and 264 edges and is too big to be presented here. It is Deadlock-free, like the different component LTSs.

3.2 Synchronization of SOGs

The above result allows to define the meta-state product $M = M_1 \times M_2$: a meta-state obtained by synchronizing two meta-states M_1 and M_2. Especially, the corresponding Deadlock attribute, $M.d$, can be computed by using the locally computed observed behaviours. Again, the meta-state product between n ($n > 2$) meta-states can be easily deduced.

Definition 7 (Meta-state 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* $\mathcal{T} = \mathcal{T}_1 \times \mathcal{T}_2 = \langle \Gamma, Obs, \rightarrow, I, F \rangle$. *Let* $M_i = \langle S_i, d_i, f_i, \lambda_i \rangle$ *be a meta-state of* $SOG(\mathcal{T}_i)$. *The* product meta-state $M = \langle S, d, f, \lambda \rangle = M_1 \times M_2$ *is defined by:*

- $S = S_1 \times S_2$,
- $d =$*true iff* $\exists (s_1, s_2) \in \Gamma \colon (\lambda_{\mathcal{T}}(s_1, s_2) = \emptyset)$,
- $f =$*true iff* $f_1 = true$ *and* $f_2 = true$,
- $\lambda = \{\psi \subseteq Obs_i \cup UnObs_i\}$ *s.t.* $\psi \in \lambda$ *iff* $\exists \gamma \subseteq \Gamma \colon \lambda(\gamma) = \psi$.

Apart from dealing with meta-states instead of singular states, the definition of the synchronized product between two SOGs is identical to the synchronized product of two LTS (Definition 6).

Figure 4 shows the synchronized product of the SOGs of Figure 2. It contains 21 nodes and 24 edges (versus 111 nodes and 264 edges in the original synchronized LTS). The obtained SOG is not Deadlock-free (like each independent SOG).

The construction of the symbolic observation graph of a synchronized product of modules consists in first building the SOGs of the individual processes and then synchronizing them. Notice that the construction of the synchronized product of the SOGs aims mainly at establishing whether the underlying processes can collaborate safely (without being in a Deadlock). Checking the Deadlock-freeness of such a synchronized product is reduced to verifying that no (product) meta-state contains a Deadlock ($\forall M \colon M.d = false$) and that there exists a final (product) meta-state ($\exists M \colon M.f = true$).

Algorithm 1 implements the synchronized product of two symbolic observation graphs. This algorithm is very similar to the construction of the synchronized product of LTSs. Function `metastate` $(M_1 \times M_2)$ constructs the meta-state product $(M_1 \times M_2).S$. It assumes that the attributes of M_1 and M_2 are computed locally as well as the observed behaviours of their states. Then, it computes $M.d$ following Definition 7. In Subsection 3.3 an efficient way of computing the deadlock attribute of the product meta-state is discussed.

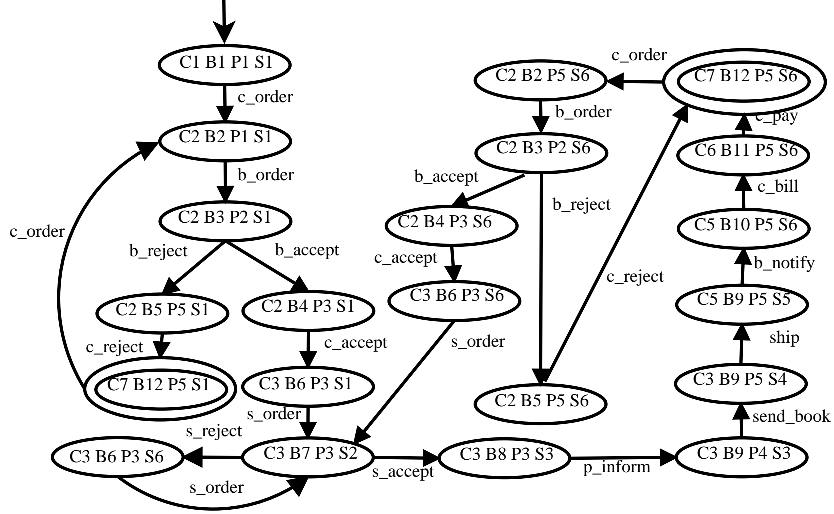

Fig. 4. Synchronized product between SOGs

In the following, we establish the main result of this paper: given two LTSs, checking the Deadlock-freeness property on their synchronized product is equivalent to checking it on the synchronized product of the corresponding SOGs.

Proposition 3. *Let $\mathcal{T}_i = \langle \Gamma_i, Act_i = Obs_i \cup UnObs_i, \rightarrow_i, I_i \rangle$ $(i \in \{1,2\})$ be two LTSs with $Act_1 \cap Act_2 \subseteq (Obs_1 \cap Obs_2)$. Then $SOG(\mathcal{T}_1 \times \mathcal{T}_2)$ and $SOG(\mathcal{T}_1) \times SOG(\mathcal{T}_2)$ are isomorphic.*

Proof. Follows from the construction.

Corollary 1. *Let $\mathcal{T}_i$ $(i \in \{1,2\})$ be two LTSs, let $\mathcal{T}$ be their synchronized product, let $SOG(\mathcal{T}_i)$ be the SOGs of $\mathcal{T}_i$ and let $\mathcal{G}$ be their synchronized product. Then the following property holds: $\mathcal{T}$ is Deadlock-free $\Leftrightarrow$ $\mathcal{G}$ is Deadlock-free.*

Proof. Consequence of Proposition 1 and Proposition 3.

3.3 Checking Deadlock-freeness on a SOGs synchronized product

According to the above results, the verification of Deadlock-freeness in a meta-state product is achieved by using the local observed behaviour mappings (i.e. $\lambda_{\mathcal{T}_1}$ and $\lambda_{\mathcal{T}_2}$). If we assume that Γ_1 and Γ_2 are the sets of states of the original LTSs $\mathcal{T}_1$ and $\mathcal{T}_2$ respectively, then the complexity of the Deadlock-freeness checking is polynomial with respect to the number of states in Γ_1 and Γ_2. However, in terms of efficiency, computing the value of these mappings for each state could reduce drastically the application of the SOG technique. In fact, the efficiency of this technique comes from the fact that it is suitable for symbolic implementation (based on set operations).

Algorithm 1: Synchronized product of 2 SOGs

Require: $SOG(T_1, Obs_1)$ and $SOG(T_2, Obs_2)$
Ensure: $SOG(T_1, Obs_1) \times SOG(T_2, Obs_2)$

```
1:  Waiting ← metastate(I_1 × I_2)
2:  while Waiting ≠ ∅ do
3:    choose M = M_1 × M_2 ∈ Waiting
4:    for all a ∈ Act'_1 ∩ Act'_2 do
5:      if M_1 --a-->_1 M'_1 ∧ M_2 --a-->_2 M'_2 then
6:        metastate(M'_1 × M'_2)
7:        arc(M, a, M'_1 × M'_2)
8:      end if
9:    end for
10:   for all a ∈ Act'_1 \ Act'_2 do
11:     if M_1 --a-->_1 M'_1 then
12:       metastate(M'_1 × M_2)
13:       arc(M, a, M'_1 × M_2)
14:     end if
15:   end for
16:   for all a ∈ Act'_2 \ Act'_1 do
17:     if M_2 --a-->_2 M'_2 then
18:       metastate(M_1 × M'_2)
19:       arc(M, a, M_1 × M'_2)
20:     end if
21:   end for
22:   Waiting ← Waiting \ {M}
23: end while
```

In the following, we propose two sufficient conditions for the existence of a Dead state within a meta-state product. Both conditions can be checked symbolically.

Proposition 4. *Let $SOG(\mathcal{T}_i)$, for $i = 1, 2$, be two SOGs corresponding to $\mathcal{T}_i = \langle \Gamma_i, Obs_i \cup UnObs_i, \rightarrow_i, I_i \rangle$. Let $M_i = \langle S_i, d_i \rangle$ be a meta-state of $SOG(\mathcal{T}_i)$ and let $M = \langle S, d \rangle = M_1 \times M_2$ be the* product meta-state *obtained by synchronizing M_1 and M_2. Then the following properties holds:*

1. $M_1.d = true \vee M_2.d = true \Rightarrow M.d = true$
2. $Out(M_1) = \emptyset \vee Out(M_2) = \emptyset \Rightarrow M.d = true$

Proof. Observe that both conditions imply $\exists (s_1, s_2) \in M_1.S \times M_2.S \colon \lambda_{\mathcal{T}_1}(s_1) \cap \lambda_{\mathcal{T}_2}(s_2) = \emptyset$.

In case the sufficient conditions of Proposition 4 are not satisfied, the following proposition establishes that the Deadlock-freeness of a meta-state product $M = M_1 \times M_2$ can be achieved by considering the projection of the λ mappings on the output states of M_1 and M_2 only instead of all states in $M_1.S$ and $M_2.S$, respectively. The number of output states (states enabling observed actions) is in general reduced with respect to the number of states of the system. This

does not change the worst complexity of the observed behaviour mapping computation. However, in practice, it could significantly reduce the time and space consumption during the computation.

Proposition 5.
Let $SOG(\mathcal{T}_i)$, be two SOGs corresponding to LTSs $= \langle \Gamma_i, Obs_i \cup UnObs_i, \rightarrow_i, I_i \rangle$, for $i = 1, 2$. Let $M_i = \langle S_i, d_i \rangle$ be a meta-state of $SOG(\mathcal{T}_i)$ and let $M = \langle S, d \rangle = M_1 \times M_2$ be the product meta-state *obtained by synchronizing M_1 and M_2. When both conditions 1 and 2 of Proposition 4 are not satisfied, then the following property holds:*
$M.d = true$ iff $\exists (s_1, s_2) \in Out(M_1.S) \times Out(M_2.S)$ s.t. $\lambda_{\mathcal{T}_1}(s_1) \cap \lambda_{\mathcal{T}_2}(s_2) = \emptyset$.

Proof. First, if $M.d = true$ then $\exists (s_1, s_2) \in M_1.S \times M_2.S \colon \lambda_{\mathcal{T}_1}(s_1) \cap \lambda_{\mathcal{T}_2}(s_2) = \emptyset$. Otherwise $M_i.d \neq true$ (for $i = 1, 2$) whence from s_1 (resp. s_2) one can reach an output state s'_1 (resp. s'_2) of M_1 (resp. M_2), with $\lambda_{\mathcal{T}_1}(s'_1) \subseteq \lambda_{\mathcal{T}_1}(s_1)$ (resp. $\lambda_{\mathcal{T}_2}(s'_2) \subseteq \lambda_{\mathcal{T}_2}(s_2)$). Then, $\lambda_{\mathcal{T}_1}(s'_1) \cap \lambda_{\mathcal{T}_2}(s'_2) = \emptyset$, which proves the proposition.

To resume, the Deadlock attribute of a product meta-state can be deduced when one of the involved meta-states contains a Dead state. Otherwise, we only need to consider the observed behaviour of the output states.

4 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.

In [17] the authors present various composition alternatives and their ability to preserve relaxed soundness [3]. The aim of this work was to analyze a list of significant composition techniques in terms of WF-nets and to prove that the composition of relaxed sound models is again relaxed sound. Hence, using these composition techniques does not preserve the deadlock-freeness property. In order to verify this property one has to explore the composed model, even though the component models are deadlock free. The approach we have presented in this paper allows verifying the deadlock-freeness property on the composition of abstract models (SOG).

In [5], the authors propose an approach for services retrieval based on behavioral specification. The idea consists in reducing the problem of service behavioral matching to a graph matching problem and then adapting existing algorithms for this purpose. The complexity of graph matchmaking algorithm used is $O(m^2 * n^2)$ in the best case and $O(m^n * n)$ in the worst case where m is the number of nodes of the request graph and n is the number of nodes of the advertised graph [5]. It is obvious that this approach is not suitable for workflow matching and composition when the number of advertised abstractions increases.

Another approach for workflow matchmaking was proposed in [10][11][12]. 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. If the *u-graph* of a workflow is isomorphic to the *c-graph* of another

workflow, then the two workflows will be considered equivalent. However, the complexity of *c-graph* construction is exponential [10] in terms of the number of nodes. Moreover, it is well known that the subgraph isomorphism detection problem is NP-complete (see for example [16]). It is also obvious that this approach is not suitable for workflow matching when the number of advertised abstractions increases whereas the complexity of our matching algorithm is $O(m*n*l)$ where m and n are the number of meta-states of the corresponding abstractions to be matched and l is the number of the common cooperative transitions.

5 Conclusion

This paper addresses the problem of the abstraction and verification of inter-organizational processes. To preserve privacy of participating processes in an inter-organization process and to enhance verification, we have used the notion of symbolic observation graph to represent process abstractions. We have in addition shown how to build the symbolic observation graph of a composite (or inter-organizational) process and established whether processes can be composed (or can collaborate) safely by checking the deadlock freeness of the obtained symbolic observation graph. Our developed approach can be used for process advertisement, discovery and interconnection.
Several future works are envisaged. The first one would be to implement a tool for the abstraction and the Deadlock-freeness verification of inter-organizational processes. The extension of this work to checking $LTL \setminus X$ properties is direct since, by detecting divergent behaviours (Deadlocks) inside meta-state, the set of maximal paths is preserved. Moreover, we already started working on developping a graph-based registry for abstract process advertisement and discovery. We are going to extend process descriptions by ontology-based semantic descriptions. Our developed algorithms for service matching will be coupled with our presented algorithms to support semantic advertisement and discovery of processes for process composition at design time and for intra-enterprise use and for process cooperation to support inter-organizational processes.

References

1. Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.
2. Tevfik Bultan, Jianwen Su, and Xiang Fu. Analyzing conversations of web services. *IEEE Internet Computing*, 10(1):18–25, 2006.
3. Juliane Dehnert and Peter Rittgen. Relaxed soundness of business processes. In *CAiSE '01: Proceedings of the 13th International Conference on Advanced Information Systems Engineering*, volume 2068 of *Lecture Notes in Computer Science*, pages 157–170, London, UK, 2001. Springer-Verlag.
4. Paul Grefen, Karl Aberer, Yigal Hoffner, and Heiko Ludwig. Crossflow: Cross-organizational workflow management in dynamic virtual enterprises. *International Journal of Computer Systems Science & Engineering*, 15(5):277–290, 2000.

5. Daniela Grigori, Juan Carlos Corrales, and Mokrane Bouzeghoub. Behavioral matchmaking for service retrieval. In *ICWS '06: Proceedings of the IEEE International Conference on Web Services*, pages 145–152, Washington, DC, USA, 2006. IEEE Computer Society.
6. Serge Haddad, Jean-Michel Ilié, and Kais Klai. Design and evaluation of a symbolic and abstraction-based model checker. In Farn Wang, editor, *ATVA*, volume 3299 of *LNCS*, pages 196–210. Springer, 2004.
7. Kais Klai and Laure Petrucci. Modular construction of the symbolic observation graph. In Jonathan Billington, Zhenhua Duan, and Maciej Koutny, editors, *ACSD*, pages 88–97. IEEE, 2008.
8. Kais Klai and Denis Poitrenaud. Mc-sog: An ltl model checker based on symbolic observation graphs. In Kees M. van Hee and Rüdiger Valk, editors, *Petri Nets*, volume 5062 of *Lecture Notes in Computer Science*, pages 288–306. Springer, 2008.
9. Niels Lohmann, Peter Massuthe, Christian Stahl, and Daniela Weinberg. Analyzing interacting ws-bpel processes using flexible model generation. *Data Knowl. Eng.*, 64(1):38–54, 2008.
10. Axel Martens. On Usability of Web Services. In Coral Calero, Oscar Daz, and Mario Piattini, editors, *Proceedings of 1st Web Services Quality Workshop (WQW 2003)*, Rome, Italy, 2003.
11. Axel Martens. Analyzing web service based business processes. In Maura Cerioli, editor, *FASE*, volume 3442 of *Lecture Notes in Computer Science*, pages 19–33. Springer, 2005.
12. Axel Martens. Simulation and Equivalence between BPEL Process Models. In *Proceedings of the Design, Analysis, and Simulation of Distributed Systems Symposium (DASD'05), Part of the 2005 Spring Simulation Multiconference (SpringSim'05)*, San Diego, California, April 2005.
13. Axel Martens, Simon Moser, Achim Gerhardt, and Karoline Funk. Analyzing compatibility of bpel processes. In *AICT-ICIW '06: Proceedings of the Advanced Int'l Conference on Telecommunications and Int'l Conference on Internet and Web Applications and Services*, page 147, Washington, DC, USA, 2006. IEEE Computer Society.
14. Peter Massuthe and Karsten Wolf. An Algorithm for Matching Nondeterministic Services with Operating Guidelines. Informatik-Berichte 202, Humboldt Universitat zu Berlin, 2006.
15. Victor Pankratius and Wolffried Stucky. A formal foundation for workflow composition, workflow view definition, and workflow normalization based on Petri nets. In *APCCM '05: Proceedings of the 2nd Asia-Pacific conference on Conceptual modelling*, pages 79–88, Darlinghurst, Australia, Australia, 2005. Australian Computer Society, Inc.
16. Ronald Read and Derek Corneil . The Graph Isomorphism Disease. *Graph Theory*, 1:339–363, 1977.
17. Juliane Siegeris and Armin Zimmermann. Workflow model compositions preserving relaxed soundness.. In *4th International Conference on Business Process Management*, volume 4102 of *Lecture Notes in Computer Science*, pages 177–192, Vienna, Austria, 2006. Springer-Verlag.
18. W.M.P. van der Aalst and M. Weske. The p2p approach to interorganizational workflows. In *Proceedings of the 13th International Conference on Advanced Information Systems Engineering*, pages 140–156. Springer-Verlag, 2001.
19. Andries van Dijk. Contracting workflows and protocol patterns. In *Business Process Management*, volume 2678 of *Lecture Notes in Computer Science*, pages 152–167. Springer, 2003.