Automated Analysis of Asynchronously Communicating Systems

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

[1] University of Grenoble Alpes, Inria, LIG, CNRS, Grenoble, France
{lakhdar.akroun,gwen.salaun}@inria.fr
[2] LRI, University Paris-Sud,
CentraleSupélec, CNRS, Université Paris-Saclay, Orsay, France
lina.ye@lri.fr

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 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.

© Springer International Publishing Switzerland 2016
D. Bošnački and A. Wijs (Eds.): SPIN 2016, LNCS 9641, pp. 1–18, 2016.
DOI: 10.1007/978-3-319-32582-8_1

In this paper, we propose a new approach for analyzing a set of peers described using Labeled Transition Systems (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 LTSs. Transitions are labeled with either send actions (exclamation marks) or receive actions (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 Fig. 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.

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

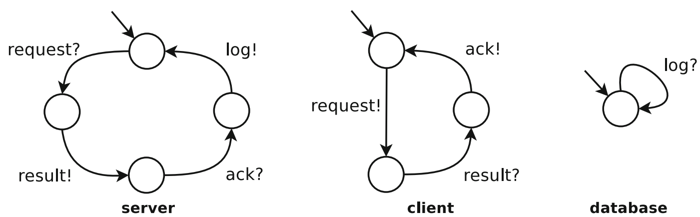

Fig. 1. Motivating example

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 majority of our examples.

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, Sect. 5 reviews related work and Sect. 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 send action has a counterpart (receive action) 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* $s' = (s'_1, Q'_1, \ldots s'_n, Q'_n) \in S_a$, *we have three possible behaviors*
  - $s \xrightarrow{m!} s' \in T_a$ *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)* $\forall k \in \{1, \ldots, n\} : k \neq i \Rightarrow s'_k = s_k$ *(send action)*
  - $s \xrightarrow{\tau} s' \in T_a$ *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)* $\forall k \in \{1, \ldots, n\} : k \neq i \Rightarrow s'_k = s_k$ *(receive action)*
  - $s \xrightarrow{\tau} s' \in T_a$ *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)* $\forall k \in \{1, \ldots, n\} : k \neq i \Rightarrow s'_k = s_k$ *(internal action)*

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 Definition 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 $(P_1|(Q_1^1| \ldots |Q_1^1))|...|(P_n|(Q_n^1| \ldots |Q_n^1))$, 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 [36] (see Sects. 1.2 and 3.3 of this book for details). Furthermore, we use $\overline{LTS_a}$ for the asynchronous composition where the receive actions are kept in the resulting LTS ($s \xrightarrow{m?} s' \in \overline{T_a}$, *receive action* rule in Definition 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 [41] (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 [33].

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 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:

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

A buffer of size k can be written as a parallel composition of k buffers of size 1 (see Sects. 1.2 and 3.3 in [36] for details), hence:

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

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

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

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

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

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* $ACTL\backslash X$ *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 send actions only. If one wants to check a property involving receive actions, a solution is to replace in the property a specific receive action by one of the send actions (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 Sect. 4).

To prove that testing the stability is an undecidable problem, we reduce the halting problem of a Turing machine to the test of stability of a set of peers communicating asynchronously. We start the proof with some preliminaries and notation, then we give an overview of the proof. Afterwards, we detail the construction of a system of two peers simulating the Turing machine and finally we prove the undecidability result.

Preliminaries and Notation. 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 and the word is not accepted at a state q, then M initiates a loop. This loop reads any symbol and moves to the right. Hence, if the word w is not accepted, then the machine executes an infinite loop by reading symbols and moving to the right. This looping behavior is not usual in classic Turing machines and acceptance semantics, but this simplifies the reduction without modifying the expressiveness of the Turing machine as shown in [18].

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).

Overview. To facilitate the understanding of the proof, we 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 $P_2^{'}$, and there exists k such that $LTS_a^{'k} \equiv_{br} LTS_a^{'}$ iff M does not halt on w, where $LTS_a^{'}$ is the asynchronous product of the system $\{P_1^{'}, P_2^{'}\}$.

Phase (i) – Construction of P_1 and P_2. 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 $uheadv\#$. We give in the following the construction of P_1 and P_2.

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 Fig. 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 Fig. 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 Fig. 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.

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.

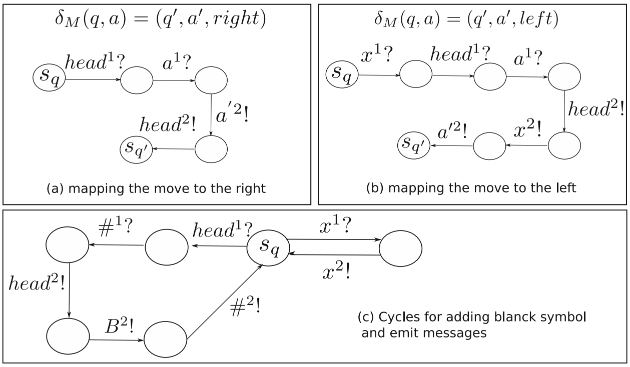

Fig. 2. Mapping the instructions of the machine M to transitions of the peer P_1

The peer P_2 is only used to read and re-send the messages. It 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} is 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 $w = a_1a_2...a_n$.
- $s_{univ} \xrightarrow{x^2?} s_1 \xrightarrow{x^1!} s_{univ} \in T_{P_2}$ where x is any symbol in $\Sigma^?_{P_2}$.

Lemma 1. *Given a Turing machine M with an input word w and the peers P_1 and P_2 constructed as above, the system composed of $\{P_1, P_2\}$ is stable whether the machine M halts on w or not.*

Proof. To prove that $\exists k$ such that $LTS^k_a \equiv_{br} LTS_a$, it is sufficient to prove that $\exists k$ such that $LTS^k_a \equiv_{br} LTS^{k+1}_a$ (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 that 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 (which occurs in our construction when the machine does not halt on the input word w). 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 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.

Now we detail the proof for cycles of type (1). The proof for cycles of type (2) is straightforward because those cycles involve receive actions only and do not make the buffer contents increase. Suppose that the machine M does not stop. Let s^k be the state of LTS_a^k representing the configuration of the machine M when starting to execute the infinite loop for the first time. 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. More precisely, the buffer of P_1 contains the following word: $head^1 \#^1 a_1^1 ... a_m^1$, 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-sends the symbol to P_1. Thus, in the configuration s^k, the first symbol is $head^1$. Then, P_1 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 contains 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 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: $s^{k''} \xrightarrow{\tau} s_1 \xrightarrow{a_m^1!} s'^k$. With our construction, any sequence of send messages which exceeds the buffer size k can be executed with a buffer size bounded by k. 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 encode the configuration of the machine M when starting to execute the infinite loop. ■

Phase (ii) – Construction of P_1' and P_2'. Until now, whenever the machine M halts on w or not, the system composed of P_1 and P_2 is always stable. Now, we extend P_1 and P_2 respectively to obtain P_1' and P_2' such that the machine M does not halt on w iff the corresponding system (composed of P_1' and P_2') is stable. This is achieved by adding to P_1 the transition system P_a to obtain P_1' and adding to P_2 the transition system P_b to obtain P_2', such that the system $\{P_a,\ P_b\}$ is not stable. The peers P_a and P_b are not formally defined, we can choose any two peers which are not stable. The 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 $\Sigma^?_{P_2}$.

Lemma 2. *Given a Turing machine M with an input word w and the peers P_1' and P_2' constructed as above, the system composed of $\{P_1', P_2'\}$ is stable iff the machine M does not halt on w.*

Proof. Suppose that the machine M halts on w, then P_1' reaches the state $s_{q_{halt}}$ (see the construction of P_1, which simulates the execution of the machine M) and it sends the message *halt* to P_2'. Hence, it 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.

Suppose now that M does not halt on w, then from the construction of P_1 simulating the execution of M, the peer P_1' never reaches the state $s_{q_{halt}}$, and the system executes an infinite loop. Hence, from Lemma 1, the system $\{P_1', P_2'\}$ is stable. ■

We can now formulate one of the main results of this paper, which asserts that testing the stability property in an undecidable problem.

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

Proof. The proof is a direct consequence of the construction given above and of both Lemmas 1 and 2. ■

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

Figure 3 overviews the main steps of our tool support. Given a set of peer LTSs, we first check as a preprocessing to our approach whether this system is branching synchronizable [34]. Synchronizability is checked comparing the synchronous composition with the 1-bounded asynchronous composition. If the system is synchronizable, the observable behavior for the synchronous and asynchronous composition always remains the same whatever buffer size is chosen. Therefore, the synchronous product can be used for analysis purposes. If the set of peers is not synchronizable, 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 using compilers, exploration tools, and equivalence checking tools available in CADP [22].

Heuristics and Search Algorithms. Each strategy consists of the computation of an initial bound k and an algorithm calculating the next bound to attempt.

- Strategy #1 starts from bound k equal to one and increment k one by one until obtaining a positive result for the equivalence check or reaching *kmax*.
- Strategy #2 computes the longest sequence of send actions in all peer LTSs, then starts from this number and uses a binary search algorithm. The intuition behind the longest sequence of send actions is that in that case all peers can

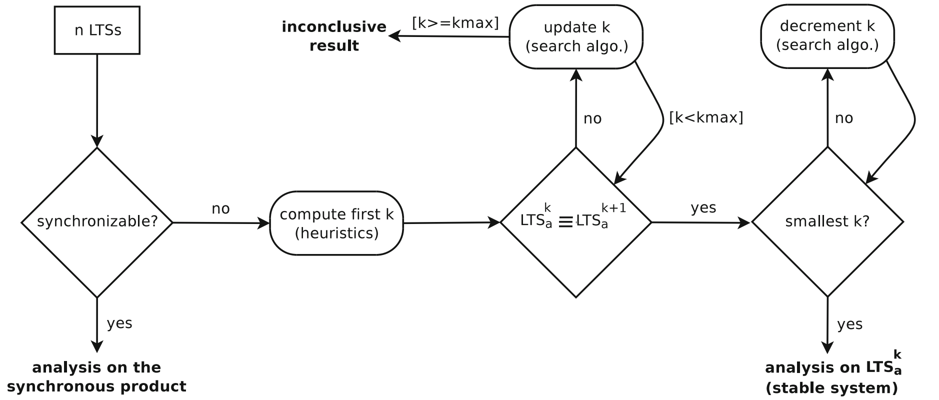

Fig. 3. Methodological aspects

at least send all their messages even if no peer consumes any message from its buffer.
- Strategy #3 uses again the longest sequence of send actions for the initial k, but then progresses by incrementing or decrementing the bound till reaching $kmax$ or the smallest k satisfying stability.
- Strategy #4 computes the maximum between the longest sequence of send actions in all peers and the highest number of send actions destinated to a same peer, and then uses the binary search algorithm (as for #2) for computing the next bounds.
- Strategy #5 uses the same initial k computation as presented for strategy #4, and then increments or decrements the bound till completion of the process as in strategy #3.

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 28 examples presented in the top part of Table 1, 23 can be analyzed using the approach proposed in this paper (10 are synchronizable and 13 are stable). 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 (29) to (35) show how LTSs and computation time grow mainly with the number of peers.

Table 1. Experimental results

Id	Description	$\|P\|$	$\|S\|/\|T\|$	k	LTS_a^k $\|S\|/\|T\|$	Time (in seconds)				
						#1	#2	#3	#4	#5
(1)	Estelle specification [28]	2	7/9	$kmax$	707/1,751	280	134	302	214	276
(2)	News server [34]	2	9/9	3	14/22	89	180	65	173	85
(3)	Client/server [7]	2	6/10	0	3/4	34				
(4)	CFSM system [28]	2	6/7	$kmax$	393/802	222	107	213	103	212
(5)	Promela program (1) [29]	2	6/6	1	3/4	52	71	67	68	66
(6)	Promela program (2) [30]	2	8/8	$kmax$	275/616	219	107	231	103	228
(7)	Figure 1	3	8/8	1	5/6	87	208	146	208	145
(8)	Web services [20]	3	13/12	0	7/7	44				
(9)	Trade system [17]	3	12/12	0	30/46	44				
(10)	Online stock broker [21]	3	13/16	$kmax$	197/452	>1 h	222	>1 h	223	>1 h
(11)	FTP transfer [6]	3	20/17	2	15/19	91	224	155	215	155
(12)	Client/server [11]	3	14/13	0	8/7	44				
(13)	Mars explorer [8]	3	34/34	2	21/25	93	176	142	170	140
(14)	Online computer sale [14]	3	26/26	0	11/12	69				
(15)	E-museum [12]	3	33/40	3	27/46	146	>1 h	138	243	182
(16)	Client/supplier [10]	3	31/33	0	17/19	44				
(17)	Restaurant service [40]	3	15/16	1	10/12	68				
(18)	Travel agency [39]	3	32/38	0	18/21	44				
(19)	Vending machine [24]	3	15/14	0	8/8	44				
(20)	Travel agency [4]	3	42/57	3	29/42	118	>1 h	113	>1 h	112
(21)	Train station [38]	4	18/18	2	19/26	114	195	137	197	165
(22)	Factory job manager [9]	4	20/20	0	12/15	54				
(23)	Bug report repository [25]	4	12/12	1	7/8	85	221	137	227	136
(24)	Cloud application [27]	4	8/10	$kmax$	26,754/83,200	352	208	339	208	337
(25)	Sanitary agency [37]	4	35/41	3	44/71	144	196	137	196	137
(26)	SQL server [35]	4	32/38	2	22/31	165	195	137	199	170
(27)	SSH protocol [31]	4	26/28	0	16/18	97				
(28)	Booking system [32]	5	45/53	1	27/35	179	285	165	>1 h	>1 h
(29)	Hand-crafted example	5	396/801	4	17,376/86,345	227	>1 h	184	313	189
(30)	——	6	16/18	5	202/559	278	641	188	641	188
(31)	——	7	38/38	6	1,716/6,468	363	763	391	767	393
(32)	——	10	48/47	8	14,904/57,600	624	800	294	804	294
(33)	——	14	85/80	4	19,840/113,520	506	1,449	483	1,442	485
(34)	——	16	106/102	3	22,400/132,400	478	1,620	454	1,621	453
(35)	——	20	128/116	4	80,640/522,480	728	2,194	698	2,183	699

Strategies #2 and #4 are less efficient than the others in terms of performance because binary search may take time before converging to the result and may return high values for k, which implies calculating asynchronous systems with larger state spaces. In contrast, the advantage of binary search is that for non-stable systems, k increases quite fast and quickly reaches $kmax$, see rows (1) and (4) for instance. Strategies #3 and #5 are better than the others in most cases. This gain is not clear for small examples and examples requiring a small buffer bound, but it becomes obvious for examples involving more peers and for those requiring larger buffer bounds, see the hand-crafted examples in Table 1.

Let us focus again on the example presented in Fig. 1 (row (7) in Table 1) in order to illustrate how our approach works in practice using strategy #1. First, we compute the synchronous composition and the 1-bounded asynchronous composition, both are shown in Fig. 4. We can see that these two systems are not equivalent (*i.e.*, not synchronizable) because in the asynchronous composition the client can submit a second request before the server sends its log file to the database. Therefore, we compute the 2-bounded asynchronous composition, which is equivalent to the 1-bounded asynchronous composition. This means that the system is stable from bound 1 and can be analyzed using model checking techniques for that bound and, if the properties are satisfied for that bound, they will be satisfied as well for upper bounds. Note that only send actions are preserved in the asynchronous compositions for comparison purposes. The 1-bounded composition with send and receive actions consists of 16 states and 24 transitions.

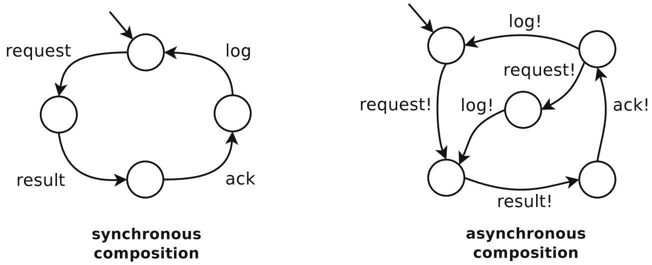

Fig. 4. Synchronous (left) and 1-bounded asynchronous (right) compositions

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 [28] 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 [29], 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 send/receive actions outgoing from a same state in [15,23], 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. However, they present this problem as decidable and propose a decision procedure for checking whether a system is stable. We have demonstrated here that the stability problem is undecidable. Since branching bisimulation is a particular case of language equivalence, 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 [33]. 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 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 send actions but also receive actions. 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.

Acknowledgments. We would like to sincerely thank the anonymous reviewers for their helpful comments on this paper. This work has been supported by the OpenCloudware project (2012-2015), which is funded by the French *Fonds national pour la Société Numérique* (FSN), and is supported by *Pôles* Minalogic, Systematic, and SCS.

References

1. Abdulla, P.A., Bouajjani, A., Jonsson, B.: On-the-fly analysis of systems with unbounded, lossy FIFO channels. In: Vardi, M.Y. (ed.) CAV 1998. LNCS, vol. 1427, pp. 305–318. Springer, Heidelberg (1998)
2. Basu, S., Bultan, T.: Automatic verification of interactions in asynchronous systems with unbounded buffers. In: Proceedings of ASE 2014, pp. 743–754 (2014)
3. Basu, S., Bultan, T., Ouederni, M.: Deciding choreography realizability. In: Proceedings of the POPL 2012, pp. 191–202. ACM (2012)
4. Bennaceur, A., Chilton, C., Isberner, M., Jonsson, B.: Automated mediator synthesis: combining behavioural and ontological reasoning. In: Hierons, R.M., Merayo, M.G., Bravetti, M. (eds.) SEFM 2013. LNCS, vol. 8137, pp. 274–288. Springer, Heidelberg (2013)
5. Bouajjani, A., Emmi, M.: Bounded phase analysis of message-passing programs. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 451–465. Springer, Heidelberg (2012)
6. Bracciali, A., Brogi, A., Canal, C.: A formal approach to component adaptation. J. Softw. Syst. **74**(1), 45–54 (2005)
7. Brand, D., Zafiropulo, P.: On communicating finite-state machines. J. ACM **30**(2), 323–342 (1983)
8. Brogi, A., Popescu, R.: Automated generation of BPEL adapters. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, pp. 27–39. Springer, Heidelberg (2006)
9. Bultan, T., Ferguson, C., Fu, X.: A tool for choreography analysis using collaboration diagrams. In: Proceedings of the ICWS 2009, pp. 856–863. IEEE (2009)
10. Cámara, J., Martín, J.A., Salaün, G., Canal, C., Pimentel, E.: Semi-automatic specification of behavioural service adaptation contracts. Electr. Notes Theor. Comput. Sci. **264**(1), 19–34 (2010)
11. Canal, C., Poizat, P., Salaün, G.: Synchronizing behavioural mismatch in software composition. In: Gorrieri, R., Wehrheim, H. (eds.) FMOODS 2006. LNCS, vol. 4037, pp. 63–77. Springer, Heidelberg (2006)
12. Canal, C., Poizat, P., Salaün, G.: Model-based adaptation of behavioural mismatching components. IEEE Trans. Softw. Eng. **34**(4), 546–563 (2008)

13. Cécé, G., Finkel, A.: Verification of programs with half-duplex communication. Inf. Comput. **202**(2), 166–190 (2005)
14. Cubo, J., Salaün, G., Canal, C., Pimentel, E., Poizat, P.: A model-based approach to the verification and adaptation of WF/.NET components. In: Proceedings of the FACS 2007, vol. 215 of ENTCS, pp. 39–55 (2007)
15. Darondeau, P., Genest, B., Thiagarajan, P.S., Yang, S.: Quasi-static scheduling of communicating tasks. Inf. Comput. **208**(10), 1154–1168 (2010)
16. Deniélou, P.-M., Yoshida, N.: Buffered communication analysis in distributed multiparty sessions. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 343–357. Springer, Heidelberg (2010)
17. Deniélou, P.-M., Yoshida, N.: Multiparty session types meet communicating automata. In: Seidl, H. (ed.) Programming Languages and Systems. LNCS, vol. 7211, pp. 194–213. Springer, Heidelberg (2012)
18. Finkel, A., McKenzie, P.: Verifying identical communicating processes is undecidable. Theor. Comput. Sci. **174**(1–2), 217–230 (1997)
19. Fokkink, W.: Introduction to Process Algebra. Texts in Theoretical Computer Science, 1st edn. Springer, Heidelberg (2000)
20. X. Fu, T. Bultan, and J. Su. Analysis of Interacting BPEL Web Services. In Proc. of WWW'04, pp. 621–630. ACM Press, (2004)
21. Fu, X., Bultan, T., Su, J.: Conversation protocols: a formalism for specification and verification of reactive electronic services. Theoret. Comput. Sci. **328**(1–2), 19–37 (2004)
22. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2010: a toolbox for the construction and analysis of distributed processes. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 372–387. Springer, Heidelberg (2011)
23. Genest, B., Muscholl, A., Seidl, H., Zeitoun, M.: Infinite-state high-level MSCs: model-checking and realizability. J. Comput. Syst. Sci. **72**(4), 617–647 (2006)
24. Gierds, C., Mooij, A.J., Wolf, K.: Reducing adapter synthesis to controller synthesis. IEEE Trans. Serv. Comput. **5**(1), 72–85 (2012)
25. Gössler, G., Salaün, G.: Realizability of choreographies for services interacting asynchronously. In: Arbab, F., Ölveczky, P.C. (eds.) FACS 2011. LNCS, vol. 7253, pp. 151–167. Springer, Heidelberg (2012)
26. Gouda, M.G., Manning, E.G., Yu, Y.-T.: On the progress of communications between two finite state machines. Inf. Control **63**(3), 200–216 (1984)
27. Güdemann, M., Salaün, G., Ouederni, M.: Counterexample guided synthesis of monitors for realizability enforcement. In: Chakraborty, S., Mukund, M. (eds.) ATVA 2012. LNCS, vol. 7561, pp. 238–253. Springer, Heidelberg (2012)
28. Jéron, T., Jard, C.: Testing for unboundedness of FIFO channels. Theor. Comput. Sci. **113**(1), 93–117 (1993)
29. Leue, S., Mayr, R., Wei, W.: A scalable incomplete test for message buffer overflow in promela models. In: Graf, S., Mounier, L. (eds.) SPIN 2004. LNCS, vol. 2989, pp. 216–233. Springer, Heidelberg (2004)
30. Leue, S., Ştefănescu, A., Wei, W.: Dependency analysis for control flow cycles in reactive communicating processes. In: Havelund, K., Majumdar, R. (eds.) SPIN 2008. LNCS, vol. 5156, pp. 176–195. Springer, Heidelberg (2008)
31. Martín, J.A., Pimentel, E.: Contracts for security adaptation. J. Log. Algebraic Program. **80**(3–5), 154–179 (2011)
32. Mateescu, R., Poizat, P., Salaün, G.: Adaptation of service protocols using process algebra and on-the-fly reduction techniques. In: Bouguettaya, A., Krueger, I., Margaria, T. (eds.) ICSOC 2008. LNCS, vol. 5364, pp. 84–99. Springer, Heidelberg (2008)

33. Nicola, R.D., Vaandrager, F.W.: Action versus state based logics for transition systems. In: Guessarian, Irène (ed.) LITP 1990. LNCS, vol. 469, pp. 407–419. Springer, Heidelberg (1990)
34. Ouederni, M., Salaün, G., Bultan, T.: Compatibility checking for asynchronously communicating software. In: Fiadeiro, J.L., Liu, Z., Xue, J. (eds.) FACS 2013. LNCS, vol. 8348, pp. 310–328. Springer, Heidelberg (2014)
35. Poizat, P., Salaün, G.: Adaptation of open component-based systems. In: Bonsangue, M.M., Johnsen, E.B. (eds.) FMOODS 2007. LNCS, vol. 4468, pp. 141–156. Springer, Heidelberg (2007)
36. Milner, R.: Communication and Concurrency. Prentice-Hall Inc, Upper Saddle River (1989)
37. Salaün, G., Bordeaux, L., Schaerf, M.: Describing and reasoning on web services using process algebra. In: Proceedings of the ICWS 2004, pp. 43–50. IEEE Computer Society (2004)
38. Salaün, G., Bultan, T., Roohi, N.: Realizability of choreographies using process algebra encodings. IEEE Trans. Serv. Comput. **5**(3), 290–304 (2012)
39. Seguel, R., Eshuis, R., Grefen, P.W.P.J.: Generating minimal protocol adaptors for loosely coupled services. In: Proceedings of the ICWS 2010, pp. 417–424. IEEE CS (2010)
40. van der Aalst, W.M.P., Mooij, A.J., Stahl, C., Wolf, K.: Service interaction: patterns, formalization, and analysis. In: Bernardo, M., Padovani, L., Zavattaro, G. (eds.) SFM 2009. LNCS, vol. 5569, pp. 42–88. Springer, Heidelberg (2009)
41. van Glabbeek, R.J., Weijland, W.P.: Branching time and abstraction in bisimulation semantics. J. ACM **43**(3), 555–600 (1996)