Partial Order Reductions using Compositional Confluence Detection

Frederic Lang, Radu Mateescu

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Partial Order Reductions using Compositional Confluence Detection

Frédéric Lang and Radu Mateescu

N° 7078

Octobre 2009

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--7078--FR+ENG

Partial Order Reductions using Compositional Confluence Detection

Frédéric Lang and Radu Mateescu

Thème COM — Systèmes communicants

Projet VASY

Rapport de recherche n° 7078 — Octobre 2009 — 28 pages

Abstract: Explicit state methods have proven useful in verifying safety-critical systems containing concurrent processes that run asynchronously and communicate. Such methods consist of inspecting the states and transitions of a graph representation of the system. Their main limitation is state explosion, which happens when the graph is too large to be stored in the available computer memory. Several techniques can be used to palliate state explosion, such as on-the-fly verification, compositional verification, and partial order reductions. In this paper, we propose a new technique of partial order reductions based on compositional confluence detection (CCD), which can be combined with the techniques mentioned above. CCD is based upon a generalization of the notion of confluence defined by Milner and exploits the fact that synchronizing transitions that are confluent in the individual processes yield a confluent transition in the system graph. It thus consists of analysing the transitions of the individual process graphs and the synchronization structure to identify such confluent transitions compositionally. Under some additional conditions, the confluent transitions can be given priority over the other transitions, thus enabling graph reductions. We propose two such additional conditions: one ensuring that the generated graph is equivalent to the original system graph modulo branching bisimulation, and one ensuring that the generated graph contains the same deadlock states as the original system graph. We also describe how CCD-based reductions were implemented in the CADP toolbox, and present examples and a case study in which adding CCD improves reductions with respect to compositional verification and other partial order reductions.

A short version of this report is also available as "*Partial Order Reductions using Compositional Confluence Detection*", in Jos Baeten, Ana Cavalcanti, and Dennis Dams, editors, Proceedings of the 16th International Symposium on Formal Methods FM'2009 (Eindhoven, The Netherlands), November 4–6, 2009.

Centre de recherche INRIA Grenoble – Rhône-Alpes
Inovallée, 655, avenue de l'Europe, Montbonnot, 38334 Saint Ismier Cedex (France)
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Key-words: Asynchronous concurrency, branching bisimulation, deadlock, formal method, graph reduction, model-checking, network of communicating automata, state explosion, verification

Réductions d'ordres partiels avec détection compositionnelle de confluence

Résumé : Les méthodes d'états explicites se sont révélées utiles pour vérifier des systèmes critiques constitués de processus parallèles s'exécutant de manière asynchrone et communicant. De telles méthodes consistent à inspecter les états et les transitions d'une représentation du système sous forme de graphe. Leur principale limitation est l'explosion d'états, qui se produit lorsque le graphe est trop grand pour être stocké dans la mémoire disponible de l'ordinateur. Plusieurs techniques peuvent être utilisées pour pallier l'explosion d'états, telles que la vérification à la volée, la vérification compositionnelle, et les réductions d'ordres partiels. Dans cet article, nous proposons une nouvelle technique de réduction d'ordres partiels basée sur la détection compositionnelle de confluence (*compositional confluence detection,* Ccd), qui peut être combinée avec les techniques citées ci-dessus. Ccd se base sur une généralisation de la notion de confluence définie par Milner et exploite le fait que la synchronisation de transitions confluentes dans les processus individuels produit une transition confluente dans le graphe du system complet. Il suffit donc d'analyser les transitions des graphes des processus individuels ainsi que la structure de synchronisation pour identifier de telles transitions confluentes de manière compositionnelle. En ajoutant des conditions supplémentaires, les transitions confluentes peuvent être rendues prioritaires par rapport aux autres transitions, permettant ainsi des réductions du graphe. Nous proposons deux telles conditions supplémentaires : l'une garantit que le graphe généré est équivalent au graphe d'origine modulo la bisimulation de branchement, et l'autre garantit que le graphe généré contient les mêmes états d'interblocage que le le graphe d'origine. Nous décrivons aussi comment les réductions basées sur Ccd ont été mises en application dans la boîte à outils Cadp, et nous présentons des exemples et une étude de cas dans lesquels l'utilisation de Ccd améliore les réductions par rapport à la vérification compositionnelle et d'autres réductions d'ordres partiels.

Mots-clés : Bisimulation de branchement, explosion d'états, interblocage, méthode formelle, model-checking, parallélisme asynchrone, réduction de graphe, réseau d'automates communicants, vérification

1 Introduction

This paper deals with systems, hereafter called *asynchronous systems*, which can be modeled by a composition of individual processes that execute in parallel at independent speeds and communicate. Asynchronous systems can be found in many application domains, such as communication protocols, embedded software, hardware architectures, distributed systems, etc.

Industrial asynchronous systems are often subject to strong constraints in terms of development cost and/or reliability. A way to address these constraints is to use methods allowing the identification of bugs as early as possible in the development cycle. Explicit state verification is such a method, and consists of verifying properties by systematic exploration of the states and transitions of an abstract model of the system.

Although appropriate for verifying asynchronous systems, explicit state verification may be limited by the combinatorial explosion of the number of states and transitions (called *state explosion*). Among the numerous techniques that have been proposed to palliate state explosion, the following have proved to be effective:

- *On-the-fly verification* (see e.g., [9, 8, 20, 29, 27]) consists of enumerating the states and transitions in an order determined by a property of interest, thus enabling one to find property violations before the whole system graph has been generated.
- *Compositional verification* (see e.g., [7, 26, 38, 41, 43, 6, 36, 16, 23, 37, 14, 11]) consists of replacing individual processes by property-preserving abstractions of limited size.
- *Partial order reductions* (see e.g., [15, 39, 33, 19, 40, 34, 35, 18, 31, 3, 32]) consist of choosing not to explore interleavings of actions that are not relevant with respect to either the properties or the graph equivalence of interest.

Regarding partial order reductions, two lines of work coexist. The first addresses the identification of a subset called *persistent* [15] (or *ample* [33], or *stubborn* [39], see [34] for a survey[1]) of the operations that define the transitions of the system, such that all operations outside this subset are independent of all operations inside this subset. This allows the operations outside the persistent subset to be ignored in the current state. Depending on additional conditions, persistent subsets may preserve various classes of properties (e.g., deadlocks, LTL-X, CTL-X, etc.) and/or graph equivalence relations (e.g., branching equivalence [42], weak trace equivalence [5], etc). Other methods based on the identification of independent transitions, such as *sleep sets* [15], can be combined with persistent sets to obtain more reductions.

[1] In this paper, the term *persistent* will refer equally to persistent, ample, or stubborn.

The second line of work addresses the detection of particular non-observable transitions (non-observable transitions are also called τ-transitions) that satisfy the property of confluence [30, 19, 18, 44, 2, 3, 32], using either symbolic or explicit-state techniques. Such transitions can be given priority over the rest of the transitions of the system, thus avoiding exploration of useless states and transitions while preserving branching (and observational) equivalence. Among the symbolic detection techniques, the proof-theoretic technique of [3] statically generates a formula encoding the confluence condition from a μCRL program, and then solves it using a separate theorem prover. Among the explicit-state techniques, the global technique of [18] computes the maximal set of strongly confluent τ-transitions and reduces the graph with respect to this set. A local technique was proposed in [2], which computes on-the-fly a representation map associating a single state to each connected subgraph of confluent τ-transitions. Another technique was proposed in [32], which reformulates the detection as the resolution of a BES (*Boolean Equation System*) and prioritizes confluent τ-transitions in the individual processes before composing them, using the fact that branching equivalence is a congruence for the parallel composition of processes. Compared to persistent subset methods, whose practical effectiveness depends on the accuracy of identifying independent operations (by analyzing the system description), confluence detection methods are able to detect all confluent transitions (by exploring the system graph), potentially leading to better reductions.

In this paper, we present a new compositional partial order reduction method for systems described as networks of communicating automata. This method, named CCD (*Compositional Confluence Detection*), exploits the confluence of individual process transitions that are not necessarily labeled by τ and thus cannot be prioritized in the individual processes. CCD relies on the fact that synchronizing such transitions always yields a confluent transition in the graph of the composition. As an immediate consequence, if the latter transition is labeled by τ (i.e., hidden after synchronization), then giving it priority preserves branching equivalence. We also describe conditions to ensure that even transitions that are not labeled by τ can be prioritized, while still preserving the deadlocks of the system.

The aim of CCD is to use compositionality to detect confluence more efficiently than explicit-state techniques applied directly to the graph of the composition, the counterpart being that not all confluent transitions are necessarily detected (as in persistent subset methods). Nevertheless, CCD and persistent subset methods are orthogonal, meaning that neither method applied individually performs better than both methods applied together. Thus, CCD can be freely added in order to improve the reductions achieved by persistent subset methods. Moreover, the definition of confluent transitions is language-independent (i.e., it does not rely upon the description language — in our case EXP.OPEN 2.0 [24] — but only upon the system graph), making CCD suitable for networks of communicating automata produced from any description language equipped with interleaving semantics.

CCD was implemented in the CADP toolbox [12] and more particularly in the existing EXP.OPEN 2.0 tool for compositional verification, which provides on-the-fly verification of compositions of processes. A new procedure was developed, which searches and annotates the confluent (or strictly confluent) transitions of a graph, using a BES to encode the confluence property. This procedure is invoked on the individual processes so that EXP.OPEN 2.0 can then generate a reduced graph for the composition, possibly combined with already available persistent subset methods.

Experimental results show that adding CCD may improve reductions with respect to compositional verification and persistent subset methods.

Paper outline. Section 2 gives preliminary definitions and theorems. Section 3 formally presents the semantic model that we use to represent asynchronous systems. Section 4 presents the main result of the paper. Section 5 describes how the CCD technique is implemented in the CADP toolbox. Section 6 presents several experimental results. Section 7 reports about the application of CCD in an industrial case-study. Finally, Section 8 gives concluding remarks.

2 Preliminaries

We consider the standard LTS (*Labeled Transition System*) semantic model [30], which is a graph consisting of a set of *states*, an *initial state*, and a set of *transitions* between states, each transition being labeled by an action of the system.

Definition 1 (Labeled Transition System) Let $\mathcal{A}$ be a set of symbols called *labels*, which contains a special symbol τ, called the *unobservable label*. An LTS is a quadruple $(Q, A, \rightarrow, q_0)$, where Q is the set of *states*, $A \subseteq \mathcal{A}$ is the set of *labels*, $\rightarrow \subseteq Q \times A \times Q$ is the *transition relation*, and $q_0 \in Q$ is the *initial state* of the LTS. As usual, we may write $q_1 \xrightarrow{a} q_2$ instead of $(q_1, a, q_2) \in \rightarrow$. Any sequence of the form $q_1 \xrightarrow{a_1} q_2 \xrightarrow{a_2} \ldots q_n \xrightarrow{a_n} q_{n+1}$ is called a *path of length* n from q_1 to q_{n+1} ($n \geq 0$). We write $q_1 \longrightarrow^n q_{n+1}$ if there exists such a path. The transition relation is acyclic if every path from a state to itself has length 0. □

Branching equivalence [42] is a weak bisimulation relation between states of an LTS that removes some τ-transitions while preserving the branching structure of the LTS. Therefore, branching equivalence is of interest when verifying branching-time temporal logic properties that concern only observable labels.

Definition 2 (Branching equivalence [42]) As usual, we write $\xrightarrow{\tau*}$ the reflexive and transitive closure of $\xrightarrow{\tau}$. Two states $q_1, q_2 \in Q$ are *branching equivalent* if and only if there exists a relation $R \subseteq Q \times Q$ such that $R(q_1, q_2)$ and (1) for each transition $q_1 \xrightarrow{a} q_1'$, either $a = \tau$ and $R(q_1', q_2)$ or there is a path

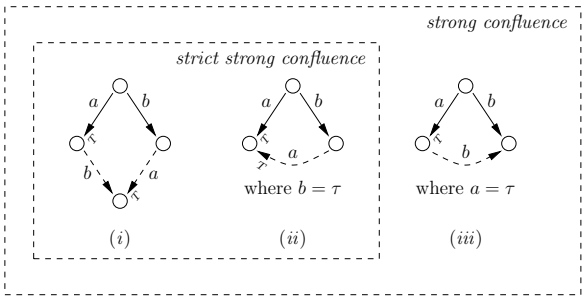

Figure 1: Graphical definition of *strong confluence* and *strict strong confluence*

$q_2 \xrightarrow{\tau*} q_2' \xrightarrow{a} q_2''$ such that $R(q_1, q_2')$ and $R(q_1', q_2'')$, and (2) for each transition $q_2 \xrightarrow{a} q_2'$, either $a = \tau$ and $R(q_1, q_2')$ or there is a path $q_1 \xrightarrow{\tau*} q_1' \xrightarrow{a} q_1''$ such that $R(q_1', q_2)$ and $R(q_1'', q_2')$. □

The following definition of *strong confluence* is a synthesis of the definitions of *confluence* by Milner [30], which is a property of processes, and *partial strong τ-confluence* by Groote and van de Pol [18], which is a property of τ-transitions. We thus generalize Groote and van de Pol's definition to transitions labeled by arbitrary symbols, as was the case of Milner's original definition. In addition, we distinguish between the property of strong confluence, and a slightly more constrained property, named *strict* strong confluence.

Definition 3 (Strong confluence) Let $(Q, A, \rightarrow, q_0)$ be an LTS and $T \subseteq \rightarrow$. We write $q \xrightarrow{a}_T q'$ if $(q, a, q') \in T$. We write $q \xrightarrow{\overline{a}} q'$ if either $q \xrightarrow{a} q'$ or $q = q'$ and $a = \tau$, and similarly for $q \xrightarrow{\overline{a}}_T q'$. T is *strongly confluent* if for every pair of distinct transitions $q_1 \xrightarrow{a}_T q_2$ and $q_1 \xrightarrow{b} q_3$, there exists a state q_4 such that $q_3 \xrightarrow{\overline{a}}_T q_4$ and $q_2 \xrightarrow{\overline{b}} q_4$. T is *strictly strongly confluent* if for every pair of distinct transitions $q_1 \xrightarrow{a}_T q_2$ and $q_1 \xrightarrow{b} q_3$, there exists a state q_4 such that $q_3 \xrightarrow{a}_T q_4$ and $q_2 \xrightarrow{\overline{b}} q_4$. A transition is *strongly confluent* (respectively *strictly strongly confluent*) if there exists a strongly confluent set (respectively strictly strongly confluent set) $T \subseteq \rightarrow$ containing that transition. □

Figure 1 gives a graphical picture of strong confluence. Plain arrows denote transitions quantified universally, whereas dotted arrows denote transitions quantified existentially. For strict strong confluence, case (*iii*) is excluded.

Strong τ-confluence is strong confluence of τ-transitions. Weaker notions of τ-confluence have been defined [19, 44], but are out of the scope of this paper. For brevity, we use below the terms confluent and strictly confluent instead of strongly confluent and strictly strongly confluent, respectively.

Prioritization consists of giving priority to some transitions. Definition 4 below generalizes the definition of [18], which was restricted to τ-transitions.

Definition 4 (Prioritization [18]) Let $(Q, A, \rightarrow_1, q_0)$ be an LTS and $T \subseteq \rightarrow_1$. A *prioritization* of $(Q, A, \rightarrow_1, q_0)$ with respect to T is any LTS of the form $(Q, A, \rightarrow_2, q_0)$, where $\rightarrow_2 \subseteq \rightarrow_1$ and for all $q_1, q_2 \in Q, a \in A$, if $q_1 \xrightarrow{a}_1 q_2$ then (1) $q_1 \xrightarrow{a}_2 q_2$ or (2) there exists $q_3 \in Q$ and $b \in A$ such that $q_1 \xrightarrow{b}_2 q_3 \in T$. □

In [18], Groote and van de Pol proved that branching bisimulation is preserved by prioritization of τ-confluent transitions, provided the LTS does not contain cycles of τ-transitions. Theorem 1 below relaxes this constraint by only requiring that the set of prioritized τ-confluent transitions does not contain cycles (which is similar to the cycle-closing condition for ample sets [33]).

Theorem 1 Let $(Q, A, \rightarrow, q_0)$ be an LTS and $T \subseteq \rightarrow$ such that T is acyclic and contains only τ-confluent transitions. Any prioritization of $(Q, A, \rightarrow, q_0)$ with respect to T yields an LTS that is branching equivalent to $(Q, A, \rightarrow, q_0)$. □

Proof. See [18]. The proof remains correct, despite the fact that the assumption "every cycle of $(Q, A, \rightarrow_1, q_0)$ contains at least one transition that does not belong to $\rightarrow_2$" is slightly more general than the assumption stated in [18] "the LTS must not contain cycles of τ-transitions". Indeed, the justification of the assumption in [18] is precisely to avoid prioritizing all τ-transitions in the same cycle of τ-confluent transitions. This issue is related to the problem known in the setting of persistent sets as the *ignoring problem*, which is solved similarly. □

Theorem 2 below states that deadlock states can always be reached without following transitions that are in choice with strictly confluent transitions. This allows prioritization of strictly confluent transitions, while ensuring that at least one (minimal) diagnostic path can be found for each deadlock state.

Theorem 2 Let $(Q, A, \rightarrow, q_0)$ be an LTS, $T \subseteq \rightarrow$ a strictly confluent set of transitions, and $q_\delta \in Q$ be a deadlock state. If $q_1 \longrightarrow^n q_\delta$ and $q_1 \xrightarrow{a}_T q_2$, then $q_2 \longrightarrow^m q_\delta$ with $m < n$. □

Proof. We proceed by induction on n. If $n = 0$ then $q_1 = q_\delta$, and $q_1 \xrightarrow{a}_T q_2$ is not possible since a deadlock state has no outgoing transition.

If $n = 1$ then, since q_δ is a deadlock state, $q_2 = q_\delta$, otherwise

the length of the shortest paths from q_1 to q_δ. If the length is 0 then $q_1 = q_\delta$ and the property trivially holds.

We assume that the property holds up to an arbitrary length n. Let the shortest paths from q_1 to q_δ have length $n + 1$. Then there exists $a_1 \in A$, $q_2 \in Q$ and a shortest path from q_1 to q_δ whose first transition has the form $q_1 \xrightarrow{a_1} q_2$. By

the induction hypothesis, we know that there exists a (possibly empty) shortest path from q_2 to q_δ of the form $q_2 \xrightarrow{a_2} q_3 \xrightarrow{a_3} \ldots \xrightarrow{a_n} q_{n+1} = q_\delta$ such that for all $i \in 2..n$ either q_i is not the source of any strictly confluent transition or $q_i \xrightarrow{a_i} q_{i+1}$ is strictly confluent. We consider two cases:

- If q_1 is not the source of any confluent transition or if $q_1 \xrightarrow{a_1} q_2$ is confluent then the path $q_1 \xrightarrow{a_1} q_2 \xrightarrow{a_2} \ldots \xrightarrow{a_n} q_{n+1} = q_\delta$ is a shortest path from q_1 to q_δ such that for all $i \in 1..n$ either q_i is not the source of any strictly confluent transition or $q_i \xrightarrow{a_i} q_{i+1}$ is strictly confluent.
- Otherwise, there exists a strictly confluent transition of the form $q_1 \xrightarrow{a} q_1'$. By definition of strict confluence, either $q_1' = q_2$ and $a = a_1$ or there exists a state q_1'' such that $q_2 \xrightarrow{a} q_1''$ is strictly confluent. From the induction hypothesis, we know that if $q_2 \xrightarrow{a} q_1''$ is strictly confluent then $a = a_2$ and $q_1'' = q_3$. Therefore, the path $q_1 \xrightarrow{a_1} q_2 \xrightarrow{a_2} \ldots \xrightarrow{a_n} q_{n+1} = q_\delta$ is a shortest path from q_1 to q_δ such that for all $i \in 1..n$ either q_i is not the source of any strictly confluent transition or $q_i \xrightarrow{a_i} q_{i+1}$ is strictly confluent. □

Therefore, any prioritization of $(Q, A, \rightarrow, q_0)$ with respect to T yields an LTS that has the same deadlock states as $(Q, A, \rightarrow, q_0)$.

Note. Theorem 2 is not true for non-strict confluence, as illustrated by the LTS consisting of the transition $q_1 \xrightarrow{a} q_\delta$ and the (non-strictly) confluent transition $q_1 \xrightarrow{\tau} q_1$.

3 Networks of LTSs

This section introduces networks of LTSs [24, 25], a concurrent model close to MEC [1] and FC2 [4], which consists of a set of LTSs composed in parallel and synchronizing following general synchronization rules.

Definition 5 (Vector) A *vector* of *length* n over a set T is an element of T^n. Let $\boldsymbol{v}$, also written $(v_1, \ldots, v_n)$, be a vector of length n. The elements of $1..n$ are called the *indices* of $\boldsymbol{v}$. For each $i \in 1..n$, $\boldsymbol{v}[i]$ denotes the i^{th} element v_i of $\boldsymbol{v}$. □

Definition 6 (Network of LTSs) Let $\bullet \notin \mathcal{A}$ be a special symbol denoting *inaction*. A *synchronization vector* is a vector over $\mathcal{A} \cup \{\bullet\}$. Let $\boldsymbol{t}$ be a synchronization vector of length n. The *active components* of $\boldsymbol{t}$, written $\text{act}(\boldsymbol{t})$, are defined as the set $\{i \in 1..n \mid \boldsymbol{t}[i] \neq \bullet\}$. The *inactive components* of $\boldsymbol{t}$, written $\text{inact}(\boldsymbol{t})$, are defined as the set $1..n \setminus \text{act}(\boldsymbol{t})$. A *synchronization rule* of length n is a pair $(\boldsymbol{t}, a)$, where $\boldsymbol{t}$ is a synchronization vector of length n and $a \in \mathcal{A}$. The elements $\boldsymbol{t}$ and a are called respectively the *left-* and *right-hand sides* of the

synchronization rule. A *network of* LTSs N of length n is a pair $(\boldsymbol{S}, V)$ where $\boldsymbol{S}$ is a vector of length n over LTSs and V is a set of synchronization rules of length n. □

In the sequel, we may use the term network instead of network of LTSs. A network $(\boldsymbol{S}, V)$ therefore denotes a product of LTSs, where each rule expresses a constraint on the vector of LTSs $\boldsymbol{S}$. In a given state of the product, each rule $(\boldsymbol{t}, a) \in V$ yields a transition labeled by a under the condition that, assuming $\text{act}(\boldsymbol{t}) = \{i_0, \ldots, i_m\}$, the LTSs $\boldsymbol{S}[i_0], \ldots, \boldsymbol{S}[i_m]$ may synchronize altogether on transitions labeled respectively by $\boldsymbol{t}[i_0], \ldots, \boldsymbol{t}[i_m]$. This is described formally by the following definition.

Definition 7 (Network semantics) Let N be a network of length n defined as a couple $(\boldsymbol{S}, V)$ and for each $i \in 1..n$, let $\boldsymbol{S}[i]$ be the LTS $(Q_i, A_i, \rightarrow_i, q_{0i})$. The *semantics* of N, written $lts(N)$ or $lts(\boldsymbol{S}, V)$, is an LTS $(Q, A, \rightarrow, \boldsymbol{q}_0)$ where $Q \subseteq Q_1 \times \ldots \times Q_n$, $\boldsymbol{q}_0 = (q_{01}, \ldots, q_{0n})$ and $A = \{a \mid (\boldsymbol{t}, a) \in V\}$. Given a synchronization rule $(\boldsymbol{t}, a) \in V$ and a state $\boldsymbol{q} \in Q_1 \times \ldots \times Q_n$, we define the *successors* of $\boldsymbol{q}$ by rule $(\boldsymbol{t}, a)$, written $\text{succ}(\boldsymbol{q}, (\boldsymbol{t}, a))$, as follows:

$$\text{succ}(\boldsymbol{q}, (\boldsymbol{t}, a)) = \{\boldsymbol{q}' \in Q_1 \times \ldots \times Q_n \mid \begin{array}{l} (\forall i \in \text{act}(\boldsymbol{t}))\ \boldsymbol{q}[i] \xrightarrow{\boldsymbol{t}[i]}_i \boldsymbol{q}'[i] \wedge \\ (\forall i \in \text{inact}(\boldsymbol{t}))\ \boldsymbol{q}[i] = \boldsymbol{q}'[i]\} \end{array}$$

The state set Q and the transition relation $\rightarrow$ of $lts(N)$ are the smallest set and the smallest relation such that $\boldsymbol{q}_0 \in Q$ and:

$$q \in Q \wedge (\boldsymbol{t}, a) \in V \wedge \boldsymbol{q}' \in \text{succ}(\boldsymbol{q}, (\boldsymbol{t}, a)) \Rightarrow \boldsymbol{q}' \in Q \wedge \boldsymbol{q} \xrightarrow{a} \boldsymbol{q}'.$$ □

Synchronization rules must obey the following admissibility condition, which forbids cutting, synchronization and renaming of the τ transitions present in the individual LTSs. This is suitable for a process algebraic framework, most parallel composition, hiding, renaming, and cutting operators of which can be translated into rules obeying these conditions. This also ensures that weak trace equivalence and stronger relations (e.g., safety, observational, branching, and strong equivalences) are congruences for synchronization rules [24].

Definition 8 (Network admissibility) The network $(\boldsymbol{S}, V)$ is *admissible* if for each q, q', i such that $q \xrightarrow{\tau}_i q'$ there exists a rule $(\boldsymbol{t}_i, \tau) \in V$ where $\boldsymbol{t}_i[i] = \tau$, $(\forall j \neq i)\ \boldsymbol{t}_i[j] = \bullet$, and $(\forall (t, a) \in V \setminus \{(\boldsymbol{t}_i, \tau)\})\ t[i] \neq \tau$. Below, every network will be assumed to be admissible. □

Example 1 We consider the simple network of LTSs consisting of the vector of LTSs $(Sender_1, Bag, Sender_2)$ depicted in Figure 2 (the topmost node being the initial state of each LTS), and of the following four synchronization rules: $((s_1, s_1, \bullet), \tau)$, $((\bullet, s_2, s_2), \tau)$, $((\bullet, r_1, \bullet), r_1)$, $((\bullet, r_2, \bullet), r_2)$.

This network represents two processes $Sender_1$ and $Sender_2$, which send their respective messages s_1 and s_2 via a communication buffer that contains one

place for each sender and uses a *bag* policy (received messages can be delivered in any order). Every transition in the individual LTSs of this network is strictly confluent. The LTS (i) depicted in Figure 3, page 14, represents the semantics of this network.

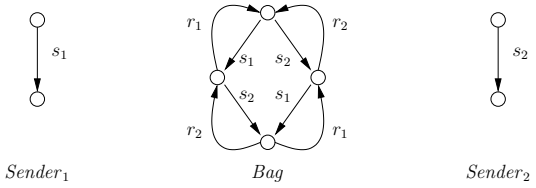

Figure 2: Individual LTSs of the network defined in Example 1

4 Compositional Confluence Detection

Although prioritizing confluent transitions yields LTS reductions, finding confluent transitions in large LTSs such as those obtained by parallel composition of smaller LTSs can be quite expensive in practice. Instead, the aim of CCD is to infer confluence in the large LTS from the (much cheaper to find) confluence present in the smaller LTSs that are composed.

Definition 9 Let $(\boldsymbol{S}, V)$ be a network, $(\boldsymbol{t}, a) \in V$, and $\boldsymbol{q}$, $\boldsymbol{q}'$ be states of $lts(\boldsymbol{S}, V)$. We write all_conf$(\boldsymbol{q}, (\boldsymbol{t}, a), \boldsymbol{q}')$ for the predicate "$\boldsymbol{q}' \in \text{succ}(\boldsymbol{q}, (\boldsymbol{t}, a)) \wedge (\forall i \in \text{act}(\boldsymbol{t}))\ \boldsymbol{q}[i] \xrightarrow{\boldsymbol{t}[i]}_i \boldsymbol{q}'[i]$ is confluent". We write all_conf_strict for the same predicate, where "*strictly confluent*" replaces "*confluent*". □

Theorem 3 below presents the main result of this paper: synchronizations involving only confluent (resp. strictly confluent) transitions in the individual LTSs produce confluent (resp. strictly confluent) transitions in the LTS of the network.

Theorem 3 (Compositional confluence detection) Let $(\boldsymbol{S}, V)$ be a network, $(\boldsymbol{t}, a) \in V$, and $\boldsymbol{q}$, $\boldsymbol{q}'$ be states of $lts(\boldsymbol{S}, V)$. (1) If all_conf$(\boldsymbol{q}, (\boldsymbol{t}, a), \boldsymbol{q}')$, then $\boldsymbol{q} \xrightarrow{a} \boldsymbol{q}'$ is confluent and (2) if all_conf_strict$(\boldsymbol{q}, (\boldsymbol{t}, a), \boldsymbol{q}')$, then $\boldsymbol{q} \xrightarrow{a} \boldsymbol{q}'$ is strictly confluent. □

Proof. We give the proof for non-strict confluence. The proof for strict confluence is simpler since we have to consider fewer relations of the form $q \xrightarrow{\overline{a}} q'$ where $a = \tau$ and $q = q'$.

For each $i \in \text{act}(\boldsymbol{t})$, we name T_i the smallest confluent set that contains $\boldsymbol{q}[i] \xrightarrow{\boldsymbol{t}[i]}_i \boldsymbol{q}'[i]$. For brevity, we write $\longrightarrow_{T_i}$ instead of $\longrightarrow_{iT_i}$. We also define T as the set of transitions $\{\boldsymbol{p} \xrightarrow{a} \boldsymbol{p}' \mid \text{all\_conf}(\boldsymbol{p}, (\boldsymbol{t}, a), \boldsymbol{p}')\}$. Note that T contains $\boldsymbol{q} \xrightarrow{a} \boldsymbol{q}'$. We show that T is a confluent set, which implies that $\boldsymbol{q} \xrightarrow{a} \boldsymbol{q}'$ is confluent.

Let $\boldsymbol{p}_1 \xrightarrow{a}_T \boldsymbol{p}_2$. By definition of T, $\boldsymbol{p}_2 \in \text{succ}(\boldsymbol{p}_1, (\boldsymbol{t}, a))$ and $\text{all\_conf}(\boldsymbol{p}_1, (\boldsymbol{t}, a), \boldsymbol{p}_2)$ holds. Let $(\boldsymbol{t}', b) \in V$ and $\boldsymbol{p}_3$ be such that $\boldsymbol{p}_3 \in \text{succ}(\boldsymbol{p}_1, (\boldsymbol{t}', b))$, i.e., $\boldsymbol{p}_1 \xrightarrow{b} \boldsymbol{p}_3$. Note that $(\boldsymbol{t}', b)$ and $\boldsymbol{p}_3$ necessarily exist, e.g., $(\boldsymbol{t}', b) = (\boldsymbol{t}, a)$ and $\boldsymbol{p}_3 = \boldsymbol{p}_2$. By Definition 7 and the facts $\boldsymbol{p}_2 \in \text{succ}(\boldsymbol{p}_1, (\boldsymbol{t}, a))$, $\boldsymbol{p}_3 \in \text{succ}(\boldsymbol{p}_1, (\boldsymbol{t}', b))$, and $\text{all\_conf}(\boldsymbol{p}_1, (\boldsymbol{t}, a), \boldsymbol{p}_2)$, we can establish the following:

(P1) If $i \in \text{inact}(\boldsymbol{t})$ then $\boldsymbol{p}_1[i] = \boldsymbol{p}_2[i]$.

(P2) If $i \in \text{act}(\boldsymbol{t})$ then $\boldsymbol{p}_1[i] \xrightarrow{\boldsymbol{t}[i]}_{T_i} \boldsymbol{p}_2[i]$.

(P3) If $i \in \text{inact}(\boldsymbol{t}')$ then $\boldsymbol{p}_1[i] = \boldsymbol{p}_3[i]$.

(P4) If $i \in \text{act}(\boldsymbol{t}')$ then $\boldsymbol{p}_1[i] \xrightarrow{\boldsymbol{t}'[i]}_i \boldsymbol{p}_3[i]$.

We show that there exists $\boldsymbol{p}_4$ such that $\boldsymbol{p}_3 \xrightarrow{\overline{a}}_T \boldsymbol{p}_4$ and $\boldsymbol{p}_2 \xrightarrow{\overline{b}} \boldsymbol{p}_4$. We build $\boldsymbol{p}_4$ as follows:

(Q1) For each $i \in \text{inact}(\boldsymbol{t})$, $\boldsymbol{p}_4[i]$ is defined as $\boldsymbol{p}_3[i]$. We have $\boldsymbol{p}_1[i] = \boldsymbol{p}_2[i]$ (by P1) and:

- **(Q1a)** if $i \in \text{inact}(\boldsymbol{t}')$ then $\boldsymbol{p}_1[i] = \boldsymbol{p}_3[i]$ (by P3), and therefore $\boldsymbol{p}_2[i] = \boldsymbol{p}_4[i]$
- **(Q1b)** if $i \in \text{act}(\boldsymbol{t}')$ then $\boldsymbol{p}_1[i] \xrightarrow{\boldsymbol{t}'[i]}_i \boldsymbol{p}_3[i]$ (by P4), and therefore $\boldsymbol{p}_2[i] \xrightarrow{\boldsymbol{t}'[i]}_i \boldsymbol{p}_4[i]$

(Q2) For each $i \in \text{act}(\boldsymbol{t}) \cap \text{inact}(\boldsymbol{t}')$, $\boldsymbol{p}_4[i]$ is defined as $\boldsymbol{p}_2[i]$.

Since $\boldsymbol{p}_1[i] \xrightarrow{\boldsymbol{t}[i]}_{T_i} \boldsymbol{p}_2[i]$ (by P2), we also have $\boldsymbol{p}_1[i] \xrightarrow{\boldsymbol{t}[i]}_{T_i} \boldsymbol{p}_4[i]$.

(Q3) For each $i \in \text{act}(\boldsymbol{t}) \cap \text{act}(\boldsymbol{t}')$, we have $\boldsymbol{p}_1[i] \xrightarrow{\boldsymbol{t}[i]}_{T_i} \boldsymbol{p}_2[i]$ (by P2) and $\boldsymbol{p}_1[i] \xrightarrow{\boldsymbol{t}'[i]}_i \boldsymbol{p}_3[i]$ (by P4). By definition of confluence, there exists a state q such that $\boldsymbol{p}_2[i] \xrightarrow{\overline{\boldsymbol{t}'[i]}}_i q$ and $\boldsymbol{p}_3[i] \xrightarrow{\overline{\boldsymbol{t}[i]}}_{T_i} q$. $\boldsymbol{p}_4[i]$ is defined as q.

By construction, the following therefore holds:

(R1) For each $i \in \text{inact}(\boldsymbol{t})$, $\boldsymbol{p}_3[i] = \boldsymbol{p}_4[i]$ (by Q1) and for each $i \in \text{act}(\boldsymbol{t})$, $\boldsymbol{p}_3[i] \xrightarrow{\overline{\boldsymbol{t}[i]}}_{T_i} \boldsymbol{p}_4[i]$ (by Q2 and Q3). If for some $i \in \text{act}(\boldsymbol{t})$ we have $\boldsymbol{t}[i] = \tau$ and $\boldsymbol{p}_3[i] = \boldsymbol{p}_4[i]$, then we also have $a = \tau$ and $\boldsymbol{p}_3 = \boldsymbol{p}_4$ due to the network

admissibility conditions (Definition 8). Otherwise, for each $i \in \text{act}(\boldsymbol{t})$, we have $\boldsymbol{p}_3[i] \xrightarrow{\boldsymbol{t}[i]}_{T_i} \boldsymbol{p}_4[i]$, i.e., $\boldsymbol{p}_4 \in \text{succ}(\boldsymbol{p}_3, (\boldsymbol{t}, a))$ (by Definition 7) and $\text{all\_conf}(\boldsymbol{p}_3, (\boldsymbol{t}, a), \boldsymbol{p}_4)$, which implies $\boldsymbol{p}_3 \xrightarrow{a}_T \boldsymbol{p}_4$. Therefore in both cases, $\boldsymbol{p}_3 \xrightarrow{\overline{a}}_T \boldsymbol{p}_4$ holds.

(R2) For each $i \in \text{inact}(\boldsymbol{t}')$, we have $\boldsymbol{p}_2[i] = \boldsymbol{p}_4[i]$ (by Q1a and Q2) and for each $i \in \text{act}(\boldsymbol{t}')$, we have $\boldsymbol{p}_2[i] \xrightarrow{\overline{\boldsymbol{t}'[i]}}_i \boldsymbol{p}_4[i]$ (by Q1b and Q3). If for some $i \in \text{act}(\boldsymbol{t}')$ we have $\boldsymbol{t}'[i] = \tau$ and $\boldsymbol{p}_2[i] = \boldsymbol{p}_4[i]$, then we also have $b = \tau$ and $\boldsymbol{p}_2 = \boldsymbol{p}_4$ due to the network admissibility conditions (Definition 8). Otherwise, for each $i \in \text{act}(\boldsymbol{t}')$, we have $\boldsymbol{p}_2[i] \xrightarrow{\boldsymbol{t}'[i]}_i \boldsymbol{p}_4[i]$, i.e., $\boldsymbol{p}_2 \xrightarrow{b} \boldsymbol{p}_4$ (by Definition 7). Therefore in both cases, $\boldsymbol{p}_2 \xrightarrow{\overline{b}} \boldsymbol{p}_4$ holds.

From $\boldsymbol{p}_3 \xrightarrow{\overline{a}}_T \boldsymbol{p}_4$ (by R1) and $\boldsymbol{p}_2 \xrightarrow{\overline{b}} \boldsymbol{p}_4$ (by R2), we can conclude that T is confluent.

We call *deadlock preserving reduction using* Ccd a prioritization of transitions obtained from synchronization of strictly confluent transitions (which indeed preserves the deadlocks of the system following Theorems 2 and 3), and *branching preserving reduction using* Ccd a prioritization of τ-transitions obtained from synchronization of confluent transitions, provided they are acyclic (which indeed preserves branching bisimulation following Theorems 1 and 3). The major differences between both reductions are thus the following: (1) branching preserving reduction does not require strict confluence; (2) deadlock preserving reduction does not require any acyclicity condition; and (3) deadlock preserving reduction does not require the prioritized transitions to be labeled by τ, which preserves the labels of diagnostic paths leading to deadlock states.

Example 2 Figure 3 depicts three Ltss corresponding to the network presented in Example 1, page 10. Lts (*i*) corresponds to the semantics of the network, generated without reduction. Lts (*ii*) is the same generated with branching preserving reduction using Ccd and thus is branching equivalent to Lts (*i*). Lts (*iii*) is the same generated with deadlock preserving reduction using Ccd and thus has the same deadlock state as Lts (*i*).

As persistent subset methods, Ccd is able to detect commuting transitions by a local analysis of the network. For persistent subsets, a relation of independence between the transitions enabled in the current state is computed dynamically by inspection of the transitions enabled in the individual Ltss and of their interactions (defined here as synchronization rules). By contrast, Ccd performs a static analysis of the individual Ltss to detect which transitions are locally confluent, the dynamic part being limited to checking whether a transition of the network can be obtained by synchronizing only locally confluent transitions.

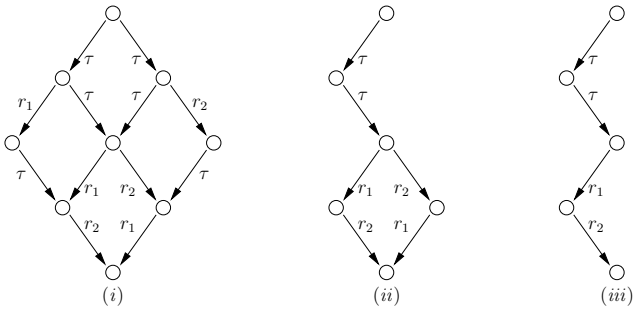

Figure 3: Three LTSs corresponding to the semantics of the network of Example 1, one generated without CCD (*i*) and two generated using CCD preserving respectively branching equivalence (*ii*) and deadlocks (*iii*)

Branching preserving reduction using CCD does not require detection of all confluent transitions in the individual LTSs of the network, but can be restricted to those active in a synchronization rule of the form $(\boldsymbol{t}, \tau)$. In a network $(\boldsymbol{S}, V)$ of length n, we thus compute for each $i \in 1..n$ a subset $C_i \subseteq A_i$ of labels that contains all labels $\boldsymbol{t}[i] \neq \bullet$ such that there exists $(\boldsymbol{t}, \tau) \in V$. For deadlock preserving reduction, the subset C_i is defined as A_i.

The problem of detecting confluence in the individual LTSs is reformulated in terms of the local resolution of a BES (*Boolean Equation System*), following the scheme we proposed in [32].

Definition 10 (Confluence BES) Let $(Q, A, \rightarrow, q_0)$ be an LTS and $C \subseteq A$ a set of actions. The maximal set of confluent transitions labeled by actions $a \in C$ is encoded by the maximal fixed point BES below:

$$B = \left\{ X_{q_1 a q_2} \stackrel{\nu}{=} \bigwedge_{q_1 \xrightarrow{b} q_3} \bigvee_{q_2 \xrightarrow{\overline{b}} q_4} \left(\bigvee_{q_3 \xrightarrow{\overline{a}} q_4'} (q_4 = q_4' \wedge X_{q_3 a q_4}) \vee (a = \tau \wedge q_3 = q_4) \right) \right\}_{q_1, q_2 \in Q, a \in C}$$

Each boolean variable $X_{q_1 a q_2}$, where $q_1, q_2 \in Q_i$ and $a \in C_i$, evaluates to true if and only if $q_1 \xrightarrow{a}_i q_2$ is confluent. The BES has maximal fixed point semantics because we seek to determine the maximal set of confluent transitions contained in an LTS. For strict confluence, $\bigvee_{q_3 \xrightarrow{\overline{a}}_i q_4'}$ must be merely replaced by $\bigvee_{q_3 \xrightarrow{a}_i q_4'}$.

A few additional notions are needed in order to prove the correctness of this encoding. Let $(Q, A, \rightarrow, q_0)$ be an LTS and $C \subseteq A$ a set of actions. Consider the two lattices $\langle \mathsf{Bool}^{|Q|\cdot|C|\cdot|Q|}, \sqsubseteq, \mathsf{false}^{|Q|\cdot|C|\cdot|Q|}, \mathsf{true}^{|Q|\cdot|C|\cdot|Q|}, \sqcup, \sqcap \rangle$ and $\langle 2^{Q \times C \times Q}, \subseteq, \emptyset, Q \times C \times Q, \cup, \cap \rangle$, where the relation $\sqsubseteq$ and the operations $\sqcup$, $\sqcap$ are defined as the pointwise extensions of the boolean connectors $\Rightarrow$, $\vee$, and $\wedge$, respectively. These lattices are isomorphic, being related by the function $\Gamma : \mathsf{Bool}^{|Q|\cdot|C|\cdot|Q|} \rightarrow 2^{Q \times C \times Q}$ defined below:

$$\Gamma(\langle b_{q_1 a q_2} \rangle_{q_1, q_2 \in Q, a \in C}) = \{ q_1 \xrightarrow{a} q_2 \mid b_{q_1 a q_2} = \mathsf{true} \}.$$

Γ is an isomorphism, i.e., it is a bijection preserving the compatibility of operations ($\boldsymbol{b} \sqsubseteq \boldsymbol{b}' \Leftrightarrow \Gamma(\boldsymbol{b}) \subseteq \Gamma(\boldsymbol{b}')$, $\Gamma(\mathsf{false}^{|Q|\cdot|C|\cdot|Q|}) = \emptyset$, $\Gamma(\mathsf{true}^{|Q|\cdot|C|\cdot|Q|}) = Q \times C \times Q$, $\Gamma(\boldsymbol{b} \sqcup \boldsymbol{b}') = \Gamma(\boldsymbol{b}) \cup \Gamma(\boldsymbol{b}')$, and $\Gamma(\boldsymbol{b} \sqcap \boldsymbol{b}') = \Gamma(\boldsymbol{b}) \cap \Gamma(\boldsymbol{b}')$).

The interpretation $[\![B]\!]$ of the BES given in Definition 10 is defined as the maximal fixed point $\nu\Phi$, where $\Phi : \mathsf{Bool}^{|Q|\cdot|C|\cdot|Q|} \to \mathsf{Bool}^{|Q|\cdot|C|\cdot|Q|}$ is the (monotonic) functional associated to B:

$$\begin{aligned}\Phi(\langle b_{q_1 a q_2}\rangle_{q_1,q_2\in Q,a\in C}) \;=\; \langle [\![\textstyle\bigwedge_{q_1 \xrightarrow{b} q_3} \bigvee_{q_2 \xrightarrow{\overline{b}} q_4} (\bigvee_{q_3 \xrightarrow{\overline{a}} q_4'} (q_4 = q_4' \wedge X_{q_3 a q_4}) \vee \\ (a = \tau \wedge q_3 = q_4))]\!] [b_{q_1 a q_2} / X_{q_1 a q_2}] \rangle_{q_1,q_2\in Q,a\in C}.\end{aligned}$$

From Tarski's theorem, the maximal fixed point $\nu\Phi$ can be computed as follows:

$$\nu\Phi = \bigsqcup \{\boldsymbol{b} \in \mathsf{Bool}^{|Q|\cdot|C|\cdot|Q|} \mid \boldsymbol{b} \sqsubseteq \Phi(\boldsymbol{b})\}.$$

The following lemma provides a link between sets of confluent transitions and the functional associated to the BES above.

Lemma 1 *Let $(Q, A, \to, q_0)$ be an* LTS*, $C \subseteq A$ a set of actions, and let $\boldsymbol{b} \in \mathsf{Bool}^{|Q|\cdot|C|\cdot|Q|}$. Then:*

$$\boldsymbol{b} \sqsubseteq \Phi(\boldsymbol{b}) \;\; \textit{if and only if} \;\; \Gamma(\boldsymbol{b}) \textit{ is confluent.}$$

Proof. **If.** Let $\boldsymbol{b} = \langle b_{q_1 a q_2}\rangle_{q_1,q_2\in Q,a\in C}$ such that $\Gamma(\boldsymbol{b})$ is confluent. We must show that $\boldsymbol{b} \sqsubseteq \Phi(\boldsymbol{b})$.

Let $q_1, q_2 \in Q$ and $a \in C$ such that $b_{q_1 a q_2} = \mathsf{true}$. From the definition of Γ, this implies $q_1 \xrightarrow{a} q_2 \in \Gamma(\boldsymbol{b})$. Since $\Gamma(\boldsymbol{b})$ is confluent, from Definition 3 this implies that for all $q_1 \xrightarrow{b} q_3$, there exists $q_4 \in Q$ such that $q_3 \xrightarrow{\overline{a}}_{\Gamma(\boldsymbol{b})} q_4$ and $q_2 \xrightarrow{\overline{b}} q_4$.

Let $q_1 \xrightarrow{b} q_3$ be a transition. From the condition above, there exists $q_4 \in Q$ such that $q_2 \xrightarrow{\overline{b}} q_4$ and $q_3 \xrightarrow{\overline{a}}_{\Gamma(\boldsymbol{b})} q_4$. The latter condition means that $q_3 \xrightarrow{\overline{a}} q_4 \in \Gamma(\boldsymbol{b})$. Two cases are possible: either $q_3 \xrightarrow{a} q_4 \in \Gamma(\boldsymbol{b})$, which from the definition of Γ implies $b_{q_3 a q_4} = \mathsf{true}$, and we can choose $q_4' = q_4$; or $a = \tau$ and $q_3 = q_4$. In either case, the boolean formula

$$\textstyle\bigvee_{q_3 \xrightarrow{\overline{a}} q_4'} (q_4 = q_4' \wedge X_{q_3 a q_4}) \vee (a = \tau \wedge q_3 = q_4)$$

is true, making the right-hand side of the equation defining variable $X_{q_1 a q_2}$ in B true. This means $(\Phi(\boldsymbol{b}))_{q_1 a q_2} = \mathsf{true}$ and, since q_1, q_2, a were chosen arbitrarily, $\boldsymbol{b} \sqsubseteq \Phi(\boldsymbol{b})$.

Only if. Let $\boldsymbol{b} = \langle b_{q_1 a q_2}\rangle_{q_1,q_2\in Q,a\in C}$ such that $\boldsymbol{b} \sqsubseteq \Phi(\boldsymbol{b})$. We must show that $\Gamma(\boldsymbol{b})$ is confluent, i.e., it satisfies Definition 3.

Let $q_1 \xrightarrow{a} q_2 \in \Gamma(\boldsymbol{b})$. From the definition of Γ, this implies $b_{q_1 a q_2} = \mathsf{true}$. Since $\boldsymbol{b} \sqsubseteq \Phi(\boldsymbol{b})$, from Definition 10 and the interpretation of boolean formulas, this implies:

$$[\![\bigwedge_{q_1 \xrightarrow{b} q_3} \bigvee_{q_2 \xrightarrow{\overline{b}} q_4} (\bigvee_{q_3 \xrightarrow{\overline{a}} q'_4} (q_4 = q'_4 \wedge X_{q_3 a q_4}) \vee (a = \tau \wedge q_3 = q_4))]\!] [b_{q_1 a q_2} / X_{q_1 a q_2}]_{q_1, q_2 \in Q, a \in C} = \mathsf{true}.$$

Let $q_1 \xrightarrow{b} q_3$ be a transition. From the condition above, each conjunct associated to such a transition must be true, i.e., there must exist a transition $q_2 \xrightarrow{\overline{b}} q_4$ such that the boolean formula below is true:

$$\bigvee_{q_3 \xrightarrow{\overline{a}} q'_4} (q_4 = q'_4 \wedge X_{q_3 a q_4}) \vee (a = \tau \wedge q_3 = q_4)$$

Two cases are possible: either the first disjunct is true, i.e., there exists $q'_4 \in Q$ such that $q_3 \xrightarrow{\overline{a}} q'_4$, $q_4 = q'_4$, and $b_{q_3 a q_4} = \mathsf{true}$, which from the definition of Γ implies $q_3 \xrightarrow{a} q_4 \in \Gamma(\boldsymbol{b})$; or the second disjunct is true, i.e., $a = \tau$ and $q_3 = q_4$. Since both cases match Definition 3, it follows that $\Gamma(\boldsymbol{b})$ is confluent. □

A useful property of confluent sets is that they are closed under union, i.e., the union of two confluent sets is also confluent. This property can be easily shown for our notion of confluence in the same way it was shown for τ-confluence in [18].

The theorem below states the correctness of the BES encoding of confluence.

Theorem 4 (Correctness of confluence BES) *Let $(Q, A, \rightarrow, q_0)$ be an* LTS*, $C \subseteq A$ a set of actions, and $T \subseteq \rightarrow$ the maximal confluent set of transitions included in $\{q_1 \xrightarrow{a} q_2 \mid a \in C\}$. Then:*

$$\Gamma([\![B]\!]) = T.$$

Proof.

$\Gamma([\![B]\!])$	$= \Gamma(\nu\Phi)$	by Definition 10
	$= \Gamma(\bigsqcup \{\boldsymbol{b} \mid \boldsymbol{b} \in \mathsf{Bool}^{\lvert Q\rvert \cdot \lvert C\rvert \cdot \lvert Q\rvert} \wedge \boldsymbol{b} \sqsubseteq \Phi(\boldsymbol{b})\})$	by Tarski's theorem
	$= \bigcup \{\Gamma(\boldsymbol{b}) \mid \boldsymbol{b} \in \mathsf{Bool}^{\lvert Q\rvert \cdot \lvert C\rvert \cdot \lvert Q\rvert} \wedge \boldsymbol{b} \sqsubseteq \Phi(\boldsymbol{b})\}$	by Γ isomorphism
	$= \bigcup \{\Gamma(\boldsymbol{b}) \mid \boldsymbol{b} \in \mathsf{Bool}^{\lvert Q\rvert \cdot \lvert C\rvert \cdot \lvert Q\rvert} \wedge \Gamma(\boldsymbol{b}) \text{ is confluent}\}$	by Lemma 1
	$= \bigcup \{U \subseteq Q \times C \times Q \mid U \text{ is confluent}\}$	by Γ bijection
	$= T$	by closure under union.

□

5 Implementation

CCD was implemented in CADP[2] (*Construction and Analysis of Distributed Processes*) [12], a toolbox for the design of communication protocols and distributed

[2] `http://www.inrialpes.fr/vasy/cadp`

systems, which offers a wide set of functionalities, ranging from step-by-step simulation to massively-parallel model checking. CADP is designed in a modular way and puts the emphasis on intermediate formats and programming interfaces. Related to the current work are the following:

- BCG is a compact graph format and a set of tools and libraries that allow coding and manipulation of LTSs, including stochastic and probabilistic LTSs, represented explicitly as a list of states and transitions. The BCG_MIN tool allows minimization of BCG graphs modulo strong and branching bisimulation.

- OPEN/CÆSAR [10] is a set of C programming interfaces and libraries dedicated to verification. In particular, OPEN/CÆSAR allows an implicit representation of LTSs in the form of an initial state and a successor state function, which we call an *implicit* LTS. OPEN/CÆSAR draws a clear separation between languages that can be compiled into implicit LTSs, and on-the-fly graph traversal tools, which take implicit LTSs as input. The GENERATOR tool implements a simple breadth-first traversal of the implicit LTS and stores it on disk as an explicit LTS in the BCG format.

- EXP.OPEN 2.0 (an extension of the previous version EXP.OPEN 1.0 of Bozga, Fernandez, and Mounier) is a compiler into OPEN/CÆSAR implicit LTSs of systems made of BCG graphs composed using synchronization vectors and parallel composition, hiding, renaming, and cutting operators taken from the CCS [30], CSP [37], LOTOS [21], E-LOTOS [22], and μCRL [17] process algebras. As an intermediate step, those systems are translated into the network of LTSs model presented in Definition 6. EXP.OPEN 2.0 has several partial order reduction options that allow standard persistent set methods (generalizations of Ramakrishna and Smolka's method presented in [35]) to be applied on-the-fly, among which **-branching** preserves branching bisimulation, **-ratebranching** preserves stochastic branching bisimulation[3], **-deadpreserving** preserves deadlocks, and **-weaktrace** preserves weak trace equivalence (i.e., observable traces).

- CAESAR_SOLVE [28] is a generic library for on-the-fly resolution of BESs, developed within OPEN/CÆSAR. BESs are manipulated by means of their implicit dependency graphs, represented in a way similar to the implicit LTSs of OPEN/CÆSAR. The library currently provides nine resolution algorithms based on various traversals of the dependency graphs (breadth-first search, depth-first search with or without detection of strongly connected components, etc.). CAESAR_SOLVE serves as verification engine for several on-the-fly verification tools of CADP, such as the model checker EVALUATOR, the equivalence checker BISIMULATOR, and the LTS reduction tool REDUCTOR.

[3] This option is similar to **-branching** and additionally gives priority to τ-transitions over stochastic transitions.

We developed in the Exp.Open 2.0 tool a new procedure that takes as input a Bcg graph, a file that contains a set of labels represented using a list of regular expressions, and a boolean parameter for strictness. For each transition whose label matches one of the regular expressions, this procedure checks whether this transition is confluent (or strictly confluent if the boolean parameter is set to true). The Bes encoding the confluence detection problem is solved using a global algorithm based on the same principles as the Caesar_Solve library. This produces as output an Lts in the Bcg format, the transition labels of which are prefixed by a special tag indicating confluence when appropriate.

We also added to Exp.Open 2.0 a new **-confluence** option, which can only be used in combination with one of the partial order reduction options already available (**-branching**, **-deadpreserving**, **-ratebranching**, **-weaktrace**[4]). In this case, Exp.Open 2.0 first computes the labels for which confluence detection is useful, and then calls the above procedure (setting the boolean parameter to true if Exp.Open was called with the **-deadpreserving** option) on the individual Ltss, providing these labels as input. Finally, it uses the information collected in the individual Ltss to prioritize the confluent transitions on the fly.

6 Experimental Results

We applied partial order reductions using Ccd to several examples. To this aim, we used a 2 GHz, 16 GB RAM, dual core AMD Opteron 64-bit computer running 64-bit Linux. Examples identified by a two digit number xy (01, 10, 11, etc.) correspond to Lts compositions extracted from an official Cadp demo available at `ftp://ftp.inrialpes.fr/pub/vasy/demos/demo_`xy. These include telecommunication protocols (01, 10, 11, 18, 20, 27), distributed systems (25, 28, 35, 36, 37), and asynchronous circuits (38). Examples $st(1)$, $st(2)$, and $st(3)$ correspond to process compositions provided to us by the STMicroelectronics company, which uses Cadp to verify critical parts of their future-generation multiprocessor systems on chip.

In each example, the individual Ltss were first minimized (compositionally) modulo branching bisimulation using Bcg_Min. This already achieves more reduction than the compositional τ-confluence technique presented in [32], since minimization modulo branching bisimulation subsumes τ-confluence reduction. The Lts of their composition was then generated using Exp.Open 2.0 and Generator following different strategies: (1) using no partial order reduction at all, (2) using persistent sets, and (3) using both persistent sets and Ccd. Figure 4 reports the size (in states/transitions) of the resulting Lts obtained when using option **-branching** (top) or **-deadpreserving** (bottom). The symbol "–" indicates that the number of states and/or transitions is the same as in the column immediately to the left.

[4] Note that branching preserving reduction using Ccd also preserves weaker relations such as weak trace equivalence.

Branching preserving reduction

Example	No partial order reduction	Persistent sets	Persistent sets + CCD
01	112/380	−/328	−/−
10	688/2,540	−/2,200	−/−
11	2,995/9,228	−/−	−/9,200
18	129,728/749,312	−/746,880	−/−
20	504,920/5,341,821	−/−	−/5,340,117
25	11,031/34,728	−/−	−/−
27(1)	1,530/5,021	−/−	−/−
27(2)	6,315/22,703	−/−	−/−
28	600/1,925	−/−	−/−
35	156,957/767,211	−/−	−/−
36	23,627/84,707	21/20	−/−
37	22,545/158,318	−/−	541/2,809
38	1,404/3,510	−/3,504	390/591
$st(1)$	6,993/100,566	−/−	−/79,803
$st(2)$	1,109,025/7,448,719	−/−	−/6,163,259
$st(3)$	5,419,575/37,639,782	−/−	5,172,660/24,792,525

Deadlock preserving reduction

Example	No partial order reduction	Persistent sets	Persistent sets + CCD
01	112/380	92/194	−/−
10	688/2,540	568/1,332	−/−
11	2,995/9,228	2,018/4,688	−/4,670
18	129,728/749,312	124,304/689,760	90,248/431,232
20	504,920/5,341,821	481,406/4,193,022	481,397/4,191,555
25	11,031/34,728	6,414/11,625	−/−
27(1)	1,530/5,021	1,524/4,811	−/−
27(2)	6,315/22,703	6,298/22,185	−/−
28	600/1,925	375/902	−/−
35	156,957/767,211	−/−	−/−
36	23,627/84,707	171/170	−/−
37	22,545/158,318	−/−	76/128
38	1,404/3,510	−/3,474	492/673
$st(1)$	6,993/100,566	6,864/96,394	1/1
$st(2)$	1,109,025/7,448,719	−/7,138,844	101,575/346,534
$st(3)$	5,419,575/37,639,782	5,289,255/34,202,947	397,360/1,333,014

Figure 4: LTS sizes in states/transitions for branching and deadlock preserving reductions

As a reference of the maximal amount of reductions achievable, Figure 5 provides also, for each example, the size of the LTSs obtained by eliminating all τ-confluence (while still preserving branching bisimulation), using the REDUCTOR tool of CADP.

These experiments show that CCD may improve the reductions obtained using persistent sets and compositional verification, most particularly in examples 37, 38, $st(1)$, $st(2)$, and $st(3)$. Indeed, in these examples the individual LTSs are themselves obtained by parallel compositions of smaller processes. This tends to generate confluent transitions, which are detected locally by CCD. On the other hand, it is not a surprise that neither CCD nor persistent sets methods preserving branching bisimulation reduce examples 25, 27(1), 27(2) and 28, since the resulting LTSs corresponding to these examples contain no confluent transitions.

Example	States / transitions
01	4/4
10	12/20
11	2,836/8,677
18	108,864/683,712
20	439,443/4,627,974
25	11,031/34,728
27(1)	1,530/5,021
27(2)	6,315/22,703
28	600/1,925
35	2/1
36	11/10
37	433/2,276
38	6/7
$st(1)$	2,103/31,428
$st(2)$	709,776/3,729,649
$st(3)$	1,629,825/11,010,192

Figure 5: Maximal reductions achieved when eliminating all confluent τ-transitions

	No partial order reduction		Persistent sets + Ccd	
	total time (s)	peak memory (MB)	total time (s)	peak memory (MB)
$st(1)$	0.72	5.6	0.91	5.6
$st(2)$	271	312	287	271
$st(3)$	2,116	1,390	1,588	981

Figure 6: Resources used to generate and reduce Ltss modulo branching bisimulation

One might be amazed by the reduction of $st(1)$ to an Lts with only one state and one transition in the deadlock preserving case. The reason is that one Lts of the network has a strictly confluent self looping transition that is independent from the other Ltss. Therefore, the network cannot have a deadlock and is reduced by Ccd to this trivial, deadlock-free Lts.

For $st(1)$, $st(2)$, and $st(3)$, we also compared the total time and peak memory needed to generate the product Lts (using Exp.Open 2.0/Generator) and then minimize it modulo branching bisimulation (using Bcg_Min), without using any partial order reduction and with persistent sets combined with Ccd. This includes time and memory used by the tools Exp.Open 2.0, Generator and Bcg_Min. Figure 6 shows that Ccd may significantly reduce the total time and peak memory (for $st(3)$, 30% and 40%, respectively) needed to generate a minimal Lts.

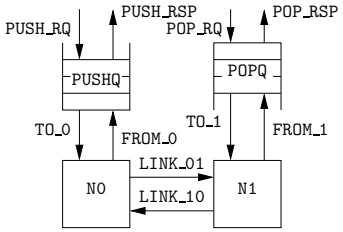

Figure 7: Two xSTream queues connected via a NoC with four routers

	without Ccd			with Ccd		
	intermediate	time	mem.	intermediate	time	mem.
itf_POPQ+N1	244,569/1,320,644	18.56	51	179,706/587,187	9.66	26
N0+PUSHQ	22,674/120,222	1.35	17	22,674/86,528	1.12	17
N0+N1+PUSHQ	140,364/828,930	12.62	32	95,208/444,972	6.40	22
NOC4	324,261/2,549,399	11.32	93	310,026/1,073,316	9.77	46

Figure 8: Performance of Lts generation and minimization with and without Ccd (times are in seconds and memory usage in megabytes)

7 Case study

We present here in more detail the use of Ccd in the context of the Multival project[5], which aims at the formal specification, verification, and performance evaluation of multiprocessor multithreaded architectures developed by Bull, the Cea/Leti, and STMicroelectronics. The case-study below concerns xSTream, a multiprocessor dataflow architecture designed by STMicroelectronics for high performance embedded multimedia streaming applications. In this architecture, computation nodes (e.g., filters) communicate using xSTream queues connected by a NoC (*Network on Chip*) composed of routers connected by direct communication links.

We used as input the network of communicating Ltss produced from a Lotos specification of two xSTream queues connected via a NoC with four routers. The architecture of the system is depicted in Figure 7, where the components `N0` and `N1` denote the routers involved in the communication between `PUSHQ` and `POPQ`, the behaviour of which incorporates perturbations induced by the other two routers of the NoC.

The Lts of the system can be generated and minimized compositionally using the following verification script written in the Svl [11] scripting language of Cadp. This script first generates an interface graph `"itf_POPQ+N1.bcg"`, which is used to constrain the Lts `"N0+PUSHQ.bcg"` using the semi-composition operator "`-|[...]|`" [23]. The Lts of the NoC is constructed by gradually

[5] `http://www.inrialpes.fr/vasy/multival`

composing the component LTSs altogether, hiding internal gates, and minimizing the resulting intermediate LTSs.

```
"itf_POPQ+N1.bcg" = branching reduction of leaf safety reduction of
   hide all but LINK_01, LINK_10 in "N1.bcg" |[TO_1, FROM_1]| "POPQ.bcg";

"N0+PUSHQ.bcg" = branching reduction of
   (hide TO_0, FROM_0 in "PUSHQ.bcg" |[TO_0, FROM_0]| "N0.bcg")
   -|[ LINK_01, LINK_10 ]| "itf_POPQ+N1.bcg";

"N0+N1+PUSHQ.bcg" = branching reduction of
   (hide LINK_01, LINK_10 in "N0+PUSHQ.bcg" |[LINK_01, LINK_10]| "N1.bcg")
   -|[ TO_1, FROM_1 ]| "POPQ.bcg";

"NOC4.bcg" = branching reduction of
   hide TO_1, FROM_1 in "N0+N1+PUSHQ.bcg" |[TO_1, FROM_1]| "POPQ.bcg";
```

The SVL script above was executed first with CCD deactivated, then with CCD activated. For each case, Figure 8 gives the following information: The "*intermediate*" column indicates the size (in states/transitions) of the intermediate LTS generated by the EXP.OPEN tool, before minimization modulo branching bisimulation; The "*time*" and "*mem.*" columns indicate respectively the cumulative time (in seconds) and memory peak (in megabytes) taken by LTS generation (including confluence detection when relevant) and minimization modulo branching bisimulation.

Figure 8 shows that CCD may reduce both the time (the LTSs `"itf_POPQ+NI.bcg"` and `"N0+N1+PUSHQ.bcg"` were generated and minimized twice faster with CCD than without CCD) and memory (`"itf_POPQ+NI.bcg"` and `"NOC4.bcg"` were generated using about half as much memory with CCD as without CCD).

8 Conclusion

CCD (*Compositional Confluence Detection*) is a partial order reduction method that applies to systems of communicating automata. It detects confluent transitions in the product graph, by first detecting the confluent transitions in the individual automata and then analysing their synchronizations. Confluent transitions of the product graph can be given priority over the other transitions, thus yielding graph reductions. We detailed two variants of CCD: one that preserves branching bisimilarity with the product graph, and one that preserves its deadlocks.

CCD was implemented in the CADP toolbox. An encoding of the confluence property using a BES (*Boolean Equation System*) allows the detection of all confluent transitions in an automaton. The existing tool EXP.OPEN 2.0, which supports modeling and verification of systems of communicating automata, was

extended to exploit on-the-fly the confluence detected in the individual automata.

CCD can be combined with both compositional verification and other partial order reductions, such as persistent sets. We presented experimental results showing that CCD may significantly reduce both the size of the system graph and the total time and peak memory needed to generate a minimal graph.

As future work, we plan to combine CCD reductions with distributed graph generation [13] in order to further scale up its capabilities. This distribution can be done both at automata level (by launching distributed instances of confluence detection for each automaton in the network or by performing the confluence detection during the distributed generation of each automaton) and at network level (by coupling CCD with the distributed generation of the product graph).

Acknowledgements. We are grateful to W. Serwe (INRIA/VASY) and to E. Lantreibecq (STMicroelectronics) for providing the specifications of the xSTREAM NOC. We also warmly thank the anonymous FM'09 referees for their useful remarks that greatly helped to improve this paper.

References

[1] André Arnold. MEC: A System for Constructing and Analysing Transition Systems. In Joseph Sifakis, editor, *Proceedings of the 1st Workshop on Automatic Verification Methods for Finite State Systems (Grenoble, France)*, volume 407 of *Lecture Notes in Computer Science*, pages 117–132. Springer Verlag, June 1989.

[2] S.C.C. Blom. Partial τ-Confluence for Efficient State Space Generation. Technical Report SEN–R0123, Centrum voor Wiskunde en Informatica, 2001.

[3] Stefan Blom and Jaco van de Pol. State Space Reduction by Proving Confluence. In *Computer Aided Verification 2002*, volume 2404 of *Lecture Notes in Computer Science*, 2002.

[4] Amar Bouali, Annie Ressouche, Valérie Roy, and Robert de Simone. The Fc2Tools set: a Toolset for the Verification of Concurrent Systems. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the 8th Conference on Computer-Aided Verification (New Brunswick, New Jersey, USA)*, volume 1102 of *Lecture Notes in Computer Science*. Springer Verlag, August 1996.

[5] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A Theory of Communicating Sequential Processes. *Journal of the ACM*, 31(3):560–599, July 1984.

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

[7] Jean-Claude Fernandez. *ALDEBARAN : un système de vérification par réduction de processus communicants*. Thèse de Doctorat, Université Joseph Fourier (Grenoble), May 1988.

[8] Jean-CLaude Fernandez, Claude Jard, Thierry Jéron, and Laurent Mounier. "On the Fly" Verification of Finite Transition Systems. *Formal Methods in System Design*, 1992.

[9] Jean-Claude Fernandez and Laurent Mounier. Verifying Bisimulations "On the Fly". In Juan Quemada, José Manas, and Enrique Vázquez, editors, *Proceedings of the 3rd International Conference on Formal Description Techniques FORTE'90 (Madrid, Spain)*. North-Holland, November 1990.

[10] Hubert Garavel. OPEN/CÆSAR: An Open Software Architecture for Verification, Simulation, and Testing. In Bernhard Steffen, editor, *Proceedings of the First International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'98 (Lisbon, Portugal)*, volume 1384 of *Lecture Notes in Computer Science*, pages 68–84, Berlin, March 1998. Springer Verlag. Full version available as INRIA Research Report RR-3352.

[11] Hubert Garavel and Frédéric Lang. SVL: a Scripting Language for Compositional Verification. In Myungchul Kim, Byoungmoon Chin, Sungwon Kang, and Danhyung Lee, editors, *Proceedings of the 21st IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems FORTE'2001 (Cheju Island, Korea)*, pages 377–392. IFIP, Kluwer Academic Publishers, August 2001. Full version available as INRIA Research Report RR-4223.

[12] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes. In Werner Damm and Holger Hermanns, editors, *Proceedings of the 19th International Conference on Computer Aided Verification CAV'2007 (Berlin, Germany)*, volume 4590 of *Lecture Notes in Computer Science*, pages 158–163. Springer Verlag, July 2007.

[13] Hubert Garavel, Radu Mateescu, Damien Bergamini, Adrian Curic, Nicolas Descoubes, Christophe Joubert, Irina Smarandache-Sturm, and Gilles Stragier. DISTRIBUTOR and BCG_MERGE: Tools for Distributed Explicit State Space Generation. In Holger Hermanns and Jens Palberg, editors, *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2006 (Vienna, Austria)*, volume 3920 of *Lecture Notes in Computer Science*, pages 445–449. Springer Verlag, March–April 2006.

[14] D. Giannakopoulou. *Model Checking for Concurrent Software Architectures*. PhD thesis, Imperial College of Science, Technology and Medicine — University of London — Department of Computer Science, January 1999.

[15] Patrice Godefroid. Using Partial Orders to Improve Automatic Verification Methods. In R. P. Kurshan and E. M. Clarke, editors, *Proceedings of the 2nd Workshop on Computer-Aided Verification (Rutgers, New Jersey, USA)*, volume 3 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 321–340. AMS-ACM, June 1990.

[16] S. Graf, B. Steffen, and G. Lüttgen. Compositional Minimization of Finite State Systems using Interface Specifications. *Formal Aspects of Computation*, 8(5):607–616, September 1996.

[17] J. F. Groote and A. Ponse. The Syntax and Semantics of μCRL. In A. Ponse, C. Verhoef, and S.F.M. van Vlijmen, editors, *Algebra of Communicating Processes'94*, Workshops in Computing Series, pages 26–62. Springer Verlag, 1995.

[18] J.F. Groote and J. van de Pol. State Space Reduction using Partial τ-Confluence. In Mogens Nielsen and Branislav Rovan, editors, *Proceedings of the 25th International Symposium on Mathematical Foundations of Computer Science MFCS'2000 (Bratislava, Slovakia)*, volume 1893 of *Lecture Notes in Computer Science*, pages 383–393, Berlin, August 2000. Springer Verlag. Also available as CWI Technical Report SEN-R0008, Amsterdam, March 2000.

[19] J.F. Groote and M.P.A. Sellink. Confluence for process verification. *Theoretical Computer Science*, 170(1–2):47–81, December 1996.

[20] G.J. Holzmann. On-The-Fly Model Checking. *ACM Computing Surveys*, 28(4), 1996.

[21] ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève, September 1989.

[22] ISO/IEC. Enhancements to LOTOS (E-LOTOS). International Standard 15437:2001, International Organization for Standardization — Information Technology, Genève, September 2001.

[23] Jean-Pierre Krimm and Laurent Mounier. Compositional State Space Generation from LOTOS Programs. In Ed Brinksma, editor, *Proceedings of TACAS'97 Tools and Algorithms for the Construction and Analysis of Systems (University of Twente, Enschede, The Netherlands)*, volume 1217 of *Lecture Notes in Computer Science*, Berlin, April 1997. Springer Verlag. Extended version with proofs available as Research Report VERIMAG RR97-01.

[24] Frédéric Lang. EXP.OPEN 2.0: A Flexible Tool Integrating Partial Order, Compositional, and On-the-fly Verification Methods. In Jaco van de Pol, Judi Romijn, and Graeme Smith, editors, *Proceedings of the 5th International Conference on Integrated Formal Methods IFM'2005 (Eindhoven, The Netherlands)*, volume 3771 of *Lecture Notes in Computer Science*, pages 70–88. Springer Verlag, November 2005. Full version available as INRIA Research Report RR-5673.

[25] Frédéric Lang. Refined Interfaces for Compositional Verification. In Elie Najm, Jean-François Pradat-Peyre, and Véronique Viguié Donzeau-Gouge, editors, *Proceedings of the 26th IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems FORTE'2006 (Paris, France)*, volume 4229 of *Lecture Notes in Computer Science*, pages 159–174. Springer Verlag, September 2006. Full version available as INRIA Research Report RR-5996.

[26] J. Malhotra, S. A. Smolka, A. Giacalone, and R. Shapiro. A Tool for Hierarchical Design and Simulation of Concurrent Systems. In *Proceedings of the BCS-FACS Workshop on Specification and Verification of Concurrent Systems (Stirling, Scotland)*, pages 140–152, Swinton, UK, July 1988. British Computer Society.

[27] Radu Mateescu. A Generic On-the-Fly Solver for Alternation-Free Boolean Equation Systems. In Hubert Garavel and John Hatcliff, editors, *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2003 (Warsaw, Poland)*, volume 2619 of *Lecture Notes in Computer Science*, pages 81–96. Springer Verlag, April 2003. Full version available as INRIA Research Report RR-4711.

[28] Radu Mateescu. CAESAR_SOLVE: A Generic Library for On-the-Fly Resolution of Alternation-Free Boolean Equation Systems. *Springer International Journal on Software Tools for Technology Transfer (STTT)*, 8(1):37–56, February 2006. Full version available as INRIA Research Report RR-5948, July 2006.

[29] Radu Mateescu and Mihaela Sighireanu. Efficient On-the-Fly Model-Checking for Regular Alternation-Free Mu-Calculus. *Science of Computer Programming*, 46(3):255–281, March 2003.

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

[31] Ratan Nalumasu and Ganesh Gopalakrishnan. An Efficient Partial Order Reduction Algorithm with an Alternative Proviso Implementation. *Formal Methods in System Design*, 20(3), May 2002.

[32] Gordon Pace, Frédéric Lang, and Radu Mateescu. Calculating τ-Confluence Compositionally. In Jr Warren A. Hunt and Fabio Somenzi, editors, *Proceedings of the 15th International Conference on Computer Aided Verification CAV'2003 (Boulder, Colorado, USA)*, volume 2725 of *Lecture Notes in*

Computer Science, pages 446–459. Springer Verlag, July 2003. Full version available as INRIA Research Report RR-4918.

[33] D.A. Peled. Combining partial order reduction with on-the-fly model-checking. In *Computer Aided Verification 1994*, volume 818 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[34] D.A. Peled, V.R. Pratt, and G.J. Holzmann, editors. *Partial Order Methods in Verification*, volume 29 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1997.

[35] Y.S. Ramakrishna and S.A. Smolka. Partial-Order Reduction in the Weak Modal Mu-Calculus. In A. Mazurkiewicz and J. Winkowski, editors, *Proceedings of the 8th International Conference on Concurrency Theory CONCUR'97*, volume 1243 of *Lecture Notes in Computer Science*, pages 5–24. Springer Verlag, 1997.

[36] Andrew W. Roscoe, Poul H.B. Gardiner, Michael H. Goldsmith, Jason R. Hulance, David M. Jackson, and J. Bryan Scattergood. Hierarchical compression for model-checking CSP *or* how to check 10^{20} dining philosophers for deadlock. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 1995.

[37] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1998.

[38] K. C. Tai and V. Koppol. Hierarchy-Based Incremental Reachability Analysis of Communication Protocols. In *Proceedings of the IEEE International Conference on Network Protocols (San Francisco, CA)*, pages 318–325, Piscataway, NJ, October 1993. IEEE Press.

[39] A. Valmari. A Stubborn Attack on State Explosion. In R. P. Kurshan and E. M. Clarke, editors, *Proceedings of the 2nd Workshop on Computer-Aided Verification (Rutgers, New Jersey, USA)*, volume 3 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 25–42. AMS-ACM, June 1990.

[40] A. Valmari. Stubborn Set Methods for Process Algebras. In *Workshop on Partial Order Methods in Verification*, volume 29 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1997.

[41] Antti Valmari. Compositional State Space Generation. In *Proceedings of Advances in Petri Nets*, volume 674 of *Lecture Notes in Computer Science*, pages 427–457. Springer Verlag, 1993.

[42] 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. Also in proc. IFIP 11th World Computer Congress, San Francisco, 1989.

[43] W. J. Yeh. *Controlling State Explosion in Reachability Analysis*. PhD thesis, Software Engineering Research Center (SERC) Laboratory, Purdue University, December 1993. Technical Report SERC-TR-147-P.

[44] Mingsheng Ying. Weak confluence and τ-inertness. *Theoretical Computer Science*, 238(1–2):465–475, May 2000.

Centre de recherche INRIA Grenoble – Rhône-Alpes
Inovallée, 655, avenue de l'Europe, Montbonnot - 38334 Saint Ismier Cedex (France)

Centre de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes, 4, rue Jacques Monod - Bât. G - 91893 Orsay Cedex (France)
Centre de recherche INRIA Nancy – Grand Est : 615, rue du Jardin Botanique - 54600 Villers-lès-Nancy (France)
Centre de recherche INRIA Rennes – Bretagne Atlantique : Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399