Compatibility Checking for Asynchronously Communicating Software

Meriem Ouederni, Gwen Salaün, Tevfik Bultan

Compatibility Checking for Asynchronously Communicating Software

Meriem Ouederni[1], Gwen Salaün[2], and Tevfik Bultan[3]

[1] Toulouse INP, IRIT, France
meriem.ouederni@irit.fr
[2] Grenoble INP, Inria, LIG, France
gwen.salaun@inria.fr
[3] UCSB, USA
bultan@cs.ucsb.edu

Abstract. Compatibility is a crucial problem that is encountered while constructing new software by reusing and composing existing components. A set of software components is called compatible if their composition preserves certain properties, such as deadlock freedom. However, checking compatibility for systems communicating asynchronously is an undecidable problem, and asynchronous communication is a common interaction mechanism used in building software systems. A typical approach in analyzing such systems is to bound the state space. In this paper, we take a different approach and do not impose any bounds on the number of participants or the sizes of the message buffers. Instead, we present a sufficient condition for checking compatibility of a set of asynchronously communicating components. Our approach relies on the synchronizability property which identifies systems for which interaction behavior remains the same when asynchronous communication is replaced with synchronous communication. Using the synchronizability property, we can check the compatibility of systems with unbounded message buffers by analyzing only a finite part of their behavior. We have implemented a prototype tool to automate our approach and we have applied it to many examples.

1 Introduction

A widely accepted view in software development is that the software systems should be built by reusing and composing existing pieces of code. Moreover, recent trends in computing technology promote development of software applications that are intrinsically concurrent and distributed. For example, service-oriented computing promotes development of Web-accessible software systems that are composed of distributed services that interact with each other by exchanging messages over the Internet. Cyber-physical systems, on the other hand, involve integration of physical and computational components that interact in a variety of ways to implement a common functionality. Finally, pervasive systems combine large numbers of sensors and computational elements integrated into everyday environment and require their coordination in a dynamic setting. All

these computing paradigms involve concurrent execution of distributed components that are required to interact with each other to achieve a shared goal.

A central problem in composing distributed components is checking their compatibility. Compatibility checking is used to identify if composed components can interoperate without errors. This verification is crucial for ensuring correct execution of a distributed system at runtime. Compatibility errors that are not identified during the design phase can make a distributed system malfunction or deadlock during its execution, which can result in delays, financial loss, and even physical damage in the case of cyber-physical systems.

In this paper, we focus on the compatibility checking problem for closed systems involving composition of distributed components. We call the components that participate in a composed system *peers*. A set of peers is compatible if, when they are composed, they satisfy a certain property. We call such a property a compatibility notion. It is worth observing that the compatibility problem depends on several parameters: the behavioral model used to describe the peers (finite state machines, Petri nets, etc.), the communication model (synchronous *vs.* asynchronous, pairwise *vs.* broadcast/multicast, ordered *vs.* unordered buffers, lossy channels, etc.), and the compatibility notion. In this paper, we use Labeled Transition Systems (LTSs) to describe peer behaviors. We focus on pairwise asynchronous communication model (which corresponds to message-based communication via FIFO buffers). Pairwise communication means that each individual message is exchanged between two peers (no broadcast communication). As for compatibility, there are several compatibility notions existing in the literature. Here, we focus on two widely used notions, namely deadlock-freedom (*DF*) [15] and unspecified receptions (*UR*) [11, 34]. A set of peers is *DF* compatible if their composition does not contain any deadlock, *i.e.*, starting from their initial states peers can either progress by following transitions in their respective LTSs or terminate if they are in final states. A set of peers is *UR* compatible if they do not deadlock and for each message that is sent there is a peer that can receive that message.

Most results in the literature for verifying the compatibility of behavioral models assume two interacting peers and synchronous communication, *e.g.*, [34, 15, 13, 9]. However, asynchronous communication is more suitable than synchronous communication in a distributed setting, since asynchronous communication is non-blocking. In asynchronous communication the sender does not have to wait for the receiver when it needs to emit a message. Analyzing asynchronously communicating systems is more complicated than synchronously communicating systems since it is necessary to represent the contents of the message buffers during analysis of a system that uses asynchronous communication. Moreover, asynchronous communication with unbounded message buffers leads to infinite state spaces. This means that, in general, verification techniques based on explicit state space exploration will not be sound for such systems. Analysis of asynchronously communicating systems has been investigated extensively during the last 30 years, *e.g.*, [11, 24, 26, 14, 31]. A common approach used in analyzing asynchronously communicating systems is to bound the state

space by bounding the number of cycles, peers, or buffers. Bounding buffers to an arbitrary size during its execution is not a satisfactory solution since, if at some point buffers' sizes change (due to changes in memory requirements for example), it is not possible to know how the system would behave compared to its former version and new unexpected errors can show up. This is the case for instance of the simplified news server protocol shown in Figure 1. Transitions are labeled with either emissions (exclamation marks) or receptions (question marks). Initial states are marked with incoming half-arrow and final states have no outgoing transitions. With buffer size 1, the system executes correctly (no deadlock). However, if we increase the buffer size to 2, a deadlock appears when the news server sends message sendnews! followed by stop!. In that situation, the news server is in a final state, but the reader is not able to read the stop message from its buffer and cannot interact properly with the news server.

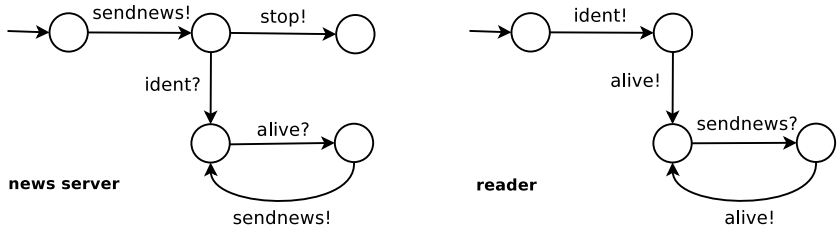

Fig. 1. Motivating Example (1)

Figure 2 shows another simple example involving three peers: a client (cl), a server (sv), and a database (db), which exchange three messages request, result, and log. Peer sv receives a request, sends a result, and loops. Peer cl sends a request, receives a result, sends a log message, and loops. Peer db receives log messages. If we try to generate the LTS corresponding to the composition of these three peers interacting asynchronously through unbounded buffers, this results in an infinite state system. Indeed, the peers sv and cl can loop infinitely, and the peer db can consume from its input buffer whenever it wants, meaning that its buffer can grow arbitrarily large. Analyzing such system is therefore a complicated task (undecidable in general [11]), and to the best of our knowledge, existing approaches cannot analyze compatibility of such systems, because they cannot handle systems that communicate with asynchronous communication via unbounded buffers.

It was recently shown that it is decidable to check certain properties of distributed systems interacting asynchronously through unbounded buffers using the *synchronizability* property [3, 4]. A set of peers is synchronizable if and only if the system generates the same sequences of messages under synchronous and unbounded asynchronous communication (considering only the ordering of the send actions and ignoring the ordering of receive actions). It was shown that synchronizability can be verified by checking the equivalence of synchronous and 1-bounded asynchronous (where buffer sizes are bounded to be 1) versions

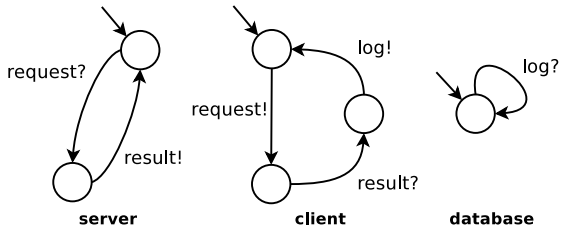

Fig. 2. Motivating Example (2)

of the given system [3, 4]. Hence, synchronizability checking can be achieved using equivalence checking techniques for finite state spaces, although the system consisting of peers interacting asynchronously can result in infinite state spaces. For example, the system described in Figure 2 is synchronizable because the synchronous system consists of sequences of interactions on request, result, and log, and this order is the same in the 1-bounded asynchronous system considering only send actions. Focusing only on send actions and ignoring receive actions makes sense for checking synchronizability because: (i) send actions are the actions that transfer messages to the network and are therefore observable, (ii) receive actions correspond to local consumptions by peers from their buffers and can therefore be considered to be local and private information.

In this paper, we propose a new approach for checking the compatibility of a set of peers interacting asynchronously through unbounded FIFO buffers. Peers are described using LTSs and exhibit their internal behaviors in these models (*e.g.*, replacing conditional constructs with non-deterministic choices of internal actions). Compatibility checking relies on synchronizability, which ensures that the synchronous system behaves like the asynchronous one for any buffer size. Thus, we can check the compatibility on the synchronous version of the system and the results hold for the asynchronous versions. We propose a branching notion of synchronizability to take internal actions present in the peer models into account. We also need to check that the system is *well-formed*, meaning that every message sent to a buffer will be eventually consumed. We show that our approach can be used to check *DF* and *UR* compatibility. Many systems involving loops do respect the synchronizability property. Thus, these systems can be analyzed using the approach proposed in this paper, whereas they could not be analyzed using existing approaches. This is the case for the example given in Figure 2. This set of peers is synchronizable and the synchronous system is deadlock-free for instance. Therefore, we can conclude using our result that the asynchronous version of this system is also deadlock-free compatible even if buffers are unbounded.

Our approach is fully automated through an encoding of the peer model into the process algebra LOTOS [23], one of the input languages of the CADP verification toolbox [19]. By doing so, we can reuse all CADP tools and particularly state space exploration tools for generating synchronous and asynchronous systems, equivalence checking techniques for verifying synchronizability, and model checking techniques for searching deadlocks. We have validated our approach on

many case studies, most of them borrowed from real-world scenarios found in the literature. The evaluation shows that (i) most systems are synchronizable and can be analyzed using our approach, and (ii) this check is achieved in a reasonable time (seconds for examples involving up to ten peers, and minutes for systems up to 18 peers).

Our contributions with respect to earlier results on formal analysis of behavioral models for synchronizability and compatibility checking are the following:

- A general framework for verifying the compatibility of synchronizable systems interacting asynchronously through unbounded buffers;
- A generalization of synchronizability and well-formedness results to branching time equivalences for peer models involving internal actions;
- A fully automated tool support that implements the presented approach for checking asynchronous compatibility.

The organization of the rest of this paper is as follows. Section 2 defines our models for peers and their composition. Section 3 presents a branching notion of synchronizability. In Section 4, we present our solution for checking asynchronous compatibility. Section 5 illustrates our approach on a case study. Section 6 describes our tool support and experiments we carried out to evaluate our approach. Finally, Section 7 reviews related work and Section 8 concludes.

2 Behavioral Models

2.1 Peer Model

We use Labeled Transition Systems (LTSs) for modeling peers. This behavioral model defines the order in which a peer executes the send and receive actions.

Definition 1 (Peer). *A peer is an LTS* $\mathcal{P} = (S, s^0, \Sigma, T)$ *where* S *is a finite set of states,* $s^0 \in S$ *is the initial state,* $\Sigma = \Sigma^! \cup \Sigma^? \cup \{\tau\}$ *is a finite alphabet partitioned into a set of send messages, receive messages, and the internal action, and* $T \subseteq S \times \Sigma \times S$ *is a transition relation.*

We write $m!$ for a send message $m \in \Sigma^!$ and $m?$ for a receive message $m \in \Sigma^?$. We use the symbol τ (`tau` in figures) for representing internal activities. A transition is represented as $(s, l, s') \in T$ where $l \in \Sigma$.

Finally, we assume that peers are deterministic on observable messages meaning that if there are several transitions going out from one peer state, and if all the transition labels are observable, then they are all different from one another. However, nondeterminism can result from internal actions when several transitions (at least two) outgoing from a same state are labeled with τ.

It is crucial to represent internal activities in the peer model using τ actions, particularly when we reason in terms of synchronous communication. These internal actions are used to model internal choices, that is, if/while constructs in programming languages for instance. Figure 3 shows a simple example where we see that two peers `p1` and `p2` are deadlock-free if we do not explicitly show the internal actions. If we consider an abstraction closer to reality by modeling the internal actions, we observe that the peers (`p1'` and `p2`) actually deadlock.

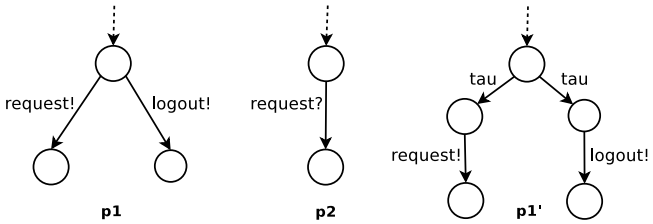

Fig. 3. p1 and p2 are Deadlock-free; p1' and p2 Deadlock

2.2 Synchronous Composition

The synchronous composition of a set of peers corresponds to the system in which the peer LTSs communicate using synchronous communication. In this context, a communication between two peers occurs if both agree on a synchronization label, *i.e.*, if one peer is in a state in which a message can be sent, then the other peer must be in a state in which that message can be received. One peer can evolve independently from the others through an internal action.

Definition 2 (Synchronous Composition). *Given a set of peers* $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ *with* $\mathcal{P}_i = (S_i, s_i^0, \Sigma_i, T_i)$*, the synchronous composition* $(\mathcal{P}_1 \mid \ldots \mid \mathcal{P}_n)$ *is the labeled transition system* $LTS_s = (S_s, s_s^0, \Sigma_s, T_s)$ *where:*

- $S_s = S_1 \times \ldots \times S_n$
- $s_s^0 \in S_s$ *such that* $s_s^0 = (s_1^0, \ldots, s_n^0)$
- $\Sigma_s = \cup_i \Sigma_i$
- $T_s \subseteq S_s \times \Sigma_s \times S_s$*, and for* $s = (s_1, \ldots, s_n) \in S_s$ *and* $s' = (s'_1, \ldots, s'_n) \in S_s$

(interact) $s \xrightarrow{m} s' \in T_s$ *if* $\exists i, j \in \{1, \ldots, n\}$ *where* $i \neq j : m \in \Sigma_i^! \cap \Sigma_j^?$ *where* $\exists$ $s_i \xrightarrow{m!} s'_i \in T_i$*, and* $s_j \xrightarrow{m?} s'_j \in T_j$ *such that* $\forall k \in \{1, \ldots, n\}, k \neq i \wedge k \neq j \Rightarrow s'_k = s_k$

(internal) $s \xrightarrow{\tau} s' \in T_s$ *if* $\exists i \in \{1, \ldots, n\}$, $\exists$ $s_i \xrightarrow{\tau} s'_i \in T_i$ *such that* $\forall k \in \{1, \ldots, n\}, k \neq i \Rightarrow s'_k = s_k$

2.3 Asynchronous Composition

In the asynchronous composition, the peers communicate with each other asynchronously through FIFO buffers. Each peer $\mathcal{P}_i$ is equipped with an unbounded message buffer Q_i. A peer can either send a message $m \in \Sigma^!$ to the tail of the receiver buffer Q_j at any state where this send message is available, read a message $m \in \Sigma^?$ from its buffer Q_i if the message is available at the buffer head, or evolve independently through an internal action. Since reading from the buffer is not considered as an observable action, it is encoded as an internal action in the asynchronous system.

Definition 3 (Asynchronous Composition). *Given a set of peers* $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ *with* $\mathcal{P}_i = (S_i, s_i^0, \Sigma_i, T_i)$*, and* Q_i *being its associated buffer, the asynchronous composition* $((\mathcal{P}_1, Q_1) \mid\mid \ldots \mid\mid (\mathcal{P}_n, Q_n))$ *is the labeled transition system* $LTS_a = (S_a, s_a^0, \Sigma_a, T_a)$ *where:*

- $S_a \subseteq S_1 \times Q_1 \times \ldots \times S_n \times Q_n$ *where* $\forall i \in \{1, \ldots, n\}$, $Q_i \subseteq (\Sigma_i^?)*$
- $s_a^0 \in S_a$ *such that* $s_a^0 = (s_1^0, \epsilon, \ldots, s_n^0, \epsilon)$ *(where* ϵ *denotes an empty buffer)*
- $\Sigma_a = \cup_i \Sigma_i$
- $T_a \subseteq S_a \times \Sigma_a \times S_a$*, and for* $s = (s_1, Q_1, \ldots, s_n, Q_n) \in S_a$ *and* $s' = (s'_1, Q'_1, \ldots s'_n, Q'_n) \in S_a$

(send) $s \xrightarrow{m!} s' \in T_a$ *if* $\exists i, j \in \{1, \ldots, n\}$ *where* $i \neq j : m \in \Sigma_i^! \cap \Sigma_j^?$*, (i)* $s_i \xrightarrow{m!} s'_i \in T_i$*, (ii)* $Q'_j = Q_j m$*, (iii)* $\forall k \in \{1, \ldots, n\} : k \neq j \Rightarrow Q'_k = Q_k$*, and (iv)* $\forall k \in \{1, \ldots, n\} : k \neq i \Rightarrow s'_k = s_k$

(consume) $s \xrightarrow{\tau} s' \in T_a$ *if* $\exists i \in \{1, \ldots, n\} : m \in \Sigma_i^?$*, (i)* $s_i \xrightarrow{m?} s'_i \in T_i$*, (ii)* $mQ'_i = Q_i$*, (iii)* $\forall k \in \{1, \ldots, n\} : k \neq i \Rightarrow Q'_k = Q_k$*, and (iv)* $\forall k \in \{1, \ldots, n\} : k \neq i \Rightarrow s'_k = s_k$

(internal) $s \xrightarrow{\tau} s' \in T_a$ *if* $\exists i \in \{1, \ldots, n\}$*, (i)* $s_i \xrightarrow{\tau} s'_i \in T_i$*, (ii)* $\forall k \in \{1, \ldots, n\} : Q'_k = Q_k$*, and (iii)* $\forall k \in \{1, \ldots, n\} : k \neq i \Rightarrow s'_k = s_k$

We use LTS_a^k to define the *bounded asynchronous composition*, where each message buffer is bounded to size k. The definition of LTS_a^k can be obtained from Def. 3 by allowing send transitions only if the message buffer that the message is being written to has less than k messages in it.

3 Branching Synchronizability and Well-Formedness

Although peers are represented with finite models, their parallel execution could be an infinite state system due to the communication over unbounded buffers. This makes the exhaustive analysis of all executed communication traces impossible and most verification tasks in this setting are undecidable [11]. However, this issue can be avoided for systems that are synchronizable, *i.e.*, if the sequences of send actions generated by the peer composition remains the same under synchronous and asynchronous communication semantics. Thus, the *synchronizability* condition [4] enables us to analyze asynchronous systems, even those generating an infinite state space, using the synchronous version of the given system (which has a finite state space). The results presented below show that synchronizability can be checked by bounding buffers to $k = 1$ and comparing interactions in the synchronous system with the interactions in the asynchronous system.

In this paper, the peer model and corresponding compositions take internal behaviors into account. Therefore, we need to extend synchronizability to branching time semantics [32][4]. This is crucial for considering models closer to

[4] We assume that the reader is familiar with branching time bisimulations, refer to [32] otherwise.

reality (see Fig. 3) and for analyzing the internal structure to detect possible issues at this level. In this paper, we refer to branching equivalence as $\equiv_{br}$.

Definition 4 (Branching Synchronizability). *Given a set of peers $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$, their synchronous composition $LTS_s = (S_s, s_s^0, L_s, T_s)$, and their asynchronous composition $LTS_a = (S_a, s_a^0, L_a, T_a)$, we say that LTS_a is branching synchronizable, $\mathcal{SYNC}_{br}(LTS_a)$, if and only if $LTS_s \equiv_{br} LTS_a$.*

Theorem 1. *A LTS_a defined over a set of peers $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ is branching synchronizable if and only if $LTS_s \equiv_{br} LTS_a^1$. In other words: $LTS_s \equiv_{br} LTS_a^1 \Leftrightarrow LTS_s \equiv_{br} LTS_a$*

Proofs of the theorems from this section are available on the first author Webpage.

Below we define the well-formedness property and present two theorems related to well-formedness.

Definition 5. *An asynchronous system is well-formed if and only if every message that is sent is eventually consumed.*

Given a labeled transition system LTS_a defined over a set of peers $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$, we use $\mathcal{WF}(LTS_a)$ to denote that LTS_a is well-formed.

Theorem 2. *A synchronizable system LTS_a is well-formed if and only if LTS_a^1 is well-formed, i.e., $\mathcal{WF}(LTS_a^1) \Leftrightarrow \mathcal{WF}(LTS_a)$.*

Theorem 3. *Every asynchronous system LTS_a that is branching synchronizable and composed of observationally deterministic peers is always well-formed.*

4 Compatibility

In this section, we present how to check the compatibility of a set of peers communicating asynchronously over unbounded FIFO buffers. This problem is undecidable in the general case [11] since unbounded buffers may lead to infinite state spaces. We present the compatibility checking for synchronous communication, and then show how we extend these results to asynchronous communication. We first focus on *DF* and *UR* compatibility notions. We use *DF* to detect blocking behaviors where system remains infinitely in a pending state with no further execution. We use *UR* to detect cases where some emissions are never received. As a second step, we show how other compatibility notions can also be considered such as bidirectional complementarity and goal oriented compatibility (*BC* and *GOC* for short, respectively). *BC* requires that every emission must be received and every message that is expected to be received must be sent during peer communication. *GOC* describes a temporal logic-based compatibility (expressed in Linear Time Logic for example), that must be respected by the peers. It is worth noting that here we focus on checking properties related to ordering of message exchanges among peers, leaving properties such as state reachability out of the scope of this paper.

4.1 Synchronous Compatibility

Given n communicating peers described using LTSs $(S_i, s_i^0, \Sigma_i, T_i)$, we define a *global state* as a tuple of states $(s_1, \ldots, s_n)$ where s_i is the current state of LTS_i. We refer to a label l as a message in Σ together with its direction ($d \in \{!, ?\}$), *i.e.*, $l = m!|m?$. Two labels $l_1 = m_1 d_1$ and $l_2 = m_2 d_2$ are considered *compatible*, *lab-comp*(l_1, l_2), if and only if $m_1 = m_2$ and $\overline{d_1} = d_2$ where $\overline{!} = ?$ and $\overline{?} = !$.

Compatibility checking requires to verify the interaction at every global state *reachable* during system execution. Reachability returns the set of global states that n interoperating peers can reach from a current global state $(s_1, \ldots, s_n)$ through independent evolutions (internal behaviors) or synchronizations.

The *DF* compatibility is defined as follows. Given a set of peers, we call them *DF* compatible if and only if, starting from their initial global state, they can always evolve until reaching a global state where every peer state has no outgoing transition (correct termination).

The *UR* compatibility is defined as follows. Given a set of peers, we call them *UR* compatible if, when one peer can send a message at a reachable state, there is another peer which must eventually receive that emission, and the system is deadlock-free. A set of peers can be compatible even if one peer is able to receive a message that cannot be sent by any of the other peers, *i.e.*, there might be additional receptions. It is also possible that one peer holds an emission that will not be received by its partners as long as the state from which this emission goes out is unreachable when those peers interact together.

More details about these compatibility notions (*DF* and *UR* but also *BC* and *GOC*) as well as their formal definitions can be found in [17].

4.2 Asynchronous Compatibility

In this section we present sufficient conditions for checking asynchronous compatibility. The behaviors of synchronizable systems remain identical for any buffer size, therefore, we can check compatibility of synchronizable systems using existing techniques for checking synchronous compatibility. A set of communicating peers $\{P_1, \ldots, P_n\}$ is asynchronous compatible if the following conditions hold:

- **Synchronizability.** Peer composition LTSs are branching synchronizable (Theorem 1).
- **Well-formedness.** Every message sent to a buffer is eventually consumed (Theorems 2 and 3).
- **Compatibility.** The set of peers is compatible under synchronous communication semantics (Section 4.1).

In the rest of this section, we define the asynchronous *DF* and *UR* compatibility (DF_a and UR_a for short, resp.) and we finally show how our asynchronous checking can be generalized to check other notions, *e.g.*, BC_a and OGC_a.

Deadlock-Freedom. An asynchronous system LTS_a defined over a set of peers $\{P_1, \ldots, P_n\}$, is DF_a compatible if $\mathcal{SYNC}_{br}(LTS_a)$ and $\mathcal{WF}(LTS_a)$, and the corresponding LTS_s is *DF* (referred to as *DF*(LTS_s)).

Theorem 4. $(\mathcal{SYNC}(LTS_a) \wedge \mathcal{WF}(LTS_a) \wedge DF(LTS_s)) \Rightarrow DF_a(LTS_a)$

Proof. $LTS_s \equiv_{br} LTS_a$ follows from $\mathcal{SYNC}(LTS_a)$ (Theorem 1). Then, we have $DF(LTS_s) \Rightarrow DF_a(LTS_a)$. ■

Unspecified Receptions. Although both *DF* and *UR* compatibility are different under the synchronous communication semantics, in the asynchronous setting, they can be checked similarly. Recall that *UR* compatibility requires us to check that (i) every reachable sent message must be received (*i.e.*, consumed from the buffer where it has been stored), and (ii) the system must be deadlock-free.

Theorem 5. $(\mathcal{SYNC}(LTS_a) \wedge \mathcal{WF}(LTS_a) \wedge DF(LTS_s)) \Rightarrow UR_a(LTS_a)$

Proof. Condition (i) for *UR* compatibility is ensured by well-formedness. Thus, this claim follows directly from *UR* compatibility definition and Theorem 1. ■

Property 1. Our condition for checking DF_a and UR_a is not a necessary condition.

Proof. Let us consider the example given in Figure 4. The asynchronous system starts with an interleaving of both emissions that can be executed in peer 1 and peer 2, whereas no synchronization is possible under synchronous communication. Thus, this example is not synchronizable and we cannot conclude anything about its compatibility. Yet the asynchronous version of this system is deadlock-free compatible. As a result, our condition for asynchronous compatibility is sufficient but not necessary. ■

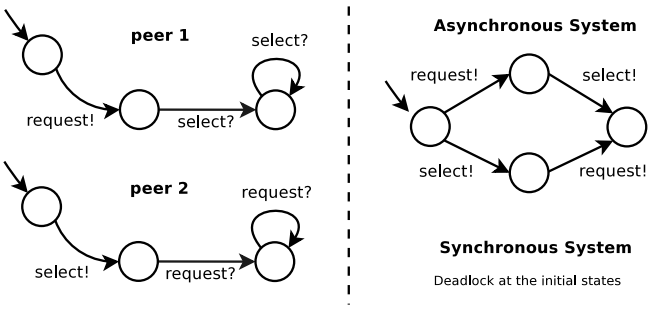

Fig. 4. Asynchronous but not Synchronous *DF* Compatible Example

Note that finding a necessary and sufficient condition for asynchronous compatibility of behavioral peers is still an open problem.

Generalization. The former results can be generalized to define a sufficient condition for verifying any notion of compatibility CN_a on synchronizable systems. Examples of other notions that can be derived are BC_a and OGC_a. For instance, OGC_a can be formalized in terms of liveness and safety properties, *e.g.*, $G(\phi \Rightarrow F\psi)$ and $G(\neg\phi)$ in LTL, resp.

Theorem 6. $(\mathcal{SYNC}(LTS_a) \wedge \mathcal{WF}(LTS_a) \wedge CN(LTS_s)) \Rightarrow CN_a(LTS_a)$

Proof. The claim follows from Theorems 1 and 3. ■

Complexity. The complexity of our asynchronous compatibility checking lies on the cost of checking the synchronizability and the compatibility on the synchronous composition. Branching bisimulation complexity is $O(S' \times T')$ [20] where S' and T' are the total number of states and transitions in LTS_s and LTS_a^1. As for compatibility checking, given n LTSs (S, s^0, Σ, T), $S = \prod_{i=1}^{n} |S_i|$ represents an upper bound of the number of possible global states, and $T = \sum_{i=1}^{n} |T_i|$ represents an upper bound for the number of transitions available from any particular global state. S and T are greater than or equal to the number of states reachable from $(I_1, \ldots, I_n)$. Both UR_a and DF_a compatibilities have a time complexity of $O(S \times T)$ and BC_a has a time complexity of $O(S^2 \times T^2)$.

5 Illustrative Example

We consider a simplified version of a Web application involving four peers: a client, a Web interface, a Web server, and a database. Figure 5 shows the peer LTSs. The *client* starts with a request (request!), and expects an acknowledgment (ack?). Then, the client either interacts with the Web server as long as it needs (access!), or decides to terminate its processing (terminate!). This internal choice is modeled using a branching of internal actions. Finally, the client waits for an invoice (invoice?). The *server* first receives a setup request (setup?). Then, the server is accessed by the client (access?) and it expects to either be released (free?) or receive an alarm if an error occurs (alarm?). Finally, the server submits information to be stored (log!), *e.g.*, start/end time and used resources. Every time a client request is received (request?), the *interface* triggers a setup request (setup!) and sends back an acknowledgment (ack!) to the client. Then, if a termination message is received (terminate?), the interface asks the Web server to be freed (free!). If an error occurs (error?), the interface sends an alarm message (alarm!). Finally, the *database* waits for some information to be stored (log?).

Synchronizability. LTS_s and LTS_a^1 are branching equivalent and therefore $\mathcal{SYNC}_{br}(LTS_a)$. Figure 6 (left) shows LTS_s, where transitions are labeled with the messages on which the peers can synchronize as presented in Definition 2.

Well-formedness. The set of peers are observationally deterministic and $\mathcal{SYNC}_{br}(LTS_a)$, hence $\mathcal{WF}(LTS_a)$.

Synchronous Compatibility. This system cannot be compatible *wrt. DF*, *UR*, and *BC* notions since the peers deadlock at the last state in Figure 6 (left). In that situation, all peers are in their initial states and may continue interacting with each other, except the client, which is expecting an invoice that is not provided by any of the partners.

Asynchronous Compatibility. Since LTS_a is branching synchronizable and well-formed, we can use results for synchronous compatibility for this system.

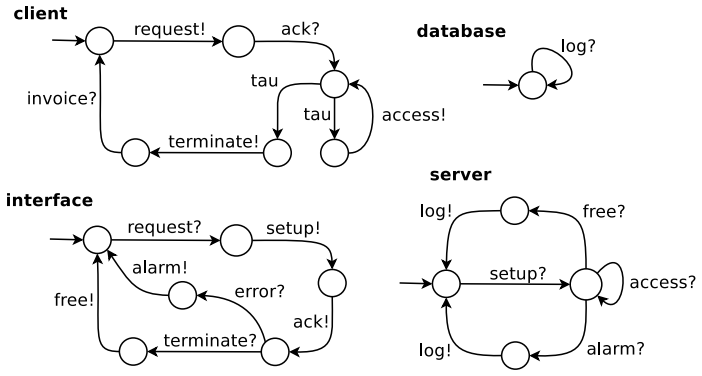

Fig. 5. Peer LTSs

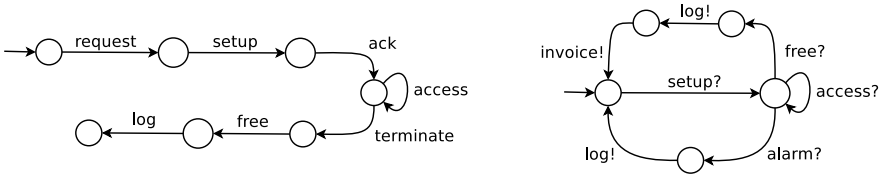

Fig. 6. Synchronous Peer Composition, V1 (left), Web Server Peer, V2 (right)

The system is not DF_a and UR_a compatible because there is a deadlock in LTS_s. We can fix this issue by, *e.g.*, adding the missing invoice! message to the server peer (Fig. 6, right). Thus, the new system is branching synchronizable (see the resulting synchronous composition in Fig. 7), well-formed, and LTS_s is deadlock-free, so it is DF_a and UR_a compatible. However, LTS_a is still not compatible *wrt.* BC_a, because there are still messages, *e.g.*, error? in the interface peer, that have no counterpart in any other peer. This issue could also be detected using GOC compatibility and checking the following LTL formula: $LTS_s \models \Diamond\Box$error.

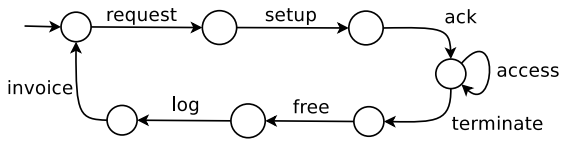

Fig. 7. Synchronous Peer Composition, V2

Note that the second version of our example with peers communicating over unbounded buffers has an infinite state space since the client, the server, and the interface peers can loop arbitrary many times while the database peer does

never consume the log? messages from its buffer. Although this is not a finite state system, we can analyze it using the techniques we propose in this paper.

6 Tool Support and Evaluation

Our approach for checking the asynchronous compatibility is fully automated. This is achieved by a translation we implemented from peer models to the LOTOS process algebra. The CADP verification toolbox [19] accepts LOTOS as input and provides efficient tools for generating LTSs from LOTOS specifications and for analyzing these LTSs using equivalence and model checking techniques, which enable us to check all the notions of compatibility presented in this paper.

6.1 LOTOS Encoding

LOTOS [23] is an abstract formal language for specifying concurrent processes, communicating via messages. We chose LOTOS because (i) it provides expressive operators for encoding LTSs and generating their compositions, and (ii) it is supported by state-of-the-art verification tools (CADP) that can be used for analyzing LOTOS specifications. With regards to compatibility checking, we first encode peer LTSs into LOTOS processes following the state machine pattern (one process is generated per state in the LTS). Each peer comes with an input buffer. Buffers are processes, which interact with the peers and store/handle messages using classic structures (lists) and operations on them (add, remove, etc.). Finally, we use the LOTOS parallel composition for specifying the synchronous and asynchronous composition of peers.

Based on this encoding, we first use state space exploration tools to generate LTSs corresponding to the LOTOS specification, in particular for synchronous and 1-bounded asynchronous system. Then, we check the synchronizability condition using branching equivalence checking, and finally we check compatibility conditions using the deadlock-freedom check or model checking of properties written in MCL [29], which subsumes both LTL and CTL.

6.2 Experiments

We carried out experiments on a Mac OS machine running on a 2.53 GHz Intel dual core processor with 4 GB of RAM. Our database of examples includes 160 examples of communicating systems: 10 case studies taken from the literature (Web services, cloud computing, e-commerce, etc.), 86 examples of Singularity channel contracts [1], which is a contract notation for Microsoft's Singularity operating system, and 64 hand-crafted examples. We emphasize that out of the 96 real-world examples, only 5 are not branching synchronizable and well-formed. Thus, 91 examples out of 96 can be analyzed using our approach.

Tables 1, 2, and 3 present experiments for some examples from our database. Each table considers DF_a compatibility for illustration purposes, but we recall that DF_a is equivalent to UR_a for asynchronous systems. Each table shows, for

each example, the number of peers, the total number of transitions and states involved in these peers, the size of the synchronous system, the size of the 1-bounded asynchronous system, the compatibility result ("$\surd$" denotes that the system is compatible, "$\times$" denotes that the system is not compatible, and "-" denotes that the system does not satisfy the sufficient condition, *i.e.*, it is not synchronizable), the successive time for computing the synchronous and 1-bounded asynchronous system, and for checking synchronizability and deadlock-freedom, respectively.

We can see that analyzing the examples given in Tables 1 and 2 only takes a few seconds. This is due to systems involving a reasonable number of peers (up to 6 in Table 1), which results in quite small LTSs, even for the 1-bounded asynchronous composition (up to 100 states and 200 transitions in Table 1).

Table 3 presents a few examples with more than 10 peers. The number of interacting peers is the main factor of state space explosion, because it induces more parallelism in the corresponding composition. The cost in terms of computation time mainly lies on the generation of the 1-bounded asynchronous system, that is compiling LOTOS code into LTSs by enumerating all the possible behaviors (interleavings of concurrent emissions/receptions) and minimizing the resulting LTS using CADP tools. In particular, reducing LTSs with respect to a branching relation needs a certain amount of time (see examples 0115, 0153, and 0159). In contrast, checking synchronizability and deadlock-freedom using equivalence and model checking techniques takes only few seconds because LTSs obtained after reduction are much smaller.

We have also made a few experiments, increasing the buffer size (k=5, k=10, etc.). We have observed that the resulting, reduced LTS remains the same due to the synchronizability property, but the generation time increases because there are more possibilities of adding/removing messages from buffers. Consequently, computation time for our solution is much lower than approaches using arbitrary bounds for buffers.

Table 1. Case Studies From the Literature

Example	|peers|	$\|T\|/\|S\|$	LTS_s	LTS_a^1	*DF*	Analysis Time		
			$\|T\|/\|S\|$	$\|T\|/\|S\|$	Comp.	Gen.	Sync.	*DF*
Supply Chain Management Application [7]	6	20/25	20/17	216/97	$\times$	5.05s	0.35s	0.15s
Health System [12]	6	21/20	10/11	22/21	$\times$	4.48s	1.99s	2.26s
Cloud System [21]	4	19/15	10/9	29/22	$\times$	4.65s	2.25s	1.88s
Cloud System (V2) [21]	4	20/16	12/10	78/43	$\surd$	4.44s	1.96s	1.60s
Sanitary Agency [30]	4	37/27	26/21	159/100	-	4.76s	2.28s	-
E-Marketplace [18]	3	8/11	6/7	15/14	$\surd$	4.35s	1.96s	1.49s
Filter Collaboration [34]	2	10/11	10/10	14/14	$\surd$	4.18s	2.22s	1.51s
Car Rental [8]	4	17/17	9/9	59/44	-	4.99s	2.04s	-
Client/Server [11]	2	10/6	9/6	19/14	-	4.68s	2.09s	-
Airline Ticket Reservation [33]	2	9/9	7/7	15/13	$\times$	4.30s	2.01s	1.49s

Table 2. Singularity Channels Contracts [1]

Example	|peers|	$\|T\|/\|S\|$	LTS_s	LTS_a^1	DF	Analysis Time		
			$\|T\|/\|S\|$	$\|T\|/\|S\|$	Comp.	Gen.	Sync.	DF
Smb Client Manager	2	40/18	21/10	41/30	√	6,83s	3.30s	2.53s
Calculator	2	12/10	7/6	13/12	√	6.89s	2.40s	2.51s
File System Controller	2	16/10	9/6	17/14	√	6,87s	2.21s	2.30s
Tcp Contract	2	8/8	5/5	9/9	√	6.61	2.55s	2.26s
Pipe Multiplex Control	2	4/4	2/2	5/5	√	6.44s	2.10s	2.27s
Udp Connection Contract	2	134/60	69/32	136/99	√	7.26s	2.52s	2.14s
IP Contract	2	64/28	33/15	65/47	√	7.07	2.30s	2.23s
Routing Contract	2	44/20	23/11	45/33	√	6.65s	2.10s	2.27s
Reservation Session	2	16/12	9/7	23/19	-	6.66s	2.37s	-
Tpm Contract	2	38/24	20/13	44/35	-	6.80	2.36s	-

Table 3. Hand-Crafted Examples

Example	|peers|	$\|T\|/\|S\|$	LTS_s	LTS_a^1	LTS_a red.	DF	Analysis Time			
			$\|T\|/\|S\|$	$\|T\|/\|S\|$	$\|T\|/\|S\|$	Comp.	Gen.	Red.	Sync.	DF
0097	9	19/19	103/27	1,543/387	98/26	√	4.59s	2.2s	2.45s	1.43s
0101	14	42/29	4,277/649	334,379/54,433	3,402/486	√	1m15s	4m2s	2.46s	1.80s
0115	16	48/41	14,754/1,945	2,332,812/326,593	11,664/1,458	√	3m34s	11m33s	2.44s	1.45s
0153	18	38/38	4,616/577	1,179,656/147,457	4,608/576	√	7.51s	18m52s	2.60s	1.43s
0159	20	45/43	15,561/1,729	7,962,633/884,737	15,552/1,728	√	24.28s	5h58m	2.62s	1.64s

7 Related Work

One of the first approaches on compatibility checking is proposed by Brand and Zafiropulo [11]. It defines the unspecified receptions compatibility notion for interaction protocols described using Communicating Finite State Machines (CFSMs). This work focuses on the compatibility of n interacting processes executed in parallel and exchanging messages via FIFO buffers. When considering unbounded buffers, the authors show that the resulting state spaces may be infinite, and the problem becomes undecidable.

The approaches used in [15, 6] deal with two kinds of processes compatibility, namely *optimistic* and *pessimistic* notions. De Alfaro and Henzinger [15] argue for the use of the optimistic notion that considers two processes P_1 and P_2 (I/O automata) as compatible if there is an environment that can *properly* communicate with their composite process. Note that an environment is also composed of one or more processes. A proper communication holds if the composition of the interface product $P_1 \otimes P_2$ with its environment is deadlock-free. The approach introduced in [6] addresses the pessimistic notion which states that two processes P_1 and P_2 are compatible if no deadlock occurs between P_1 and P_2, in *any* environment of $P_1 \otimes P_2$. [5] defines an asynchronous compatibility for modal I/O transition systems. The authors do not propose any decision criterion but they claim that this verification is undecidable in the general case due to the buffering mechanism which may lead to infinite state spaces.

[22] treats different compatibility problems for non-ordered buffers and for open systems using Petri nets. [28, 31, 25, 27] rely on an extension of Petri nets,

namely open nets to model and verify behavioral interfaces of processes described as workflows, assuming asynchronous communication over message buffers. This model provides a graphical representation, and can be computed from existing programming languages. [28] rely on the *usability* concept to analyze the compatibility of processes represented as workflows. This compatibility notion is an environment-aware compatibility where two processes A and B are considered compatible if there is an environment E, which uses the composed system $A \otimes B$. In such a case, $A \otimes B$ is considered usable, meaning that its composition with E is deadlock-free. The condition, yet necessary, is not sufficient in the case of n processes. A similar compatibility definition used in the literature is that of *controllability* [31, 25, 27]. A process A is controllable if it has a compatible partner B in the sense that the composite process $A \otimes B$ is deadlock-free. As far as asynchronous semantics is considered, controllability has proven to be undecidable for unbounded open nets. For implementing controllability, the authors require that open nets are bounded and satisfy k-limited communication, for some given k. Consequently, using a Petri net-based model requires a much higher computational and space complexity than our approach.

Darondeau and colleagues [14] identify a decidable class of systems consisting of non-deterministic communicating processes that can be scheduled while ensuring boundedness of buffers. Abdulla *et al.* [2] propose some verification techniques for CFSMs. They present a method for performing symbolic forward analysis of unbounded *lossy* channel systems. Jeron and Jard [24] propose a sufficient condition for testing unboundedness, which can be used as a decision procedure for checking reachability for CFSMs. In [26], the authors present an incomplete boundedness test for communication channels in Promela and UML RT models. They also provide a method to derive *upper bound* estimates for the maximal occupancy of each individual message buffer. More recently, [16] proposed a causal chain analysis to determine upper bounds on buffer sizes for multi-party sessions with asynchronous communication. Recently, Bouajjani and Emmi [10] consider a bounded analysis for message-passing programs, which does not limit the number of communicating processes nor the buffers' size. However, they limit the number of communication cycles. They propose a decision procedure for reachability analysis when programs can be sequentialized. By doing so, program analysis can easily scale while previous related techniques quickly explode.

8 Conclusion

In this paper, we have presented results that go beyond all existing works on checking the compatibility of systems communicating asynchronously by message exchange over unbounded buffers. In our approach, we do not have any restrictions on the number of participants, on the presence of communication cycles in behavioral models, or on the buffer sizes. Instead, we focus on the class of synchronizable systems and propose a sufficient condition for analyzing asynchronous compatibility. This results in a generic framework for verifying whether

a set of peers respect some property such as deadlock-freedom or unspecified receptions. In order to obtain these results for peer models involving internal behaviors, we have extended synchronizability results to branching time. Finally, we have implemented a prototype tool which enables us to automatically check the asynchronous compatibility using the CADP toolbox, and we have conducted experiments on many examples. In the future we plan to develop techniques for enforcing the asynchronous compatibility of a set of peers when the compatibility check fails, by automatically generating a set of distributed controllers as advocated in [21] for enforcing choreography realizability.

References

1. Singularity Design Note 5: Channel Contracts. Singularity RDK Documentation (v1.1). http://www.codeplex.com/singularity, 2004.
2. P. A. Abdulla, A. Bouajjani, and B. Jonsson. On-the-Fly Analysis of Systems with Unbounded, Lossy FIFO Channels. In *Proc. CAV'98*, volume 1427 of *LNCS*, pages 305–318. Springer, 1998.
3. S. Basu and T. Bultan. Choreography Conformance via Synchronizability. In *Proc. of WWW'11*, pages 795–804. ACM Press, 2011.
4. S. Basu, T. Bultan, and M. Ouederni. Deciding Choreography Realizability. In *Proc. of POPL'12*, pages 191–202. ACM, 2012.
5. S. S. Bauer, R. Hennicker, and S. Janisch. Interface Theories for (A)synchronously Communicating Modal I/O-Transition Systems. In *Proc. of FIT'10*, volume 46 of *EPTCS*, pages 1–8, 2010.
6. S. S. Bauer, P. Mayer, A. Schroeder, and R. Hennicker. On Weak Modal Compatibility, Refinement, and the MIO Workbench. In *Proc. of TACAS'10*, volume 6015 of *LNCS*, pages 175–189. Springer, 2010.
7. D. Beyer, A. Chakrabarti, and T. Henzinger. Web Service interfaces. In *Proc. of WWW'05*, pages 148–159. ACM, 2005.
8. D. Bianculli, D. Giannakopoulou, and C. S. Pasareanu. Interface Decomposition for Service Compositions. In *Proc. of ICSE'11*, pages 501–510. ACM, 2011.
9. L. Bordeaux, G. Salaün, D. Berardi, and M. Mecella. When are Two Web Services Compatible? In *Proc. of TES'04*, volume 3324 of *LNCS*, pages 15–28. Springer, 2004.
10. A. Bouajjani and M. Emmi. Bounded Phase Analysis of Message-Passing Programs. In *Proc. of TACAS'12*, volume 7214 of *LNCS*, pages 451–465. Springer, 2012.
11. D. Brand and P. Zafiropulo. On Communicating Finite-State Machines. *J. ACM*, 30(2):323–342, 1983.
12. A. Bucchiarone, H. Melgratti, and F. Severoni. Testing Service Composition. In *Proc. of ASSE'07*, 2007.
13. C. Canal, E. Pimentel, and J. M. Troya. Compatibility and Inheritance in Software Architectures. *Science of Computer Programming*, 41(2):105–138, 2001.
14. P. Darondeau, B. Genest, P. S. Thiagarajan, and S. Yang. Quasi-Static Scheduling of Communicating Tasks. In *Proc. CONCUR'08*, volume 5201 of *LNCS*, pages 310–324. Springer, 2008.
15. L. de Alfaro and T. Henzinger. Interface Automata. In *Proc. of ESEC/FSE'01*, pages 109–120. ACM Press, 2001.

16. P.-M. Deniélou and N. Yoshida. Buffered Communication Analysis in Distributed Multiparty Sessions. In *Proc. CONCUR'10*, volume 6269 of *LNCS*, pages 343–357. Springer, 2010.
17. F. Durán, M. Ouederni, and G. Salaün. A Generic Framework for N-Protocol Compatibility Checking. *Science of Computer Programming*, 77(7-8):870–886, 2012.
18. H. Foster, S. Uchitel, J. Kramer, and J. Magee. Compatibility Verification for Web Service Choreography. In *Proc. of ICWS'04*. IEEE Computer Society, 2004.
19. H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes. In *Proc. of TACAS'11*, volume 6605 of *LNCS*, pages 372–387. Springer, 2011.
20. J. F. Groote and F. W. Vaandrager. An Efficient Algorithm for Branching Bisimulation and Stuttering Equivalence. In *Proc. of ICALP'90*, volume 443 of *LNCS*, pages 626–638. Springer, 1990.
21. M. Güdemann, G. Salaün, and M. Ouederni. Counterexample Guided Synthesis of Monitors for Realizability Enforcement. In *Proc. of ATVA'12*, volume 7561 of *LNCS*, pages 238–253. Springer, 2012.
22. S. Haddad, R. Hennicker, and M. H. Møller. Channel Properties of Asynchronously Composed Petri Nets. In *Proc. of Petri Nets 2013*, volume 7927 of *LNCS*, pages 369–388. Springer, 2013.
23. ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, ISO, 1989.
24. T. Jéron and C. Jard. Testing for Unboundedness of FIFO Channels. *Theor. Comput. Sci.*, 113(1):93–117, 1993.
25. K. Kaschner and K. Wolf. Set Algebra for Service Behavior: Applications and Constructions. In *Proc. of BPM'09*, volume 5701 of *LNCS*, pages 193–210. Springer, 2009.
26. S. Leue, R. Mayr, and W. Wei. A Scalable Incomplete Test for Message Buffer Overflow in Promela Models. In *Proc. SPIN'04*, volume 2989 of *LNCS*, pages 216–233. Springer, 2004.
27. N. Lohmann. Why Does My Service Have No Partners? In *Proc. of WS-FM'08*, volume 5387 of *LNCS*, pages 191–206. Springer, 2008.
28. A. Martens, S. Moser, A. Gerhardt, and K. Funk. Analyzing Compatibility of BPEL Processes. In *Proc. of AICT/ICIW'06*, pages 147–156. IEEE Computer Society, 2006.
29. R. Mateescu and D. Thivolle. A Model Checking Language for Concurrent Value-Passing Systems. In *Proc. of FM'08*, volume 5014 of *LNCS*. Springer, 2008.
30. G. Salaün, L. Bordeaux, and M. Schaerf. Describing and Reasoning on Web Services using Process Algebra. *International Journal on Business Process and Integration Management*, 1(2):116–128, 2006.
31. W. M. P. van der Aalst, A. J. Mooij, C. Stahl, and K. Wolf. Service Interaction: Patterns, Formalization, and Analysis. In *Proc. of SFM'09*, volume 5569 of *LNCS*, pages 42–88. Springer, 2009.
32. R. J. van Glabbeek and W. P. Weijland. Branching Time and Abstraction in Bisimulation Semantics. *J. ACM*, 43(3):555–600, 1996.
33. P. Wong and J. Gibbons. Verifying Business Process Compatibility. In *Proc. of QSIC'08*, pages 126–131. IEEE Computer Society, 2008.
34. D. M. Yellin and R. E. Strom. Protocol Specifications and Component Adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, 1997.