Compositional state space generation with partial order reductions for Asynchronous Communicating Systems

Jean-Pierre Krimm and Laurent Mounier

VERIMAG* – Centre Equation – 2, avenue de Vignate – F-38610 Gières
Jean-Pierre.Krimm@imag,fr, Laurent.Mounier@imag.fr

Abstract. Compositional generation is an incremental technique for generating a reduced labelled transition system representing the behaviour of a set of communicating processes. In particular, since intermediate reductions can be performed after each generation step, the size of the LTS can be kept small and state-explosion can be avoided in many cases. This paper deals with compositional generation in presence of asynchronous communications via shared buffers. More precisely, we show how partial-order reduction techniques can be used in this context to define equivalence relations: that preserve useful properties, are congruence w.r.t asynchronous composition, and rely on a (syntactic) notion of preorder on execution sequences characterizing their "executability" in any buffer environment. Two such equivalences are proposed, together with dedicated asynchronous composition operators able to directly produce reduced LTS.

1 Introduction

This work takes place in the context of *formal verification* of distributed programs, those purpose is to evaluate a set of expected requirements on a formal program description. To automate this activity, one of the promising technique is the well-known *model-checking* approach, which consists of performing the verification on an explicit model of the system behaviour (e.g., a labelled transition system, or LTS). However, the main drawback of model-checking is the model explosion occurring when dealing with complex systems. This still limits its large scale utilisation in the industry.

Several interesting solutions have already been investigated to overcome this problem, for instance by avoiding an explicit storage of the whole model ("on-the-fly" techniques), or by processing it using efficient representations ("symbolic" techniques), or by generating a model simpler than the initial one ("abstraction" techniques). A particular instance of this latter solution consists of performing the verification not on the LTS S obtained from the original program description, but rather on its S/R *quotient* where R is an equivalence relation preserving the

* VERIMAG is a joint laboratory of CNRS, UJF and INPG

properties under verification. The main difficulty is then to get this quotient without generating first the initial LTS.

When the program under consideration is described by a *composition expression* between communicating LTS, and provided that R is a congruence with respect to the operators of this expression, the quotient S/R can easily be generated with a so-called *compositional approach*: it consists of (repeatedly) generating the LTS S' associated with a given sub-expression, and replacing this sub-expression in the initial one by the quotient S'/R. This approach has been widely studied [GS90,CK93,Val96,KM97], and has already been applied in some succesfull case studies. However, most of this works was done in the context of *synchronous* communicating systems (described for instance using process algebras like LOTOS [ISO87] or CSP [Hoa85]).

In this paper we propose a way to efficiently extend this compositional generation strategy to *asynchronous* systems communicating by message exchange through shared buffers. In fact, this communication scheme is very suitable for describing distributed systems or communication protocols, and it is the underlying model of popular specification formalisms such as the international standard SDL [IT92], or the PROMELA language [Hol91].

One of the main difficulties encountered during a compositional generation is to correctly handle the effect of the environment (i.e., the rest of the system) in order to restrict the generation of a given subset of components (otherwise the model obtained for this subset may be larger than the one corresponding to the whole system). This problem was addressed in [GS90,CK93,KM97] by expressing the constraints provided from the environment in terms of *process interfaces*, allowing to "cut off" some parts of a component behaviour. Unfortunately, this solution is not applicable in case of asynchronous communications, since the effects of the external buffers cannot precisely be statically approximated. Thus, many useless interleavings are computed when generating a subsystem independently of its buffer environment.

To avoid these interleavings, the solution we propose relies on the (well-known) *partial order* approach which consists of identifying *independent* execution sequences that can be safely sequentialized (instead of being fully interleaved). Such techniques have already rather intensively been studied, and their efficiency has been established in practice, in particular for asynchronous communicating systems [Val90,GW91,Pel96,KLM$^+$98]. However, to our knowledge, their application in this framework is original by its combination of two aspects:

- First, partial order reductions are usually performed on the whole system, considering the explicit behaviour of each of its components. *A contrario*, the approach we describe here can be applied on a partial sub-system, and it allows generation of a reduced LTS (with less interleavings) that can be re-used during further compositions.
- Second, the reductions we consider are not only based on a symmetrical *independence* relation of actions (leading to an equivalence relation between independent execution sequences), but also on an asymmetrical notion of *precedence* relation of actions, leading to a preorder between execution se-

quences. According to this preorder, smallest sequences are always "*more executable*" than larger ones in any buffer environment.

This notion of non commutative independence relation between actions was first introduced by [Lip75] to study the correctness of concurrents processes synchronized by means of semaphores. It was also used in [AJKP98] within a symbolic verification framework.

The paper is organized as follows:
First, we give in section 2 the program syntax we consider (a set of asynchronous communicating processes), and we briefly explain how the LTS denoting a program semantics can be compositionally generated in this framework. Then, we introduce in section 3 a (syntactic) notion of preorder between execution sequences, and we show how it characterizes the executability of an execution sequence in any buffer environment. Using this preorder, we consider in section 4 a first equivalence relation $\approx_\delta$, deadlock preserving, and which is a congruence w.r.t asynchronous composition. We then propose a new asynchronous composition operator, allowing to directly compute a reduced LTS w.r.t $\approx_\delta$ and thus avoiding many useless interleavings. Finally, in section 5 we extend these results to a stronger equivalence relation $\approx_o$, able to preserve the language w.r.t a set of observable actions.

2 Asynchronous communicating systems

In this section we give the abstract syntax and semantics used to represent asynchronous communicating systems by means of a parallel composition of labelled transition systems. Then we indicate how the global state space of such systems can be obtained in a compositional way.

2.1 Program syntax and semantics

A Labelled Transition System (LTS, for short) is a tuple $S = (Q, A, T, q_0)$ where Q is a finite set of (reachable) states, A a finite set of actions (or labels), $T \subseteq Q \times A \times Q$ a transition relation, and $q_0 \in Q$ the initial state of S. As usual, we shall note $p \xrightarrow{a}_T q$ instead of $(p, a, q) \in T$.

Let $\mathcal{M}$ be a set of *message* names, and Buf a set of *unbounded buffers* over $\mathcal{M}$. A buffer $B \in Buf$ is an abstract type with the following signature, and those concrete implementation depends on the exact nature of the buffer (e.g., bags, stacks, fifo queues, ...):

- $\perp$ is an empty buffer;
- **first**: $\mathcal{M} \times B \to \mathbf{bool}$.
  first(m, B) is true iff message m can be consumed in buffer B.
- **remove**: $\mathcal{M} \times B \to B$.
  When **first**(m, B) holds **remove**(m, B) returns the new buffer obtained from B by eliminating message m, otherwise B is returned unchanged.
- **append**: $\mathcal{M} \times B \to B$.
  append(m, B) adds the message m to buffer B.

Finally, a *program* is a couple $\mathcal{P} = (\mathcal{S}_n, \mathcal{B}_p)$ where $\mathcal{S}_n = \{S_1, S_2, \ldots S_n\}$ is a finite set of *elementary processes* represented by LTS $S_i = (Q_i, A_i, T_i, q_{0i})$, and $\mathcal{B}_p = \{B_1, B_2, \ldots B_p\}$ is a finite set of buffers over $\mathcal{M}$. Moreover, for each S_j in $\mathcal{S}_n$, action sets $A_j \subseteq \mathcal{A} = \mathcal{A}^+ \cup \mathcal{A}^- \cup \{\tau\}$ where:

$$\mathcal{A}^+ = \{+(i,m) \mid i \in [1,p] \ \wedge \ m \in \mathcal{M}\} \ \ ; \ \ \mathcal{A}^- = \{-(i,m) \mid i \in [1,p] \ \wedge \ m \in \mathcal{M}\}$$

Informally, for an LTS S_j, action $+a = +(i,m)$ denotes the output of message m to the buffer B_i, action $-a = -(i,m)$ denotes the input of message m from buffer B_i and τ denotes any internal (non communication) action.

Definition 1 (Program semantics). *The semantics of a program* $\mathcal{P} = (\mathcal{S}_n, \mathcal{B}_p)$ *is defined as the* LTS $\mathbf{sem}(\mathcal{P}) = (Q, \mathcal{A}, T, q_0)$ *where:*

- $Q \subseteq Q_1 \times Q_2 \times \cdots \times Q_n \times B_1 \times B_2 \times \cdots \times B_p$
- $q_0 = (q_{0_1}, q_{0_2}, \ldots, q_{0_n}, \perp, \ldots \perp)$
- *Q and T are the smallest sets obtained when applying the following rules:*

$$q_0 \ \in \ Q \qquad \text{[R0]}$$

$$\frac{p = (p_1, \ldots, p_j, \ldots p_n, B_1, \ldots, B_i, \ldots B_p) \in Q, \ p_j \overset{-(i,m)}{\longrightarrow}_{T_j} q_j, \ \mathbf{first}(m, B_i)}{q = (p_1, \ldots, q_j, \ldots p_n, B_1, \ldots, \mathbf{remove}(m, B_i), \ldots B_p) \in Q, \ p \overset{-(i,m)}{\longrightarrow}_{T} q} \qquad \text{[R1]}$$

$$\frac{p = (p_1, \ldots, p_j, \ldots p_n, B_1, \ldots, B_i, \ldots B_p) \in Q, \ p_j \overset{+(i,m)}{\longrightarrow}_{T_j} q_j}{q = (p_1, \ldots, q_j, \ldots p_n, B_1, \ldots, \mathbf{append}(m, B_i), \ldots B_p) \in Q, \ p \overset{+(i,m)}{\longrightarrow}_{T} q} \qquad \text{[R2]}$$

$$\frac{p = (p_1, \ldots, p_j, \ldots p_n, B_1, \ldots, B_i, \ldots B_p) \in Q, \ p_j \overset{\tau}{\longrightarrow}_{T_j} q_j}{q = (p_1, \ldots, q_j, \ldots p_n, B_1, \ldots, B_i, \ldots B_p) \in Q, \ p \overset{\tau}{\longrightarrow}_{T} q} \qquad \text{[R3]}$$

2.2 Compositional state space generation

The generation of $\mathbf{sem}(\mathcal{P})$ using definition 1, needs to consider *simultaneously* the whole sets of buffers and elementary processes. However, this resulting LTS can also be built in a more compositional way by taking into account each program component (i.e., buffer or elementary process) incrementally. To this purpose we first introduce two auxiliary operators, the *asynchronous product* between LTS and the execution of an LTS within a given *buffer environment*.

The asynchronous product ($||$) between two LTS $S_i = (Q_i, A_i, T_i, q_{0i})$ is defined in the usual manner: $S_1 \mid\mid S_2$ is the LTS $S = (Q, A, T, q_0)$ where $Q = Q_1 \times Q_2$, $T = \{((p_1, p_2), a, (q_1, q_2)) \mid (p_1 \overset{a}{\longrightarrow}_{T_1} q_1 \ \wedge \ p_2 = q_2) \ \vee \ (p_2 \overset{a}{\longrightarrow}_{T_2} q_2 \ \wedge \ p_1 = q_1)\}$, $A = A_1 \cup A_2$, $q_0 = (q_{01}, q_{02})$.

Definition 2 (Execution of an LTS **within a buffer environment).** *For an* LTS $S = (Q, A, T, q_0)$ *and a buffer environment* $\mathcal{B}_p$*, we note* $S[\mathcal{B}_p]$ *the* LTS (Q_s, A, T_s, q_{s_0}) *obtained by executing* S *within* $\mathcal{B}_p$*, and defined as follows:*

- $Q_s \subseteq Q \times B_1 \times B_2 \times \cdots \times B_p$

– $q_{s_0} = (q_0, \bot, \ldots, \bot)$
– *Q_s and T_s are the smallest sets obtained when applying the following rules:*

$$q_{s_0} \in Q_s \qquad \textit{[R0]}$$

$$\frac{p_s = (p, B_1, \ldots, B_i, \ldots B_p) \in Q_s,\ p \xrightarrow{-(i,m)}_T q,\ \mathbf{first}(m, B_i)}{q_s = (q, B_1, \ldots, \mathbf{remove}(m, B_i), \ldots B_p) \in Q_s,\ p_s \xrightarrow{-(i,m)}_{T_s} q_s} \qquad \textit{[R1]}$$

$$\frac{p_s = (p, B_1, \ldots, B_i, \ldots B_p) \in Q_s,\ p \xrightarrow{+(i,m)}_T q}{q_s = (q, B_1, \ldots, \mathbf{append}(m, B_i), \ldots B_p) \in Q_s,\ p_s \xrightarrow{+(i,m)}_{T_s} q_s} \qquad \textit{[R2]}$$

$$\frac{p_s = (p, B_1, \ldots, B_i, \ldots B_p) \in Q_s,\ p \xrightarrow{\tau}_T q}{q_s = (q, B_1, \ldots, B_i, \ldots B_p) \in Q_s,\ p_s \xrightarrow{\tau}_{T_s} q_s} \qquad \textit{[R3]}$$

It is easy to show that the global LTS $\mathbf{sem}(\mathcal{P})$ can be obtained by considering first the asynchronous product of its elementary processes, then executing it w.r.t its buffer environment:

Proposition 1. *For a program $\mathcal{P} = (\mathcal{S}_n, \mathcal{B}_p)$, $\mathbf{sem}(\mathcal{P}) = (S_1 \mid\mid S_2 \mid\mid \ldots \mid\mid S_n)[\mathcal{B}_p]$.*

Furthermore, this approach can be made even more flexible by partially distributing buffers $\mathcal{B}_p$ w.r.t a subset of elementary processes. More formally:

Proposition 2. *Let S_1 and S_2 be two* LTS *and $\mathcal{B}_p$ a buffer environment.*

Consider a split of $\mathcal{B}_p$ into three sets $\mathcal{B}_{p1}$, $\mathcal{B}_{p2}$ and $\mathcal{B}_{p3}$ such that: buffers of $\mathcal{B}_{p1}$ are not accessed *by S_2, buffers of $\mathcal{B}_{p2}$ are* not accessed *by S_1, and buffers of $\mathcal{B}_{p3}$ are accessed by both S_1 and S_2. (such a split always exists since $\mathcal{B}_{p1}$ and $\mathcal{B}_{p2}$ can be empty).*

Then, the following holds: $(S_1 \mid\mid S_2)[\mathcal{B}_p] = (S_1[\mathcal{B}_{p1}] \mid\mid S_2[\mathcal{B}_{p2}])[\mathcal{B}_{p3}]$

Finally, depending on the program properties under consideration, intermediate LTS reductions can now be introduced between successive generation steps. Furthermore, since internal communications within a sub-system can be abstracted away *before* its composition with the other program components, powerful reduction operations are possible when only the external program behaviour is relevant. In particular most of the usual bisimulation based weak equivalence relations (such as observational equivalence [Mil89], branching bisimulation [vGW89] or safety equivalence [BFG+91]) happen to be congruences w.r.t. operators $\mid\mid$ and [...] and can be used in this framework.

However, due to asynchronous nature of communications, this (straightforward) compositional approach may still suffer from state explosion problems. In fact, when generating a subsystem, each **append** or **remove** operations concerning external buffers is considered as fully asynchronous. This leads to many possible interleavings, and, therefore, the size of the resulting intermediate LTS may become very large.

We propose in this paper a solution to decrease the number of these useless interleavings by taking advantage of some (well-known) considerations about the concurrent execution of *independent* actions.

3 Equivalence and preorder on execution sequences

First, we give some notations related to the execution sequences of an LTS. Then we introduce some equivalence and preorder relations between execution sequences.

Definition 3 (Execution sequences of an LTS).
Let $S = (Q, A, T, q_0)$ be an LTS, *and p a given set of Q:*

- *$Act(p)$ is the set of actions the state p can perform, and $Pre(p)$ the set of actions that may reach it:*

$$Act(p) = \{a \in \mathcal{A} \mid \exists q \,.\, p \xrightarrow{a}_T q\} \quad ; \quad Pre(p) = \{a \in \mathcal{A} \mid \exists q \,.\, q \xrightarrow{a}_T p\}$$
$$Act^-(p) = Act(p) \cap \mathcal{A}^- \quad ; \quad Act^+(p) = Act(p) \cap \mathcal{A}^+$$

- *An (*execution*)* sequence *σ from p is an element $\sigma = a_1.a_2.\cdots a_n$ of A^* such that: $\sigma = p \xrightarrow{a_1}_T p_1 \xrightarrow{a_2}_T \cdots \xrightarrow{a_n}_T p_n$ We shall also use the notation $p \xrightarrow{\sigma}_T p_{n+1}$, or simply $p \xrightarrow{\sigma}_T$.*

3.1 Equivalence between execution sequences

The equivalence relation between execution sequences we consider is based on an *independency* relation I on actions. Roughly speaking, two actions a_1 and a_2 will be considered as independent ($(a_1, a_2) \in I$) if, whenever they are both enabled in a given state p, their execution order has no influence on the subsequent execution sequences p will be able to perform.

Definition 4 (Independance of actions).
A relation $I \subseteq \mathcal{A} \times \mathcal{A}$ is an independency *relation for an* LTS $S = (Q, A, T, q_0)$ *if, for all $p \in Q$, and for all $(a_1, a_2) \in I$ then:*

$$a_1, a_2 \subseteq Act(p) \Rightarrow \begin{cases} \forall q_1, q_2 \in Q \,.\, p \xrightarrow{a_1}_T q_1 \ \wedge\ p \xrightarrow{a_2}_T q_2 \\ \qquad\qquad\qquad \Rightarrow\ (a_2 \in Act(q_1) \ \wedge\ a_1 \in Act(q_2)) \\ \wedge \\ \forall q \in Q \,.\, p \xrightarrow{a_1.a_2}_T q \ \Leftrightarrow\ p \xrightarrow{a_2.a_1}_T q \end{cases}$$

We give below some examples of independency relations defined on communication actions performed by *distinct* processes, depending on the kind of buffers that are considered.

Example 1. When buffers are defined as *bags*, the order of two **append** operations does not matter. Therefore, two **append** (resp. **remove**) operations are always independent each others. Moreover, an **append** and a **remove** operation will be independent if they occur in two different bags. Therefore, I_{bag} is defined as follows: $I_{bag} = \mathcal{A}^+ \times \mathcal{A}^+ \cup \mathcal{A}^- \times \mathcal{A}^- \cup \{(\pm(i_1, m_1), \mp(i_2, m_2)) \mid i_1 \neq i_2\}$ When buffers are defined as fifo queues, the order of two **append** or **remove** operations does not matter only if they occur in different queues. The corresponding independency relation is then:

$$I_{fifo} = \{(\pm(i_1, m_1), \pm(i_2, m_2)) \mid i_1 \neq i_2\} \cup \{(\pm(i_1, m_1), \mp(i_2, m_2)) \mid i_1 \neq i_2\}$$

Note that internal transitions (τ) performed by distinct processes are always independent. $\Box$.

Independency relations allow to define equivalence relations on execution sequences: two sequences u and v will be considered as equivalent iff u can be obtained from v by repeatedly permuting two of its *adjacent* independent actions.

Definition 5 (Equivalence between execution sequences).
Let I be an independency relation. For two sequences $u, v \in \mathcal{A}^*$, write $u \sim^1_I v$ if there exist sequences w_1, w_2 and actions a, b such that $(a, b) \in I$, $u = w_1abw_2$ and $v = w_1baw_2$. Let $\sim_I$ be the reflexive and transitive closure of the relation $\sim^1_I$. We say that u is I-equivalent with v if $u \sim_I v$.

Intuitively, if two equivalent sequences σ_1 and σ_2 are enabled on a state p, then, any buffer environment allowing the execution of σ_1 also allows the execution of σ_2 (and conversely). Furthermore, buffer contents are updated similarly during execution of σ_1 or σ_2. More formally:

Proposition 3. *Let $S = (Q, A, T, q_0)$ be an* Lts*, I an independence relation, p a state of Q, and σ_1 and σ_2 two execution sequences of S such that $\exists\ q_1, q_2 \in Q$, $p \xrightarrow{\sigma_1}_T q_1$, $p \xrightarrow{\sigma_2}_T q_2$ and $\sigma_1 \sim_I \sigma_2$.*

For a given buffer environment $\mathcal{B}_p$, let $S' = S[\mathcal{B}_p]$ where $S' = (Q', A, T', q'_0)$. Then, for any state $(p, b_1, b_2, \ldots, b_p)$ of Q', the following holds:

$$(p, b_1, b_2, \ldots, b_p) \xrightarrow{\sigma_1}_{T'} (q_1, b'_1, b'_2, \ldots, b'_p) \Leftrightarrow (p, b_1, b_2, \ldots, b_p) \xrightarrow{\sigma_2}_{T'} (q_2, b'_1, b'_2, \ldots, b'_p)$$

3.2 Preorder between execution sequences

As stated above, the equivalence relation between execution sequences *exactly* preserves the executability within any buffer environment. We introduce here a weaker relation, able to characterize the fact that a given sequence σ_1 is *more executable* than another sequence σ_2 (that is, whenever σ_2 is executable, then σ_1 is). This preorder relation between execution sequence relies itself on a *precedency* relation P between actions:

Definition 6 (Precedence of actions).
A relation $P \subseteq \mathcal{A} \times \mathcal{A}$ is a precedency *relation for an* Lts *$S = (Q, A, T, q_0)$ if, for all $p \in Q$, and for all $(a_1, a_2) \in P$ then:*

$$a_1, a_2 \subseteq Act(p) \Rightarrow \begin{cases} \forall q_2 \in Q \,.\, p \xrightarrow{a_2}_T q_2 \Rightarrow a_1 \in Act(q_2) \\ \wedge \\ \forall q \in Q \,.\, p \xrightarrow{a_2.a_1}_T q \Rightarrow p \xrightarrow{a_1.a_2}_T q \end{cases}$$

Example 2. When communications buffers are defined as *unbounded* bags, an **append** action performed by a process cannot prevent any **append** or **remove** action performed by another process. The precedency relation on communication actions between distinct processes is then: $P_{bag} = I_{bag} \cup \mathcal{A}^+ \times \mathcal{A}^-$ $\Box$.

This preorder on $\mathcal{A}$ is then extended to $\mathcal{A}^*$: a sequence σ_1 is smaller than a sequence σ_2 iff σ_1 can be obtained from σ_2 by repeatedly permuting any pair of its adjacent action belonging to the precedency relation.

Definition 7 (Preorder between execution sequences).
For two sequences $u, v \in \mathcal{A}^*$, write $u \lesssim^1_P v$ if there exist sequences w_1, w_2 and actions a, b such that $(a, b) \in P$, $u = w_1abw_2$ and $v = w_1baw_2$. Let $\lesssim_P$ be the reflexive and transitive closure of $\lesssim^1_P$. We say that u is smaller than v (or more executable) if $u \lesssim_P v$.

Proposition 3 can now be rephrased as follows:

Proposition 4. *Let $S = (Q, A, T, q_0)$ be an* Lts*, P a precedence relation, p a state of Q, and σ_1 and σ_2 two execution sequences of S such that $\exists\ q_1, q_2 \in Q$, $p \xrightarrow{\sigma_1}_T q_1$ and $p \xrightarrow{\sigma_2}_T q_2$ and $\sigma_1 \lesssim_P \sigma_2$. For a given buffer environment $\mathcal{B}_p$, let $S' = S[\mathcal{B}_p]$ where $S' = (Q', A, T', q_0')$. Then, for any state $(p, b_1, b_2, \ldots, b_p)$ of Q', the following holds:*

$$(p, b_1, b_2, \ldots, b_p) \xrightarrow{\sigma_2}_{T'} (q_1, b_1', b_2', \ldots, b_p') \ \Rightarrow \ (p, b_1, b_2, \ldots, b_p) \xrightarrow{\sigma_1}_{T'} (q_2, b_1', b_2', \ldots, b_p')$$

In the following sections we show how this preorder on execution sequences allows to define equivalence relations between Lts that are able to preserve various kinds of reachability properties. Moreover, since this preorder characterizes the executability of execution sequences, it turns out that these equivalence relations are congruence w.r.t. the [..] operator and therefore can be used during a compositional state space generation.

Note 1. We will consider in the sequel that buffers are *unbounded bags*. Thus, we shall note $\lesssim$ instead of $\lesssim_{P_{bag}}$. The extension of this work to *fifo* queues will be briefly discussed in the conclusion.

4 Deadlock preservation

We consider here a first property based on a simple reachability analysis, the *deadlock freedom* of a given program $\mathcal{P}$. More precisely, this property can be verified by compositionally generating a reduced Lts S', equivalent to $\mathbf{sem}(\mathcal{P})$ w.r.t. its deadlock states. To this purpose, we introduce an equivalence relation $\approx_\delta$ preserving the reachability of any ("equivalent") potential deadlock states. Then, we show that $\approx_\delta$ is a congruence w.r.t. operators $||$ and [...]. Finally, we propose a new asynchronous composition operator for the direct generation of a reduced Lts w.r.t to $\approx_\delta$.

4.1 A deadlock preserving equivalence between Lts

In our framework the only "blocking" actions performed by a program component are the **remove** operations. Consequently, potential deadlock states are the state not able to perform any **append** (or internal) operation. This set of states can be even reduced by considering that a subsequence of adjacent potential deadlock states of a same execution sequence can be collapsed into a single one (the first state of this subsequence). Furthermore, two potential deadlock states will be considered as equivalent iff a same buffer environment is able to "unlock" them (i.e., they can perform the same sets of consecutive **remove** operations).

More formally, these potential deadlock states are defined as the *stable states* of an LTS:

Definition 8 (Stable state).
Let $S = (Q, A, T, q_0)$ be an LTS*. For each state q of Q:*

$$stable(q) \equiv \ (q = q_0) \ \vee \ (Act(q) \subseteq \mathcal{A}^- \ \wedge \ Pre(q) \cap \mathcal{A}^+ \neq \emptyset) \ \vee \ (Act(q) = \emptyset)$$

We note stable(S) *the set of stable states of S. The equivalence $\sim_\delta$ between stable states q_1 and q_2 is then the following:*

$$q_1 \sim_\delta \ q_2 \equiv \begin{cases} \forall \sigma_1 \in {\mathcal{A}^-}^* \ . \ q_1 \xrightarrow{\sigma_1}_T \Rightarrow \ \ \exists \sigma_2 \ . \ q_2 \xrightarrow{\sigma_2}_T \ \wedge \sigma_2 \sim \sigma_1 \\ \wedge \\ \forall \sigma_2 \in {\mathcal{A}^-}^* \ . \ q_2 \xrightarrow{\sigma_2}_T \Rightarrow \ \ \exists \sigma_1 \ . \ q_1 \xrightarrow{\sigma_1}_T \ \wedge \sigma_1 \sim \sigma_2 \end{cases}$$

The purpose of equivalence $\approx_\delta$ is to preserve reachability of $\sim_\delta$-equivalent stable states in any buffer environment. Thus, a sufficient definition would be to consider two LTS S_1 and S_2 as equivalent if, for any stable state of S_1 reachable by an execution sequence σ_1, it corresponds an equivalent stable state of S_2, reachable by an execution sequence σ_2, such that $\sigma_2 \lesssim \sigma_1$ (and reciprocally for any stable state of S_2). However, we will use here a stronger definition, which better corresponds to the behaviour of the composition operator we will introduce later (see section 4.2).

Definition 9 (Equivalence between LTS**).**
Let $S_i = (Q_i, A_i, T_i, q_{0_i})_{i=1,2}$ be two LTS*. $\approx_\delta \ \subseteq Q_1 \times Q_2$ is the largest symmetrical relation verifying:*

$$p_1 \approx_\delta p_2 \Leftrightarrow \forall q_1 \in stable(S_1) \ . \ p_1 \xrightarrow{\sigma_1}_{T_1} q_1 \Rightarrow \exists q_2 \in stable(S_2) \ . \\ p_2 \xrightarrow{\sigma_2}_{T_2} q_2 \ \wedge \ q_1 \sim_\delta \ q_2 \ \wedge \ \sigma_2 \lesssim \sigma_1 \ \wedge \ q_1 \approx_\delta q_2$$

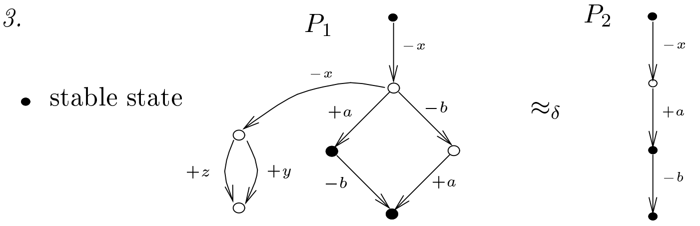
We extend $\approx_\delta$ to LTS *saying that $S_1 \approx_\delta S_2$ iff $q_{0_1} \approx_\delta q_{0_2}$.*

Relation $\approx_\delta$ preserves deadlocks in any buffer environment:

Proposition 5. *Let $S_i = (Q_i, A_i, T_i, q_{0_i})_{i=1,2}$ be two* LTS *and $\mathcal{B}_p$ a buffer environment. For a given* LTS *S let* **sink***(S) denote the set of state of S without any successors by its transition relation. Then:*

$$S_1 \approx_\delta S_2 \ \Rightarrow \ (\mathbf{sink}(S_1[\mathcal{B}_p]) = \emptyset \ \Leftrightarrow \ \mathbf{sink}(S_2[\mathcal{B}_p]) = \emptyset)$$

Example 3.



To each execution sequence of P_1 leading to a stable state there exists a *smaller* execution sequence of P_2, leading to an equivalent stable state (and reciprocally). In particular, sequence $-x.-x.(+y.+z)^*$ of P_1 which not lead to any stable state is not preserved by $\approx_\delta$ (since it will never lead to a deadlock even after further compositions). $\Box$.

Finally, proposition 6 states that relation $\approx_\delta$ is a congruence w.r.t operators $||$ (asynchronous composition) and [...] (execution within a given buffer environment). The proof of this proposition will rely on the following lemma:

Lemma 1. *For two execution sequences σ_1 and σ_2 of $\mathcal{A}^*$, we note $\sigma_1 \,||\, \sigma_2$ the set of sequences obtained by "asynchronous composition" of σ_1 and σ_2. $\sigma_1 \,||\, \sigma_2$ contains any sequence of $\mathcal{A}^*$ resulting of an interleaving of σ_1 and σ_2. Then, the following holds:*

$$\forall \sigma \in \mathcal{A}^* \,.\, \sigma_1 \lesssim \sigma_2 \;\Rightarrow\; \forall \sigma_2' \in (\sigma_2 \,||\, \sigma) \,.\, \exists\, \sigma_1' \in (\sigma_1 \,||\, \sigma) \text{ such that } \sigma_1' \lesssim \sigma_2'$$

Proposition 6 (Congruence of $\approx_\delta$). *Let S_1, S_2 and S be three* LTS, *and $\mathcal{B}_p$ a buffer environment. If $S_1 \approx_\delta S_2$ then the following holds:*

$$S_1[\mathcal{B}_p] \;\approx_\delta\; S_2[\mathcal{B}_p] \tag{1}$$
$$S_1 \,||\, S \;\approx_\delta\; S_2 \,||\, S \tag{2}$$

4.2 A deadlock preserving composition operator

The deadlock preserving composition operator $S_1 \otimes_\delta S_2$ is based on the standard operator $||$ of asynchronous composition between processes. Intuitively, the resulting LTS could be defined by "cutting off" any non minimal sequences of $S_1 \,||\, S_2$ leading to a stable state (according to the pre-order $\lesssim$, definition 7).

In practice, this LTS will be obtained by considering as atomic some particular subsequences of S_1 and S_2, thus avoiding their full interleaving. Moreover, this generation can be performed "on-the-fly" without generating $S_1 \,||\, S_2$. More precisely, atomic subsequences that we consider are delimited not only by stable states, but also using a particular set of states. These distinguished states are called "interleaving" in the sequel and are defined as follows:

Definition 10 (Interleaving states).
Let $P = (Q, A, T, q_0)$ an LTS. *We note* $\mathrm{int}(P)$ *the set of interleaving states of P:*
$int(P) = stable(P) \cup \{q \in Q \mid Act^-(p) \neq \emptyset \;\wedge\; Act^+(p) \neq \emptyset \;\wedge\; Pre(p) \cap \mathcal{A}^+ \neq \emptyset\}$

Formally, *atomic subsequences* are defined as follows:

$$atom(\sigma) \equiv\; \sigma = p_1 \xrightarrow{-a_1} p_1' \xrightarrow{-a_i^*} p_1'' \xrightarrow{+b_i^*} q_1$$

where p_1 is an interleaving state, q_1 a stable state, and each p_i'' such that $\mathrm{Act}^-(p_i'') \neq \emptyset$ is an interleaving state.

The deadlock preserving composition operator between processes can now be defined as follows:

Definition 11 (Deadlock preserving composition operator between LTS).

Let $P = P_1 \otimes_\delta P_2$ with $P = (Q, A, T, q_0)$ and $P_i = (Q_i, A_i, T_i, q_{0_i})_{i=1,2}$ s.t.:

- $q_0 = (q_{0_1}, q_{0_2})$;
- $A \subseteq A_1 \cup A_2$;
- *Q is the smallest set reachable from q_0 using T.*
- *The set of transitions T is computed using the four following rules. For each of them we note $\mathcal{H}$ the statement:*

$$\mathcal{H} = p_1 \xrightarrow{\sigma_1}_{T_1} q_1 \ \wedge\ p_2 \xrightarrow{\sigma_2}_{T_2} q_2 \ \wedge\ p_1 \in int(P_1) \ \wedge\ p_2 \in int(P_2) \\ \wedge\ atom(\sigma_1) \ \wedge\ atom(\sigma_2) \ \wedge\ stable(q_1) \ \wedge\ stable(q_2)$$

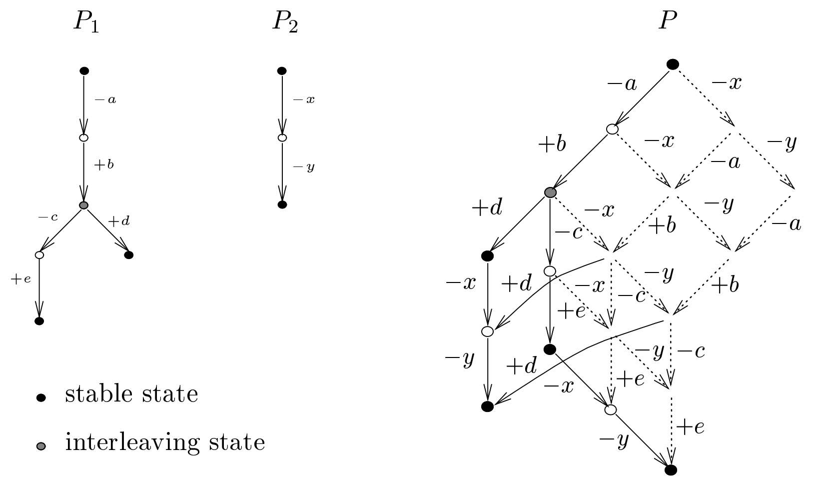
$$\frac{\mathcal{H},\ \sigma_1 \notin A^{-^*},\ \sigma_2 \notin A^{-^*}}{(p_1,p_2) \xrightarrow{\sigma_1}_T (q_1,p_2) \xrightarrow{\sigma_2}_T (q_1,q_2),\ (p_1,p_2) \xrightarrow{\sigma_2}_T (p_1,q_2) \xrightarrow{\sigma_1}_T (q_1,q_2)} \quad \textit{[R1]}$$

$$\frac{\mathcal{H},\ \sigma_1 \in A^{-^*},\ \sigma_2 \notin A^{-^*}}{(p_1,p_2) \xrightarrow{\sigma_2}_T (p_1,q_2) \xrightarrow{\sigma_1}_T (q_1,q_2)} \quad \textit{[R2]}$$

$$\frac{\mathcal{H},\ \sigma_1 \notin A^{-^*},\ \sigma_2 \in A^{-^*}}{(p_1,p_2) \xrightarrow{\sigma_1}_T (q_1,p_2) \xrightarrow{\sigma_2}_T (q_1,q_2)} \quad \textit{[R3]}$$

$$\frac{\mathcal{H},\ \sigma_1 \in A^{-^*},\ \sigma_2 \in A^{-^*}}{(p_1,p_2) \xrightarrow{\sigma_1}_T (q_1,p_2) \xrightarrow{\sigma_2}_T (q_1,q_2)\ or\ (p_1,p_2) \xrightarrow{\sigma_2}_T (p_1,q_2) \xrightarrow{\sigma_1}_T (q_1,q_2)} \quad \textit{[R4]}$$

Example 4. Let P_1 and P_2 be the two LTS represented below. LTS P is the product $P_1 \otimes_\delta P_2$. Dotted arrows indicate non minimal subsequences of $P_1 \,||\, P_2$ that have been “cut off”.



□.

Note 2.
For applying this method, all actions of σ_1 must be independent with actions of σ_2, which is the case when buffers are bags.

It remains to prove that this new operator of composition between processes preserves $\approx_\delta$ w.r.t. the standard asynchronous composition. This is expressed in the following proposition:

Proposition 7. *Let P_1 and P_2 be two* LTS*. Then we have $P_1 \mid\mid P_2 \approx_\delta P_1 \otimes_\delta P_2$.*

5 Observable language preservation

We consider now another kind of reachability property, the (finite) observable language generated by a given program $\mathcal{P}$. Here again, our objective is to compositionally generate a reduced LTS S', able to produce the same set of observable execution sequences as $\mathbf{sem}(\mathcal{P})$. Therefore, we introduce a relation $\approx_o$ preserving the language equivalence over a distinguished set $\mathcal{O} \subseteq \mathcal{A}$ of observable actions. Then, we show that $\approx_o$ is a congruence w.r.t. operators $\mid\mid$ and [...], and we propose another asynchronous composition operator preserving $\approx_o$.

5.1 A language preserving equivalence

For a given LTS S, we denote by $L_{\mathcal{O}}(S)$ the set of (finite) execution sequences S can perform up to a set of observable actions $\mathcal{O}$. Thus, *observable* states of S are the states able to perform any observable actions, and two (observable) states will be considered as equivalent iff they can perform the same observable actions.

Definition 12 (Observable language, observable states).
Let $S = (Q, A, T, q_0)$ be an LTS*. The observable language over $\mathcal{O}$ of S is the following set:*

$$L_{\mathcal{O}}(S) = \{\sigma_o \in \mathcal{O}^* \mid \sigma_o = o_1.o_2.\cdots.o_n \ \wedge \\ \exists \sigma = x_0^*.o_1.x_1^*.o_2.\cdots.x_{n-1}^*.o_n.x_n^* \ . \ q_0 \xrightarrow{\sigma}_T \ \wedge x_i \notin \mathcal{O}\}$$

For each state q of Q: $obs(q) \equiv \ Act(q) \cap \mathcal{O} \neq \emptyset$. We note $\mathrm{obs}(S)$ *the set of stable states of S and $\sim_o$ the equivalence relation between two states q_1 and q_2 defined as follows: $q_1 \sim_o \ q_2 \equiv \ (Act(q_1) \cap \mathcal{O} = Act(q_2) \cap \mathcal{O})$*

Clearly, to preserve the observable language of an LTS it is sufficient to preserve the reachability of each of its observable states (in any buffer environment) by execution sequences identical w.r.t. observable actions. Consequently, by replacing "stable" by "observable" (and $\sim_\delta$ by $\sim_o$) in definition 9, one could easily obtain a suitable equivalence relation.

Unfortunately this straightforward definition of $\approx_o$ is not satisfying, at least for two reasons:

1. Since it completely ignores the effect of execution sequences not containing any observable state, the resulting equivalence is not a congruence w.r.t. the $||$ operator [1]. Therefore, such execution sequences also have to be explicitly taken into account, this can be done in practice by preserving not only observable states but also the "interleaving" states introduced in section 4.2.
2. A "composed" state $(p_1, p_2, \dots p_n)$ becomes observable as soon as one of its component p_i is able to perform an observable action. Thus, asynchronous composition produces many "stuttering equivalent" observable states, not identified by this definition (since they are reachable by execution sequences not comparable w.r.t. $\lesssim$). Relation $\lesssim$ should be weakened into a new relation $\lesssim^{\#}$ in order to not distinguish these "stuttering equivalent" observable states.

Relation $\lesssim^{\#}$ relies on the following observation: since an **append** operation performed by a given component can never be prevented by its environment (and resp. a **remove** operation may always be prevented), execution sequence $+a.\omega$ can be considered as "more executable" than ω (resp. sequence $\omega.a-$ is "less executable" than ω). This suggests to extend the precedency relation P to the relation $P^{\#}$ such that: $P^{\#} = P \cup \{\mathcal{A}^+ \times \{\epsilon\}\} \cup \{\{\epsilon\} \times \mathcal{A}^-\}$

Relation $\lesssim^{\#}$ is then the extension of $P^{\#}$ to execution sequences (applying definition 7, where $\lesssim^{\#} = \lesssim_{P^{\#}}$). It is easy to see that proposition 4 still holds for $\lesssim^{\#}$, that is, according to this new preorder, smallest execution sequences are always more executable than largest ones in any buffer environment.

The definition of the language preserving equivalence $\approx_o$ is now the following:

Definition 13 (Equivalence between LTS**).**
Let $S_i = (Q_i, A_i, T_i, q_{0_i})_{i=1,2}$ be two LTS. *$\approx_o \subseteq Q_1 \times Q_2$ is the largest symmetrical relation verifying:*

$$\begin{aligned} p_1 \approx_o p_2 \Leftrightarrow \forall q_1 \in (obs(S_1) \cup int(S_1)) \ . \ p_1 \xrightarrow{\sigma_1}_{T_1} q_1 \Rightarrow \exists q_2 \in (obs(S_2) \cup int(S_2)) \ . \\ p_2 \xrightarrow{\sigma_2}_{T_2} q_2 \ \wedge \ q_1 \sim_{\emptyset} \ q_2 \ \wedge \ \sigma_2 \lesssim^{\#} \sigma_1 \ \wedge \ q_1 \approx_o q_2 \ \wedge \\ \forall \omega_1 \in \mathcal{A}^* \ . \ q_1 \xrightarrow{\omega_1}_{T_1} \Rightarrow \exists \omega_2 \in \mathcal{A}^* \ . \ q_2 \xrightarrow{\omega_2}_{T_2} \ \wedge \ \omega_2 \lesssim^{\#} \omega_1 \end{aligned}$$

We say that $S_1 \approx_o S_2$ iff $q_{0_1} \approx_o q_{0_2}$.

Relation $\approx_o$ preserves observable language over $\mathcal{O}$

Proposition 8. *Let S_1 and S_2 be two* LTS *and $\mathcal{B}_p$ a buffer environment.*

$$S_1 \approx_o S_2 \ \Rightarrow \ L_{\mathcal{O}}(S_1[\mathcal{B}_p]) = L_{\mathcal{O}}(S_2[\mathcal{B}_p])$$

Finally, we show that relation $\approx_o$ is a congruence w.r.t operators $||$ (asynchronous composition) and [...] (execution within a given buffer environment). Here again, the proof of this proposition will rely on lemma 1, which also applies to preorder $\lesssim^{\#}$.

[1] this problem did not occur with $\approx_\delta$ because execution sequences without stable state cannot lead to a deadlock even after composition with other components.

Proposition 9. *Let S_1, S_2 and $S = (Q, A, T, q_0)$ be three* LTS*, and $\mathcal{B}_p$ a buffer environment. If $S_1 \approx_o S_2$ then the following holds:*

$$S_1 \mid\mid S \approx_o S_2 \mid\mid S \tag{3}$$

$$S_1[\mathcal{B}_p] \approx_o S_2[\mathcal{B}_p] \tag{4}$$

5.2 A language preserving composition operator

We briefly explain here how the composition operator $\otimes_\delta$ defined in section 4.2 can be modified into a $\otimes_o$ operator preserving $\approx_o$-equivalence. The underlying idea is now to consider as *atomic* parts of execution sequences delimited either by "interleaving" states or observable states. However, the set of interleaving states considered in definition 10 have to be augmented in order to deal with "terminal" subsequences which do no contain any interleaving state (such sequences are necessarily ended by a loop of $\mathcal{A}^+$-actions). A practical way is to add to the interleaving set of states any element of this $\mathcal{A}^+$-loop (these states are computed during the construction of $S_1 \otimes_o S_2$).

Such sequences are then of the form: $atom(\sigma) \equiv \ \sigma = p_1 \xrightarrow{-a_i^*} p_1' \xrightarrow{+b_i^*} q_1$

Moreover, as in section 4.2, a complete interleaving between a pair of atomic sequences σ_1 and σ_2 is required only when both σ_1 and σ_2 contain a combination of **append** and **remove** actions (otherwise it is enough to consider only the smallest element of the ordered set $\{\sigma_1.\sigma_2, \sigma_2.\sigma_1\}$).

Formally, operator $\otimes_o$ is obtained by modifying definition of $\otimes_\delta$ (definition 11) as follows:

Definition 14 (Language preserving composition operator).
Let $P = P_1 \otimes_o P_2$ with $P = (Q, A, T, q_0)$ and $P_i = (Q_i, A_i, T_i, q_{0_i})_{i=1,2}$ s.t.:

- *$q_0 = (q_{0_1}, q_{0_2})$;*
- *$A \subseteq A_1 \cup A_2$;*
- *Q is the smallest set reachable from q_0 using T.*
- *The set of transitions T is computed using the following four rules. For each of them, we note $\mathcal{H}$ the following statement:*

$$\mathcal{H} = p_1 \xrightarrow{\sigma_1}_{T_1} q_1 \ \wedge \ p_2 \xrightarrow{\sigma_2}_{T_2} q_2 \ \wedge \ p_1 \in int(P_1) \cup obs(P_1) \ \wedge \ p_2 \in int(P_2) \cup obs(P_2)$$
$$\wedge \ atom(\sigma_1) \ \wedge \ atom(\sigma_2) \ \wedge \ (int(q_1) \ \vee \ obs(q_1)) \ \wedge \ (int(q_2) \ \vee \ obs(q_2))$$

$$\frac{\mathcal{H},\ \sigma_1 \notin A^{-^*},\ \sigma_2 \notin A^{-^*} \sigma_1 \notin A^{+^*},\ \sigma_2 \notin A^{+^*}}{(p_1, p_2) \xrightarrow{\sigma_1}_T (q_1, p_2) \xrightarrow{\sigma_2}_T (q_1, q_2),\ (p_1, p_2) \xrightarrow{\sigma_2}_T (p_1, q_2) \xrightarrow{\sigma_1}_T (q_1, q_2)} \quad \textit{[R1]}$$

$$\frac{\mathcal{H},\ \sigma_1 \in A^{-^*},\ \sigma_2 \notin A^{-^*}}{(p_1, p_2) \xrightarrow{\sigma_2}_T (p_1, q_2) \xrightarrow{\sigma_1}_T (q_1, q_2)} \quad \textit{[R2]}$$

$$\frac{\mathcal{H},\ \sigma_1 \notin A^{-^*},\ \sigma_2 \in A^{-^*}}{(p_1, p_2) \xrightarrow{\sigma_1}_T (q_1, p_2) \xrightarrow{\sigma_2}_T (q_1, q_2)} \quad \textit{[R3]}$$

$$\frac{\mathcal{H},\ (\sigma_1 \in A^{-^*} \ \wedge \ \sigma_2 \in A^{-^*}) \ \vee \ (\sigma_1 \in A^{+^*} \ \wedge \ \sigma_2 \in A^{+^*})}{(p_1, p_2) \xrightarrow{\sigma_1}_T (q_1, p_2) \xrightarrow{\sigma_2}_T (q_1, q_2) \ \textit{or} \ (p_1, p_2) \xrightarrow{\sigma_2}_T (p_1, q_2) \xrightarrow{\sigma_1}_T (q_1, q_2)} \quad \textit{[R4]}$$

Using similar arguments than in section 4.2 it is possible to show that this operator preserves $\approx_o$ w.r.t. the standard asynchronous composition:

Proposition 10. *Let P_1 and P_2 be two* LTS. *Then we have $P_1 \mid\mid P_2 \approx_o P_1 \otimes_o P_2$*

6 Conclusion and future works

We have proposed a state space generation method for asynchronous communicating processes which combines the benefits of both *compositionality* (generation and reduction steps are performed incrementally), and *partial-order reduction* techniques (only some representative elements of the set of execution sequences are considered).

More precisely, our approach was based on a syntactic notion of *precedence* of communication actions, leading to a preorder between execution sequences able to characterize their "executability" in any external buffer environments (smallest sequences are the most executable). Using this preorder, we proposed two equivalence relations between LTS, based on a similar notion of reachability of a distinguished set of states through most executable execution sequences. These two equivalence relations respectively preserve deadlock states and the system language up to a given set of observable actions. Moreover, they are congruences w.r.t. asynchronous composition. Finally, we have also defined two asynchronous composition operators, able to directly generate reduced LTS w.r.t. each of these relations. These operators differ on the standard one by considering as *atomic* particular subsequences of each process, thus saving many useless interleavings.

A first prototype implementation has been experimented within the IF environment developed at VERIMAG for the verification of asynchronous communicating systems [BFG+99]. The results obtained on a "benchmark" example (a leader election algorithm) largely confirm the interest of this compositional approach (about 5 000 generated states instead of 20 000 using a simultaneous composition, when verifying observable language preservation). It now remains to extend this experience to others case-studies, in particular to see how our approach compares with more "classical" partial-order reduction techniques (for instance the one implemented in SPIN [Hol91]).

One of the practical motivation behind this work is to apply compositional generation techniques to the verification of industrial size SDL specifications. To this purpose, the results proposed here will have to be (fully) extended to the case of asynchronous communications via *fifo queues* (instead of their abstraction in terms of *bags*). In this case, the "purely syntactic" definition of *precedence* relation between actions we considered here may be to strict, and it would be interesting to see how it can be enlarged using more sophisticated *static analysis* techniques (for instance depending on the communication topology between processes). To this purpose, a suitable framework could be provided by the notion of *conditional independence* proposed in [KP92].

Acknowledgements: Parts of this work were largely improved during fruitful discussions with M. Bozga, S. Graf and J. Sifakis. Thanks are also due to the anonymous referees for their helpful comments and suggestions.

References

[AJKP98] P. Abdulla, B. Jonsson, M. Kindhal, and D. Peled. A General Approach to Partial Order Reductions in Symbolic Verification. In *Proceedings of CAV'98, Vancouver, Canada*, volume 1427 of *LNCS*, June 1998.

[BFG+91] Ahmed Bouajjani, Jean-Claude Fernandez, Susanne Graf, Carlos Rodríguez, and Joseph Sifakis. Safety for Branching Time Semantics. In *Proceedings of 18th ICALP*. Springer Verlag, July 1991.

[BFG+99] M. Bozga, J.-C. Fernandez, L. Ghirvu, S. Graf, J.P. Krimm, and L. Mounier. IF: An Intermediate Representation and Validation Environment for Timed Asynchronous Systems. In *Proceedings of FM'99, Toulouse, France*, LNCS 1708, 1999.

[CK93] S.C. Cheung and J. Kramer. Enhancing Compositional Reachability Analysis with Context Constraints. In *Proceedings of the 1st ACM International Symposium on the Foundations of Software Engineering*, pages 115–125, Los Angeles, California, December 1993.

[GS90] S. Graf and B. Steffen. Compositional Minimization of Finite State Processes. In *Workshop on Computer-Aided Verification*, Rutgers, USA, June 1990. DIMACS, R.P. Kurshan and E.M. Clarke.

[GW91] P. Godefroid and P. Wolper. Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties. In K. G. Larsen, editor, *Proceedings of CAV'91 (Aalborg, Denmark)*, July 1991.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[Hol91] Gerard J. Holzmann. *Design and Validation of Computer Protocols.* Software Series. Prentice Hall, 1991.

[ISO87] ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. Information Processing Systems — OSI , Genève, July 1987.

[IT92] ITU-T. Specification and Description Language (SDL). ITU-T Recommendation Z.100, International Telecommunication Union, Genève, 1992.

[KLM+98] R. Kurshan, V. Levin, M. Minea, D. Peled, and H. Yenigün. Static Partial Order Reduction. In *Proceedings of TACAS'98, Lisbon, Portugal*, volume 1384 of *LNCS*, 1998.

[KM97] Jean-Pierre Krimm and Laurent Mounier. Compositional State Space Generation from Lotos Programs. In Ed Brinksma, editor, *Proceedings of TACAS'97*, Enschede, The Netherlands, April 1997. Springer Verlag.

[KP92] S. Katz and D. Peled. Defining conditional independence usin collapses. *Theoretical Computer Science*, 101(1):337–359, 1992.

[Lip75] Lipton. Reduction, a method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, dec 1975.

[Mil89] Robin Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[Pel96] Doron Peled. Combining partial-order reductions with on-the-fly model-checking. *Formal Methods in System Design*, 8:39–64, 1996.

[Val90] A. Valmari. A Stubborn Attack on State Explosion. In *Workshop on Computer-Aided Verification*, Rutgers, USA, June 1990. DIMACS, R.P. Kurshan and E.M. Clarke.

[Val96] Antti Valmari. *Compositionality in State Space Verification.* In *Application and Theory of Petri Nets*, volume 1091 of *LNCS*, pages 29–56. Springer Verlag, June 1996.

[vGW89] R. J. van Glabbeek and W. P. Weijland. Branching-Time and Abstraction in Bisimulation Semantics (extended abstract). CS R8911, Centrum voor Wiskunde en Informatica, Amsterdam, 1989.