Checking Soundness of Business Processes Compositionally Using Symbolic Observation Graphs

Kais Klai, Jörg Desel

Checking Soundness of Business Processes Compositionally Using Symbolic Observation Graphs

Kais Klai[1] and Jörg Desel[2]

[1] LIPN, CNRS UMR 7030, Université Paris 13, France
[2] FernUniversität in Hagen, 58084 Hagen, Germany

Abstract. The Symbolic Observation Graph (SOG) associated with a labelled transition system and a subset of its labels is an efficient BDD-based abstraction representing the behavior of a system. The goal of this paper is to compose SOGs such that the resulting SOG is still small but represents the behavior of the composed business process in an appropriate way. In particular, we would like to deduce the properties of a composed business process by analysing the composition of the SOGs associated with its components. This question was already answered for the deadlock-freeness property in previous work. In this paper, we extend this result to other generic properties: the so-called soundness properties. These properties guarantee the absence of livelocks, deadlocks and other anomalies that can be formulated without domain knowledge. Thus, we show how the SOG can be adapted and used so that the verification of several variants of the soundness property can be performed modularly.

1 Introduction

Behavioral correctness of a process model can be defined in various ways, depending on the properties considered. For different notions of correctness, and for different process modeling languages, there exist a variety of tools to check correctness. The most important challenge with checking correctness is its inherent high complexity; since correctness refers to the model behavior, each straightforward algorithm requires the construction of a behavioral representation of the model, which is often very large or even infinite. If a business process model is obtained by composition of other models, then concurrency between the respective activities leads to the well-known state explosion problem. Our approach to tackle the state explosion problem is to: (1) provide a behavioral model of a single business process model which is of manageable size but contains sufficient information about the process' behavior such that the relevant properties can be checked using this model, and (2) provide an efficient composition operation on this behavioral model such that analysis of this composed model yields results on the composed business process.

As a behavioral model we use the Symbolic Observation Graph (SOG) [7] which is an efficient BDD-based abstraction of the behavior of a system model.

Formally, a SOG can be viewed as a coarse representation of the state graph of a system model. Algorithmically, this state graph does not have to be constructed explicitly because the SOG can directly, and on the fly, be obtained from the original model. In this paper, the example models will be WF-nets [2]. However, since we do not restrict our work to a particular modeling language, we nevertheless start with state-based representations of process models, namely with Labeled Transition Systems (LTS). Recall that this is done only for presentation purposes and does not mean that an LTS has to be constructed in our approach.

In business process modelling, *soundness* represents a relevant property which is frequently studied. There exist various variants of soundness notions that weaken or strengthen the original definition given in [1]. Roughly speaking, soundness requires that every task of a business process model can actually occur and that it is always possible to reach a legal final state. The notion of *relaxed soundness* is introduced in [5]. This notion allows for potential deadlocks and livelocks, however, each task should occur in at least one proper execution (leading to a final state). In [13] the notion of *weak soundness*, allowing for dead transition, is proposed. Finally, *easy soundness* [17] only requires that the final state is reachable from the initial state. Other variants of soundness addressing problems related to multiple instantiation of the workflow model (e.g., k-soundness and generalized soundness [18]) or focusing on termination conditions (e.g., lazy soundness [15]) are not considered in this paper.

We first translate the definition of these variants of the soundness property, originally defined for Petri nets, to the LTS notation. Then, we show how checking these properties can be done on a SOG instead of the underlying LTS. Finally, we establish that, when the components of a composed business process are proved to be sound, how to check using SOGs whether the composition is sound or not. The last task is performed by considering only the collaboration activities of the model components. In other words, what has been already checked locally is not checked again after composition.

The paper is organized as follows: In Section 2, we give definitions and useful notations. In Section 3, we describe an example of an interorganizational workflow to illustrate the presented concepts and to progressively apply our approach. The Symbolic Observation Graph and the preservation results are presented in Section 4. Composition operators are defined in Section 5 while Section 6 is dedicated to discussing related works and to comparing our approach with existing ones. Section 7 concludes the paper and provides some future perspectives.

2 Preliminaries

The technique presented in this paper applies to different languages for business process modeling that can map to Labeled Transition Systems (one prominent example is the language of WF-nets). For the sake of simplicity and generality, we choose to present it directly for Labeled Transition Systems.

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 nonempty finite set of* states
– *Act is a nonempty finite set of* actions
– $\rightarrow \subseteq \Gamma \times Act \times \Gamma$ *is a* transition relation
– $I \subseteq \Gamma$ *is a nonempty set of* initial states
– $F \subseteq \Gamma$ *is a nonempty set of* final states

In this paper, we restrict the set of states Γ to those that are reachable from an initial state in I. Moreover, we assume that final states are terminal, i.e., no final state has a successor. We distinguish observed actions, denoted by the set Obs, from unobserved actions, denoted by $UnObs$ (with $Obs \cup UnObs = Act$ and $Obs \cap UnObs = \emptyset$). Here, the observed actions are those belonging to the interface (i.e., collaborative actions) while unobserved actions are those performing local activities.

– For $s, s' \in \Gamma$ and $a \in Act$, we denote by $s \xrightarrow{a} s'$ that $(s, a, s') \in \rightarrow$ and by $s \xrightarrow{a}$ that $s \xrightarrow{a} s''$ for some state 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 the length of σ. $s \xrightarrow{\sigma} s'$ denotes that $\exists s_1,\ s_2, \cdots s_{n-1} \in \Gamma \colon s \xrightarrow{a_1} s_1 \xrightarrow{a_2} \cdots s_{n-1} \xrightarrow{a_n} s'$ and is called a path.
– For a state s, the set *Enable(s)* denotes the set of actions a such that $s \xrightarrow{a}$. For a set of states S, *Enable(S)* denotes $\bigcup_{s \in S} Enable(s)$.
– For $s \in (\Gamma \setminus F)$, $s \not\rightarrow$ 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 using unobserved actions only. For $S \subseteq \Gamma$, $Sat(S) = \bigcup_{s \in S} Sat(s)$.
– $s \Rightarrow s'$ means that state s' is reachable from state s (possibly through observed actions).
– For $s \in \Gamma$, $s \not\Rightarrow$ denotes that no state of $Sat(s)$ is final or enables an observed action, i.e., $Sat(s) \cap F = \emptyset \wedge Enable(Sat(s)) \cap Obs = \emptyset$. Conversely, $s \Rightarrow$ means that either a final state or a state enabling an observed action is reachable from s.
– 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 moreover $\overline{\sigma} \subseteq UnObs$ 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 $s \not\Rightarrow$, 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 dead state because these two behaviors are not distinguishable. Both will be called deadlock. In the sequel, the set *Dead(S)*, for a given subset of states S, denotes the set of states $s \in S$ satisfying $s \not\Rightarrow$.

Definition 2. *Let $\mathcal{T} = \langle \Gamma, Act, \rightarrow, I, F \rangle$ be an LTS. Then $\mathcal{T}$ is said to be:*

– *sound iff*
  • $\forall s \in \Gamma\ \exists f \in F \colon s \Rightarrow f$
  • $\forall t \in Act\ \exists s \in \Gamma \colon s \xrightarrow{t}$
– *relaxed sound iff* $\forall t \in Act\ \exists s,\ s' \in \Gamma\ \exists f \in F \colon s \xrightarrow{t} s' \ \wedge\ s' \Rightarrow f$

– *weakly sound iff* $\forall s \in \Gamma\ \exists f \in F\colon s \Rightarrow f$
– *easily sound iff* $\exists i \in I\ \exists f \in F\colon i \Rightarrow f$

The soundness property, originally defined in [1], has two requirements. The first one is that it is always possible to reach a final state. If we assume an appropriate notion of fairness, then this requirement implies that a final state is eventually reached from an initial state. If we require termination without such an assumption, all models allowing loops in their execution sequences would be unsound, which is clearly undesirable. Relaxed soundness [5] allows for potential deadlocks and livelocks. However, each action should occur in at least one "good" execution path. Weak soundness [13] allows for dead transitions as long as a final state is reachable from any state. Finally, easy soundness [17] requires that a final state is reachable from some initial state. It is obvious that soundness implies both relaxed and weak soundness, which are incomparable, and that each other soundness notion implies easy soundness.

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

Definition 3 (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 minimal LTS* $\mathcal{T}_1 \times \mathcal{T}_2 = \langle \Gamma, Act, \rightarrow, I, F \rangle$ *given by:*

1. $\Gamma \subseteq \Gamma_1 \times \Gamma_2$
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. *The set of states* Γ *contains all (and by minimality only) reachable states:*
   $\Gamma = \{(s_1, s_2) \in \Gamma_1 \times \Gamma_2 \mid \exists (i_1, i_2) \in I_1 \times I_2\ \exists \sigma \in Act^* : (i_1, i_2) \xrightarrow{\sigma} (s_1, s_2)\}$
5. $I = I_1 \times I_2$
6. $F = (F_1 \times F_2) \cap \Gamma$

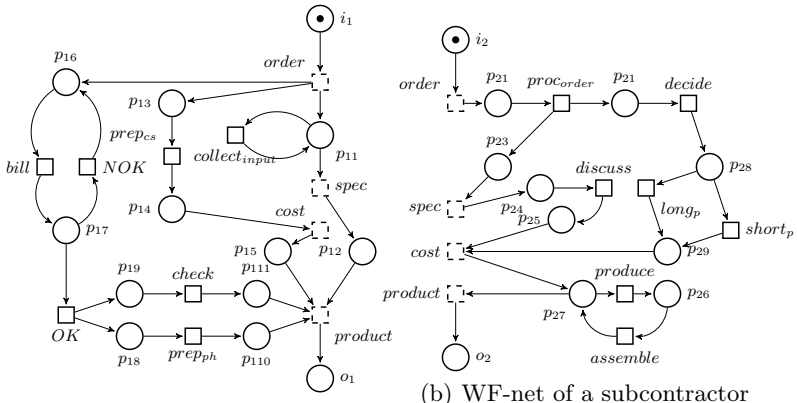
Every state of the synchronized product is a pair of states, the first component indicating the respective state of the first LTS, the second component indicating the respective state of the second LTS. Each LTS can still do its private activities autonomously, i.e., only one component of the pair representing a state of the composed LTS is changed by such an action. For common activities both components of the state are changed synchronously.

3 Running example

To introduce the problems tackled in this paper we use an example, taken from [4], of an interorganizational workflow involving two business partners: a contractor and a subcontractor. Figure 1 illustrates the WF-nets associated with

these business processes. We choose WF-nets to represent these processes instead of the corresponding LTSs because the LTSs are too large (38 states and 104 edges for the contractor's LTS, 14 states and 22 edges for the subcontractor's LTS). The collaborative tasks are represented by dashed transitions and are the only observed actions. The main scenario of the collaboration between these two partners is the following: First, the contractor sends an order to the subcontractor. Then, the contractor sends a detailed specification to the subcontractor and the subcontractor sends a cost statement to the contractor. Based on the specification, the subcontractor manufactures the desired product and sends it to the contractor. Several transitions (tasks) have been added to the original WF-net of the contractor. They are only of local interest, e.g., between the sending of an order and creation of the specification, the task called $collect_{input}$ may be executed multiple times. Internal transitions were also added to the original WF-net of the subcontractor. Both processes are sound (hence relaxed, weakly and easily sound). The same holds for the process model obtained by composing these two WF-nets (obtained by merging the common transitions) and for the corresponding LTSs.

For both models, the initial marking represents the only initial state, and the only final marking is the one with one token in o_1 (o_2 respectively) and no token elsewhere.



(a) WF-net of a contractor

(b) WF-net of a subcontractor

Fig. 1. The WF-nets of a contractor and of a subcontractor

4 Symbolic Observation Graphs

In this section, we show how *Symbolic Observation Graphs* [7] (*SOGs*) can be used to abstract processes while allowing their analysis with respect to the var-

ious soundness notions. The construction of a *SOG* associated with an LTS is guided by a subset of *observed* actions. The *SOG* is defined as a graph where each node is a set of states linked by unobserved actions and each arc is labeled by an observed action. Nodes of the *SOG* are called *aggregates* and may be represented and managed efficiently using decision diagram techniques (e.g., BDDs). In practice, the size of a SOG is proportional to the number of observed actions (see [7,10,9] for experimental results). Thus, by observing only the collaborative actions of a business process, one can hide the internal behavior and hope for a reduced size of the SOG when building and analysing composed business processes, especially when the components are loosely coupled.

Definition 4 (aggregate). *Let $\mathcal{T} = \langle \Gamma, Act, \rightarrow, I, F \rangle$ be a Labeled Transition System with $Act = Obs \cup UnObs$. An* aggregate *is a tuple $a = \langle S, d, f \rangle$ defined as follows:*

1. *S is a non-empty subset of Γ satisfying $Sat(S) = S$*
2. *$d \in \{true, false\}$; $d = true$ iff $Dead(S) \neq \emptyset$*
3. *$f \in \{true, false\}$; $f = true$ iff $S \cap F \neq \emptyset$*

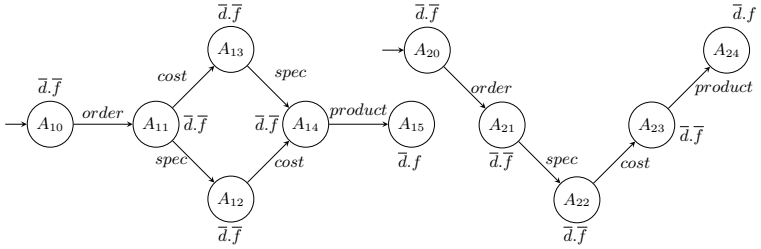
From now on, $a.S$, $a.d$ and $a.f$ denote the corresponding attributes of an aggregate a.

Definition 5 (Symbolic Observation Graph). *A* symbolic observation graph *associated with an LTS $\mathcal{T} = \langle \Gamma, Obs \cup UnObs, \rightarrow, I, F \rangle$ is a five-tuple $\langle \mathcal{A}, Act', \rightarrow', I', F' \rangle$ where:*

1. *$\mathcal{A}$ is a finite set of aggregates satisfying:*
   - *there is an aggregate $a_0 \in \mathcal{A}$ with $a_0.S = Sat(I)$*
   - *if, for some $a \in \mathcal{A}$ and $o \in Obs$, the set $Ext(a, o) := \{s' \notin a.S \mid \exists s \in a.S, s \xrightarrow{o} s'\}$ is not empty, then it is a pairwise disjoint union of non-empty sets $S_1 \ldots S_k$, and for $i = 1 \ldots k$, there is an aggregate $a_i \in \mathcal{A}$ with $a_i.S = Sat(S_i)$*
2. *$Act' = Obs$*
3. *$\rightarrow' \subseteq \mathcal{A} \times Act' \times \mathcal{A}$ is the transition relation satisfying:*
   - *if $a \neq a'$ then $(a, o, a') \in \rightarrow'$ iff $a'.S = Sat(S')$ for some $S' \subseteq Ext(a, o)$*
   - *$(a, o, a) \in \rightarrow'$ iff $Sat(\{s' \in \Gamma \mid \exists s \in a.S, s \xrightarrow{o} s'\}) = a.S$*
4. *$I' = \{a_0\}$ (where $a_0.S = Sat(I)$)*
5. *$F' = \{a \in \mathcal{A} \mid a.S \cap F \neq \emptyset\}$ (=$\{a \in \mathcal{A} \mid a.f = true\}$)*

Notice that Definition 5 does not guarantee the uniqueness of a SOG for a given LTS. In fact, it supplies a certain flexibility for its implementation. In particular, the SOG can be nondeterministic even if the original LTS is not. Actually, one can take advantage of such nondeterminism to obtain smaller aggregates. Even if the SOG obtained in this way has more aggregates than a deterministic one, its construction might consume less time and memory.

Definition 6. *Let $\mathcal{G} = \langle \mathcal{A}, Act', \rightarrow', I', F' \rangle$ be a SOG over a set of observed actions Obs, corresponding to an LTS $\mathcal{T} = \langle \Gamma, Act, \rightarrow, I, F \rangle$. Let $Live(a)$, for an aggregate a, be the set of non dead-states i.e., $Live(a) := a.S \setminus Dead(a.S)$, and let $L(a) := \{t \in Enable(Live(a)) \mid Succ(Live(a), t) \cap Live(a) \neq \emptyset\}$, where $Succ(S, t) := \{s' \mid \exists s \in S : s \xrightarrow{t} s'\}$. Then $\mathcal{G}$ is said to be:*



(a) SOG of the contractor (b) SOG of the subcontractor

Fig. 2. SOGs of the contractor and of the subcontractor

- *sound iff*
  - $\forall a \in \mathcal{A}\colon a.d = false$ *and* $\exists f \in F'\colon a \Rightarrow f$ *and*
  - $\bigcup_{a \in A} Enable(a.S) = Act$
- *relaxed sound iff* $\forall t \in Act\ \exists a \in \mathcal{A}\ \exists f \in F'\colon t \in L(a) \wedge a \Rightarrow f$
- *weakly sound iff* $\forall a \in \mathcal{A}\colon a.d = false$ *and* $\exists f \in F'\colon a \Rightarrow f$
- *easily sound iff* $\exists f \in F'\colon a_0 \Rightarrow f$

The soundness notions are extended to SOGs in order to ensure an equivalence between the soundness of a SOG and the soundness of the underlying LTS. The translation is immediate for all the variants except relaxed soundness. In order to check if each action (observed or not) belongs to a proper execution sequence, we exclude all the dead states (see Section 5.3 for an efficient computation of $Dead(a.S)$) from each aggregate and check wether the obtained subset allows to reach a final aggregate. The absence of dead actions is checked in a similar way.

Proposition 1. *Let $\mathcal{G}$ be a SOG over an arbitrary set of observed actions Obs corresponding to an LTS $\mathcal{T}$. Then the following holds:*

1. *$\mathcal{T}$ is sound $\Leftrightarrow$ $\mathcal{G}$ is sound*
2. *$\mathcal{T}$ is relaxed sound $\Leftrightarrow$ $\mathcal{G}$ is relaxed sound*
3. *$\mathcal{T}$ is weakly sound $\Leftrightarrow$ $\mathcal{G}$ is weakly sound*
4. *$\mathcal{T}$ is easily sound $\Leftrightarrow$ $\mathcal{G}$ is easily sound*

Figure 2 shows two (deterministic) SOGs associated with the WF-nets of the contractor (Figure 2(a)) and the subcontractor (Figure 2(b)) of Figure 1. Each aggregate a is indexed with its attributes $a.d$ and $a.f$. The symbol d (resp. $\overline{d}$) is used when a contains (resp. does not contain) a dead state and the symbol f (resp. $\overline{f}$) is used when a contains (resp. does not contain) a final state. Notice that states of the corresponding LTS are partitioned into aggregates which is not necessary the case in general (i.e., a single state may belong to two (or more) different aggregates).

Note that the corresponding LTSs contain 38 nodes and 104 edges, and 14 nodes and 22 edges, respectively. None of the aggregates of the contractor's (resp. the subcontractor's) SOG contains a deadlock. Both are sound.

5 Composition of SOGs

It is well known that deadlock-freeness is not preserved by composition.

Figure 3(a) presents a WF-net which is almost the same as the WF-net of Figure 1(a). Only the additional place *cs* has been added between transitions *cost* and *spec* to order the corresponding tasks. An alternative WF-net for the subcontractor is represented in Figure 3(b). This workflow contains three new transitions: *decide*, $proc_1$, and $proc_2$. After an order has been received, a decision is made. Based on this decision, one of two possible procedures is executed. In one procedure (transition $proc_1$), the specification is processed before a cost statement is created. In the other procedure (transition $proc_2$), the cost statement is created before the specification is processed. Although both WF-nets are deadlock-free, the synchronization of these models by merging related transitions is not. In fact, the composed process gets stuck as soon as the contractor sends an order to the subcontractor who decides to process it by procedure $proc_1$.

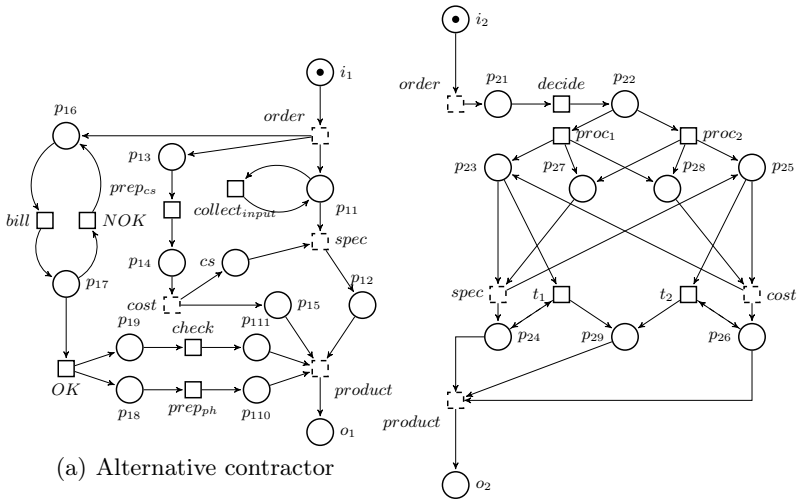
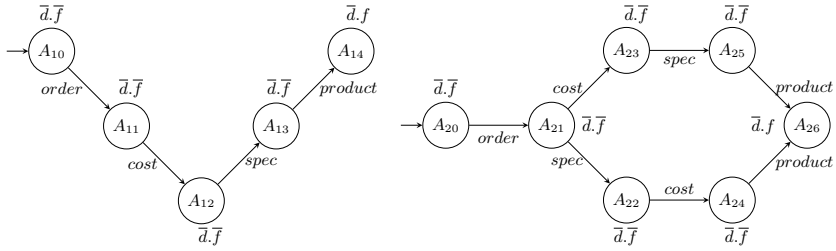


Fig. 3. Alternative WF-nets of a contractor and a subcontractor

Instead of analysing the synchronized product of the underlying LTSs (134 nodes and 480 edges), and detecting such an incorrect behavior, we propose in this section to compose the corresponding SOGs (see Figure 4) in such a way that this behavior is detectable. This approach presents several advantages: First, the verification of the composition takes into account the local verification process. We only focus on the common activities between the processes to be composed. The main task at this stage is to check whether, due to the composition, the



(a) A SOG of the modified contractor(b) A SOG of the modified subcontractor

Fig. 4. Two SOGs of the new contractor and of the new subcontractor

desirable properties have been violated. Second, such an approach allows to reduce the state space explosion due to the composition. Finally, by abstracting a business process with a SOG, we hide the local behavior of the process which might represent internal organisation and private information. This allows to respect the privacy feature of the enterprise and to avoid to expose irrelevant or sensitive information.

5.1 Observed behavior

In the following, we show how, using local information of two aggregates, one can compute the attributes of the aggregate resulting from their synchronisation. Before we define an aggregate a obtained by composition of two aggregates a_1 and a_2, let us define the following particular mapping (called *observed behavior*) applied to states of an LTS $\mathcal{T}$, and extend it progressively to aggregates. It will be established that the *observed behavior* associated with an aggregate is the necessary and sufficient local information to be retained so that soundness properties can be checked on the composition of two process models. For this purpose, and for the remaining part of this paper, we assume the existence of an additional *virtual* observed action *"term"* belonging to Obs.

Definition 7 (Observed behavior mapping).
Let $\mathcal{T} = \langle \Gamma, Obs \cup UnObs, \rightarrow, I, F \rangle$ be an LTS. Let a be an aggregate of a SOG associated with $\mathcal{T}$. The observed behavior is progressively defined by :

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. $\lambda_{\mathcal{T}} : 2^{\Gamma} \rightarrow 2^{Obs}$
   $\lambda_{\mathcal{T}}(S) = \{\lambda_{\mathcal{T}}(s) \mid s \in S\}$
3. $\lambda_a = \{X \in \lambda_{\mathcal{T}}(a.S) \mid \nexists Y \in \lambda_{\mathcal{T}}(a.S) : \ Y \subset (X \setminus \{term\})\}$.

Informally, for each state s of an LTS $\mathcal{T}$, the observed behavior of s, $\lambda_{\mathcal{T}}(s)$, represents 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. The observed behavior $\lambda_{\mathcal{T}}$ associated with a set of states S is a set of sets of observed actions. This set contains the observed behavior of the states of S. Finally, the observed behavior of an aggregate a, namely λ_a, is the minimal set of subsets (w.r.t. the set inclusion relation) of $\lambda_{\mathcal{T}}(a.S)$. The inclusion relation does not concern the *term* action. For instance, if there exist two states $s, s' \in a.S$ such that $\lambda_{\mathcal{T}}(s) = \emptyset$ and $\lambda_{\mathcal{T}}(s') = \{term\}$, then both sets $\emptyset$ and $\{term\}$ will belong to λ_a. This way we distinguish a dead state from a final state reached in $a.S$.

C_1		SC_1		C_2		SC_2	
a	λ_a	a	λ_a	a	λ_a	a	λ_a
A_{10}	$\{\{order\}\}$	A_{20}	$\{\{order\}\}$	A_{10}	$\{\{order\}\}$	A_{20}	$\{\{order\}\}$
A_{11}	$\{\{spec\},\{cost\}\}$	A_{21}	$\{\{spec\}\}$	A_{11}	$\{\{cost\}\}$	A_{21}	$\{\{spec\},\{cost\}\}$
A_{12}	$\{\{spec\}\}$	A_{22}	$\{\{cost\}\}$	A_{12}	$\{\{spec\}\}$	A_{22}	$\{\{cost\}\}$
A_{13}	$\{\{cost\}\}$	A_{23}	$\{\{product\}\}$	A_{13}	$\{\{product\}\}$	A_{23}	$\{\{spec\}\}$
A_{14}	$\{\{product\}\}$	A_{24}	$\{\{term\}\}$	A_{14}	$\{\{term\}\}$	A_{24}	$\{\{product\}\}$
A_{15}	$\{\{term\}\}$	-	-	-	-	A_{25}	$\{\{product\}\}$
-	-	-	-	-	-	A_{26}	$\{\{term\}\}$

Table 1. Illustration of the observed behavior function

Table 5.1 illustrates the observed behavior of each aggregate of the SOGs associated with both versions of our running example. C_1 and SC_1 (resp. C_2 and SC_2) stand for the SOGs associated with the contractor and the subcontractor of Figure 2 (resp. Figure 4) respectively.

The observed behavior associated with an aggregate a allows us to get rid of the attributes $a.d$ and $a.f$ which can be directly deduced from λ_a as follows:

Proposition 2. *Let a be an aggregate of a SOG $\mathcal{G}$ (associated with an LTS $\mathcal{T}$).*

1. *$a.d = true$ if and only if $\emptyset \in \lambda_a$*
2. *$a.f = true$ if and only if $\exists O \in \lambda_a : term \in O$*

From now on, an aggregate a is identified by its observed behavior λ_a. In fact, the set of states $a.S$ of an aggregate a has not to be stored explicitly within an aggregate. Once the SOG is built (and the soundness properties checked), it will not play any further role in the composition process.

When composing several processes, each SOG is computed locally by taking into account the observed behavior of each aggregate. The obtained SOGs are then composed leading to a new SOG. The observed behavior of each aggregate of this SOG is deduced from those of the composed aggregates, as follows:

Definition 8. *For $i = 1, 2$, let $\mathcal{G}_i$ be two SOGs corresponding to $\mathcal{T}_i = \langle \Gamma_i, Obs_i \cup UnObs_i, \rightarrow_i, I_i, F_i \rangle$ and let $a_i = \langle \lambda_{a_i} \rangle$ be an aggregate of $\mathcal{G}_i$. The* product aggregate *$a = \langle \lambda_a \rangle = a_1 \times a_2$ is defined by:*

$$\lambda_a = \{(x \cap y) \cup (x \cap (Obs_1 \setminus Obs_2)) \cup (y \cap (Obs_2 \setminus Obs_1)) \mid x \in \lambda_{a_1},\ y \in \lambda_{a_2}\}$$

Note first that the sets of observed actions Obs_1 and Obs_2 are not necessarily identical (but they share at least the virtual action $term$). When we compose a_1 and a_2, if a_1 can progress in $\mathcal{G}_1$ by using locally observed actions (i.e., actions that are observed in $\mathcal{G}_1$ but not shared by $\mathcal{G}_2$), the product aggregate a should be able to do the same. If this is not the case, then a has to have the same behavior as a_1 and a_2 conjointly. In this way, the observed behavior associated with a product aggregate is helpful to deduce whether the involved set of (pairs of) states contains a deadlock. Moreover, once computed, the observed behavior of $a = a_1 \times a_2$ still respects Proposition 2: The product aggregate contains a deadlock iff the corresponding observed behavior contains the empty set, and it is a final aggregate iff $term$ belongs to one of its observed behavior's elements. Typically, a composed deadlock is a dead state $\langle s_1, s_2 \rangle$ where the shared observed transitions that are enabled in s_1 are all not enabled in s_2 (or viceversa).

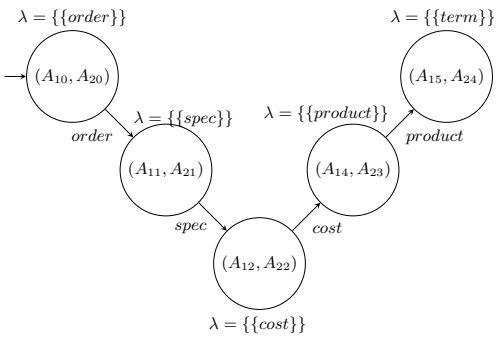
The following definition characterizes the *composed deadlocks*: the deadlocks that are only due to the composition.

Definition 9. *Let $\mathcal{G}$ be the SOG obtained by synchronizing two SOGs $\mathcal{G}_1$ and $\mathcal{G}_2$. $\mathcal{G}$ is said to be containing a composed deadlock iff it contains an aggregate $a = a_1 \times a_2$ such that $\exists (x, y) \in \lambda_{a_1} \times \lambda_{a_2}$ satisfying:*

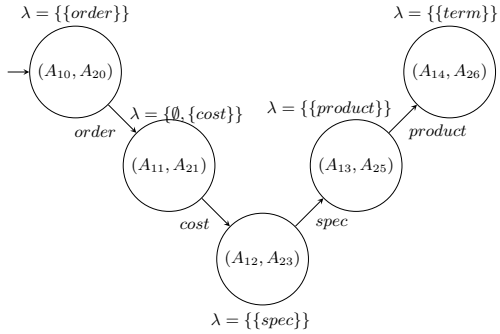
1. *$x \neq \emptyset \wedge y \neq \emptyset$ and $(x \cap y) \cup (x \cap (Obs_1 \setminus Obs_2)) \cup (y \cap (Obs_2 \setminus Obs_1)) = \emptyset$, or*
2. *$x = \emptyset \wedge \emptyset \subset y \subseteq ((Obs_1 \cap Obs_2) \setminus \{term\})$, or*
3. *$\emptyset \subset x \subseteq ((Obs_1 \cap Obs_2) \setminus \{term\}) \wedge y = \emptyset$*

5.2 Synchronous Composition

Given two (or more) LTSs that have been analysed locally and proved to be correct (w.r.t. soundness notions), we would like to reduce the verification of their composition to the verification of the composition of the underlying SOGs. The synchronized product of two SOGs can be defined similarly to the synchronized product of two LTSs (Definition 3). The only difference is that we deal with aggregates (carrying additional information) instead of states. In [11,8] it has been demonstrated that the synchronized product of two SOGs associated with two LTSs is a SOG associated with the synchronized product of these LTSs. Such an approach presents several advantages: First, the verification of the composition takes into account the local verification process. We only focus on the common activities between the processes to be composed. The main task at this stage is to check whether, due to the composition, the desirable properties have been violated. Second, such an approach allows to reduce the state space explosion induced by the concurrency between the activities of the composed components. In fact, these activities are hidden in aggregates of the associated SOGs. Finally, by abstracting a business process with a SOG, we hide the local behavior of



(a) Synchronized SOG's product



(b) Alternative synchronized SOG's product

Fig. 5. The SOG's synchronized products

the process which would represent internal organisation and private information. This allows to respect the privacy feature of the enterprise and to avoid to expose irrelevant or sensitive information.

Figure 5(a) and Figure 5(b) illustrate the SOGs obtained by synchronizing the SOGs of Figure 2 and Figure 4. The left synchronized SOG inherits the same properties of the involved processes. The right synchronized SOG contains a deadlock (aggregate (A_{11}, A_{21})). In fact, $\lambda_{A_{11}} = \{\{p_{cost}\}\}$ and $\lambda_{A_{21}} = \{\{p_{spec}\}, \{p_{cost}\}\}$ which lead to $\lambda_{(A_{11},A_{21})} = \{\emptyset, \{p_{cost}\}\}$, and thus $\{A_{11}, A_{21}\}$ contains a deadlock (a *composed deadlock*).

Proposition 3. *Let $\mathcal{G}_1$ and $\mathcal{G}_2$ be SOGs corresponding to the LTSs $\mathcal{T}_1$ and $\mathcal{T}_2$ with respect to observed actions Obs_1 and Obs_2 respectively.*
Let $\mathcal{G} = \langle \mathcal{A}, Obs_1 \cup Obs_2, \rightarrow, I, F \rangle$ be the synchronized product of $\mathcal{G}_1$ and $\mathcal{G}_2$. Then the following holds:

1. *if $\mathcal{G}_1$ and $\mathcal{G}_2$ are sound then $\mathcal{G}$ is sound iff*
   - $\forall a \in \mathcal{A}\colon \emptyset \notin \lambda_a \wedge \exists f \in F\colon a \Rightarrow f$

– $\forall o \in Obs_1 \cap Obs_2\ \exists a \in \mathcal{A}\colon a \xrightarrow{o}$

2. *if* $\mathcal{G}_1$ *and* $\mathcal{G}_2$ *are relaxed sound then:*
   $\mathcal{G}$ *does not contain a composed deadlock* $\Rightarrow$ $\mathcal{G}$ *is relaxed sound*
3. *if* $\mathcal{G}_1$ *and* $\mathcal{G}_2$ *are weakly sound then* $\mathcal{G}$ *is weakly sound iff*
   $\forall a \in \mathcal{A}\colon \emptyset \notin \lambda_a \wedge \exists f \in F\colon a \Rightarrow f$
4. *if* $\mathcal{G}_1$ *and* $\mathcal{G}_2$ *are easily sound then* $\mathcal{G}$ *is easily sound iff* $\exists f \in F\colon a_0 \Rightarrow f$

The soundness notion of a synchronized product of two SOGs involves only the common observed actions. The enabledness of such actions can be checked after composition as well as the deadlock attribute of the composed aggregate. Local infomation has not to be recalculated because it can not be the reason of soundness violation. Hence, the soundness of the synchronized SOG (except the relaxed variant) can be be decided modularly. Concerning, the relaxed soundness, only the absence of *composed deadlock* in the synchronized SOG implies the satisfaction of this property. In fact, the existence of such a deadlock does not allow to know whether such a property still hold or not for the composition.

Corollary 1. *Let* $\mathcal{T}_1$ *and* $\mathcal{T}_2$ *be two LTSs whose synchronized product is* $\mathcal{T}$*. Let* $\mathcal{G}_1$ *and* $\mathcal{G}_2$ *be SOGs corresponding to* $\mathcal{T}_1$ *and* $\mathcal{T}_2$ *with respect to observed actions* Obs_1 *and* Obs_2 *respectively. Let* $\mathcal{G}$ *be the synchronized product of* $\mathcal{G}_1$ *and* $\mathcal{G}_2$*. Then the following holds:*

1. *If* $\mathcal{T}_1$ *and* $\mathcal{T}_2$ *are sound then* $\mathcal{T}$ *is sound iff* $\mathcal{G}$ *is sound.*
2. *If* $\mathcal{T}_1$ *and* $\mathcal{T}_2$ *are relaxed sound then:*
   $\mathcal{G}$ *does not contain a composed deadlock* $\Rightarrow$ $\mathcal{T}$ *is relaxed sound.*
3. *If* $\mathcal{T}_1$ *and* $\mathcal{T}_2$ *are weakly sound then* $\mathcal{T}$ *is weakly sound iff* $\mathcal{G}$ *is weakly sound.*
4. *If* $\mathcal{T}_1$ *and* $\mathcal{T}_2$ *are easily sound then* $\mathcal{T}$ *is easy sound iff* $\mathcal{G}$ *is easily sound.*

Although in this section we deal with synchronous composition, our technique can also be used for components of a process communicating asynchronously. As long as the whole system is finite, the buffers ensuring the communication between the components together with the associated collaborative actions can be isolated in order to form an intermediate component. The asynchronous composition between two components is thus transformed to a synchronous composition of three components (see. [11,8] for details).

Table 5.2 summarizes the application of our approach to our running examples. We consider the contractor and subcontractor processes of Figure 1 and their alternatives of Figure 3. Both synchronous and asynchronous compositions are considered. For each obtained model we provide the size (the number of nodes in the first column and the number of edges in the second column) of the corresponding reachability graph (R. G.) and of the SOG. Soundness (S), Relaxed soundness (R. S.), weak soundness (W. S.) and easy soundness (E. S.) are also checked.

5.3 The observed behavior computation algorithm

A direct implementation of the observed behavior of a given aggregate (following Definition 7) implies to consider each state belonging to the aggregate separately.

Model	R. G.		SOG		S	R. S.	W. S	E. S.
$Contractor_1$	38	104	6	6	yes	yes	yes	yes
$Contractor_2$	26	66	5	4	yes	yes	yes	yes
$Subontractor_1$	14	22	5	4	yes	yes	yes	yes
$Subontractor_2$	21	22	7	7	yes	yes	yes	yes
$Synchronous_1$	134	480	5	4	yes	yes	yes	yes
$Synchronous_2$	99	320	5	4	no	yes	no	yes
$Asynchronous_1$	248	889	5	4	yes	yes	yes	yes
$Asynchronous_2$	109	373	5	4	no	yes	no	yes

Table 2. Checking Soundness on RG VS SOG

This would considerably decrease the efficiency of the approach. However, since each aggregate is encoded by a BDD, 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 inputs of Algorithm 1 are an aggregate A, a set of observed actions Obs, a set of unobserved actions $UnObs$, and a set of final states F. It computes the observed behavior associated with A (i.e., $A.\lambda$).

We use a map (called R) whose elements are pairs 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 composed of its keys 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 $term$ is the termination observed action, and S is the set of the immediate predecessors of the final states in $A.S$. 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 $A.S$ enabeling o (using function $Enable()$). 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 both actions in O and actions in O' and should be associated with the set $O \cup O'$.

The final part of the algorithm (lines $19-29$) is dedicated to the analysis of the deadlock states inside the aggregate A. Recall that a dead state is either a (non final) terminal state, or a state belonging to a strong livelock (a terminal cycle). If a deadlock state is found in $A.S$ then the empty set is added to λ. A terminal state is found (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

Algorithm 1: Computing the Observed Behavior

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

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 $A.\lambda$.

6 Related Work

The importance of dealing with business processes on the one hand and business process composition on the other hand is reflected in the literature by several publications (e.g., [3,14,12]). To the best of our knowledge, none of the existing approaches combines symbolic abstraction (using BDDs) and modular verification to check the correctness of inter-organisational processes. The originality of our technique is to exploit the efficiency of the SOG's implementation while

respecting the privacy of the enterprise, i.e., without exposing irrelevant or sensitive information. Moreover, the SOGs are computed once and locally for each process which reduces the state explosion problem compared to a non-modular approach. Below we discuss some related approaches.

In [16] the authors present various composition alternatives and their ability to preserve relaxed soundness [5]. 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. Our approach can be applied to any kind of models (not only WF-nets) and allows to check several kinds of soundness including relaxed soundness. In [6], the authors propose an approach for service retrieval based on behavioral specifications. The idea consists of reducing the problem of behavioral matching to a graph matching problem and then adapting existing algorithms for this purpose. The complexity of the 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 [6]. 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 [14]. It assumes that two workflows match if they are equivalent. To reach this end, the author introduces the notions 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. However, the complexity of the *c-graph* construction is exponential in terms of the number of nodes [14].

7 Conclusion

We addressed the problem of checking correctness of inter-organizational business processes compositionally. By correctness we mean soundness with various variants. We established that and how Symbolic Observation Graphs can be extended and efficiently used for that purpose. Moreover, we showed how our approach can be used when the different processes communicate either synchronously or asynchronously.

Our immediate future works follow three directions: First, we are implementing a tool for the abstraction and the verification of inter-organizational business processes. The verification concerns generic properties like deadlock freeness, soundness or specific properties that are expressed by linear-time temporal logics.This helps to check our techniques for concrete applications and makes them available to the scientific community and for practical applications. Second, we plan to extend our approach to deal with resources. Finally, our approach can be used for developing a graph-based registry for abstract process advertisement and discovery.

References

1. Wil M. P. van der Aalst. Verification of workflow nets. In *ICATPN*, LNCS, pages 407–426, London, UK, 1997. Springer-Verlag.
2. Wil M. P. van der Aalst. The application of Petri nets to workflow management. *Journal of Circuits, Systems, and Computers*, 8(1):21–66, 1998.
3. Wil M. P. van der Aalst. Loosely coupled interorganizational workflows: Modeling and analyzing workflows crossing organizational boundaries. *Information and Management*, 37:67–75, 2000.
4. Wil M. P. van der Aalst. Inheritance of interorganizational workflows: How to agree to disagree without loosing control? *Information Technology and Management*, 4:345–389, October 2003.
5. Juliane Dehnert and Peter Rittgen. Relaxed soundness of business processes. In *CAiSE'01*, volume 2068 of *LNCS*, pages 157–170. Springer-Verlag, 2001.
6. 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. IEEE, 2006.
7. Serge Haddad, Jean-Michel Ilié, and Kais Klai. Design and evaluation of a symbolic and abstraction-based model checker. In *ATVA*, LNCS, pages 196–210. Springer-Verlag, 2004.
8. Kais Klai and Hanen Ochi. Modular verification of inter-enterprise business processes. In *eKNOW 2012, The Fourth International Conference on Information, Process, and Knowledge Management*, pages 155–161. IEEE, 2012.
9. Kais Klai and Laure Petrucci. Modular construction of the symbolic observation graph. In *ACSD*, pages 88–97. IEEE, 2008.
10. Kais Klai and Denis Poitrenaud. Mc-sog: An LTL model checker based on symbolic observation graphs. In *ICATPN*, LNCS, pages 288–306. Springer-Verlag, 2008.
11. Kais Klai, Samir Tata, and Jörg Desel. Symbolic abstraction and deadlock-freeness verification of inter-enterprise processes. *Data Knowl. Eng.*, 70(5):467–482, 2011.
12. Niels Lohmann and Karsten Wolf. Petrifying operating guidelines for services. In *ACSD*, pages 80–88. IEEE, 2009.
13. Axel Martens. On compatibility of web services. *Petri Net Newsletter, Special Interest Groups on Petri Nets and Related Systems Models, Gesellschaft fur Informatik e.V.*, 65:12–20, 2003.
14. Axel Martens. On Usability of Web Services. In Coral Calero, Oscar Daz, and Mario Piattini, editors, *Web Services Quality Workshop*, 2003.
15. Frank Puhlmann and Mathias Weske. Interaction soundness for service orchestrations. In *Service-Oriented Computing - ICSOC 2006*, volume 4294 of *LNCS*, pages 302–313. Springer-Verlag, 2006.
16. Juliane Siegeris and Armin Zimmermann. Workflow model compositions preserving relaxed soundness.. In *4th International Conference on Business Process Management*, volume 4102 of *LNCS*, pages 177–192. Springer-Verlag, 2006.
17. Wil M. P. van der Aalst, Kees M. van Hee, and Robert A. van der Toorn. Component-based software architectures: a framework based on inheritance of behavior. *Sci. Comput. Program.*, 42(2-3):129–171, 2002.
18. Kees M. van Hee, Natalia Sidorova, and Marc Voorhoeve. Soundness and separability of workflow nets in the stepwise refinement approach. In *ICATPN*, volume 2679 of *LNCS*, pages 337–356. Springer, 2003.