Automated Analysis of Asynchronously Communicating Systems

Lakhdar Akroun[1], Gwen Salaün[1], and Lina Ye[2]

[1] University of Grenoble Alpes, Inria, LIG, CNRS, France
[2] CentraleSupélec, LRI, France

Abstract. Analyzing systems communicating asynchronously via reliable FIFO buffers is an undecidable problem. A typical approach is to check whether the system is bounded, and if not, the corresponding state space can be made finite by limiting the presence of communication cycles in behavioral models or by fixing the buffer size. In this paper, our focus is on systems that are likely to be unbounded and therefore result in infinite systems. We do not want to restrict the system by imposing any arbitrary bound. We introduce a notion of stability and prove that once the system is stable for a specific buffer bound, it remains stable whatever larger bounds are chosen for buffers. This enables one to check certain properties on the system for that bound and to ensure that the system will preserve them whatever larger bounds are used for buffers. We also prove that computing this bound is undecidable but we show how we succeed in computing these bounds for many typical examples using heuristics and equivalence checking.

1 Introduction

Most software systems are constructed by reusing and composing existing components or peers. This is the case in many different areas such as component-based systems, distributed cloud applications, Web services, or cyber-physical systems. Software entities are often stateful and therefore described using behavioral models. Moreover, asynchronous communication via FIFO buffers is a classic communication model used for such distributed, communicating systems. A crucial problem in this context is to check whether a new system consisting of a set of interacting peers respects certain properties. Analyzing asynchronously communicating software has been studied extensively in the last 30 years and is known to be undecidable in general [7]. A common approach to circumvent this issue is to bound the state space by restricting the cyclic behaviors or imposing an arbitrary bound on buffers. Bounding buffers to an arbitrary size during the execution is not a satisfactory solution: 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.

In this paper, we propose a new approach for analyzing a set of peers described using LTSs, communicating asynchronously via reliable (no loss of messages) and possibly unbounded FIFO buffers. We do not want to restrict the

system by imposing any arbitrary bound on cyclic behaviors or buffers. We introduce a notion of stability for the asynchronous versions of the system. A system is stable if asynchronous compositions exhibit the same observable behavior (send actions) from some buffer bound. This property can be verified in practice using equivalence checking techniques on finite state spaces by comparing bounded asynchronous compositions, although the system consisting of peers interacting asynchronously via unbounded buffers can result in infinite state spaces. We prove that once the system is stable for a specific buffer bound, it remains stable whatever larger bounds are chosen for buffers. This enables one to check temporal properties on the system for that bound (using model checking techniques for instance) and ensures that the system will preserve them whatever larger bounds are used for buffers. We also prove that computing this bound is undecidable, but we show how we succeed in computing such bounds in practice for many examples.

Figure 1 gives an example where peers are modeled using Labeled Transition Systems (LTSs). Transitions are labeled with either emissions (exclamation marks) or receptions (question marks). Initial states are marked with incoming half-arrows. In the asynchronous composition, each peer is equipped with one input buffer, and we consider only the ordering of the send actions, ignoring the ordering of receive actions. Focusing only on send actions makes sense for verification purposes 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. We can use our approach to detect that when each peer is equipped with a buffer bound fixed to 2, the observable behavior of the system depicted in Figure 1 is stable. This means that we can check properties, such as the absence of deadlocks, on the 2-bounded asynchronous version of the system and the results hold for any asynchronous version of the system where buffer bounds are greater or equal to 2.

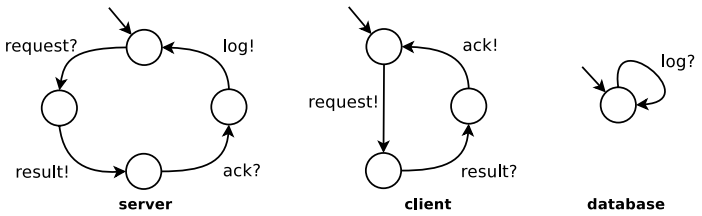

Fig. 1. Motivating example

We implemented our approach in a tool that first encodes the peer LTSs and their compositions into process algebra, and then uses heuristics, search algorithms, and equivalence checking techniques for verifying whether the system satisfies the stability property. If this is the case, we return the smallest bound respecting this property. Otherwise, when we reach a certain maximal bound,

our check returns an inconclusive result. Heuristics and search algorithms aim at guiding the approach towards the smallest bound satisfying stability whereas equivalence checking techniques are used for checking the stability property given a specific bound k. All the steps of our approach are fully automated (no human intervention). We applied our tool support to more than 300 examples of communicating systems, many of them taken from the literature on this topic. These experiments show that a large number of these examples are stable and can therefore be formally analyzed using our approach.

The contributions of this paper are summarized as follows:

- The introduction of the stability property for asynchronously communicating systems that, once acquired for a bound k, is preserved for upper bounds;
- A proof demonstrating that computing such a bound k is undecidable;
- A fully automated tool support that shows that the bound exists for a large majority of our dataset of examples found in the literature.

The organization of the rest of this paper is as follows. Section 2 defines our model for peers and their asynchronous composition. Section 3 presents the stability property and our results on stable systems. Section 4 describes our tool support and experiments we carried out to evaluate our approach. Finally, Section 5 reviews related work and Section 6 concludes.

2 Communicating Systems

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. *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, a set of 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 for representing internal activities. A transition is represented as $s \xrightarrow{l} s' \in T$ where $l \in \Sigma$. This can be directly extended to $s \xrightarrow{\sigma} s'$, $\sigma \in \Sigma^*$, where $\sigma = l_1, ..., l_n$, $s \xrightarrow{l_1} s_1, \ldots, s_i \xrightarrow{l_{i+1}} s_{i+1}, \ldots, s_{n-1} \xrightarrow{l_n} s' \in T$. In the following, for the sake of simplicity, we will denote this by $s \xrightarrow{\sigma} s' \in T^*$.

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. Nondeterminism can also result from internal choices when several transitions (at least two) outgoing from a same state are labeled with τ. Given a set of peers $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$, we assume that each message has a unique sender and a unique receiver: $\forall i, j \in [1, n]$, $i \neq j$, $\Sigma_i^! \cap \Sigma_j^! = \emptyset$ and $\Sigma_i^? \cap \Sigma_j^? = \emptyset$. Furthermore, each message is exchanged between two different peers: $\Sigma_i^! \cap \Sigma_i^? = \emptyset$ for all i. We also

assume that each emission has a reception counterpart in another peer (closed systems): $\forall i \in [1,n]$, $\forall m \in \Sigma_i^! \implies \exists j \in [1,n]$, $i \neq j$, $m \in \Sigma_j^?$.

In the asynchronous composition, the peers communicate with each other asynchronously via FIFO buffers. Each peer $\mathcal{P}_i$ is equipped with an unbounded input message buffer Q_i. A peer $\mathcal{P}_i$ can either send a message $m \in \Sigma_i^!$ to the tail of the receiver buffer Q_j of $\mathcal{P}_j$ at any state where this send message is available, read a message $m \in \Sigma_i^?$ from its buffer Q_i if the message is available at the buffer head, or evolve independently through an internal transition. We focus on send actions in this paper. We consider that reading from the buffer is private non-observable information, which is encoded as an internal transition in the asynchronous system.

Definition 2. *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* $(P_1|Q_1)|...|(P_n|Q_n)$ *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 $LTS_a^k = (S_a^k, s_a^0, \Sigma_a^k, T_a^k)$ to define the *bounded* asynchronous composition, where each message buffer bounded to size k is denoted Q_i^k, for $i \in [1,n]$. The definition of LTS_a^k can be obtained from Def. 2 by allowing send transitions only if the message buffer of the receiving peer has less than k messages in it. Otherwise, the sender is blocked, *i.e.*, we assume reliable communication without message losses. The k-bounded asynchronous product can be denoted $(P_1|Q_1^k)|...|(P_n|Q_n^k)$ or $(P_1|(Q_1^1|\ ...\ |Q_1^1))|...|(P_n|(Q_n^1|\ ...\ |Q_n^1))$, where each peer is in parallel with the parallel composition of k buffers of size one. Let us emphasize that the encoding of an ordered bounded buffer following this pattern based on parallel composition was originally proposed by R. Milner in [37] (see Sections 1.2 and 3.3 of this book for details). Furthermore, we use $\overline{LTS_a}$ for the asynchronous composition where the receptions are kept in the resulting LTS ($s \xrightarrow{m?} s' \in \overline{T_a}$, *consume* rule in Def. 2) instead of being encoded as τ.

3 Stability-based Verification

In this section, we show that systems consisting of a finite set of peers involving cyclic behaviors and communicating over FIFO buffers may stabilize from a specific buffer bound k. We call this property *stability* and we say that the corresponding systems are *stable*. The class of systems that are stable corresponds to systems whose asynchronous compositions remain the same from some buffer bound when we observe send actions only (we ignore receive actions and buffer contents). Since stable systems produce the same behavior from a specific bound k, they can be analyzed for that bound to detect for instance the presence of deadlocks or to check whether they satisfy any kind of temporal properties. Stability ensures that these properties will be also satisfied for larger bounds. The stability definition relies on branching bisimulation checking [42] (Definition 3). We chose branching bisimulation because in this work receive actions are hidden as internal behaviors, and branching bisimulation is the finest equivalence notion in presence of internal behaviors. This equivalence preserves properties written in ACTL\X logic [34].

Definition 3. *Given two LTSs LTS_1 and LTS_2, they are branching bisimilar, denoted by $LTS_1 \equiv_{br} LTS_2$, if there exists a symmetric relation R (called a branching bisimulation) between the states of LTS_1 and LTS_2 satisfying the following two conditions: (i) The initial states are related by R; (ii) If $R(r, s)$ and $r \xrightarrow{\delta} r\prime$, then either $\delta = \tau$ and $R(r\prime, s)$, or there exists a path $s \xrightarrow{\tau^*} s_1 \xrightarrow{\delta} s\prime$, such that $R(r, s_1)$ and $R(r\prime, s\prime)$. For the sake of simplicity, in the following, $R(r, s)$ is also denoted by $r \equiv_{br} s$.*

Definition 4. *Given a set of peers $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$, we say that this system is stable if and only if $\exists k$ such that $LTS_a^k \equiv_{br} LTS_a^q$ $(\forall q > k)$.*

As a first result, we show a sufficient condition to ensure stability: if there exists a bound k such that the k-bounded and the $(k+1)$-bounded asynchronous systems are branching equivalent, then we prove that the system remains stable, meaning that the observable behavior is always the same for any bound greater than k.

Theorem 1. *Given a set of peers $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$, if $\exists k \in \mathbb{N}$, such that $LTS_a^k \equiv_{br} LTS_a^{k+1}$, then we have $LTS_a^k \equiv_{br} LTS_a^q, \forall q > k$.*

Proof. We will prove the theorem by induction, starting with the following base case: If $LTS_a^k \equiv_{br} LTS_a^{k+1}$ then $LTS_a^k \equiv_{br} LTS_a^{k+2}$. Let us recall that the strong and branching bisimulations are congruences with respect to the operators of process algebras [19], that is, if P and P' are branching bisimilar, then for every Q we have $P|Q \equiv_{br} P'|Q$. Now, suppose that $\exists k \in \mathbb{N}$, such that $LTS_a^k \equiv_{br} LTS_a^{k+1}$, then:

$$(P_1|Q_1^k)|...|(P_n|Q_n^k) \equiv_{br} (P_1|Q_1^{k+1})|...|(P_n|Q_n^{k+1}) \quad (1)$$

A buffer of size k can be written as a parallel composition of k buffers of size 1 (see Sections 1.2 and 3.3 in [37] for details), hence:

$$(P_1|Q_1^{k+2})|...|(P_n|Q_n^{k+2}) \equiv_{br} (P_1|Q_1^{k+1}|Q_1^1)|...|(P_n|Q_n^{k+1}|Q_n^1) \quad (2)$$

Then, by congruence and using equation (1) we have:

$$(P_1|Q_1^{k+2})|...|(P_n|Q_n^{k+2}) \equiv_{br} (P_1|Q_1^{k}|Q_1^1)|...|(P_n|Q_n^{k}|Q_n^1) \quad (3)$$

$$(P_1|Q_1^{k+2})|...|(P_n|Q_n^{k+2}) \equiv_{br} (P_1|Q_1^{k+1})|...|(P_n|Q_n^{k+1}) \quad (4)$$

$$(P_1|Q_1^{k+2})|...|(P_n|Q_n^{k+2}) \equiv_{br} (P_1|Q_1^{k})|...|(P_n|Q_n^{k}) \quad (5)$$

The same argument can be used to prove the induction case, *i.e.*, we suppose that $LTS_a^k \equiv_{br} LTS_a^{k+i}$ and we demonstrate that $LTS_a^k \equiv_{br} LTS_a^{k+i+1}$. This proves that if $LTS_a^k \equiv_{br} LTS_a^{k+1}$, then we have $LTS_a^k \equiv_{br} LTS_a^q, \forall q > k$. ■

The main interest of the stability property is that any temporal property can be analyzed using existing model checking tools on the minimal k-bounded version of the system, and this result ensures that these properties are preserved when buffer bounds are increased or if buffers are unbounded.

Proposition 1. *Given a set of peers $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$, if $\exists k$ s.t. $LTS_a^k \equiv_{br} LTS_a^q$ $(\forall q > k)$, and for some property P written in ACTL\X logic, $LTS_a^k \models P$, then $LTS_a^q \models P$ $(\forall q > k)$.*

Classic properties, such as liveness or safety properties, can be verified considering emissions only. If one wants to check a property involving receptions, a solution is to replace in the property a specific reception by one of the emissions (if there is one) occurring next in the corresponding peer LTS.

We prove now that determining whether a system is stable is an undecidable problem. Yet there are many cases in which stability is satisfied and the corresponding bound can be computed in those cases using heuristics, search algorithms, and equivalence checking (see Section 4).

Theorem 2. *Given a set of peers $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$, it is undecidable whether the corresponding asynchronous system is stable.*

Proof. We reduce the halting problem of a Turing machine to the test of stability of a set of peers communicating asynchronously. The construction is inspired from [18]. The Turing machine used is a deterministic one-way-infinite single tape model. A Turing machine is defined as $M = (Q_M, \Sigma_M, \Gamma_M, q_0, q_{halt}, B, \delta_M)$ where Q_M is the set of states, Σ_M is the input alphabet, Γ_M is the tape alphabet, $q_0 \in Q_M$ is the initial state and q_{halt} is the accepting state. $B \in \Gamma_M$ is the blank symbol and $\delta_M : Q_M \times \Gamma_M \rightarrow Q_M \times \Gamma_M \times \{left, right\}$ is the transition function. The machine M accepts an input word $w = a_1, ..., a_m$ iff M halts on w. If M does not halt on w, then M moves its tape head far to the right. A *configuration* of the Turing machine M is a word $uqv\#$ where uv is a word from the tape alphabet, q is a state of M (meaning that M is in the state q and the head

pointing on the first symbol of v), and # is a fixed symbol which is not in Γ_M (used to indicate the end of the word on the tape).
To ease the understanding, we shall present the reduction in two phases. In a first phase (i), starting from a Turing machine M and an input word w, we construct a pair of peers P_1 and P_2, such that whenever the machine M halts on w or not, there always exists a k such that $LTS_a^k \equiv_{br} LTS_a$, where LTS_a is the asynchronous product of the system $\{P_1, P_2\}$. In a second phase (ii), we extend P_1 and P_2 respectively to $P_1^{'}$ and $P_2^{'}$ such that M halts iff $\nexists k$ such that $LTS_a^{'k} \equiv_{br} LTS_a^{'}$, where $LTS_a^{'}$ is the asynchronous product of the system $\{P_1^{'}, P_2^{'}\}$.
Phase (i): The peer P_1 simulates the execution of the machine M on w while P_2 is used to receive and re-send messages to P_1. A configuration of M of the form $uqv\#$ is encoded with the buffer of P_1 with the content $uheadv\#$.
The peer P_1 is defined as $(S_{P_1}, s_{q_0}, \Sigma_{P_1}, T_{P_1})$ where S_{P_1} is the set of states, s_{q_0} is the initial state where q_0 is the initial state of M. The alphabet $\Sigma_{P_1} = \Sigma_{P_1}^! \cup \Sigma_{P_1}^?$ is defined as follows:

- $\Sigma_{P_1}^! = \Sigma_M \cup \Gamma_M \cup \{head\} \cup \{\#\}$ where all messages sent from P_1 to P_2 are indexed with 2 (*e.g.,* P_1 sends B^2 instead of sending the blank symbol, inversely P_2 sends B^1 instead of sending B to P_1).
- $\Sigma_{P_1}^? = \Sigma_M \cup \Gamma_M \cup \{head\} \cup \{\#\}$ where all messages received from P_2 are indexed with 1.

Now we present how each action of the machine M is encoded.

- For each transition of M of the form $\delta_M(q, a) = (q', a', right)$ we have the following transitions in T_{P_1}: $s_q \xrightarrow{head^1?} s_1 \xrightarrow{a^1?} s_2 \xrightarrow{a'^2!} s_3 \xrightarrow{head^2!} s_{q'}$.
  If the peer is in the state q and the buffer starts with $head^1a^1$ then the two messages are read and the peer P_1 sends the next configuration to P_2 as depicted in Figure 2(a). s_i's are fresh intermediary states.
- For each transition of M of the form $\delta_M(q, a) = (q', a', left)$ and for each $x \in \Gamma_M$ we have the following transitions in T_{P_1}:
  $s_q \xrightarrow{x^1?} s_1 \xrightarrow{head^1?} s_2 \xrightarrow{a^1?} s_3 \xrightarrow{head^2!} s_4 \xrightarrow{x^2!} s_5 \xrightarrow{a'^2!} s_{q'}$. P_1 starts by reading the letter before *head*, then it reads *head*, the next letter, and sends the new configuration to P_2 as depicted in Figure 2(b).
- For each state s_q where q is a state of M we have the following cycle in T_{P_1}:
  $s_q \xrightarrow{head^1?} s_1 \xrightarrow{\#^1?} s_2 \xrightarrow{head^2!} s_3 \xrightarrow{B^2!} s_4 \xrightarrow{\#^2!} s_q$.
  As depicted in Figure 2(c), the configuration of M is extended to the right with a blank symbol. Peer 1 starts by reading the current configuration of the machine, then sends the next configuration of M to P_2 (P_1 adds a blank symbol before #).
- For each letter $x \in \Gamma_M \cup \{\#\}$ and each s_q where q is a state of M, we have the following cycle: $s_q \xrightarrow{x^1?} s_1 \xrightarrow{x^2!} s_q$ where P_1 reads x indexed with 1, then sends x indexed with 2.

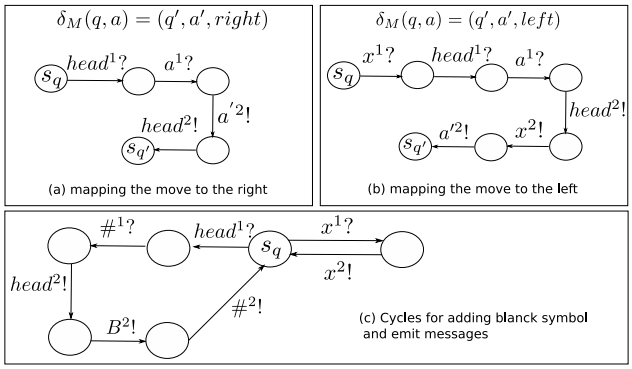

Fig. 2. Mapping the instructions of the machine M to transitions of the peer P_1

Note that at a state s_q representing a state q of the machine M, there is only one outgoing transition labeled with $head^1?$, hence P_1 is deterministic.

The peer P_2 is defined as $(S_{P_2}, s_{init}, \Sigma_{P_2}, T_{P_2})$ where S_{P_2} is the set of states and s_{init} the initial state. P_2 starts by sending the initial configuration of M to P_1, then reaches the state s_{univ}, which contains a set of cycles used to receive any message from P_1 and re-send them. Σ_{P_2}=$\Sigma^!_{P_2} \cup \Sigma^?_{P_2}$ is defined as follows:

- $\Sigma^!_{P_2}$= $\Sigma_M \cup \Gamma_M \cup \{head\} \cup \{\#\}$ where all messages sent from P_2 to P_1 are indexed with 1.
- $\Sigma^?_{P_2}$= $\Sigma_M \cup \Gamma_M \cup \{head\} \cup \{\#\}$ where all messages received from P_1 are indexed with 2.

P_2 contains the following transitions:

- $s_{init} \xrightarrow{head^1!} s_1 \xrightarrow{a_1^1!} ... \xrightarrow{a_n^1!} s_n \xrightarrow{\#^1!} s_{univ} \in T_{P_2}$ where $w = a_1 a_2 ... a_n$.
- $s_{univ} \xrightarrow{x^2?} s_1 \xrightarrow{x^1!} s_{univ} \in T_{P_2}$ where x is any symbol in $\Sigma^?_{P_2}$.

To prove that $\exists k$ such that $LTS_a^k \equiv_{br} LTS_a$, it is sufficient to prove that $\exists k$ such that $LTS_a^k \equiv_{br} LTS_a^{k+1}$ (from Theorem 1). Suppose that M halts on w. Then, the number of configurations of the machine is finite. Hence, from our construction, the asynchronous product of $\{P_1, P_2\}$ is finite, so there exists a k such that $LTS_a^k \equiv_{br} LTS_a^{k+1}$, hence $LTS_a^k \equiv_{br} LTS_a$.
Now suppose the machine does not halt on w. Then, the corresponding communicating system executes infinitely two cycles: (1) one adding a blank symbol, (2) another reading blank symbols and moving to the right. Hence, for a given bound k, the behavior of the system resulting from the execution of one of the two cycles in LTS_a^{k+1} may not be reproduced in LTS_a^k, due to the buffer bound, then $LTS_a^k \not\equiv_{br} LTS_a^{k+1}$. We will prove that, with our construction, this case never happens, that is $LTS_a^k \equiv_{br} LTS_a$ when the machine M does not halt on

w. This is possible because with our construction, each border state[3] can make a reception before sending a message, hence we obtain a confluent diamond and the branching bisimulation is maintained [27]. Now we will detail the proof for cycles of type (1), because this is the only case where the peer P_1 will increment the buffer by adding a blank symbol. Suppose the machine M does not stop. Let s^k be the configuration of LTS_a^k representing the configuration of the machine M when starting to execute the infinite loop. From our construction, at s^k the system can execute the first cycle adding a blank symbol. Note that in s^k the buffer of P_1 is full (size equal to k) and the buffer of P_2 is empty (which corresponds to the configuration of M). More precisely, the buffer of P_1 contains the following word: $head^1\#^1a_1^1...a_m^1$, where $m = k-2$. It is easy to verify that such a state exists, because at a state s_q representing a state q of the machine M, P_1 can enter two cycles, one starting by reading $head^1$, the other one reading any other symbol. Hence, if the first symbol of the buffer of P_1 is not $head^1$, P_1 reads the symbol and sends it to P_2 which re-emits the symbol to P_1. Thus, in the configuration s^k, the first symbol is $head^1$. Then, it executes the cycle which adds a blank symbol: $s^k \xrightarrow{head^1?} s_1 \xrightarrow{\#^1?} s_2 \xrightarrow{head^2!} s_3 \xrightarrow{B^2!} s_4 \xrightarrow{\#^2!} s^{k'}$.
At $s^{k'}$ the buffer of P_1 contains $k-2$ messages and the buffer of P_2 three messages. The sum of the two buffers is $k+1$ messages, due to the addition of the blank symbol, but $s^{k'}$ is still in LTS_a^k. From our construction, at the configuration $s^{k'}$, P_1 sends a_1^2,..., a_{m-1}^2: $s^{k'} \xrightarrow{a_1^2!} s_1 \xrightarrow{a_2^2!} ... \xrightarrow{a_{m-1}^2!} s^{k''}$.
At $s^{k''}$ the buffer of P_2 contains k messages and the buffer of P_1 one message. At this configuration, P_1 sends the message a_m^2, and the system reaches a configuration s^{k+1} which is in LTS_a^{k+1} but not in LTS_a^k.
Hence, LTS_a^{k+1} and LTS_a^k can send the same sequences of messages from the initial state to the border state $s^{k''}$. Then, LTS_a^{k+1} can send the message a_m^2. Moreover, in the configuration $s^{k''}$, P_2 can read a message (because it is in the state s_{univ}). Thus, in LTS_a^k there is the following sequence: $s^{k''} \xrightarrow{\tau} s_1 \xrightarrow{a_m^1!} s'^k$.
Hence, we obtain a confluent diamond from $s^{k''}$. With our construction, each border state can make a reception before sending a message, hence we obtain a confluent diamond and the branching bisimulation is maintained. Hence, if the machine does not halt on w, then $\exists k$ such that $LTS_a^k \equiv_{br} LTS_a$.
Note that, since proving $LTS_a^k \equiv_{br} LTS_a^{k+1}$ is sufficient to prove $LTS_a^k \equiv_{br} LTS_a$, we do not need to prove our statement for buffer size containing more than k or $k+1$ messages. The bound k depends on the execution of the machine M and the word w, where k represents the buffer size needed to represent the configuration of the machine M when starting to execute the infinite loop.

Phase (ii): Until now, whenever the machine M halts on w or not, there exists a k such that $LTS_a^k \equiv_{br} LTS_a$. Now we extend P_1 and P_2 respectively to obtain P_1' and P_2' such that if the machine M halts on w, then there exists no k such that $LTS_a'^k \equiv_{br} LTS_a'$. This is achieved by adding to P_1 the transition

[3] A border state s^k in LTS_a^k is a state where at least one buffer contains k messages and there is an outgoing transition from s^k labeled with a send message which reaches a state in LTS_a^{k+1}.

system P_a to obtain P_1' and adding to P_2 the transition system P_b to obtain P_2', such that in the system $\{P_a, P_b\}$ there is no k where $LTS_a^{''k} \equiv_{br} LTS_a^{''}$. The peers P_a and P_b are not formally defined, we can choose any two peers which are not stable. The set of additional transitions used to connect P_1 to P_a and P_2 to P_b are listed below:

- s_0 and s_0' are respectively the initial states of P_a and P_b. The messages exchanged between P_a and P_b do not appear in P_1 and P_2.
- $s_{q_{halt}} \xrightarrow{halt!} s_0 \in T_{P_1'}$.
- $s_{univ} \xrightarrow{halt?} s_0' \in T_{P_2'}$.
- $s_0 \xrightarrow{x?} s_0 \in T_{P_1'}$, where x is any letter in $\Sigma^?_{P_1}$.
- $s_0' \xrightarrow{y?} s_0' \in T_{P_2'}$, where y is any letter in $\Sigma^?_{P_2}$.

With this construction, if the machine M halts on w, $P_1^{'}$ reaches the state $s_{q_{halt}}$. Then it sends the message *halt* to $P_2^{'}$ and reaches the state s_0. $P_2^{'}$ is in the state s_{univ}, hence, it reads the message *halt* and reaches the state s_0'. At s_0 and s_0', the two peers empty their buffers, start executing P_a and P_b, and thus the stability is violated. Note that the executions where the two peers do not empty their buffers deadlocks, and in that case, the stability is maintained.
With this reduction we prove that, the machine M halts on w iff $\nexists k$ such that $LTS_a^{'k} \equiv_{br} LTS_a^{'}$, and this shows that checking the stability is undecidable. ■

Another result concerns well-formed systems [3]. A system consisting of a set of peers is well-formed iff whenever the size of the buffer, Q_i, of the i-th peer is non-empty, the system can move to a state where Q_i is empty. In other words, well-formedness concerns the ability of a system to eventually consume all messages in any of its buffers. In order to check this property, we have to keep receive messages and thus analyze the system on its asynchronous composition $\overline{LTS_a}$ instead of LTS_a .

Definition 5. *Given a set of peers $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$, it is well-formed, denoted by $WF(\overline{LTS_a})$, if $\forall s = (s_1, Q_1, ..., s_n, Q_n) \in \overline{S_a}, \forall Q_i$, it holds that if $|Q_i| > 0$, then $\exists s \xrightarrow{\sigma} s\prime \in \overline{T_a}^*$, where $s\prime = (s_1\prime, Q_1\prime, ..., s_n\prime, Q_n\prime) \in \overline{S_a}, |Q_i\prime| = 0$. The well-formedness property can be checked with the CTL temporal formula on $\overline{LTS_a}$: $AG(|Q_i| > 0 \Rightarrow EF(|Q_i| = 0))$.*

One can check whether a stable system is well-formed for the smallest k satisfying stability for instance. If a system is both stable and well-formed for this smallest k, then it remains well-formed for larger bound q greater than k.

Theorem 3. *Given a set of peers $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$, if $\exists k$ s.t. $LTS_a^k \equiv_{br} LTS_a^q$ $(\forall q > k)$ and $WF(\overline{LTS_a^k})$, then we have $WF(\overline{LTS_a^q})$ $(\forall q > k)$.*

Proof. Suppose that there exists a k such that $LTS_a^k \equiv_{br} LTS_a^q$ $(\forall q > k)$. We know from Proposition 1 that the stability preserves properties written in ACTL\X logic, *i.e.,* when the system is stable and $LTS_a^k \models P$, then $LTS_a^q \models P$

$(\forall q > k)$, where P is a property written in this logic. Well-formedness is a property expressed in ACTL\X logic. Hence, $WF(\overline{LTS_a^k})$ implies $WF(\overline{LTS_a^q})$ $(\forall q > k)$ when the system is stable. ■

4 Tool Support

Our tool support works as follows. Given a set of peer LTSs, we compute an initial bound k. For that bound, we verify whether the k-bounded asynchronous system is branching equivalent to the $(k + 1)$-bounded system. If this is the case, the system is stable for bound k, and properties can be analyzed using that bound. If the equivalence check returns false, we modify k and apply the check again. We repeat the process up to a certain arbitrary bound *kmax* that makes the approach abort inconclusively if attained. All these checks are achieved automatically using compilers, exploration tools, and equivalence checking tools available in CADP [22].

Several heuristics and search algorithms can be used for computing an initial bound k and the next bound to attempt, resp. In the experiments we present in this section, the initial k is computed as the maximum between the longest sequence of emissions in all peers and the highest number of emissions destinated to a same peer. The intuition behind the longest sequence of emissions is that all peers can at least send all their messages even if no peer consumes any message from its buffer. The second part of the maximum function corresponds to the case in which the buffer size prevents some peers to send messages because that buffer is already full. Next values of k are computed by decrementing / incrementing the bound till reaching *kmax* or the smallest k satisfying stability. Other heuristics and search algorithms have been attempted (*e.g.*, binary search) but turned out to be less efficient.

Experimental Results. We used a Mac OS laptop running on a 2.3 GHz Intel Core i7 processor with 16 GB of memory and carried out experiments on more than 300 examples. Table 1 presents experimental results for some real-world examples as well as larger (hand-crafted) examples for showing how our approach scales. The table gives for each example the number of peers (P), the total number of states (S) and transitions (T) involved in these peers, the bound k if the system is stable (0 if the synchronous and 1-bounded asynchronous composition are equivalent, and *kmax* if this upper bound is reached during the analysis process), the size of the k-bounded asynchronous system (minimized modulo branching reduction), and the time for applying the whole process. During our experiments, we used a bound *kmax* arbitrarily fixed to 10.

Out of the 27 examples found in the literature, 23 are stable and can be analyzed using the approach proposed in this paper. In most cases, LTSs are quite small and computation times reasonable (up to a few minutes). These times increase due to the computation of intermediate state spaces, which grow with the size of the buffer bounds. Examples (28) to (34) show how LTSs and computation time grow mainly with the number of peers. It is worth observing that some examples that are presented as unbounded and then impossible to

Id	Description	$\|P\|$	$\|S\|/\|T\|$	k	LTS_a^k ($\|S\|/\|T\|$)	Time (sec)
(1)	Estelle specification [29]	2	7/9	5	28/54	120
(2)	News server [35]	2	9/9	3	14/22	85
(3)	Client/server [7]	2	6/10	0	3/4	34
(4)	CFSM system [29]	2	6/7	$kmax$	393/802	212
(5)	Promela program (1) [30]	2	6/6	1	3/4	66
(6)	Promela program (2) [31]	2	8/8	$kmax$	275/616	228
(7)	Web services [20]	3	13/12	0	7/7	44
(8)	Trade system [17]	3	12/12	0	30/46	44
(9)	Online stock broker [21]	3	13/16	$kmax$	197/452	>1h
(10)	FTP transfer [6]	3	20/17	2	15/19	155
(11)	Client/server [11]	3	14/13	0	8/7	44
(12)	Mars explorer [8]	3	34/34	2	21/25	140
(13)	Online computer sale [14]	3	26/26	0	11/12	69
(14)	E-museum [12]	3	33/40	3	27/46	182
(15)	Client/supplier [10]	3	31/33	0	17/19	44
(16)	Restaurant service [41]	3	15/16	1	10/12	68
(17)	Travel agency [40]	3	32/38	0	18/21	44
(18)	Vending machine [24]	3	15/14	0	8/8	44
(19)	Travel agency [4]	3	42/57	3	29/42	112
(20)	Train station [39]	4	18/18	2	19/26	165
(21)	Factory job manager [9]	4	20/20	0	12/15	54
(22)	Bug report repository [25]	4	12/12	1	7/8	136
(23)	Cloud application [28]	4	8/10	$kmax$	26,754/83,200	337
(24)	Sanitary agency [38]	4	35/41	3	44/71	137
(25)	SQL server [36]	4	32/38	2	22/31	170
(26)	SSH protocol [32]	4	26/28	0	16/18	97
(27)	Booking system [33]	5	45/53	1	27/35	>1h
(28)	Hand-crafted example	5	396/801	4	17,376/86,345	189
(29)	——	6	16/18	5	202/559	188
(30)	——	7	38/38	6	1,716/6,468	393
(31)	——	10	48/47	8	14,904/57,600	294
(32)	——	14	85/80	4	19,840/113,520	485
(33)	——	16	106/102	3	22,400/132,400	453
(34)	——	20	128/116	4	80,640/522,480	699

Table 1. Experimental Results

analyze in the literature are actually stable, see, *e.g.*, (1), which means that they can be analyzed for the minimal k satisfying stability and these properties still hold for larger buffer bounds.

5 Related Work

Brand and Zafiropulo show in [7] that the verification problem for FSMs interacting via (unbounded) FIFO buffers is undecidable. Gouda *et al.* [26] presents sufficient conditions to compute a bound k from which two finite state machines

communicating through 1-directional channels are guaranteed to progress indefinitely. Jeron and Jard [29] propose a sufficient condition for testing unboundedness, which can be used as a decision procedure in order to check reachability for CFSMs. Abdulla *et al.* [1] propose some verification techniques for CFSMs. They present a method for performing symbolic forward analysis of unbounded *lossy* channel systems. In [30], 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. Cécé and Finkel [13] focus on the analysis of infinite half-duplex systems and present several (un)decidability results. For instance, they prove that a symbolic representation of the reachability set is computable in polynomial time and show how to use this result to solve several verification problems.

A notion of existential-boundedness was introduced in [23] for communicating automata. The idea is to assume unbounded channels, but to consider only executions that can be rescheduled on bounded ones. Darondeau *et al.* [15] identify a decidable class of systems consisting of non-deterministic communicating processes that can be scheduled while ensuring boundedness of buffers. [16] proposed a causal chain analysis to determine upper bounds on buffer sizes for multi-party sessions with asynchronous communication. Bouajjani and Emmi [5] 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.

Compared to all these results, we do not impose any bound on the number of peers, cycles, or buffer bounds. Another main difference is that we do not want to ensure or check (universal) boundedness of the systems under analysis. Contrarily, we are particularly interested in unbounded (yet possibly stable) systems. Existential boundedness in turn assumes structural hypothesis on models, *e.g.*, at most one sending transition and no mix of emissions/receptions outgoing from a same state in [23, 15], whereas we do not impose any restriction on our LTS models.

In [2], the authors rely on language equivalence and propose a result similar to the stability property introduced here (Theorem 1). However, they present this problem as decidable and propose a decision procedure for checking whether a system is stable. We have demonstrated here that this theorem is wrong. Since branching bisimulation is a particular case of language equivalence, hence testing stability is undecidable for language equivalence as well. Moreover, [2] uses LTL logic whereas we consider a finest notion of equivalence in this paper (branching), which allows one to check properties written with ACTL\X logic [34]. The tool support provided in [2] does not provide any result (infinite loop, inconclusive result, or error) for more than half of the examples presented in Table 1.

6 Conclusion

We have presented in this paper a framework for formally analyzing systems communicating via (possibly unbounded) FIFO buffers. This work focuses on cyclic behavioral models, namely Labeled Transition Systems. We have introduced the stability property, which shows that several systems become stable from a specific buffer bound k when focusing on send messages. The stability problem is undecidable in the general case, but for many systems we can determine whether those systems are stable using heuristics, search algorithms, and branching equivalence checking. Experiments showed that many real-world examples satisfy this property and this can be identified in a reasonable time. Model checking techniques can then be used on the asynchronous version of the system with buffers bound to the smallest k satisfying stability. If a stable system satisfies a specific property for that k, the property will be satisfied too if buffer bounds are increased or if buffers are unbounded.

As far as future work is concerned, a first perspective is to investigate whether our results stand or need to be adjusted for different communication models, *e.g.*, when each peer is equipped with one buffer per message type or when each couple of peers in a system is equipped with a specific communication buffer. Many properties on send messages can be formalized using temporal logic and verified using our approach. However, in some specific cases, one may also want to write properties on receive messages or on both send and receive messages. Thus, we plan to extend our results and define a notion of stability involving not only emissions but also receptions. A last perspective aims at identifying subclasses of systems preserving the stability property. Such a sufficient condition could be achieved by statically analyzing cycle dependencies.

References

1. 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.
2. S. Basu and T. Bultan. Automatic Verification of Interactions in Asynchronous Systems with Unbounded Buffers. In *Proc. of ASE'14*, pages 743–754, 2014.
3. S. Basu, T. Bultan, and M. Ouederni. Deciding Choreography Realizability. In *Proc. of POPL'12*, pages 191–202. ACM, 2012.
4. A. Bennaceur, C. Chilton, M. Isberner, and B. Jonsson. Automated Mediator Synthesis: Combining Behavioural and Ontological Reasoning. In *Proc. of SEFM'13*, volume 8137 of *LNCS*, pages 274–288. Springer, 2013.
5. 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.
6. A. Bracciali, A. Brogi, and C. Canal. A Formal Approach to Component Adaptation. *Journal of Software Systems*, 74(1):45–54, 2005.
7. D. Brand and P. Zafiropulo. On Communicating Finite-State Machines. *J. ACM*, 30(2):323–342, 1983.

8. A. Brogi and R. Popescu. Automated Generation of BPEL Adapters. In *Proc. of ICSOC'06*, volume 4294 of *LNCS*, pages 27–39. Springer, 2006.
9. T. Bultan, C. Ferguson, and X. Fu. A Tool for Choreography Analysis Using Collaboration Diagrams. In *Proc. of ICWS'09*, pages 856–863. IEEE, 2009.
10. J. Cámara, J. A. Martín, G. Salaün, C. Canal, and E. Pimentel. Semi-Automatic Specification of Behavioural Service Adaptation Contracts. *Electr. Notes Theor. Comput. Sci.*, 264(1):19–34, 2010.
11. C. Canal, P. Poizat, and G. Salaün. Synchronizing Behavioural Mismatch in Software Composition. In *Proc. of FMOODS'06*, volume 4037 of *LNCS*, pages 63–77. Springer, 2006.
12. C. Canal, P. Poizat, and G. Salaün. Model-Based Adaptation of Behavioural Mismatching Components. *IEEE Transactions on Software Engineering*, 34(4):546–563, 2008.
13. G. Cécé and A. Finkel. Verification of Programs with Half-duplex Communication. *Inf. Comput.*, 202(2):166–190, 2005.
14. J. Cubo, G. Salaün, C. Canal, E. Pimentel, and P. Poizat. A Model-Based Approach to the Verification and Adaptation of WF/.NET Components. In *Proc. of FACS'07*, volume 215 of *ENTCS*, pages 39–55, 2007.
15. P. Darondeau, B. Genest, P. S. Thiagarajan, and S. Yang. Quasi-static Scheduling of Communicating Tasks. *Inf. Comput.*, 208(10):1154–1168, 2010.
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. P.-M. Deniélou and N. Yoshida. Multiparty Session Types Meet Communicating Automata. In *Proc. of ESOP'12*, volume 7211 of *LNCS*, pages 194–213. Springer, 2012.
18. A. Finkel and P. McKenzie. Verifying Identical Communicating Processes is Undecidable. *Theor. Comput. Sci.*, 174(1-2):217–230, 1997.
19. W. Fokkink. *Introduction to Process Algebra.* Texts in Theoretical Computer Science. Springer, 2000.
20. X. Fu, T. Bultan, and J. Su. Analysis of Interacting BPEL Web Services. In *Proc. of WWW'04*, pages 621–630. ACM Press, 2004.
21. X. Fu, T. Bultan, and J. Su. Conversation Protocols: A Formalism for Specification and Verification of Reactive Electronic Services. *Theoretical Computer Science*, 328(1-2):19–37, 2004.
22. 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.
23. B. Genest, A. Muscholl, H. Seidl, and M. Zeitoun. Infinite-state High-level MSCs: Model-checking and Realizability. *J. Comput. Syst. Sci.*, 72(4):617–647, 2006.
24. C. Gierds, A. J. Mooij, and K. Wolf. Reducing Adapter Synthesis to Controller Synthesis. *IEEE T. Services Computing*, 5(1):72–85, 2012.
25. G. Gössler and G. Salaün. Realizability of Choreographies for Services Interacting Asynchronously. In *Proc. of FACS'11*, volume 7253 of *LNCS*, pages 151–167. Springer, 2011.
26. M. G. Gouda, E. G. Manning, and Y.-T. Yu. On the Progress of Communications between Two Finite State Machines. *Information and Control*, 63(3):200–216, 1984.
27. J. F. Groote and M. P. A. Sellink. Confluence for Process Verification. *Theor. Comput. Sci.*, 170(1-2):47–81, 1996.

28. 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.
29. T. Jéron and C. Jard. Testing for Unboundedness of FIFO Channels. *Theor. Comput. Sci.*, 113(1):93–117, 1993.
30. 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.
31. S. Leue, A. Stefanescu, and W. Wei. Dependency Analysis for Control Flow Cycles in Reactive Communicating Processes. In *Proc. of SPIN'08*, volume 5156 of *LNCS*, pages 176–195. Springer, 2008.
32. J. A. Martín and E. Pimentel. Contracts for Security Adaptation. *J. Log. Algebr. Program.*, 80(3-5):154–179, 2011.
33. R. Mateescu, P. Poizat, and G. Salaün. Adaptation of Service Protocols Using Process Algebra and On-the-Fly Reduction Techniques. In *Proc. of ICSOC'08*, volume 5364 of *LNCS*, pages 84–99. Springer, 2008.
34. R. D. Nicola and F. W. Vaandrager. Action versus State Based Logics for Transition Systems. In *Semantics of Concurrency*, volume 469 of *LNCS*, pages 407–419. Springer, 1990.
35. M. Ouederni, G. Salaün, and T. Bultan. Compatibility Checking for Asynchronously Communicating Software. In *Proc. of FACS'13*, volume 8348 of *LNCS*, pages 310–328. Springer, 2013.
36. P. Poizat and G. Salaün. Adaptation of Open Component-based Systems. In *Proc. of FMOODS'07*, volume 4468 of *LNCS*, pages 141–156. Springer, 2007.
37. R. Milner. *Communication and Concurrency*. Prentice-Hall, Inc., 1989.
38. G. Salaün, L. Bordeaux, and M. Schaerf. Describing and Reasoning on Web Services using Process Algebra. In *Proc. of ICWS'04*, pages 43–50. IEEE Computer Society, 2004.
39. G. Salaün, T. Bultan, and N. Roohi. Realizability of Choreographies Using Process Algebra Encodings. *IEEE Transactions on Services Computing*, 5(3):290–304, 2012.
40. R. Seguel, R. Eshuis, and P. W. P. J. Grefen. Generating Minimal Protocol Adaptors for Loosely Coupled Services. In *Proc. of ICWS'10*, pages 417–424. IEEE CS, 2010.
41. 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.
42. R. J. van Glabbeek and W. P. Weijland. Branching Time and Abstraction in Bisimulation Semantics. *J. ACM*, 43(3):555–600, 1996.