Refined Interfaces for Compositional Verification

Frédéric Lang

Inria Rhône-Alpes / Vasy
655 avenue de l'Europe, 38 334 St Ismier Cedex, France
Phone: +33 (0)4 76 61 55 11; Fax: +33 (0)4 76 61 52 52
Frederic.Lang@inria.fr

Abstract. The compositional verification approach of Graf & Steffen aims at avoiding state space explosion for individual processes of a concurrent system. It relies on interfaces that express the behavioural constraints imposed on each process by synchronization with the other processes, thus preventing the exploration of states and transitions that would not be reachable in the global state space. Krimm & Mounier, and Cheung & Kramer proposed two techniques to generate such interfaces automatically. In this paper, we propose a refined interface generation technique, in which the interface of a process is derived automatically from the examination of (a subset of) concurrent processes. This technique is applicable to formalisms in which concurrent processes are composed either using synchronization vectors or process algebra parallel composition operators (including those of Ccs, Csp, μCrl, Lotos, and E-Lotos), for which we developed a tool. Several experiments indicate state space reductions by more than two orders of magnitude for the largest processes.

1 Introduction

Enumerative verification is a popular technique that consists in exploring and checking reachable states and transitions of a concurrent system. It is confronted with the *state explosion* problem, which occurs when the number of states grows exponentially as the number of concurrent processes increases. To avoid or reduce state explosion, various approaches have been proposed, among which symbolic verification, on-the-fly verification, partial order reductions, symmetries, data-flow analysis, and compositional verification. This paper deals with the latter approach, which assumes that the concurrent system under study can be expressed as a collection of communicating sequential processes, the behaviours of which are modeled as finite state machines or Ltss (*Labelled Transition Systems*). The sequential processes are composed in parallel, either in a flat or hierarchical manner.

In its simplest forms [10,28,32,38,33,34,36,31], compositional verification (also called incremental reduction [32], incremental reachability analysis [33,34], compositional state space generation [36], or inductive compression [31]) consists in replacing each sequential process by an *abstraction*, simpler than the original process but still preserving the properties to be verified on the whole system.

E. Najm et al. (Eds.): FORTE 2006, LNCS 4229, pp. 159–174, 2006.

Quite often, abstracting a process is done by minimizing its corresponding LTS modulo an appropriate equivalence or preorder relation (e.g., a bisimulation relation, such as strong, branching, or observational equivalence). If the system has a hierarchical structure, minimization can also be applied at every intermediate level in the hierarchy. Although this simple form of compositional verification has been applied successfully to some complex systems (e.g., [11,5] in the case of the LOTOS language [22]), it may be counter-productive in some other cases: generating the LTS of each process separately may lead to state explosion, whereas the generation of the whole system of concurrent processes might succeed if processes constrain each other when composed in parallel. Indeed, there may be many states of a process that, although useful in a general environment, are useless (i.e., never explored) in a particular environment.

This issue has been addressed by enhanced compositional verification approaches [19,7,37,8,9,18,26,6,16], which permit the generation of the LTS of an individual process by taking into account *interface constraints* (also known as *environment constraints* or *context constraints*). These constraints express the behavioural restrictions imposed on the considered process by synchronization with its neighbour processes. Taking into account the environment of a process permits local elimination of states and transitions unreachable in the LTS of the whole system.

In general, interface constraints are expressed in the form of an LTS simply called *interface*. There exist two approaches to restrict the behaviour of a process w.r.t. an interface. In the first one, the process is composed in parallel with the interface, which must have been transformed beforehand so that the composition does not affect the global behaviour of the system (a property known as *context transparency*) [6,7,8,9]. This approach is supported in the framework of CSP by the TRACTA tool [16]. In the second approach, the process is constrained using a specific *semi-composition* operator [19,18,26], which cuts the process states and transitions that cannot be reached when considering the traces of the interface as the only possible interactions between the process and its environment. This approach is supported in the framework of LOTOS by the PROJECTOR [26] and SVL [12] tools of CADP (*Construction and Analysis of Distributed Processes*) [13] and was used in the verification of an industrial protocol [35].

Interfaces can be either written by the user (and possibly checked automatically [26]), or generated automatically. Although automated generation has the neat advantage to relieve users from the burden of calculating appropriate constraints, existing automated interface generation techniques undergo two main limitations: first, they are specific to a given composition operator and thus not directly applicable in the framework of concurrent languages featuring different and/or more general operators; second, as already observed in [7], they may fail to capture effective interface constraints due to deficiencies in their analysis of synchronizations between processes[1].

In this paper, we propose to generate interfaces automatically using a new technique that relies on a translation of the system into an intermediate

[1] See in particular Examples 2 and 3, Section 3 of this paper.

concurrent model, named *network of* LTSs, which describes the synchronization between processes in a flat manner. This intermediate representation permits the derivation of effective interface constraints imposed on a given process by a set of its neighbour processes automatically, independently of the hierarchy of processes and of the nature of the composition operators. This permits combination of constraints induced by distant processes, and improvement of the accuracy of interfaces by exploiting more precisely the synchronizations between processes. For this reason, we qualify as *refined* the interfaces generated using this technique.

As regards practical aspects, we implemented refined interface generation in the EXP.OPEN 2.0 tool for on-the-fly verification of networks of LTSs [27] of CADP. Interfaces can be generated automatically from systems made of LTSs composed using operators taken from several languages (CCS [29], CSP [4], μCRL [21], LOTOS [22], the E-LOTOS international standard [24], and general concurrent specification formalisms). In the framework of LOTOS specifications, the SVL scripting language was also extended to facilitate the combined use of the various CADP tools involved to use refined interfaces in a compositional verification task. For behavioural restriction, we rely on PROJECTOR and its semi-composition operator, which is general enough to be applicable in the framework of the above concurrent languages, although originally designed for LOTOS.

Using a flat intermediate concurrent model such as networks of LTSs is not new, as most model-checkers start by flattening the process hierarchy, for instance generating an intermediate Petri net [14] in the case of LOTOS, *Linear Process Equations* in the case of μCRL [20], or using a *supercombinator*-based compilation mechanism called *supercompilation* [17] in the case of CSP. The model we use in this paper is close to MEC *synchronization vectors* [1] and FC2 *synchronization networks* [3]. The originality of our work resides in both the treatment we make on the intermediate model to generate interfaces, and the effective use of this model to handle many different operators in a compositional verification setting.

This paper focuses on communication by *rendez-vous* between processes which run asynchronously (i.e., at independent speeds). It naturally generalizes to communication through bounded buffers if buffers are represented as finite processes communicating by *rendez-vous* with the rest of the system[2]. The current approach can be used to constrain such buffers in the same way as any process. Approaches to constrain processes communicating through buffers that are not bounded *a priori* (i.e., the bound of each buffer, if any, is not known statically but determined at execution time) have been proposed [25] but are out of the scope of this paper.

The paper is organized as follows: Section 2 presents the technical background. Section 3 recalls semi-composition and discusses the limitations of existing interface generation methods. Section 4 defines refined interface generation, which

[2] See `http://www.inrialpes.fr/vasy/cadp/case-studies` which references more than 80 case studies in various application domains, many of which use bounded buffers.

improves over existing interface generation methods. Section 5 describes the implementation of refined interface generation in CADP. Section 6 presents some experimental results. Section 7 finally concludes.

2 Technical Background

Definition 1 (Vectors). A *vector* of *length* n over a set S is an element of S^n, written $\boldsymbol{t}$ or $(t_1, \ldots, t_n)$. For $i \in 1..n$, $\boldsymbol{t}[i]$ denotes the ith element t_i of $\boldsymbol{t}$, and $\boldsymbol{t}[i \leftarrow t'_i]$ represents a copy of $\boldsymbol{t}$ where $\boldsymbol{t}[i]$ is replaced by t'_i. Given $t \in S$, we write t^n the vector of length n such that $(\forall i \in 1..n)\ t^n[i] = t$. Given $I \subseteq 1..n$, the *projection* $\boldsymbol{t}_{\downarrow I}$ is defined by: $\boldsymbol{t}_{\downarrow I} = (\boldsymbol{t}[k_1], \ldots, \boldsymbol{t}[k_m])$ where $\{k_i \mid i \in 1..m\} = I$ and $(\forall i < j)\ k_i < k_j$.

Definition 2 (Labelled Transition System). Let $\mathcal{A}$ be a set of symbols called *observable actions*, and $\tau \notin \mathcal{A}$ the *unobservable action*. Given $A \subseteq \mathcal{A}$, we write A_τ the set $A \cup \{\tau\}$. An LTS is a quadruple $S = (Q, A, T, q_0)$, where Q is the set of *states*, $A \subseteq \mathcal{A}$ — also written $act(S)$ — is the set of *observable actions*, $T \subseteq Q \times A_\tau \times Q$ is the *transition relation*, and $q_0 \in Q$ is the *initial state*. As usual, we may write $q_1 \xrightarrow{a}_T q_2$ (or $q_1 \xrightarrow{a} q_2$ when T is clear from the context) instead of $(q_1, a, q_2) \in T$. A *trace* of S is a sequence of actions $a_1 \ldots a_{n \geq 0} \in (A_\tau)^n$, such that $(\exists q_1, \ldots, q_n \in Q)\ (\forall i \in 0..n-1)\ q_i \xrightarrow{a_{i+1}}_T q_{i+1}$ (note that the sequence starts in the initial state q_0 of S). An *observable trace* is a trace in which all occurrences of τ have been removed. We write $\mathcal{L}(S)$ the set of observable traces of S. An action $a \in A$ is *reachable* if there is a trace containing a. A state $q \in Q$ is *reachable* if there exists a trace such that $q_n = q$. A transition $(q_1, a, q_2) \in T$ is *reachable* if q_1 is reachable. Two LTSs S_1, S_2 are *equal*, written $S_1 = S_2$, if and only if they have the same initial states and reachable transitions.

3 Semi-composition

Semi-composition [26] (implemented in the PROJECTOR tool of CADP) permits restriction of the behaviour of a process *on-the-fly* by taking into account interface constraints, usually derived from its environment. Since semi-composition was designed in the framework of LOTOS, its definition is tightly related to the following LOTOS-like parallel composition and hiding operators.

Definition 3 (Parallel Composition, Hiding). Let $S_i = (Q_i, A_i, T_i, q_{0_i})$ $(i = 1, 2)$ be two LTSs, and $A \subseteq \mathcal{A}$. The *parallel composition* "$S_1 \parallel_A S_2$" models the concurrent execution of S_1 and S_2 with forced synchronization on A. It is defined as the LTS $(Q, A_1 \cup A_2, T, (q_{01}, q_{02}))$, where Q and T are the smallest sets satisfying both $(q_{01}, q_{02}) \in Q$ and the following properties:

$$\frac{(q_1, q_2) \in Q,\ q_1 \xrightarrow{a}_{T_1} q'_1,\ q_2 \xrightarrow{a}_{T_2} q'_2,\ a \in A}{(q'_1, q'_2) \in Q,\ (q_1, q_2) \xrightarrow{a}_T (q'_1, q'_2)}$$

$$\frac{(q_1, q_2) \in Q,\ q_1 \xrightarrow{a}_{T_1} q'_1,\ a \notin A}{(q'_1, q_2) \in Q,\ (q_1, q_2) \xrightarrow{a}_T (q'_1, q_2)} \qquad \frac{(q_1, q_2) \in Q,\ q_2 \xrightarrow{a}_{T_1} q'_2,\ a \notin A}{(q_1, q'_2) \in Q,\ (q_1, q_2) \xrightarrow{a}_T (q_1, q'_2)}$$

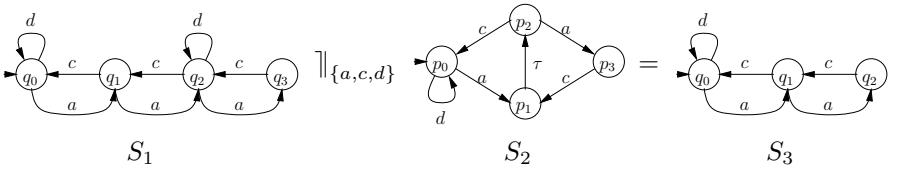
Note that, by construction, the states belonging to Q are reachable. A state p of S_1 (respectively S_2) is said *reachable* in $S_1 \parallel_A S_2$ if there is a state (p, q) (resp. (q, p)) in $S_1 \parallel_A S_2$. Similarly, a transition $p \xrightarrow{a} p'$ of S_1 (respectively S_2) is said *reachable* in $S_1 \parallel_A S_2$ if there is a transition $(p, q) \xrightarrow{a} (p', q')$ (resp. $(q, p) \xrightarrow{a} (q', p')$) in $S_1 \parallel_A S_2$. The expression "hide A in S_1" denotes the LTS $(Q_1, A_1 \setminus A, T_1', q_{01})$, where T_1' is defined as follows:

$$\frac{q \xrightarrow{a}_{T_1} q',\ a \in A}{q \xrightarrow{\tau}_{T_1'} q'} \qquad \frac{q \xrightarrow{a}_{T_1} q',\ a \notin A}{q \xrightarrow{a}_{T_1'} q'}$$

Semi-composition takes as input two LTSs S_1, S_2 and a set of actions A, and returns the LTS which contains exactly the states and transitions of S_1 that are reachable in $S_1 \parallel_A S_2$.

Definition 4 (Semi-Composition). Let $S_i = (Q_i, A_i, T_i, q_{0i})$ $(i = 1, 2)$ be two LTSs, $A \subseteq \mathcal{A}$, and $(Q', A', T', q_0') = S_1 \parallel_A S_2$. The *semi-composition of S_1 and S_2*, written "$S_1 \between_A S_2$", is the LTS (Q, A_1, T, q_{01}), where $Q = \{p \mid (p, q) \in Q'\}$ and $T = T_1 \cap \{(p_1, a, p_2) \mid (p_1, q_1) \xrightarrow{a}_{T'} (p_2, q_2)\}$. A is called the *synchronization set* and the pair (A, S_2) is called the *interface*[3]. We say that an action $a \in A_1$ is *controlled* by the interface (A, S_2) if $a \in A$.

Example 1. The following holds:



State q_3 and transitions $q_2 \xrightarrow{d} q_2, q_2 \xrightarrow{a} q_3$, and $q_3 \xrightarrow{c} q_2$ do not belong to S_3 because they are not reachable in $S_1 \parallel_{\{a,c,d\}} S_2$.

Three properties of semi-composition are essential to ensure its practicability:

- Semi-composition is a state space reduction, since the sets of states and transitions of $S_1 \between_A S_2$ are by definition subsets of S_1. The worst case is when $\mathcal{L}(\text{hide } (\mathcal{A} \setminus A) \text{ in } S_1) \subseteq \mathcal{L}(\text{hide } (\mathcal{A} \setminus A) \text{ in } S_2)$, yielding $S_1 \between_A S_2 = S_1$.
- $(S_1 \between_A S_2) \parallel_A S_2 = S_1 \parallel_A S_2$. Therefore semi-composition can be used to reduce S_1 given its environment S_2 by removing the unreachable states and transitions, without losing any temporal property of the system $S_1 \parallel_A S_2$. Note that, unlike Cheung & Kramer's approach, which requires that the interface be context transparent — and thus be transformed into a deterministic LTS using a well-known but expensive algorithm — no restriction is made here on the shape of S_2.

[3] This definition of semi-composition is simpler but equivalent to that given in [26].

– $S_1 \rceil\!\|_A S_2 = S_1 \rceil\!\|_A S_2'$ if $\mathcal{L}(\text{hide } (\mathcal{A} \setminus A) \text{ in } S_2) = \mathcal{L}(\text{hide } (\mathcal{A} \setminus A) \text{ in } S_2')$. Therefore, reductions of the interface can be achieved by first hiding uncontrolled actions and then minimizing the LTS modulo a relation preserving observable traces (e.g., *safety equivalence* [2]), which permits reduction of the number of states to explore while calculating semi-composition. Safety minimization is less expensive than determinization and, unlike determinization which can induce a dramatic growth of the LTS, yields an LTS that contains fewer states than the input. Minimization of the interface is not mandatory but important to reduce the cost of semi-composition, the complexity of which is the same as parallel composition, hence sensitive to the size of its operands.

In practice, the equation $S_1 \|_A S_2 = (S_1 \rceil\!\|_A S_2) \|_A S_2$ is not sufficient to compute interfaces in the case of systems consisting of more than two LTSs: it may happen that S_2 does not constrain S_1 but that a more distant LTS in the environment of S_1 does. Krimm & Mounier proposed a method to compute an exact interface in the framework of more general systems of communicating LTSs built upon parallel composition and action hiding. Given two LTSs S_1 and S_2 in such a system, this method permits to synthesize a synchronization set A such that S_1 can be replaced by $S_1 \rceil\!\|_A S_2$ without changing the global LTS of the system. It is defined inductively, based on the following semi-composition laws:

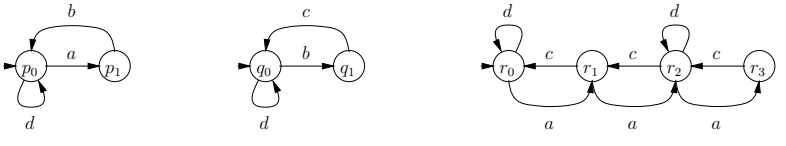
$$S_1 \|_A S_2 = (S_1 \rceil\!\|_A S_2) \|_A S_2 \tag{1}$$

$$(S_1 \|_{A_1} S_3) \|_{A_2} S_2 = ((S_1 \rceil\!\|_B S_2) \|_{A_1} S_3) \|_{A_2} S_2 \quad \text{where } B = A_2 \cap (A_1 \cup (act(S_1) \setminus act(S_3))) \tag{2}$$

$$(\text{hide } A_1 \text{ in } S_1) \|_{A_2} S_2 = (\text{hide } A_1 \text{ in } (S_1 \rceil\!\|_{A_2 \setminus A_1} S_2)) \|_{A_2} S_2 \tag{3}$$

Unfortunately, the interface (A, S_2) built using Krimm & Mounier's method generally does not give the best account of environment constraints, as illustrated by the following two examples.

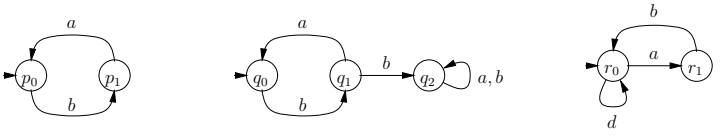
Example 2. Let $E = S_1 \|_{\{a,b,d\}} (S_2 \|_{\{c,d\}} S_3)$ with S_1, S_2, and S_3 as follows:



According to the semi-composition laws, S_3 can be replaced in E either by $S_3 \rceil\!\|_{\{a,d\}} S_1$, or by $S_3 \rceil\!\|_{\{c,d\}} S_2$, but both expressions result in S_3 itself. Yet, one can see that actions a and c are executed with some alternation in E, due to the mandatory synchronization on b between S_1 and S_2. As a consequence, state r_3 is not reachable in E. To capture such a constraint, it should be possible to build an interface that takes simultaneously into account the constraints induced by

both S_1 and S_2, even though there is no sub-expression of E containing S_1 and S_2 only. This is not possible with Krimm & Mounier's method[4].

Example 3. Let $E = S_1 \parallel_{\{a,b\}} (S_2 \parallel_{\{a\}} S_3)$ with S_1, S_2, and S_3 as follows:



S_1 S_2 S_3

According to the semi-composition laws, S_2 can be replaced by $S_2 \rceil\!\rceil_{\{a\}} S_1$, but this expression yields S_2 itself. Yet, it is clear from S_1 and the synchronizations in E that state q_2 of S_2 is unreachable in E, as two successive b actions cannot be fired without an a in between. A better interface should permit to take into account the environment constraints due to synchronizations on b, even though every b of S_1 does not necessarily synchronize with a b of S_2. Unfortunately, this is not possible using the Krimm & Mounier's method[5].

In the sequel, we propose to generate interface constraints automatically in a way that palliates these limitations.

4 Refined Interface Generation

Refined interface generation is a new method that permits the computation of an interface capturing the constraints imposed on a given process P in a concurrent system by one or several processes of its environment. This interface can then be semi-composed with P on-the-fly, so as to restrict P's behaviour.

As regards the model of concurrency on which we establish our results, we use the following network model named "*network of* LTSs", in which the composition hierarchy is completely flattened. The network of LTSs model is more general than the parallel composition operator defined in the previous section, and the parallel composition, renaming, hiding and cutting operators from many process algebras can be translated into networks of LTSs [27]. Networks of LTSs thus make our work non-specific to a particular process algebra and permit an easier way of reasoning about the synchronization structure of systems.

Definition 5 (Network of LTSs). Let $\bullet \notin \mathcal{A}_\tau$ be a special symbol denoting that a particular LTS has no role in a given synchronization. A *synchronization rule* is a pair $(\boldsymbol{t}, a)$, where $\boldsymbol{t}$ is a vector over $\mathcal{A}_\tau \cup \{\bullet\}$ (called a *synchronization vector*) and $a \in \mathcal{A}_\tau$. The components $\boldsymbol{t}$ and a are called respectively the left- and

[4] This limitation holds similarly for Cheung & Kramer's method, as mentioned in [7].

[5] Cheung & Kramer do not provide a solution to this issue as their method relies on a CSP-like parallel composition operator whose semantics states that synchronization on b is mandatory between all processes containing b in their action set.

right-hand sides of the synchronization rule. A *network of* LTSs (or simply *network*) N of *dimension* $n > 0$ is a pair $(\boldsymbol{S}, V)$ where $\boldsymbol{S}$ is a vector of LTSs of length n and V is a set of synchronization rules, whose left-hand sides are all of length n. Each left-hand side $\boldsymbol{t}$ expresses a synchronization constraint on $\boldsymbol{S}$, all components $\boldsymbol{S}[i]$ where $\boldsymbol{t}[i] \neq \bullet$ having to take a transition labeled respectively $\boldsymbol{t}[i]$ altogether so that a transition labeled with the corresponding right-hand side a be generated in the product. More formally, let $\boldsymbol{S}[i] = (Q_i, A_i, T_i, q_{0_i})$ $(i \in 1..n)$. To $N = (\boldsymbol{S}, V)$ corresponds an LTS $(Q, A, T, \boldsymbol{q_0})$, written $sem(N)$ or $sem(\boldsymbol{S}, V)$, such that $A = \{a \mid (\boldsymbol{t}, a) \in V\}$, $\boldsymbol{q_0} = (q_{0_1}, \ldots, q_{0_n})$, and Q and T are the smallest sets satisfying both $\boldsymbol{q_0} \in Q$ and:

$$\frac{\boldsymbol{q} \in Q,\ (\boldsymbol{t}, a) \in V,\ (\forall i \in 1..n)\ (\boldsymbol{t}[i] = \bullet \wedge \boldsymbol{q'}[i] = \boldsymbol{q}[i]) \vee \boldsymbol{q}[i] \xrightarrow{\boldsymbol{t}[i]}_{T_i} \boldsymbol{q'}[i]}{\boldsymbol{q'} \in Q,\ (\boldsymbol{q}, a, \boldsymbol{q'}) \in T}$$

Note that, by construction, the states that belong to Q are reachable. Synchronization rules must obey the following *admissibility* properties, which forbid cutting, synchronizations and renaming of τ transitions and therefore ensure that safety equivalence and stronger relations (e.g., observational, branching, and strong equivalences) are congruences for networks of LTS [27]:

$$((\exists i \in 1..n)\ \tau \text{ is reachable in } \boldsymbol{S}[i]) \implies (\exists (\boldsymbol{t}, \tau) \in V)\ \boldsymbol{t}[i] = \tau$$
$$(\forall (\boldsymbol{t}, a) \in V)\ ((\exists i \in 1..n)\ \boldsymbol{t}[i] = \tau) \implies (a = \tau \wedge (\forall j \in 1..n \setminus \{i\})\ \boldsymbol{t}[j] = \bullet)$$

Example 4. Systems of communicating LTSs built upon various operators can be translated into networks of LTSs. As an example, given S_1 and S_2, the parallel composition $(S_1 \parallel_A S_2)$ can be translated into $((S_1, S_2), V_{sync} \cup V_{async})$, where:

$$V_{sync} = \{((a, a), a) \mid a \in act(S_1) \cap act(S_2) \cap A\}$$
$$V_{async} = \{((a, \bullet), a) \mid a \in act(S_1)_\tau \setminus A\} \cup \{((\bullet, a), a) \mid a \in act(S_2)_\tau \setminus A\}$$

Given a network $N = (\boldsymbol{S}, V)$ and an LTS $\boldsymbol{S}[k]$ in this network, we address the problem of computing automatically an interface of the form $(\mathcal{A}, C)$ that will permit reduction of $\boldsymbol{S}[k]$ by taking into account its interactions with a subset $\{\boldsymbol{S}[i] \mid i \in I\}$ $(k \notin I)$ of LTSs in its environment. The goal is to permit the replacement of LTS $\boldsymbol{S}[k]$ by LTS $\boldsymbol{S}[k] \rceil\!\rceil_{\mathcal{A}} C$ in N without affecting the LTS of the global system. To this aim, we define the following refined interface generation procedure, whose inputs are N, k, and I. The refined interface generated consists of a product of the LTSs $\boldsymbol{S}[i]$ $(i \in I)$, synchronized by synchronization rules derived systematically from the synchronization rules of N, each rule $(\boldsymbol{t}, a)$ being transformed into a rule $(\boldsymbol{t}_{\downarrow I}, \boldsymbol{t}[k])$ if $\boldsymbol{t}[k] \neq \bullet$, or $(\boldsymbol{t}_{\downarrow I}, \tau)$ otherwise. Therefore, whenever a transition $\boldsymbol{q} \xrightarrow{a} \boldsymbol{q'}$ can be fired in $sem(N)$ using a synchronization rule $(\boldsymbol{t}, a)$ with $\boldsymbol{t}[k] \neq \bullet$, then the participating transition $\boldsymbol{q}[k] \xrightarrow{\boldsymbol{t}[k]} \boldsymbol{q'}[k]$ of $\boldsymbol{S}[k]$ is also a transition of $\boldsymbol{S}[k] \rceil\!\rceil_{\mathcal{A}} C$. Conversely, transitions of $\boldsymbol{S}[k]$ that cannot participate in any mandatory synchronization with C (i.e., the $\boldsymbol{S}[i]$'s) are eliminated by the semi-composition $\boldsymbol{S}[k] \rceil\!\rceil_{\mathcal{A}} C$.

Definition 6 (Refined Interface Generation). Let $\varphi : A_\tau \cup \{\bullet\} \to A_\tau$, defined by $\varphi(\bullet) = \tau$ and $(\forall a \in A_\tau)\ \varphi(a) = a$. Let $N = (\boldsymbol{S}, V)$ be a network

of dimension n, I a set of indices such that $\emptyset \subset I \subset 1..n$, and k an index such that $k \in 1..n \setminus I$. The *refined interface* of $\boldsymbol{S}[k]$ capturing constraints induced by $\{\boldsymbol{S}[i] \mid i \in I\}$, written $\mathit{refint}(N, k, I)$, is the interface $(\mathcal{A}, \mathit{sem}(\boldsymbol{S}_{\downarrow I}, V'))$, where $V' = \{(\boldsymbol{t}_{\downarrow I}, \varphi(\boldsymbol{t}[k])) \mid (\boldsymbol{t}, a) \in V\}$.

Example 5. Consider the network N displayed on the left below, with arbitrary LTSs $S_1, \ldots, S_4$. The refined interface of S_1 capturing constraints induced by S_3 and S_4, written $\mathit{refint}(N, 1, \{3, 4\})$, is the LTS corresponding to the network displayed on the right below. Note the projection on S_3 and S_4, and observe that the right-hand sides of synchronization rules in the result are the elements of column S_1, where $\bullet$ is renamed into τ.

$$\mathit{refint}\left(\left(\begin{array}{c}(S_1,S_2,S_3,S_4),\\ \left\{\begin{array}{l}((a_1,\ a_2,\ a_3,\ a_4),\ a),\\ ((\ \bullet,\ b_2,\ b_3,\ \bullet),\ b),\\ ((\ c_1,\ c_2,\ \bullet,\ \bullet),\ c)\end{array}\right\}\end{array}\right), 1, \{3,4\}\right) = \mathit{sem}\left(\left(\begin{array}{c}(S_3,S_4),\\ \left\{\begin{array}{l}((a_3,\ a_4),\ a_1),\\ ((\ b_3,\ \bullet),\ \tau),\\ ((\ \bullet,\ \bullet),\ c_1)\end{array}\right\}\end{array}\right)\right)$$

The following theorem states that, in an arbitrary network N, any interface $\mathit{refint}(N, k, I)$ can be used to restrict $\boldsymbol{S}[k]$ using semi-composition because the LTS of N and the LTS of N in which $\boldsymbol{S}[k]$ is replaced by its restriction are equal.

Theorem 1. Let $N = (\boldsymbol{S}, V)$ be a network of dimension n, I such that $\emptyset \subset I \subset 1..n$, $k \in 1..n \setminus I$, and $(A, C) = \mathit{refint}(N, k, I)$. If $\boldsymbol{S}' = \boldsymbol{S}\big[k \leftarrow (\boldsymbol{S}[k] \parallel_A C)\big]$ then $\mathit{sem}(\boldsymbol{S}, V) = \mathit{sem}(\boldsymbol{S}', V)$.

Proof. Since $\boldsymbol{S}[k] \parallel_A C$ is a sub-LTS of $\boldsymbol{S}[k]$ by definition of semi-composition, it follows that $\mathit{sem}(\boldsymbol{S}', V)$ is a sub-LTS of $\mathit{sem}(\boldsymbol{S}, V)$. We show that, conversely, $\mathit{sem}(\boldsymbol{S}, V)$ is a sub-LTS of $\mathit{sem}(\boldsymbol{S}', V)$. To this aim, we consider an arbitrary state $\boldsymbol{q}$ reachable in $\mathit{sem}(\boldsymbol{S}, V)$. In a first step we assume that $\boldsymbol{q}_{\downarrow I}$ is reachable in C, $(\boldsymbol{q}[k], \boldsymbol{q}_{\downarrow I})$ is reachable in $\boldsymbol{S}[k] \parallel_{\mathcal{A}} C$, $\boldsymbol{q}[k]$ is reachable in $\boldsymbol{S}'[k]$, $\boldsymbol{q}$ is reachable in $\mathit{sem}(\boldsymbol{S}', V)$ and given a transition $\boldsymbol{q} \xrightarrow{a} \boldsymbol{q}'$ of $\mathit{sem}(\boldsymbol{S}, V)$ induced by a vector $(\boldsymbol{t}, a)$, we show simultaneously that (1) $\boldsymbol{q}'_{\downarrow I}$ is reachable in C, (2) $(\boldsymbol{q}'[k], \boldsymbol{q}'_{\downarrow I})$ is reachable in $\boldsymbol{S}[k] \parallel_{\mathcal{A}} C$, which implies that $\boldsymbol{q}'[k]$ is reachable in $\boldsymbol{S}'[k]$, and (3) $\boldsymbol{q} \xrightarrow{a} \boldsymbol{q}'$ is a transition of $\mathit{sem}(\boldsymbol{S}', V)$, which implies that $\boldsymbol{q}'$ is reachable in $\mathit{sem}(\boldsymbol{S}', V)$. We consider two cases:

- If $\boldsymbol{t}[k] = \bullet$ then by definition $\boldsymbol{q}[k] = \boldsymbol{q}'[k]$ and property (3) is obvious. In addition, by definition of *refint*, the transition $\boldsymbol{q}_{\downarrow I} \xrightarrow{\tau} \boldsymbol{q}'_{\downarrow I}$ belongs to C, which implies properties (1) and (2).
- If $\boldsymbol{t}[k] \neq \bullet$ then by hypothesis $\boldsymbol{q}[k] \xrightarrow{\boldsymbol{t}[k]} \boldsymbol{q}'[k]$ belongs to $\boldsymbol{S}[k]$ and $\boldsymbol{q}_{\downarrow I} \xrightarrow{\boldsymbol{t}[k]} \boldsymbol{q}'_{\downarrow I}$ belongs to C by definition of *refint*, which implies property (1). Therefore, $(\boldsymbol{q}[k], \boldsymbol{q}_{\downarrow I}) \xrightarrow{\boldsymbol{t}[k]} (\boldsymbol{q}'[k], \boldsymbol{q}'_{\downarrow I})$ belongs to $\boldsymbol{S}[k] \parallel_{\mathcal{A}} C$, which implies property (2). By definition of semi-composition, $\boldsymbol{q}[k] \xrightarrow{\boldsymbol{t}[k]} \boldsymbol{q}'[k]$ belongs to $\boldsymbol{S}'[k]$, which implies property (3).

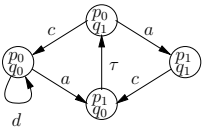
In a second step, given $\boldsymbol{q}_0$ the initial state of $\mathit{sem}(\boldsymbol{S}, V)$, we observe that $\boldsymbol{q}_{0\downarrow I}$, $(\boldsymbol{q}_0[k], \boldsymbol{q}_{0\downarrow I})$, $\boldsymbol{q}_0[k]$, and $\boldsymbol{q}_0$ are the initial states of, respectively, C, $\boldsymbol{S}[k] \parallel_{\mathcal{A}} C$,

$\boldsymbol{S}'[k]$, and $sem(\boldsymbol{S}', V)$. Given a state $\boldsymbol{q}$ reachable in $sem(\boldsymbol{S}, V)$, an induction using properties (1), (2), and (3) shows that $\boldsymbol{q}_{\downarrow I}$, $(\boldsymbol{q}[k], \boldsymbol{q}_{\downarrow I})$, $\boldsymbol{q}[k]$, and $\boldsymbol{q}$ are reachable in, respectively, C, $\boldsymbol{S}[k] \parallel_{\mathcal{A}} C$, $\boldsymbol{S}'[k]$, and $sem(\boldsymbol{S}', V)$. Therefore, every transition of $sem(\boldsymbol{S}, V)$ is also a transition of $sem(\boldsymbol{S}', V)$, which implies that $sem(\boldsymbol{S}, V)$ and $sem(\boldsymbol{S}', V)$ are equal. □

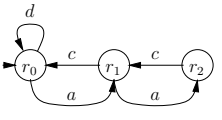
The following examples show that refined interfaces solve the issues raised in Examples 2 and 3 of Section 3.

Example 6 (back to Example 2 page 164). Expression $E = S_1 \parallel_{\{a,b,d\}} (S_2 \parallel_{\{c,d\}} S_3)$ defined in Example 2 can be translated into the network N displayed below. S_3 may be restricted using a refined interface $(\mathcal{A}, sem(N')) = refint(N, 3, \{1, 2\})$ that takes simultaneously both S_1 and S_2 into account, where N' and $sem(N')$ are displayed below. $S_3 \parallel_{\mathcal{A}} sem(N')$, also displayed below, reduces S_3 by eliminating the unreachable state r_3 and transitions $r_2 \xrightarrow{a} r_3, r_3 \xrightarrow{c} r_2$, and $r_2 \xrightarrow{d} r_2$.

$$N = \left(\begin{array}{c} (S_1, S_2, S_3), \\ \left\{ \begin{array}{l} ((\ a,\ \bullet,\ a), a), \\ ((\ b,\ b,\ \bullet), b), \\ ((\ \bullet,\ c,\ c), c), \\ ((\ d,\ d,\ d), d) \end{array} \right\} \end{array} \right) \qquad N' = \left(\begin{array}{c} (S_1, S_2), \\ \left\{ \begin{array}{l} ((\ a,\ \bullet), a), \\ ((\ b,\ b), \tau), \\ ((\ \bullet,\ c), c), \\ ((\ d,\ d), d) \end{array} \right\} \end{array} \right)$$



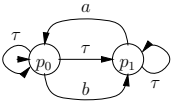
$sem(N')$



$S_3 \parallel_{\mathcal{A}} sem(N')$

Example 7 (back to Example 3 page 165). Expression $E = S_1 \parallel_{\{a,b\}} (S_2 \parallel_{\{a\}} S_3)$ defined in Example 3 can be translated into the network N displayed below. S_2 may be restricted using a refined interface $(\mathcal{A}, sem(N')) = refint(N, 2, \{1\})$ that takes S_1 into account, where N' and $sem(N')$ are displayed below. In practice, $sem(N')$ can be minimized modulo safety equivalence, yielding an LTS with 2 states and 3 transitions. $S_2 \parallel_{\mathcal{A}} sem(N')$ is isomorphic to S_1.

$$N = \left(\begin{array}{c} (S_1, S_2, S_3), \\ \left\{ \begin{array}{l} ((\ a,\ a,\ a),\ a), \\ ((\ b,\ b,\ \bullet),\ b), \\ ((\ b,\ \bullet,\ b),\ b), \\ ((\ \bullet,\ \bullet,\ d),\ d) \end{array} \right\} \end{array} \right) \qquad N' = \left(\begin{array}{c} (S_1), \\ \left\{ \begin{array}{l} ((\ a\ ),\ a), \\ ((\ b\ ),\ b), \\ ((\ b\ ),\ \tau), \\ ((\ \bullet\ ),\ \tau) \end{array} \right\} \end{array} \right)$$



$sem(N')$

This example shows that without using more LTSs from the environment of S_2 than in Example 3, but simply by taking a better account of the synchronization structure of the system, the *refint* operation permits refinement of the interface with respect to that obtained using equation (2), turning the set of observable traces of the interface from a^* with b uncontrolled in Example 3 to $a^* + b + (ba^+)^*$ in the current example. The latter set of traces does not contain any trace with

two consecutive b's, thus disabling the transition $q_1 \xrightarrow{b} q_2$ in S_2 and making state q_2 and transitions $q_2 \xrightarrow{a} q_2$, $q_2 \xrightarrow{b} q_2$ also unreachable.

The *refint* operation may create synchronization rules of the form $(\bullet^n, a)$, which induce a self-looping transition labelled a in each state of the interface (see for instance the last synchronization rule of the right-hand side network in Example 5 and the last synchronization rule of network N' in Example 7, which induces the τ-loops in states p_0 and p_1). Some of these synchronization rules can be eliminated as follows:

- Every synchronization rule of the form $(\bullet^n, \tau)$ can merely be removed. Indeed, for all $\boldsymbol{S}$ and V, $\mathcal{L}(sem(\boldsymbol{S}, V \cup (\bullet^n, \tau))) = \mathcal{L}(sem(\boldsymbol{S}, V))$.
- Every synchronization rule of the form $(\bullet^n, a)$ where $a \neq \tau$ can be removed if the set of synchronization rules does not contain another rule with the same action a as right-hand side. Indeed, for all $\boldsymbol{S}$, S', A, and V in which a does not occur as a right-hand side, $S' \rVert_A \, sem(\boldsymbol{S}, V \cup (\bullet^n, a)) = S' \rVert_{A \setminus a} \, sem(\boldsymbol{S}, V)$. Eliminating this rule transforms the synchronization set of the interface from A into $A \setminus a$.

Algorithmically, refined interface generation has the same complexity as the synchronization product of the LTSs taken into account in the environment. In practice, the cost of computing the interface can be reduced by minimizing the individual LTSs participating in the interface modulo safety equivalence, which is correct due to the above mentioned congruence property of safety equivalence. In addition, well-known partial order reductions preserving observable traces can be used to further reduce interfaces on-the-fly during their construction.

So far, refined interface generation required that each (high-level) process of the concurrent system under verification was replaced by its LTS, which apparently contradicts the claim that refined interfaces can be used to restrict processes on-the-fly. However, it is clear from Definition 6 that the states and transitions of LTS $\boldsymbol{S}[k]$ (corresponding to the process to restrict) are not needed for interface generation. In practice, only the observable actions of $\boldsymbol{S}[k]$ are needed to compute the synchronization rules of the network from higher level operators as in Example 4. To do so, $\boldsymbol{S}[k]$ can be replaced by an abstraction consisting of an arbitrary (and much smaller) LTS containing the same set of actions. In fact, the method remains correct if the abstraction contains a superset of $\boldsymbol{S}[k]$'s actions, although the reduction obtained on $\boldsymbol{S}[k]$ by semi-composition generally increases while the set of actions of the abstraction gets closer to the exact set of actions of $\boldsymbol{S}[k]$.

In practice, users must provide such an abstraction "by hand", which is not hard as it suffices to examine the gates (or channels) occurring in the process specification and the types of their data, and to enumerate actions of this type appropriately. If the abstraction provided by the user lacks some action of $\boldsymbol{S}[k]$, then the generated interface might be wrong, but this is detected automatically during the compositional verification task as explained in [26]. Calculating this abstraction automatically from source code or from an internal representation of processes would not present any difficulty.

5 Implementation in the CADP Toolbox

Our method was implemented in CADP (*Construction and Analysis of Distributed Processes*) [13], a popular toolbox for protocol engineering. Refined interface generation is implemented as an option (**-interface**) of the EXP.OPEN 2.0 tool [27] for on-the-fly verification of products of communicating LTSs, which can be combined using the following operators:

- standard parallel composition, action cutting, action hiding, and action renaming from CCS, CSP, LOTOS, and μCRL;
- networks of LTSs and generalized parallel composition from E-LOTOS, which includes n-ary parallel composition, "n among m" parallel composition, and parallel composition with synchronization interfaces [15];
- generalized forms of action hiding, action renaming, and transition cutting, where actions can be defined using regular expressions.

EXP.OPEN 2.0 also implements several partial order reductions, one of which can be used to partially reduce the interface on-the-fly while preserving its observable traces (**-weaktrace** option).

To simplify the use of refined interfaces in the more specific framework of LOTOS descriptions, we have also extended the SVL scripting language [12] with a new operator, named "`refined abstraction`", which can be used in the context of any parallel composition expression. As an example, given a LOTOS file `"file.lotos"` defining the system "`(P |[A, C]| Q) |[A, B]| R`", where `P`, `Q`, and `R` are LOTOS processes, one may write the following SVL script:

```
% DEFAULT_LOTOS_FILE="file.lotos"
"file.bcg" = root leaf strong reduction of
((refined abstraction Q, R using "act.bcg" of P) |[A, C]| Q) |[A, B]| R
```

This script computes the LTS corresponding to the system by first restricting `P` on-the-fly w.r.t. the constraints induced by `Q` and `R`, using the LTS `"act.bcg"` as the abstraction of `P`. To this aim, `Q` and `R` are first minimized modulo safety equivalence and an interface generated automatically using EXP.OPEN 2.0. Once the LTSs corresponding to processes `P` (restricted using the refined interface), `Q`, and `R` have been generated, the "`root leaf strong reduction`" operation minimizes them modulo strong bisimulation, and then minimizes their product once they have been composed in parallel. The result is stored in `"file.bcg"`.

6 Applications

We applied refined interfaces to three case studies. The first one is a LOTOS description written by J. Romijn [30] of the HAVi (*Home Audio-Video*) asynchronous leader election protocol[6], which consists of seven concurrent processes named `BUSRESET`, `DCM1`, `DCM2`, `CMM1`, `CMM2`, `MS1`, and `MS2`. Given a LOTOS process `ABS_DCM1` containing the actions of `DCM1`, we made the following experiments:

[6] See `ftp://ftp.inrialpes.fr/pub/vasy/demos/demo_27`

Exp.	Interface generated states	Interface generated trans.	Interface minimized (safety) states	Interface minimized (safety) trans.	DCM1 generated states	DCM1 generated trans.	Total time	Max memory
E1	0	0	0	0	**404,477**	**3,025,842**	99.9 s	54 Mb
E2	3,904	42,697	3	37	**365,923**	**2,514,848**	182.1 s	46 Mb
E3	704	7,145	4	45	**17,199**	**73,130**	12.1 s	5.9 Mb
E4	2,328	14,158	52	613	**645**	**2,020**	10.7 s	8.5 Mb

Fig. 1. LTS sizes, computation time and memory consumption for experiments E1-E4

E1 Generation of `DCM1` without interface.
E2 Generation of `DCM1` using an interface consisting of the LTS of the sub-system including `CMM1, CMM2, MS1`, and `MS2`, and of a synchronization set computed as defined by Krimm & Mounier's semi-composition laws.
E3 Generation of `DCM1` using a refined interface capturing the constraints induced by `CMM1`, `CMM2`, `MS1`, and `MS2`.
E4 Same as E3, capturing also the constraints induced by `BUSRESET` and `DCM2`.

The table in Figure 1 shows for each experiment E1 to E4 the size of the interface before and after safety minimization, the size of `DCM1` restricted by the interface (if any), the total computation time, and the peak memory consumption. It shows that refined interfaces permit state space reductions by more than two orders of magnitude (from 404,477 states reachable in a general environment down to 645 states reachable in an environment that takes an account of all processes — experiment E4), while globally reducing verification time by a factor of almost 10 and peak memory consumption by a factor of up to 9.

Experiments E2 and E3 take an account of the same processes to restrict `DCM1`, the difference being that E2 uses Krimm & Mounier's method and E3 the *refint* operation to compute the interface. Figure 1 thus shows that *refint* yields an LTS with more than 20 times fewer states and 35 times fewer transitions than Krimm & Mounier's method, while the execution time and peak memory consumption are reduced by factors of 15 and 8 respectively. Note that Krimm & Mounier's method does not permit the computation of an interface that takes an account of all processes in a way analogous to E4, because the processes in the environment of `DCM1` belong to different sub-expressions.

Second, we considered an ODP (*Open Distributed Processing*) trader [23], an E-LOTOS model of which was presented in [15][7]. An ODP trader is an agent that registers services that can be provided by distant servers, receives service requests from distant clients, and provides to the requesting clients the address of a server that can furnish the requested service. The client and server are then able to exchange the service directly without communicating with the trader anymore. Note that the trader is a central component in the ODP model in the sense that the ability of two agents to communicate is initiated by the trader. Such central components generally have large state spaces, especially in compositional verification settings where their LTS have to be generated outside of any context.

[7] See `ftp://ftp.inrialpes.fr/pub/vasy/demos/demo_37`

In our experiment, the components (trader, clients and services) are described in LOTOS and the synchronization structure describing their interactions in EXP.OPEN 2.0 using the "n among m" E-LOTOS parallel composition operator to model the dynamicity of object exchanges. In this example, the ODP trader executes in an environment consisting of 4 objects and 5 services. A refined interface is generated automatically from this environment to restrict the LTS corresponding to the trader, which is thus limited to 256 states instead of 1 million otherwise.

At last, we studied a standard cache coherency protocol for multiprocessor architectures, which consists of a remote directory process and several agent processes accessing the directory concurrently[8]. In a configuration with 5 agents, refined interface generation has allowed us to reduce the size of the LTS corresponding to the remote directory from 1 million states and 40 million transitions downto less than 60 states. This method has allowed us to generate easily the LTS corresponding to larger configurations, which could not be generated using other methods.

7 Conclusion

Compositional verification in which the behaviours of concurrent processes are restricted using interface constraints is an effective method to avoid the state explosion that may occur when the state space of a process is generated out of its environment. This paper alleviates the lack of efficient methods to synthesize constraints automatically, by proposing a method based on the analysis of the synchronizations between concurrent processes.

Compared to prior work [7,9,26,6], our method performs a finer analysis of synchronization constraints: our implementation in the EXP.OPEN 2.0 tool of CADP exhibits more than two orders of magnitude better state space reductions on an industrial case study studied by Romijn [30]. Moreover, it provides a systematic way of using the semi-composition operator of Krimm & Mounier [26] (which is implemented in the PROJECTOR tool of CADP) in the framework of languages whose composition operators are not limited to LOTOS parallel composition and hiding; indeed, both synchronization vectors and a large number of parallel composition operators are supported, including those of CCS, CSP, LOTOS, μCRL, and E-LOTOS. Alternatively, we believe that we can also use parallel composition instead of semi-composition as advocated by Cheung & Kramer [7,9,6]; indeed the interfaces generated for semi-composition can be transformed into "context-transparent" interfaces using the algorithm given in [7].

Acknowledgements. The author thanks the anonymous referees, and Hubert Garavel, Radu Mateescu, Gwen Salaün, and Wendelin Serwe from the VASY team at INRIA Rhône-Alpes for useful comments on this paper and on earlier versions of this paper.

[8] See `ftp://ftp.inrialpes.fr/pub/vasy/demos/demo_28`

References

1. A. Arnold. MEC: A System for Constructing and Analysing Transition Systems. In *Proc. of the 1st Workshop on Automatic Verification Methods for Finite State Systems*, LNCS vol. 407, 1989.
2. A. Bouajjani, J.-C. Fernandez, S. Graf, C. Rodríguez, and J. Sifakis. Safety for Branching Time Semantics. In *Proc. of 18th ICALP*. 1991.
3. A. Bouali, A. Ressouche, V. Roy, and R. de Simone. The Fc2Tools set: a Toolset for the Verification of Concurrent Systems. In *Proc. of CAV'96*, LNCS vol. 1102, 1996.
4. 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, 1984.
5. G. Chehaibar, H. Garavel, L. Mounier, N. Tawbi, and F. Zulian. Specification and Verification of the PowerScale Bus Arbitration Protocol: An Industrial Experiment with LOTOS. In *Proc. of FORTE/PSTV'96*. IFIP, Chapman & Hall, 1996. Full version available as INRIA Research Report RR-2958.
6. K. H. Cheung. *Compositional Analysis of Complex Distributed Systems*. PhD thesis, Hong Kong University of Science and Technology, 1998.
7. S. C. Cheung and J. Kramer. Enhancing Compositional Reachability Analysis with Context Constraints. In *Proc. of the 1st ACM SIGSOFT International Symposium on the Foundations of Software Engineering*. ACM Press, 1993.
8. S. C. Cheung and J. Kramer. Compositional Reachability Analysis of Finite-State Distributed Systems with User-Specified Constraints. In *Proc. of the 3rd ACM SIGSOFT International Symposium on the Foundations of Software Engineering*. ACM Press, 1995.
9. S. C. Cheung and J. Kramer. Context Constraints for Compositional Reachability. *ACM Transactions on Software Engineering Methodology*, 5(4):334–377, 1996.
10. J.-C. Fernandez. *ALDEBARAN : un système de vérification par réduction de processus communicants*. PhD thesis, Université Joseph Fourier (Grenoble), 1988.
11. J.-C. Fernandez, H. Garavel, L. Mounier, A. Rasse, C. Rodríguez, and J. Sifakis. A Toolbox for the Verification of LOTOS Programs. In *Proc. of ICSE*. ACM, 1992.
12. H. Garavel and F. Lang. SVL: a Scripting Language for Compositional Verification. In *Proc. of FORTE'2001*. IFIP, Kluwer Academic Publishers, 2001. Full version available as INRIA Research Report RR-4223.
13. H. Garavel, F. Lang, and R. Mateescu. An Overview of CADP 2001. *European Association for Software Science and Technology Newsletter*, 4:13–24, 2002. Also available as INRIA Technical Report RT-0254 (2001).
14. H. Garavel and J. Sifakis. Compilation and Verification of LOTOS Specifications. In *Proc. of PSTV'90*. IFIP, North-Holland, 1990.
15. H. Garavel and M. Sighireanu. A Graphical Parallel Composition Operator for Process Algebras. In *Proc. of FORTE/PSTV'99*. IFIP, Kluwer, 1999.
16. D. Giannakopoulou. *Model Checking for Concurrent Software Architectures*. PhD thesis, Imperial College, University of London, 1999.
17. M. Goldsmith. Operational Semantics for Fun and Profit. In *Proc. of the Symposium on the Occasion of 25 Years of* Csp, LNCS vol. 3525, 2005.
18. S. Graf, B. Steffen, and G. Lüttgen. Compositional Minimisation of Finite State Systems using Interface Specifications. *Formal Aspects of Computation*, 8(5):607–616, 1996.
19. S. Graf and B. Steffen. Compositional Minimization of Finite State Systems. In *Proc. of the 2nd Workshop on Computer-Aided Verification*, LNCS vol. 531, 1990.

20. J. F. Groote and M. Reniers. *Algebraic Process Verification.* In *Handbook of Process Algebra*, chapter 17. North-Holland, 2001.
21. J.F. Groote and A. Ponse. Syntax and semantics of μ-CRL. In *Proc. of* Algebra of Communicating Processes, Workshops in Computing, 1995.
22. 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, 1989.
23. ISO/IEC. Open Distributed Processing – Reference Model. International Standard 10746, International Organization for Standardization — Information Processing Systems, Genève, 1995.
24. ISO/IEC. Enhancements to LOTOS (E-LOTOS). International Standard 15437:2001, International Organization for Standardization — Information Technology, Genève, 2001.
25. J.-P. Krimm. *Application des ordres partiels à la génération compositionnelle de systèmes asynchrones.* PhD thesis, Université Joseph Fourier, Grenoble, 2000.
26. J.-P. Krimm and L. Mounier. Compositional State Space Generation from LOTOS Programs. In *Proc. of TACAS'97*, LNCS vol. 1217, 1997.
27. F. Lang. EXP.OPEN 2.0: A Flexible Tool Integrating Partial Order, Compositional, and On-the-fly Verification Methods. In *Proc. of IFM'2005*, LNCS vol. 3771, 2005. Full version available as INRIA Research Report RR-5673.
28. J. Malhotra, S. A. Smolka, A. Giacalone, and R. Shapiro. A Tool for Hierarchical Design and Simulation of Concurrent Systems. In *Proc. of the BCS-FACS Workshop on Specification and Verification of Concurrent Systems*, 1988. British Computer Society.
29. R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.
30. J. Romijn. Model Checking the HAVi Leader Election Protocol. Technical Report SEN-R9915, CWI, Amsterdam, The Netherlands, 1999.
31. A.W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall, 1998.
32. K. K. Sabnani, A. M. Lapone, and M. U. Uyar. An Algorithmic Procedure for Checking Safety Properties of Protocols. *IEEE Transactions on Communications*, 37(9):940–948, 1989.
33. K. C. Tai and V. Koppol. Hierarchy-Based Incremental Reachability Analysis of Communication Protocols. In *Proc. of the IEEE International Conference on Network Protocols.* IEEE Press, 1993.
34. K. C. Tai and V. Koppol. An Incremental Approach to Reachability Analysis of Distributed Programs. In *Proc. of the 7th International Workshop on Software Specification and Design.* IEEE Press, 1993.
35. F. Tronel, F. Lang, and H. Garavel. Compositional Verification Using CADP of the ScalAgent Deployment Protocol for Software Components. In *Proc. of FMOODS'2003*, LNCS vol. 2884, 2003. Full version available as INRIA Research Report RR-5012.
36. A. Valmari. Compositional State Space Generation. In *Proc. of Advances in Petri Nets*, LNCS vol. 674, 1993.
37. W. J. Yeh. *Controlling State Explosion in Reachability Analysis.* PhD thesis, Software Engineering Research Center Laboratory, Purdue University, 1993. Technical Report SERC-TR-147-P.
38. W. J. Yeh and M. Young. Compositional Reachability Analysis Using Process Algebra. In *Proc. of the ACM SIGSOFT Symposium on Testing, Analysis, and Verification.* ACM Press, 1991.