Compatibility Control of Asynchronous Communicating Systems with Unbounded Buffers

D. Dahmani[1], M.C. Boukala[1], H. Mountassir[2], and S. Chouali[2]

[1] MOVEP, Comp. Sci. Dept, USTHB, Algiers.
dzaouche,mboukala@usthb.dz,
[2] Femto-ST/DISC, Comp. Sci. Dept, Burgundy, Franche-Comté University
hmountassir,schouali@femto-st.fr

Abstract. The composition of heterogeneous software components is required in many domains to build complex systems. However, such compositions raise mismatches between components such as unspecified messages. Checking compatibility for asynchronously communicating systems with unbounded channels is undecidable. In this paper, we propose a compatibility control approach based on a coverability product, which is a finite abstraction of the asynchronous products of I/O-transition systems. This coverability product is used to check UR-compatibility, without requiring a synchronizabity of the peers. We distinguish between messages brought by acyclic path and those brought by cycles. This allows us to overcome over-approximation for some arcs. Furthermore, we define relationships, called patterns, to check a good choreography between peers in terms of emission and reception activities.

Keywords: communicating systems, compatibility analysis, coverability product, patterns, infinite systems.

1 Introduction

Component-based development aims at facilitating the construction of very complex applications by reusing and composing existing components. The assembly of components offers a great potential for reducing cost and time to build complex software systems and improving system maintainability and flexibility. A software component is generally developed independently and is assembled with other components, which have been designed separately, to create complex systems. Normally glue code is written to realize such an assembly. Compatibility checking is used to control if composed components can work together *without errors*. This verification is very important for ensuring correct interaction between components and may be done by using different formal models such as interface automata, I/O-transition systems, π-calcul, Petri nets and state machines [1,8,9,12].

There are several incompatibility notions: (1) *label incompatibility* concerns messages exchanged between components which can be handled differently. For instance, messages are named differently, methods are called with parameters which don't match perfectly (incompatible types, different orders, ...) [1]. (2) The *bidirectional notion* requires that when a component sends a message, there is another component which eventually receives it, and when a component is waiting to receive a message, there is another which must send that message. (3) The *unspecified reception* (UR-compatibility) is less restrictive than the *bidirectional notion*, since the reception of messages expected is not required for *unspecified reception* notion. The UR-compatibility requires that the composition of components doesn't contain any deadlock, i.e. starting from their initial states, all components can either evolve or terminate if they are in final states and their buffers are empty [1, 6].

To deal with incompatible components, pessimistic and optimistic approaches have been proposed. The pessimistic approach considers that two components are compatible if they can always work together in any environment while, in the optimistic approach, two components are compatible if they can be used together in some helpful environment [1].

Components can interact synchronously or asynchronously. In synchronous communications, a send action is a lock-step, since it is allowed only when the receiver is ready to perform the corresponding reception. Thus, send and receive actions are performed simultaneously. In asynchronous communications, a send action is not a lock-step. The messages sent are added to the receiver buffers. Such buffers may be ordered or unordered. Analyzing asynchronous communicating components with unbounded buffers is a complex task, undecidable in general [10, 13], since it may lead to an infinite state space. Several works aiming at analyzing such systems bound the size of buffers or the number of iterations per cycles. Bounding such parameters is not a good solution since new unexpected errors can occur when the values of parameters change or exceed the fixed bounds. Thus, the main problem of these approaches is to choose the appropriate bounds to study the compatibility of the infinite communicating systems. In this paper, we propose a compatibilty control approach, based on a coverability product, for asynchronous communicating components with *unordered* and *unbounded* buffers.

2 Related works

Brand and Zafiropulo are among the first to have proposed works in the area of service compatibility checking [6]. They use communicating finite state machines to model interacting processes executed in parallel and exchanging messages via FIFO buffers. Their works focus on unspecified receptions compatibility notion for interaction protocols. When considering unbounded buffers, the authors show that the resulting state spaces may be infinite, and the problem becomes undecidable.

The works in [7, 8] use Petri net models to treat incompatibility problems. Some works rely on extensions of Petri nets, like open nets to model communicating processes, assuming asynchronous communication over non-ordered message buffers. As far as asynchronous semantics is considered, compatibility analysis has been proven to be undecidable for unbounded open nets. The authors deal only with limited-communications which give bounded open nets. In [12] authors use the so-called state equation, which is based on the standard linear programming in Petri nets, to find a necessary but not sufficient condition for compatibility control.

Recently, Bouajjani and Emmi [5] consider a bounded analysis of asynchronous message-passing programs with ordered message queues. Their approach does not limit the number of communicating processes nor the buffers' size. However, the *number of iterations of communication cycles* is limited. Despite the potential for huge exploration of unbounded process contexts, the proposed bounding scheme gives rise to a simple and efficient program analysis by reduction to sequential programs.

In [13], an approach on checking the compatibility of peers communicating asynchronously by message exchange over unbounded buffers is proposed. The approach requires that peers are synchronisable. In this case, the synchronous system behaves like the asynchronous one for any buffer size. Thus, the compatibility check on the asynchronous version of the system is reduced to the synchronous version, which is finite and decidable. However, the approach cannot conclude anything about non synchronizable peers. In [11], authors focus on the verification of weak asynchronous compatibility relying on half-duplex systems instead of synchronizability and provide a decidable criterion that ensures weak asynchronous compatibility.

3 Motivation

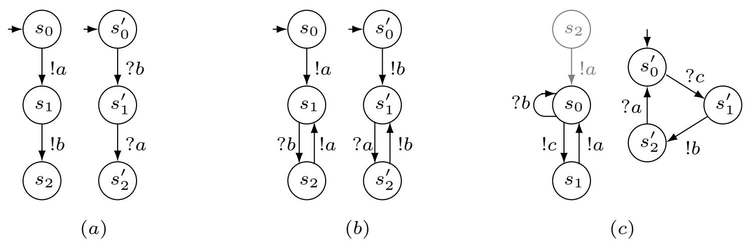

Fig. 1: Asynchronous communicating peers

Figure 1.a shows a simple example which highlights the relevance of unordered queues in a context of asynchronous communication. Peer 1 sends message a followed by b, but peer 2 consumes the messages in a reverse order of

their emission. The peers are compatible with unordered queues but not with FIFO queues. In Figure 1.b, peer 1 (resp. peer 2) sends message a (resp. b) and loops infinitely by consuming b (resp. a) and then sending a (resp. b). All works based on synchronizability property [2, 3, 13] cannot conclude anything about the compatibility of such a system, since peers are not synchronizable. However, they are free of deadlock and any message sent is consumed. Consider Figure 1.c and suppose that s_0 is the initial state of the left peer. This latter can either loop on the emission of c followed by a, or the reception of b. Peer 2 can indefinitely receive c, send b and then receive a. Despite the perfect coherence between the production and consumption patterns of these peers, there is no conclusion about their compatibility based on synchronizability property.
In this paper, we propose an approach to control the compatibility of asynchronous communicating peers without requiring the synchronizability property. Our solution is mainly based on the construction of a *coverability asynchronous product* which is always finite. Some analysis techniques are also proposed to overcome the over-approximation of a coverability asynchronous product and check the coherence of peers's production and consumption patterns in their cyclic stages.

4 Coverability product

4.1 I/O-transition systems

First we give the definition of an I/O-transition system.

Definition 1 *An I/O-transition system A is a 4-tuple $(S_A, S_A^{init}, \Sigma_A, \tau_A)$ such that:*

1. *S_A is a finite set of states.*
2. *$S_A^{init} \subset S_A$ a set of initial states.*
3. *$\Sigma_A = \Sigma_A^I \cup \Sigma_A^O \cup \Sigma^H$, where Σ_A^I , Σ_A^O and Σ_A^H are finite disjoint sets of input, output and internal actions.*
4. *$\tau_A \subseteq S_A \times \Sigma_A \times S_A$ is a set of steps.*

In an I/O-transition system, each input (resp. output, internal) action is preceded by the symbol ? (resp. !, ;). A finite execution σ in an I/O-transition system is an alternating sequence of states and actions $s_0 \xrightarrow{a_1} s_1 \ldots \xrightarrow{a_n} s_n$ such that $s_i \xrightarrow{a_{i+1}} s_{i+1} \in \tau_A$. For the sake of clarity, we suppose that S_A^{init} is reduced to a single state.

4.2 Coverability product construction

The asynchronous product of I/O-transition systems with unbounded buffers may be infinite. In this paper, we propose to build a *finite* coverability product, inspired from the standard approach of constructing a coverability graph for P/T nets [4]. The coverability graph covers all reachable states where ω, a specific symbol, is used to represent reachable states with unbounded messages.

In our approach, we distinguish between the message occurrences brought by elementary paths and those brought by cycles. The former are explicitly represented in the product, while the latter are abstracted by a set Ω. Thus, Ω gives the set of unbounded messages. In the sequel, we confuse between terms message and action.

Let $A_1 = (S_{A_1}, s_{0A_1}, \Sigma_{A_1}, \tau_1)$ and $A_2 = (S_{A_2}, s_{0A_2}, \Sigma_{A_2}, \tau_{A_2})$ be two I/O-transition systems. Automata A_1 and A_2 fulfils the following conditions: $\Sigma^I_{A_1} \cap \Sigma^O_{A_2} = \emptyset$ and $\Sigma^O_{A_1} \cap \Sigma^I_{A_2} = \emptyset$. Let $\Sigma = \Sigma_{A_1} \cup \Sigma_{A_2}$ be the set of actions of A_1 and A_2.

Definition 2 *The coverability product of A_1 and A_2, denoted by $A_1 \otimes A_2$, is a 7-tuple $(V, \Pi_1, \Pi_2, \tau, M, \Omega, v_0)$. V is the set of nodes, with $v_0 \in V$ being the root. Π_i (with $i = 1,2$) is a function from V to S_{A_i}. $\tau \subseteq V \times \Sigma \times V$ is the set of labelled arrows. M is a function from $V \times \Sigma$ to $\mathbb{N}$, assigning an ordinary marking to every node. Ω is a function from V to 2^{Σ} giving, for a node v, the set of its infinite actions (messages).*

A step of $A_1 \otimes A_2$ is either a step of A_1 or A_2. The following definitions are useful to build $A_1 \otimes A_2$.

Definition 3 *(Enabling condition) A state s of S_{A_i}, for $i \in \{1, 2\}$, enables a node v of $A_1 \otimes A_2$ iff $\exists s \xrightarrow{\delta a} s' \in \tau_{A_i}$ and $\Pi_i(v)$=s and either:*

- *$\delta \in \{! , ;\}$,*
- *or $\delta =? \Rightarrow (a \in \Omega(v) \vee M(v)(a) > 0\ )$.*

Definition 4 *Let u and v be two nodes of $A_1 \otimes A_2$. Two nodes u and v are equal, denoted by $u = v$, iff $\Pi_1(u) = \Pi_1(v)$, $\Pi_2(u) = \Pi_2(v)$, $\forall a \in \Sigma$, $M(u)(a) = M(v)(a)$ and $\Omega(u) = \Omega(v)$.*

Definition 5 *Let u and v be two nodes of $A_1 \otimes A_2$. Node u is less than v , denoted by $u < v$, iff $\Pi_1(u) = \Pi_2(v)$, $\Pi_2(u) = \Pi_2(v)$, $\exists a \in \Sigma$, $M(u)(a) < M(v)(a)$, $\forall b \in \Sigma$, $(M(u)(b) \leq M(v)(b)$ or $b \in \Omega(u))$ and $\Omega(u) \subseteq \Omega(v)$.*

Given two I/O-transition systems A_1 and A_2, the algorithm 1 constructs the coverability product $A_1 \otimes A_2$. For the sake of clarity we present the product for two I/O-transition systems. However, the algorithm can be easily extended to n I/O-transition systems.

Algorithm 1: Coverability graph construction

```
   Data: Two I/O-transition systems $A_1 = (S_{A_1}, s_{0A_1}, \Sigma_{A_1}, \tau_1)$ and
         $A_2 = (S_{A_2}, s_{0A_2}, \Sigma_{A_2}, \tau_{A_2})$
   Result: Coverability Graph $G = (V, \Pi_1, \Pi_2, \tau, M, \Omega, v_0)$
1  $v_0 \leftarrow$ new node ;
2  $V \leftarrow \{v_0\}$ ;
3  $\Pi_1(v_0) = s_{0A_1}; \Pi_2(v_0) = s_{0A_2}$ ;
4  $\forall a \in \Sigma$, $M(v_0)(a)=0$ and $\Omega(v_0) = \emptyset$ ;
5  $\tau \leftarrow \emptyset$ ;
6  *unprocessed* $\leftarrow \{v_0\}$;
7  **while** *unprocessed* $\neq \emptyset$ **do**
8      $v \leftarrow$ element of *unprocessed*;
9      **foreach** $s \xrightarrow{\delta a} s'$ *of* $\tau_{A_1}$*, s.t. state s enables node v* **do**
10         $v' \leftarrow$ new node ; $\Pi_1(v') = s'$; $\Pi_2(v') = \Pi_2(v)$ ; $\Omega(v') = \Omega(v)$ ;
11         **if** $\delta =!$ **then** $M(v')(a) = M(v)(a) + 1$ ;
12         **if** $\delta =? \wedge M(v)(a) > 0$ **then** $M(v')(a)=M(v)(a) - 1$ ;
13         **if** $\delta =? \wedge M(v)(a) = 0 \wedge a \in \Omega(v)$ **then** $M(v')(a)=M(v)(a)$ ;
14         **if** $\delta =;$ **then** $M(v')(a)=M(v)(a)$ ;
           /* $x \rightsquigarrow_\tau y$ means a path from $x$ to $y$ with arcs in $\tau$.    */
15         **if** $\exists u \rightsquigarrow_\tau v$*, with* $u < v'$ **then**
16             **foreach** $a \in \Sigma$ **do**
17                 **if** $M(u)(a) < M(v')(a)$ **then**
18                     $M(v')(a) = M(u)(a)$ ;
19                     $\Omega(v') = \Omega(v') \cup \{a\}$ ;
20                 **end**
21             **end**
22         **end**
23         **if** $\nexists w \in V$ ***s.t.*** $w = v'$ **then**
24             $V \leftarrow V \cup \{v'\}$ ;
25             *unprocessed* $\leftarrow$ *unprocessed* $\cup \{v'\}$ ;
26             $\tau \leftarrow \tau \cup \{v \overset{\delta a}{\rightarrow} v'\}$ ;
27         **else**
28             **select** $w \in V$ **s.t.** $w = v'$ ;
29             $\tau \leftarrow \tau \cup \{v \overset{\delta a}{\rightarrow} w\}$
30         **end**
31     **end**
       /* This loop is repeated for each edge $s \xrightarrow{\delta a} s'$ of $\tau_{A_2}$.    */
32     *unprocessed* $\leftarrow$ *unprocessed* $\setminus \{v\}$ ;
33 **end**
```

4.3 Reduction of I/O-transition systems

The interleaving between some actions of A_1 and A_2 are not necessary for the properties we target (dealock and UR-reception). They only increase the size of $A_1 \otimes A_2$. For instance, from state s_0 of Figure 1.c, message c is sent and then message a. These actions can be done by the left peer without any interaction with the right peer. Hence, they can be replaced by a single and an atomic action $!ca$. Furthermore, the path $s_0 \xrightarrow{!c} s_1 \xrightarrow{!a} s_0$ is reduced to $s_0 \xrightarrow{!ca} s_0$. This transition produces one occurrence of message a and another of b. We can also reduce sequences of internal or/and emission actions. However, the reduction of a sequence $s_j? \xrightarrow{\delta_0 a_0} \ldots \xrightarrow{\delta_k a_{k-1}} s_{j+k}$ of A_i, $i \in \{1,2\}$, is not always possible. Conditions required to reduce a path are given below:

1. **No interaction**: $\delta_0, \ldots \delta_{k-1} \in \{!,;\}$.
2. **No intermediary branching states**: $\forall j < h < j+k$, s_h is not a branching state.
3. **Occurrence in an elementary cycle**: Either all the arcs of the sequence are in an elementary cycle or all are not.

The first point states that there is no interaction between the peer and its correspondent, whereas the two other points allow to preserve the behavior of the peer.

Example 1 *Consider again the I/O-transition systems depicted in Figure 1.c and suppose that s_2 is the initial state of the left peer. Only the left automaton can be reduced by replacing the edges $s_0 \xrightarrow{!c} s_1$ and $s_1 \xrightarrow{!a} s_0$ by $s_0 \xrightarrow{!ca} s_0$. The coverability product, constructed by Algorithm 1, is shown in Figure 2. Unbounded actions of a node v, $\Omega(v)$ are given between brackets. The occurrences of actions brought by elementary path are explicitly represented. In the reachable state $(s_0,s'_0,a,\{c,a\})$, say u, the current state $\Pi_1(u)$ of the first (resp. $\Pi_2(u)$ of the second) I/O-transition system is s_0 (resp. s'_0). The marking M of u is reduced to one occurrence of a ($M(u)(a)$=1, $M(u)(b)$=0 and $M(u)(c)$=0) and $\Omega(u) = \{c,a\}$ meaning that the occurrences a and c are potentially infinite.*

Non-travial scc are bordred by dashed boxes and $C1$ is terminal scc. Some arcs are represented by gray lines to highlight the over-approximation notion,-which will be introduced in the following section.

Notation 1 *A strongly connected component (scc) C of a graph G is a maximal set of nodes $C \subseteq S_G$, where S_G is the set of nodes of graph G, such that for every pair of nodes u and v in C, there is a directed path from u to v and a directed path from v to u. A scc C composed of one node is said trivial, otherwise non trivial. A path or cycle in G is called elementary if no node occurs more than once. A cycle is called simple if no edge occurs more than once. A scc is terminal if it has no outgoing edge. Let σ be a sequence of transitions or an elementary cycle of $A_1 \otimes A_2$. $\#(\sigma,!a)$ (resp. $\#(\sigma,?a)$) gives the number of occurrences $!a$ (resp. $?a$) in σ and $\#(\sigma,a)$ represents $\#(\sigma,!a) - \#(\sigma,?a)$.*

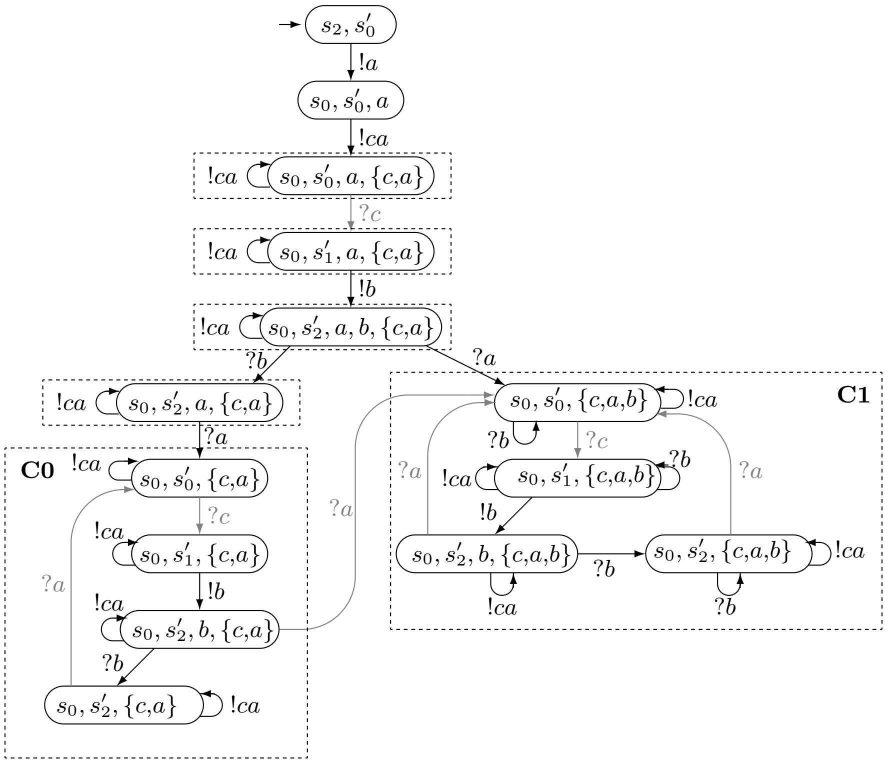

Fig. 2: Coverability product for peers of Figure 1.c

Definition 6 *The set of actions which may* ***accumulate*** *in the nodes of a cycle μ, denoted by $\Psi(\mu,!)$, corresponds to $\{a \in \Sigma |\ \#(\mu,a) > 0\}$. A cycle μ may consume accumulated actions, those in $\Psi(\mu,?) = \{a \in \Sigma | \#(\mu,a) < 0\}$.*

In Figure 2, each cycle μ labelled by $!ca$ is an infinite producer of c and a ($\#(\mu,c) = 1$ and $\#(\mu,a) = 1$). Similarly, for instance, cycle $\mu = ?c\ !b\ ?b\ ?a$ consumes actions c and a which are not produced by μ ($\#(\mu,c) = -1$ and $\#(\mu,a) = -1$) but accumulated by cycles labelled by $!ca$.

4.4 Over-approximation of coverability product

We give in Figure 3.b a part of the coverability product of peers in 3.a. In node v_1, peer A_2 does not block, since peer A_1, at state s_0, is able to provide messages a. Now consider the sequence $v_0 \xrightarrow{!a} v_1 \xrightarrow{!a} v_1 \xrightarrow{?a} v_2 \xrightarrow{!b} v_3 \xrightarrow{?b} v_4$. At the end of this sequence, peers A_1 and A_2 are in states s_1 and s'_2, respectively. Observe that A_2 waits indefinitely message a which will never be provided by peer A_1. Indeed, at state s_1, peer A_1 cannot produce message a. Hence, node v_4, contray to v_1, is a potential deadlock for peer A_2.

Both edges $v_1 \xrightarrow{?a} v_2$ and $v_4 \xrightarrow{?a} v_5$ are over-approximated. However, the first is always possible, whereas the second is only possible for some executions. An over-approximated edge $v \xrightarrow{?a} v'$ deserves special attention, since, for some executions, node v' is *never reached*. Hence, their presence makes $A_1 \otimes A_2$ not reliable. A loop $v \xrightarrow{?a} v$ is not considered as over-approximated even if $M(v)(a) = 0$.

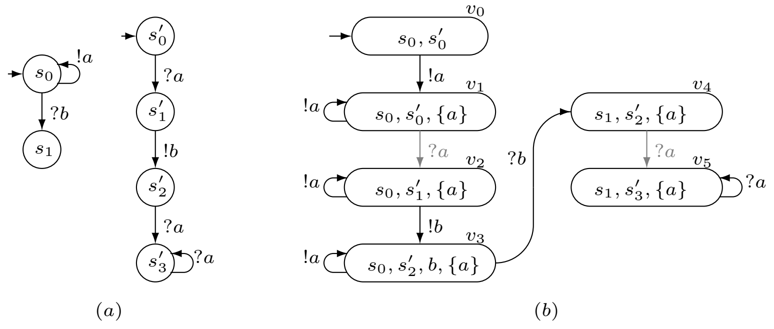

Fig. 3: Over-approximation

In this section, we propose a technique aiming at discarding gradually the over-approximated status for some edges. Algorithm 2 summarises this technique. Its input is a strongly connected component C of $A_1 \otimes A_2$. The algorithm returns *false* whenever some edges remain over-approximated, otherwise it returns *true*. The algorithm uses the sets *simpleCycles(C)* and *reliableCycles(C,v)*, with v a node of C. The former contains all simple cycles of C and the second is a subset of *simpleCycles*(C) where each cycle contains node v and is composed of non over-approximated edges only.

The algorithm builds a set of nodes called V_{over}. A node of V_{over} is an initial extremity of at least one over-approximated edge. The algorithm is iterative. At each step, V_{over} is computed. The over-approximated status of each edge $v \xrightarrow{?a} v'$, with $v \in V_{over}$ is discarded whenever there is a reliable cycle $\mu \in$*reliableCycles(C,v)* producing indefinitely action a, i.e. $a \in \Psi(\mu,!)$. At the next step, V_{over} is updated, since some edges turn into non over-approximated. Hence, new cycles become reliable.

The termination of the algorithm is ensured since there is a finite number of simple cycles, actions and edges. The algorithm is applied to the strongly connected components of $A_1 \otimes A_2$. It is obvious that if the product contains no over-approximated edges, then it is reliable and can be used to check the free of deadlock and UR-compatibility properties.
Consider again the coverability product of Figure 2. Applying the precedent algorithm to non trivial scc ($C1$) induces the following steps.

1. Gray edges (over-approximated) of $C1$ compose E_{over}.

Algorithm 2: dealOverapproxation

```
Data: a strongly connected components $C$ of $A_1 \otimes A_2$
Result: $boolean$
1  $new = true$;
2  $E_{over} = \{v \xrightarrow{?a} v'$ an edge of $C$ s.t. $M(v)(a) = 0$ and $v \neq v'\}$ ;
   /* $E_{over}$ contains the set of over-approximated edges */
3  $\forall \mu \in simpleCycles(C)$, $\mu.visited = false$ ;
4  while $new$ do
5      $V_{over} = \{v\ a\ node\ C | \exists v \xrightarrow{?a} v' \in E_{over}\}$;
6      $new = false$ ;
7      foreach $v \in V_{over}$ do
8          $actions = \{a \in \Sigma | a \in \Psi(\mu,!) \wedge \mu \in reliableCycles(C,v) \wedge \neg \mu.visited\}$ ;
9          foreach $\mu \in reliableCycles(C,v)$ do
10             $\mu.visited = true$
11         end
12         foreach over-approximated edge $e = v \xrightarrow{?a} v'$ do
13             if $a \in actions$ then
14                 $E_{over} = E_{over} \setminus \{e\}$ ;
15                 $new = true$;
16             end
17         end
18     end
19 end
20 if $E_{over} \neq \emptyset$ then return $false$ else return $true$;
```

2. In the first iteration, V_{over} contains the three nodes of $C1$ which are initial extremities of gray arcs. A reliable simple cycle, labelled by $!ca$, loops around each node of V_{over}. Thus, the set of actions $\{c,a\}$ is associated to each node of V_{over}. All gray edges are then removed from E_{over}.
3. In the second iteration, V_{over} is empty and there is no change about edge status, inducing the termination of the algorithm. The algorithm returns true.

Likewise, the algorithm 2 is applied to the other scc of $A_1 \otimes A_2$ depicted in Figure 2. We can easily see that the coverability product depicted in Figure 2 is free of deadlock.

5 UR compatibility verification

In this section, we focus on the widely notion unspecified receptions UR. A set of peers is UR-compatible if they do not deadlock and each sent message by a peer is received by another one. Consider the peers given in Figure 1.c and consider s_0 as initial state of the left peer. It is obvious that there are a good *choreography* and *proper* interactions between the peers, since the right peer requires an equality between messages a and c, a relation satisfied by the left peer of the Figure 1.c. However, when we add a cycle, for instance, $s_0 \xrightarrow{!a} s_0$ for the left peer, the equality relation between a and c is lost inducing a mess in the consumption activity. In this paper, we propose an approach to determine relationships between unbounded actions and use such relationships to check the UR compatibility through $A_1 \otimes A_2$.

5.1 Pattern of a strongly connected component

In this paper, we suppose that a turn of any cycle may produce (or consume) **at most** one occurrence of a given action. In other terms, $-1 \leq \#(\mu,a) \leq 1$ for any action a of any cycle μ of $A_1 \otimes A_2$.

Consider a non-trivial *scc* C of a **free of deadlock** coverability product. It is worth noting that, by construction, the set of unbounded actions is the same for any node of C; we use $\Omega(C)$ to denote such a set. The relationships between unbounded actions within C consists to partition set $\Omega(C)$ to $\mathcal{P}(C) = \{P_1, \ldots, P_n\}$ such that for any node v of C, any cycle $v \xrightarrow{\mu} v$ and any part $P_j \in \mathcal{P}(C)$, the accumulated actions of $\Omega(C)$ verify the following equality: $a, b \in P_j \Leftrightarrow \#(\mu,a) = \#(\mu, b)$. $\mathcal{P}(C)$ is called the pattern of C.

We inductively build $\mathcal{P}(C)$, by considering the **elementary** cycles of C. Initially, the partition associated with C is empty. At step i, a new partition is computed according to the partition in the previous step $i-1$ by considering a new *elementary* cycle μ of C. Cycle μ may alter the relationships between some actions. In other terms, it may change the relationships between the part's elements of the partition computed in step $i-1$, since it brings some actions and consumes other actions, with the **same rhythm**. Thus, taking into account the restriction presented at the beginning of this section, each part P of the old partition is split into three subsets E, R and N.

- A subset E gathering actions of P produced by μ. Each action of E wins one occurrence for each turn of μ.
- A subset R gathering actions of P consumed by μ. Each action of R loses one occurrence for each turn of μ.
- A subset N gathering actions of P which are not changed by μ.

Furthermore, the set $E \subset \Psi(\mu, !)$ (resp. $R \subset \Psi(\mu, ?)$) whose actions don't belong to any part of the old partition is added to the partition being computed. E (resp. R) constitutes a fresh equality relationship.

Example 2 *Consider a partition $\mathcal{P}_i(C) = \{\{a,b,c,d,e\},\{f,g\}\}$, obtained at the i^{th} iteration, and a new elementary cycle $\mu = !a\ !b\ ?b\ ?c\ !f\ ?e\ !h\ ?i?j$, so:*
- $\Psi(\mu,!) = \{a,f,h\}$, $\Psi(\mu,?) = \{c,e,i,j\}$,

Algorithm 3: Pattern computation

```
    Data: a strongly connected components $C$ of $A_1 \otimes A_2$
    Result: Pattern of $C$
 1  $partition = \{\emptyset\}$;
 2  foreach Elementary cycle $\mu$ of $C$ do
 3      $NewPartition = \emptyset$;
 4      foreach Part $P$ of partition do
 5          $E = \{a \in P \mid \#(\mu,a) = 1\}$;
 6          $R = \{a \in P \mid \#(\mu,a) = -1\}$;
 7          $N = \{a \in P \mid \#(\mu,a) = 0\}$;
 8          $NewPartition = NewPartition \cup \{E, R, N\}$;
 9      end
10      $E = \{a \in \Psi(\mu,!) \mid \nexists P \in partition \wedge a \in P\}$;
11      $R = \{a \in \Psi(\mu,?) \mid \nexists P \in partition \wedge a \in P\}$;
12      $NewPartition = NewPartition \cup \{E, R\}$;
13      partition$=NewPartition$;
14  end
15  return $partition$;
```

- Partition obtained at the next iteration $\mathcal{P}_{i+1}(C) = \{\{a\}, \{b,d\}, \{c,e\}, \{f\}, \{g\}, \{h\}, \{i,j\}\}$.

Part $\{a,b,c,d,e\}$ is split into three subsets, $\{a\}, \{b,d\}, \{c,e\}$ since $\#(\mu,a) = 1$, $\#(\mu,b) = \#(\mu,d) = 0$ and $\#(\mu,c) = \#(\mu,e) = -1$, idem for the part $\{f,g\}$. Cycle μ brings new actions, $h \in \Psi(\mu,!)$ and $i, j \in \Psi(\mu,?)$, inducing two parts $\{h\}$, $\{i,j\}$ in $\mathcal{P}_{i+1}(C)$. It is worth noting that cycle μ has broken the equality of a with the other actions of its part in $\mathcal{P}_i(C)$, keeps an equality between b and d (resp. c and e) and so on.

5.2 Choreography Checking

The pattern of a scc must C also take into account the elementary cycles of the ascendants scc of C. It is worth noting that the action occurrences, brought by elementary paths, are explicitly represented in the coverability product and are not included in the computation of the patterns of the scc. Consider again the coverability product of Figure 2, the pattern of scc $C0$ (resp. $C1$), computed in isolation is $\{\{c, a\}\}$ (resp. $\{\{c, a\},\{b\}\}$). The pattern of $C0$ is preserved when considering the ascendant scc, idem for $C1$.

Definition 7 *There is a choreography compatibility within a scc C of a coverability product if for any elementary cycle μ of C, $\Psi(\mu,?) \in \mathcal{P}(C)$.*

Consider an element $E \in \mathcal{P}(C)$. By construction, there is an equality relationship, in some nodes of $A_1 \otimes A_2$, between occurrences of actions of E. Any elementary cycle μ such that $\Psi(\mu,?) = E$ consumes, at each turn, a same number of elements of E preserving the relationship. Thus, the comsumption is *in a step with* the production rythm.

Example 3 *Consider the coverability graph of Figure 2. The pattern of its unique terminal Scc is $\mathcal{P}(C1) = \{\{c{,}a\}, \{b\}\}$. There is a choreography compatibility within $C1$, since cycles of $C1$ fulfill definition 7. For instance, $\Psi(\mu{,}?) = \{a{,}c\} \in \mathcal{P}(C1)$ for cycle $\mu = ?c\ !b\ ?a \in C1$, idem for $\Psi(\mu{,}?) = \{b\} \in \mathcal{P}(C1)$ for cycle $\mu = ?b \in C1$.*

5.3 UR compatibility Checking

The UR compatibility requires that each message sent is eventually received. To ensure that all sent messages are received, we must have in all terminal strongly connected components, cycles able to consume accumulated actions. These cycles are garbage collectors.

Definition 8 *A cycle μ of $A_1 \otimes A_2$ is a garbage collector iff: $\Psi(\mu{,}?) \neq \emptyset$ and $\Psi(\mu{,}!) = \emptyset$.*

A garbage collector is able to clean the actions $\Psi(\mu{,}?)$. In the other hand, the cycle does not produce residual messages. Consider again the coverability product of Figure 2. The cycle labelled by $?b$ is a garbage collector of action b, whereas the cycle labelled by $?c!b?b?a$ is a garbage collector of actions a and b.

Proposition 1 *Two automata A_1 and A_2 are UR-compatible if the coverability product $A_1 \otimes A_2$ is free of deadlock and for any terminal scc C of $A_1 \otimes A_2$ the following conditions hold:*

- *There is a choreography compatibility within C.*
- *For any node v of C and any action a of $\Omega(v)$, there is a garbage collector of a.*
- *$\forall a \in \Sigma$, $\exists$ a node v of C, such that $M(v)(a) = 0$.*

Proof. The first point states that there are proper interactions between peers, while the second ensures that peers are able to clean the accumulated actions. Finally, the last point indicates that the peer's buffers always reach a state where any action brought by an elementary path has been emptied.

The coverability product of Figure 2 holds the conditions of the previous proposition. Thus, the peers of example 1 are UR-compatible.

6 Conclusion

In this paper, we presented a new approach to check compatibility of asynchronous communicating infinite systems, using unbounded and unordered buffers. We do not have any restrictions on the number of cycle iterations, size of buffers nor on number of components. Our main result is that our approach does not require any synchronizability property. In our approach, we proposed a coverability product by constructing a *finite* state space. We define the concept of

patterns associated with messages occurring in the cycles of a strongly connected component. These patterns are jointly used with the covering graph to check the UR-compatibility of the components. The patterns allow to make an underlying analysis of strongly connected components. Thus, they show if there is a good choreography between components.

We have implemented our approach, which is under experimentation as future work, we aim to consider systems with FIFO unbounded buffers. This work could be a promising base to tackle with infinite systems.

References

1. L. Alfaro and T.A. Henzinger. Interface automata. In *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering (FSE), ACM*, pages 109–120. Press, 2001.
2. Samik Basu and Tevfik Bultan. On deciding synchronizability for asynchronously communicating systems. *Theor. Comput. Sci.*, 656:60–75, 2016.
3. Samik Basu, Tevfik Bultan, and Meriem Ouederni. Deciding choreography realizability. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 191–202, 2012.
4. Michael Blondin, Alain Finkel, Christoph Haase, and Serge Haddad. Approaching the coverability problem continuously. In *TACAS 2016, Held as Part ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, pages 480–496, 2016.
5. Ahmed Bouajjani and Michael Emmi. Bounded phase analysis of message-passing programs. *STTT*, 16(2):127–146, 2014.
6. Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *J. ACM*, 30(2):323–342, April 1983.
7. C. Canal, P. Poizat, and G. Salaun. Model-based adaptation of behavioral mismatching components. *IEEE Transactions on Software Engineering*, 34(4):546–563, 2008.
8. Djaouida Dahmani, Mohand Cherif Boukala, and Hassen Mountassir. A Petri net approach for reusing and adapting components with atomic and non-atomic synchronisation. In *International Workshop on Petri Nets and Software*, Tunis, Tunisia, Juin 2014.
9. Djaouida Dahmani, Mohand Cherif Boukala, and Hassen Mountassir. Reusing and adapting components using atomic and non-atomic strong synchronisations. In *Conférence francophone sur l'Architecture Logicielle, CAL'2014*, Paris, France, 10-11 Juin 2014.
10. Serge Haddad, Rolf Hennicker, and Mikael H. Møller. Channel properties of asynchronously composed petri nets. In *Application and Theory of Petri Nets and Concurrency - 34th International Conference, PETRI NETS 2013, Milan, Italy, June 24-28, 2013. Proceedings*, pages 369–388, 2013.
11. Rolf Hennicker, Michel Bidoit, and Thanh-Son Dang. On synchronous and asynchronous compatibility of communicating components. In *Coordination Models and Languages - 18th IFIP WG 6.1 International Conference, COORDINATION 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings*, pages 138–156, 2016.

12. Olivia Oanea and Karsten Wolf. An efficient necessary condition for compatibility. In Oliver Kopp and Niels Lohmann, editors, *ZEUS*, volume 438 of *CEUR Workshop Proceedings*, pages 81–87. CEUR-WS.org, 2009.
13. Meriem Ouederni, Gwen Salaün, and Tevfik Bultan. Compatibility checking for asynchronously communicating software. In *Formal Aspects of Component Software - 10th International Symposium, FACS 2013, Nanchang, China, October 27-29, 2013, Revised Selected Papers*, pages 310–328, 2013.