HAL open science

Smart Reduction

Pepijn Crouzen, Frédéric Lang

To cite this version:

Pepijn Crouzen, Frédéric Lang. Smart Reduction. Proceedings of Fundamental Approaches to Software Engineering (FASE'2011), Mar 2011, Saarbrucken, Germany. pp.111-126. inria-00572535

HAL Id: inria-00572535

https://inria.hal.science/inria-00572535v1

Submitted on 1 Mar 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Smart Reduction

Pepijn Crouzen[1] and Frédéric Lang[2]

[1] Computer Science, Saarland University, Saarbrücken, Germany
crouzen@cs.uni-saarland.de
[2] Vasy project team, Inria Grenoble Rhône-Alpes/Lig, Montbonnot, France
Frederic.Lang@inria.fr

Abstract. Compositional aggregation is a technique to palliate state explosion — the phenomenon that the behaviour graph of a parallel composition of asynchronous processes grows exponentially with the number of processes — which is the main drawback of explicit-state verification. It consists in building the behaviour graph by incrementally composing and minimizing parts of the composition modulo an equivalence relation. Heuristics have been proposed for finding an appropriate composition order that keeps the size of the largest intermediate graph small enough. Yet the underlying composition models are not general enough for systems involving elaborate forms of synchronization, such as multiway and/or nondeterministic synchronizations. We overcome this by proposing a generalization of compositional aggregation that applies to an expressive composition model based on synchronization vectors, subsuming many composition operators. Unlike some algebraic composition models, this model enables any composition order to be used. We also present an implementation of this approach within the Cadp verification toolbox in the form of a new operator called *smart reduction*, as well as experimental results assessing the efficiency of smart reduction.

1 Introduction

Explicit-state verification is a way of ascertaining whether a system fulfills its specification, by systematically exploring its behaviour graph. The main limitation of explicit-state verification is the exponential growth of the behaviour graph, known as state explosion. For systems consisting of asynchronous processes executing in parallel, *compositional aggregation* [11] (also known as *incremental reachability analysis* [29], *compositional state space minimization* [32, 20, 25], and *compositional reachability analysis* [9, 19]) is a way to palliate state explosion by incrementally *aggregating* (i.e., composing and then minimizing modulo an equivalence relation) parts of the system. Compositional aggregation was applied successfully to systems from various domains [8, 22, 15, 31, 5, 4, 6].

Due to their modular nature, software systems are appropriate for compositional modeling and verification. Examples of studies include software reuse [12], unit testing [30], web service performance [13], middleware specification [28], software deployment protocols [31], multi-processor multi-threaded architectures [10], and software decomposition [7], in which processes usually represent

software components, such as servers, packages, threads, objects or functions. These applications often involve the (possibly automatic) translation of (architectural) software description languages such as UML, statecharts, or BPEL, each of which provides its own composition model, to a formal model.

The efficiency of compositional aggregation depends on the order in which the concurrent processes are aggregated. In practice, the order is often specified by the designer of a concurrent system, either explicitly, or implicitly through the order and hierarchy of the concurrent processes. Since it is not possible to know precisely whether an order will be more or less efficient than another without trying them and comparing the results, the user generally has to rely on intuition. This task is difficult for large and/or not hierarchical compositions, and impractical for compositional models that are automatically generated from a higher-level description.

Heuristics to automatically determine efficient aggregation orders, based on the process interactions, have been proposed in [29] for concurrent finite state machines communicating via named channels. More recently, such heuristics have been refined and implemented in a prototype tool for processes synchronizing on their common alphabets [11]. In both works, the processes to be composed are selected using two metrics: an estimate of the proportion of internal transitions in the composition (the higher, the more the composition graph being expected to be reducible), and an estimate of the proportion of transitions that interleave. A limitation of the above works lies in the limited forms of synchronizations enabled by their composition models, which are generally insufficient to capture the semantics of the composition models of state-of-the-art software description languages: The composition model used in [29] does not enable multiway synchronization (more than two processes synchronizing all together), and neither of the composition models used in [29, 11] enables nondeterministic synchronization (a process synchronizing with one or another on a given label).

This paper presents a refinement of the compositional aggregation techniques of [29, 11], called *smart reduction*. Smart reduction uses an expressive composition model named *networks of* LTSs (*Labeled Transition Systems*) [26], inspired by synchronization vectors in the style of MEC [1] and FC2 [3], which has two major advantages. The first advantage is that it makes the compositional aggregation technique more general: networks of LTSs subsume not only the models used in [29, 11], but also many other concurrent operators. They include the parallel composition, label hiding, label renaming and label cutting (sometimes also called label restriction) found in process algebras (e.g., CCS [27], CSP [23], LOTOS [24], μCRL [21], etc.). They also include more general parallel composition such as that of E-LOTOS/LOTOS NT [18], which enables n among m synchronization (any n processes synchronizing together among a set of m) and synchronization by interfaces (all processes sharing a label in their interface synchronizing together on that label). The latter operators have been shown to be expressive enough to reflect the graphical structure of process networks [18], such as those found in graphical software description languages. In particular, synchronization by interfaces was adopted in the FIACRE intermediate model for

avionic systems [2]. The second advantage is that networks enable any aggregation order, which is not in general the case in process algebraic models, where some composition orders, possibly including the optimal order, may not be representable using the available algebraic operators. This paper also presents the implementation of smart reduction in the CADP toolbox [17], and experimental results that assess the effectiveness of smart reduction on several case studies.

Paper overview. Networks of LTSs are defined in Section 2. Compositional aggregation of networks is described and illustrated in Section 3. Metrics for selecting a good aggregation order are presented in Section 4. The implementation within CADP is described in Section 5. Experimentation on existing case studies is reported in Section 6. Finally, concluding remarks are given in Section 7.

2 Networks of LTSs

The *network of LTSs* model (or *networks* for short) was introduced in [26] as an intermediate model to represent compositions of LTSs using various operators. We first give a few background definitions before defining the model formally.

Background. Given two integers n and m, we write $n..m$ for the set of integers ranging from n to m. If $n > m$ then $n..m$ denotes the empty set. A *vector* $\boldsymbol{v}$ of size n is a set of n elements indexed by $1..n$. For $i \in 1..n$, we write $\boldsymbol{v}[i]$ for the element of v at index i. We write $()$ for the vector of size 0, (e_1) for the vector $\boldsymbol{v}$ of size 1 such that $\boldsymbol{v}[1] = e_1$, and more generally $(e_1, \ldots, e_n)$ for the vector $\boldsymbol{v}$ of size n such that $(\forall i \in 1..n)\ \boldsymbol{v}[i] = e_i$. Given $\boldsymbol{v}_1$, a vector of size n_1, and $\boldsymbol{v}_2$, a vector of size n_2, $\boldsymbol{v}_1 \oplus \boldsymbol{v}_2$ denotes the vector of size $n_1 + n_2$ obtained by concatenation of $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$, defined by $(\forall i \in 1..n_1)\ (\boldsymbol{v}_1 \oplus \boldsymbol{v}_2)[i] = \boldsymbol{v}_1[i]$ and $(\forall i \in n_1 + 1..n_1 + n_2)\ (\boldsymbol{v}_1 \oplus \boldsymbol{v}_2)[i] = \boldsymbol{v}_2[i - n_1]$. The expression $e :: \boldsymbol{v}$ denotes adjunction of e to the head of $\boldsymbol{v}$ and is defined as $(e) \oplus \boldsymbol{v}$. Given an ordered subset of $1..n$ I, such that $I = \{i_1, \ldots, i_m\}$ with $i_1 < \ldots < i_m$ $(0 \leq m \leq n)$, $\boldsymbol{v}_{|I}$ denotes the projection of $\boldsymbol{v}$ on to the set of indexes I, defined as the vector of size m such that $(\forall j \in 1..m)\ \boldsymbol{v}_{|I}[j] = \boldsymbol{v}[i_j]$. We write as $\overline{I}$ the set $1..n \setminus I$. For any set S, we write $|S|$ for the number of elements of S. An LTS (*Labeled Transition System*) is a tuple $(\Sigma, A, \longrightarrow, s_0)$, where Σ is a set of states, A is a set of labels, $\longrightarrow \subseteq \Sigma \times A \times \Sigma$ is the (labeled) transition relation, and $s_0 \in \Sigma$ is the initial state.

Networks of LTSs. A *network of LTSs* N of size n is a pair $(\boldsymbol{S}, V)$ where:

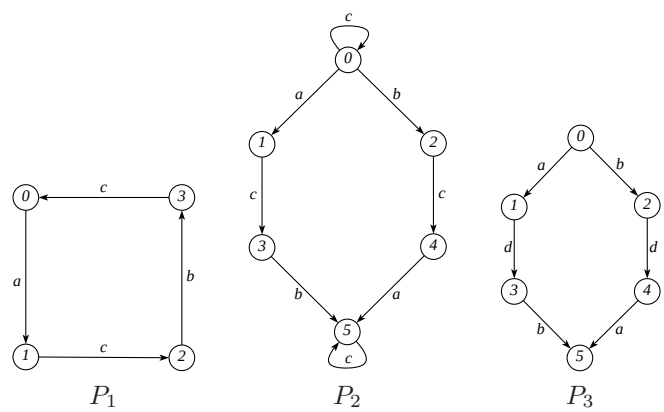
- $\boldsymbol{S}$ is a vector of LTSs (called *individual LTSs*) of size n. We write respectively $\longrightarrow_i$, Σ_i, and s_i^0 for the transition relation, the set of states, and the initial state of $\boldsymbol{S}[i]$. For a label b, we also write $\xrightarrow{b}_i$ for the largest subset of $\longrightarrow_i$ containing only transitions labeled by b.
- V is a finite set of *synchronization rules*. Each synchronization rule has the form $(\boldsymbol{t}, a)$, where a is a label and $\boldsymbol{t}$ is a vector of size n, called a *synchronization vector*, whose elements are labels and occurrences of a special symbol $\bullet$ that does not occur as a label in any individual LTS.

To a network N can be associated a (global) LTS $lts(N)$ which is the parallel composition of its individual LTSs. Each rule $(\boldsymbol{t}, a) \in V$ defines transitions labeled by a, obtained either by synchronization (if several indices i are such that $\boldsymbol{t}[i] \neq \bullet$) or by interleaving (otherwise) of individual LTS transitions. Formally, $lts(N)$ is defined as the LTS $(\Sigma, A, \longrightarrow, \boldsymbol{s}_0)$, where $\Sigma = \Sigma_1 \times \ldots \times \Sigma_n$, $A = \{a \mid (\boldsymbol{t}, a) \in V\}$, $\boldsymbol{s}_0 = (s_1^0, \ldots, s_n^0)$, and $\longrightarrow$ is the smallest transition relation satisfying:

$$(\boldsymbol{t}, a) \in V \wedge (\forall i \in 1..n)\ (\boldsymbol{t}[i] = \bullet \wedge \boldsymbol{s}'[i] = \boldsymbol{s}[i]) \vee (\boldsymbol{t}[i] \neq \bullet \wedge \boldsymbol{s}[i] \xrightarrow{\boldsymbol{t}[i]}_i \boldsymbol{s}'[i]) \Rightarrow \boldsymbol{s} \xrightarrow{a} \boldsymbol{s}'$$

If $\boldsymbol{t}[i] \neq \bullet$, we say that $\boldsymbol{S}[i]$ is *active* for the rule $(\boldsymbol{t}, a)$, otherwise we say that $\boldsymbol{S}[i]$ is *inactive*. We write $A(\boldsymbol{t})$ for the set of individual LTS indexes active for a rule, defined as $\{i \mid i \in 1..n \wedge \boldsymbol{t}[i] \neq \bullet\}$. We say that a rule $(\boldsymbol{t}, a)$ or a synchronization vector $\boldsymbol{t}$ is *controlled* by LTS $\boldsymbol{S}[i]$ if $i \in A(\boldsymbol{t})$. In other words, a rule or a synchronization vector is controlled by all the LTSs that it synchronizes.

Example 1. Let a, b, c, and d be labels, and P_1, P_2, and P_3 be the processes defined as follows, where the initial states are those numbered 0:



Consider the network $N = ((P_1, P_2, P_3), V_{123})$, where V_{123} is the set of rules $\{((a, a, \bullet), a), ((a, \bullet, a), a), ((b, b, b), b), ((c, c, \bullet), \tau), ((\bullet, \bullet, d), d)\}$. The first two synchronization rules express that a transition labeled by a in P_1 synchronizes with a transition labeled by a nondeterministically either in P_2 or in P_3. The third synchronization rule expresses a multiway synchronization on b between P_1, P_2, and P_3. The fourth synchronization rule expresses that synchronization on c between P_1 and P_2 yields a transition labeled by τ, thus is internal. The fifth synchronization rule expresses that transitions labeled by d in P_3 execute in full interleaving. The global LTS of this network is given in Figure 1.

A large set of operators can be translated to networks: An LTS P translates to a network of the form $((P), V)$ where V contains a rule of the form $((a), a)$ for each label a of P. Hiding of labels in an expression E_0 translates to the network of E_0 in which each rule $(\boldsymbol{t}, a)$ with a a label to be hidden is replaced by $(\boldsymbol{t}, \tau)$. Renaming of labels in an expression E_0 translates to the network of E_0 in which each rule $(\boldsymbol{t}, a)$ with a a label to be renamed into a' is replaced by $(\boldsymbol{t}, a')$. Cutting

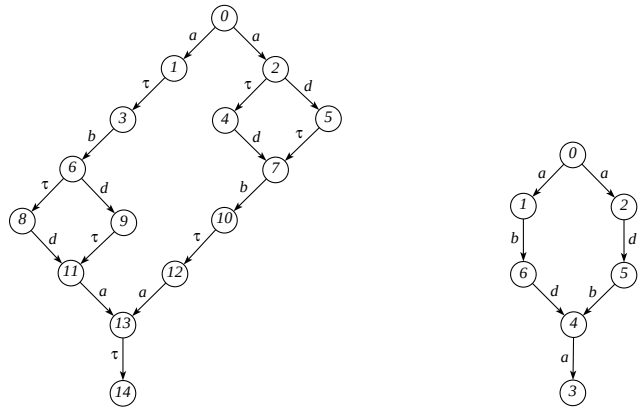


Fig. 1. Global LTS of the network of Example 1, unreduced (left) and minimized modulo branching bisimulation (right)

of labels in an expression E_0 translates to the network of E_0 in which each rule $(\boldsymbol{t}, a)$ with a a label to be cut is merely suppressed. Parallel composition of a set of expressions $E_1, \ldots, E_n$ translates to the network obtained by concatenating the vectors of LTSs of $E_1, \ldots, E_n$ and joining their synchronization rules as follows: for each subset $\{E_{i_1}, \ldots, E_{i_m}\}$ of $\{E_1, \ldots, E_n\}$ that may synchronize all together on label a, the resulting set of synchronization rules contains as many different rules of the form $(\boldsymbol{t}_1 \oplus \cdots \oplus \boldsymbol{t}_n, a)$ as possible, such that for each $i \in \{i_1, \ldots, i_m\}$ the network of E_i has a rule of the form $(\boldsymbol{t}_i, a)$ and for each $i \in 1..n \setminus \{i_1, \ldots, i_m\}$, $\boldsymbol{t}_i$ is a vector of $\bullet$ whose size is the size of the network of E_i. This translation also holds for $m = 1$, corresponding to labels that do not synchronize.

Example 2. Let P_i $(i \in 1..3)$ be LTSs with labels a and b. Each P_i translates into $((P_i), \{((a), a), ((b), b)\})$. Hiding a in P_1 translates into $((P_1), \{((a), \tau), ((b), b)\})$, renaming a to c in P_1 translates into $((P_1), \{((a), c), ((b), b)\})$, cutting a in P_1 translates into $((P_1), \{((b), b)\})$, and synchronizing P_1 and P_2 on a deterministically translates into the network with vector of LTSs (P_1, P_2) and set of rules $\{((a, a), a), ((b, \bullet), b), ((\bullet, b), b)\}$. The set of rules for synchronizing P_1 and P_2 on a nondeterministically is $\{((a, a), a), ((a, \bullet), a), ((\bullet, a), a), ((b, \bullet), b), ((\bullet, b), b)\}$. Lastly, the set of rules for 2 among 3 synchronization on a between P_1, P_2, and P_3 is $\{((a, a, \bullet), a), ((a, \bullet, a), a), ((\bullet, a, a), a), ((b, \bullet, \bullet), b), ((\bullet, b, \bullet), b), ((\bullet, \bullet, b), b)\}$.

Rules of the form $(\boldsymbol{t}, a)$ may define synchronizations between distinct labels, which is useful, notably to represent combinations of synchronizations and renamings. For instance, the (pseudo-language) expression "(**rename** $a \to c$ **in** P_1) || (**rename** $b \to c$ **in** P_2)" (where || represents synchronization on all visible labels) produces the synchronization rule $((a, b), c)$.

Many equivalence relations on LTSs exist, each preserving particular classes of properties. For instance, two LTSs that are trace equivalent have the same set

of traces. Equivalences can be used to ease the cost of explicit-state verification. For instance, the traces of an LTS P can be obtained by generating a smaller LTS P', which is trace equivalent to P. We are then interested in the smallest LTS equivalent to P. Replacing an LTS by its smallest representative with respect to an equivalence relation is called *minimizing* the LTS modulo this relation.

We are especially interested in equivalence relations that interact well with algebraic operations like parallel composition, renaming, hiding, and cutting. We say an equivalence relation R is a congruence with respect to networks if the equivalence of two LTSs P and P' implies the equivalence of any network N to a network N' obtained by replacing P by P'. Strong bisimulation and trace equivalence are congruences for networks. Branching bisimulation, observation equivalence, safety equivalence, and weak trace equivalence, are also congruences for networks provided that the synchronization rules satisfy the following standard constraints regarding the internal transitions of individual LTSs [26]:

- **No synchronization**: $(\boldsymbol{t}, a) \in V \wedge \boldsymbol{t}[i] = \tau \implies A(\boldsymbol{t}) = \{i\}$
- **No renaming**: $(\boldsymbol{t}, a) \in V \wedge \boldsymbol{t}[i] = \tau \implies a = \tau$
- **No cut**: $\xrightarrow{\tau}_i \neq \emptyset \implies (\exists(\boldsymbol{t}, \tau) \in V)\ \boldsymbol{t}[i] = \tau$

We assume that all networks in this paper satisfy these constraints.

3 Compositional Aggregation of Networks

Generating the global LTS of a network all at once may face state explosion. To overcome this, the LTS can be generated incrementally, by alternating compositions of well-chosen subsets of the individual LTSs and minimizations modulo an equivalence relation. We call *aggregation* a composition followed by a minimization of the result, and *compositional aggregation* this incremental technique.

Formally, we consider a relation R that is a congruence for networks and write $min_R(P)$ for the minimization modulo R of the LTS P. A compositional aggregation strategy to generate modulo R the LTS corresponding to a network of LTSs $N = (\boldsymbol{S}, V)$ of size n is defined by the following iterative algorithm:

1. Replace in N each $\boldsymbol{S}[i]$ $(i \in 1..n)$ by $min_R(\boldsymbol{S}[i])$.
2. Select a set I containing at least two of the individual LTSs of N. A strategy for selection will be addressed in the next section.
3. Replace N by a new network $agg(N, I)$ — defined below — corresponding to N in which the LTSs in I have been replaced by their aggregation.
4. If N still contains more than two LTSs, then continue in step 2. Otherwise return $min_R(lts(N))$.

By abuse of language, since I denotes the set of LTSs to be aggregated, we call I an aggregation. We represent I as a subset of $1..n$, corresponding to the indexes of the LTSs to be aggregated. We assume a function $\alpha(\boldsymbol{t}, a)$ that associates to each $(\boldsymbol{t}, a) \in V$ a unique label distinct from all others and define $agg(N, I)$ in Figure 2, where $min_R(lts(proj(N, I)))$ corresponds to the aggregation of the

$$
\begin{aligned}
agg(N,I) &= (min_R(lts(proj(N,I))) :: \boldsymbol{S}_{|\overline{I}}, V_{agg}) \\
\text{where } proj(N,I) &= (\boldsymbol{S}_{|I}, V_{proj}) \\
V_{agg} &= \{\ (a :: \boldsymbol{t}_{|\overline{I}}, a) \quad | \ (\boldsymbol{t}, a) \in V \wedge A(\boldsymbol{t}) \subseteq I \quad \} \cup \\
&\quad \{\ (\alpha(\boldsymbol{t}, a) :: \boldsymbol{t}_{|\overline{I}}, a) \ | \ (\boldsymbol{t}, a) \in V \wedge \emptyset \subset (I \cap A(\boldsymbol{t})) \subset A(\boldsymbol{t}) \ \} \cup \\
&\quad \{\ (\bullet :: \boldsymbol{t}_{|\overline{I}}, a) \quad | \ (\boldsymbol{t}, a) \in V \wedge (I \cap A(\boldsymbol{t})) = \emptyset \quad \} \\
\text{and} \qquad V_{proj} &= \{\ (\boldsymbol{t}_{|I}, a) \quad | \ (\boldsymbol{t}, a) \in V \wedge A(\boldsymbol{t}) \subseteq I \quad \} \cup \\
&\quad \{\ (\boldsymbol{t}_{|I}, \alpha(\boldsymbol{t}, a)) \quad | \ (\boldsymbol{t}, a) \in V \wedge \emptyset \subset (I \cap A(\boldsymbol{t})) \subset A(\boldsymbol{t}) \ \}
\end{aligned}
$$

Fig. 2. Definition of $agg(N, I)$

LTSs inside I and $\boldsymbol{S}_{|\overline{I}}$ corresponds to the LTSs outside I, which are kept non-aggregated. The synchronization rules V_{proj} of the auxiliary network $proj(N, I)$ are obtained by projection of V on to I, whereas the synchronization rules V_{agg} are obtained by synchronization of the labels in V_{proj} with the projection of V on to $\overline{I}$. The rules of V_{proj} and V_{agg} are organized in three subsets:

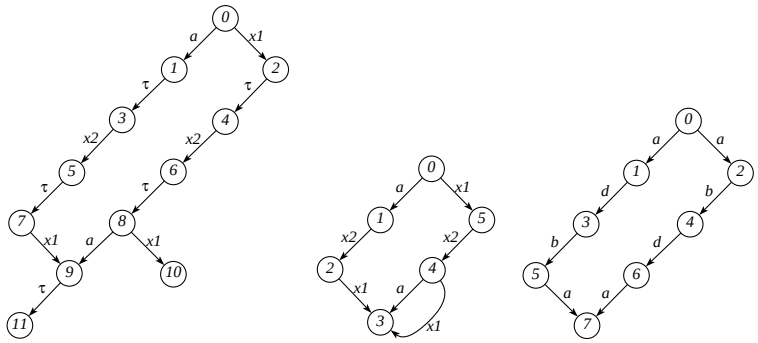
- The first subset — rules of the form $(\boldsymbol{t}_{|I}, a)$ in V_{proj} and $(a :: \boldsymbol{t}_{|\overline{I}}, a)$ in V_{agg} — represents the synchronization rules that are completely controlled by LTSs inside I. In this case, $\boldsymbol{t}_{|\overline{I}}$ is a vector of $\bullet$, which expresses that transitions of $min_R(lts(proj(N, I)))$ obtained by synchronization of LTSs inside I do not need to synchronize with LTSs outside I.
- The second subset — rules of the form $(\boldsymbol{t}_{|I}, \alpha(\boldsymbol{t}, a))$ in V_{proj} and $(\alpha(\boldsymbol{t}, a) :: \boldsymbol{t}_{|\overline{I}}, a)$ in V_{agg} — represents the synchronization rules that are controlled by LTSs both inside I and outside I. This expresses that transitions of $min_R(lts(proj(N, I)))$ obtained by synchronization of LTSs inside I still need to synchronize with LTSs outside I. The special label $\alpha(\boldsymbol{t}, a)$ is an intermediate label used for this synchronization. Note that in general, a cannot be used instead of $\alpha(\boldsymbol{t}, a)$ because (1) a can be the internal label τ (even though $\boldsymbol{t}$ synchronizes only visible labels), which cannot be synchronized, and (2) in general there can be several rules with the same label a (in particular when nondeterministic synchronization is involved) and using a could create unexpected synchronizations.
- The third subset — rules of the form $(\bullet :: \boldsymbol{t}_{|\overline{I}}, a)$ in V_{agg} — represents the synchronization rules that are completely controlled by LTSs outside I. These rules do not impose synchronization constraints on LTSs inside I, thus explaining why V_{proj} has no rule in the third subset.

If $P_1, \ldots, P_m$ are individual LTSs of N and if I is the set of their indexes, then we may write $comp(P_1, \ldots, P_m)$ for $lts(proj(N, I))$.

Since R is a congruence, $lts(agg(N, I))$ is equivalent modulo R to $lts(N)$. Therefore, $lts(N)$ remains invariantly R-equivalent to the input until the end of the algorithm, thus guaranteeing the correctness of compositional aggregation. Moreover, the size of $agg(N, I)$ is the size of N plus 1 minus the size of I (which

is at least 2). Since N is substituted by $agg(N, I)$ at each step, this guarantees that the size of N decreases and therefore that the algorithm terminates.

Example 3. We write $min_b(P)$ for minimization of P modulo branching bisimulation. The global LTS of the network of Example 1, whose LTSs P_1, P_2, and P_3 are already minimal, can be generated modulo branching bisimulation by first composing P_1 and P_2, then P_3 as follows. First, build $N_{12} = proj(N, \{1, 2\}) = ((P_1, P_2), V_{12})$ for the aggregation of P_1 and P_2, where V_{12} is the set of rules $\{((a, a), a), ((a, \bullet), \mathbf{x}_1), ((b, b), \mathbf{x}_2), ((c, c), \tau)\}$, $\mathbf{x}_1$ is the label $\alpha((a, \bullet, a), a)$, and $\mathbf{x}_2$ is the label $\alpha((b, b, b), b)$. Compute the intermediate LTS $P_{12} = min_b(lts(N_{12}))$ (see below). Second, build $N_{(12)3} = agg(N, \{1, 2\}) = ((P_{12}, P_3), V_{(12)3})$, where $V_{(12)3}$ is the set $\{((a, \bullet), a), ((\mathbf{x}_1, a), a), ((\mathbf{x}_2, b), b), ((\tau, \bullet), \tau), ((\bullet, d), d)\}$. Return $min_b(lts(N_{(12)3}))$ (see $lts(N_{(12)3})$ below and $min_b(lts(N_{(12)3}))$ in Figure 1, right), which is branching equivalent to $lts(N_{123})$ (see Figure 1, left). Note that both LTSs $lts(N_{12})$ and $lts(N_{(12)3})$ are smaller than $lts(N_{123})$.



$lts(N_{12})$ $\quad$ $min_b(lts(N_{12}))$ $\quad$ $lts(N_{(12)3})$

Other aggregation orders are possible, yielding different intermediate graphs. Figure 3 gives the sizes of those intermediate graphs corresponding to the different aggregation orders. The largest intermediate graph size of each order is indicated in bold type. This table shows that the aggregation order described above (order 1) is optimal in terms of largest intermediate graph (12 transitions).

Note that N has the same meaning as the LOTOS composition of processes "**hide** c **in** $(P_1 \,|[a, b, c]|\, (P_2 \,|[b]|\, P_3))$". Yet using LOTOS operators instead of networks, it would not be possible to aggregate P_1, P_2, and P_3 following the optimal order (P_1, P_2) then P_3, because there do not exist LOTOS operators (or compositions of operators) op_1 and op_2, such that the above term can be written in the form "$op_2(op_1(P_1, P_2), P_3)$". For instance, "**hide** c **in** $((P_1 \,|[a, b, c]|\, P_2) \,|[b]|\, P_3)$" does not work, because we lose synchronization on a between P_1 and P_3. More generally, the problem happens when the model contains nondeterministic synchronizations, which can make parallel composition non-associative. Similar examples can be built in other process algebras such as, e.g., CSP (by combining renaming and synchronization on common alphabet).

Order 1 : (P_1, P_2) then P_3	states	transitions
$comp(P_1, P_2)$	**12**	**12**
$min_b(comp(P_1, P_2))$	6	7
$comp(min_b(comp(P_1, P_2)), P_3)$	8	8
$min_b(comp(min_b(comp(P_1, P_2)), P_3))$	7	7
Order 2 : (P_1, P_3) then P_2	states	transitions
$comp(P_1, P_3)$	**19**	**23**
$min_b(comp(P_1, P_3))$	17	22
$comp(min_b(comp(P_1, P_3)), P_2)$	15	17
$min_b(comp(min_b(comp(P_1, P_3)), P_2))$	7	7
Order 3 : (P_2, P_3) then P_1	states	transitions
$comp(P_2, P_3)$	**18**	**34**
$min_b(comp(P_2, P_3))$	**18**	**34**
$comp(min_b(comp(P_2, P_3)), P_1)$	15	17
$min_b(comp(min_b(comp(P_2, P_3)), P_1))$	7	7
Order 4 : (P_1, P_2, P_3)	states	transitions
$comp(P_1, P_2, P_3)$	**15**	**17**
$min_b(comp(P_1, P_2, P_3))$	7	7

Fig. 3. Sizes of intermediate graphs for all aggregation orders

4 Smart Reduction

The most difficult issue in compositional aggregation concerns step 2 of the algorithm, namely to select, if possible automatically, an aggregation I that avoids state explosion. In this section, we present *smart reduction*, which corresponds to compositional aggregation using a heuristic based on metrics evaluated against possible aggregations.

Our metrics use an estimate of the number of global transitions in I generated by synchronization vector $\boldsymbol{t}$, written $ET(I, \boldsymbol{t})$ and defined below:

$$ET(I, \boldsymbol{t}) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } I \cap A(\boldsymbol{t}) = \emptyset \\ \prod_{i \in I \setminus A(\boldsymbol{t})} |\Sigma_i| \quad \times \quad \prod_{i \in I \cap A(\boldsymbol{t})} |\xrightarrow{\boldsymbol{t}[i]}_i| & \text{otherwise} \end{cases}$$

Informally, $ET(I, \boldsymbol{t})$ counts, for vector $\boldsymbol{t}$, the number of transitions going out of every product state of I, including unreachable states. This count equals 0 if $\boldsymbol{t}$ is not controlled by LTSs inside I (first line). Otherwise, $proj(N, I)$ has a rule of the form $(\boldsymbol{t}_{|I}, b)$. This rule generates a transition in a state of $proj(N, I)$ without condition on the states of the individual LTSs $\boldsymbol{S}[i]$ such that $i \in I \setminus A(\boldsymbol{t})$ — thus justifying the first product in the definition of $ET(I, \boldsymbol{t})$ — and provided the states of the individual LTSs $\boldsymbol{S}[i]$ such that $i \in I \cap A(\boldsymbol{t})$ have a transition labeled $\boldsymbol{t}[i]$ — thus justifying the second product. In general, the exact number of global transitions in I generated by synchronization vector $\boldsymbol{t}$ is below this count since some states may be unreachable.

We now define our metrics on networks. The *hiding metric* is defined by $HM(I) \stackrel{\text{def}}{=} HR(I)/|I|$, where $HR(I)$ (the *hiding rate*) is defined in Figure 4 (left). Informally, $HR(I)$ represents an estimate of the proportion of transitions in $proj(N, I)$ that are internal. Those transitions are necessarily created by rules completely controlled by LTSs inside I. The addition of 1 in its divisor avoids division by 0 in pathological cases. The divisor $|I|$ of $HM(I)$ aims at favouring smaller aggregations, which are likely to yield smaller intermediate LTSs.

Using $HM(I)$ is justified in the context of weak equivalence relations (e.g., branching bisimulation), because internal transitions are often eliminated by the corresponding minimizations. Although to a lesser extent, it is also justified in the context of strong bisimulation, because hiding enables abstracting away from labels that otherwise would differentiate the behaviour of equivalent states. However, in both cases, $HM(I)$ is not sufficient to avoid intermediate explosion due to the aggregation of loosely synchronized LTSs. To palliate this, the hiding metric will be combined with the *interleaving metric* $IM(I) \stackrel{\text{def}}{=} (1 - IR(I))/|I|$, where the *interleaving rate* $IR(I)$ is defined in Figure 4 (right). In this definition, $\boldsymbol{t}@i$ denotes the synchronization vector of size n defined by $(\boldsymbol{t}@i)[i] = \boldsymbol{t}[i]$ and $(\forall j \in 1..n \setminus \{i\})\ (\boldsymbol{t}@i)[j] = \bullet$. The value $IR(I)$ is therefore the quotient between an estimate of the number of global transitions of I and the number of global transitions that I would have if all individual LTSs were fully interleaving. For an aggregation I of fully interleaving individual LTSs, we thus have $IR(I)$ very close to 1. It is a refinement of the *interleaving count* defined in [11], which uses the proportion of fully interleaving (i.e., non-synchronized) individual LTS transitions out of the total number of individual LTS transitions. We believe that $IR(I)$ is more accurate, because it also measures the *partial* interleaving of synchronized transitions with the remainder of the aggregation. Taking into account both the hiding rate and the interleaving rate, we use here the *combined metric* $CM(I) \stackrel{\text{def}}{=} HM(I) + IM(I)$.

$$HR(I) \stackrel{\text{def}}{=} \frac{\displaystyle\sum_{(\boldsymbol{t},\tau)\in V \wedge A(\boldsymbol{t})\subseteq I} ET(I,\boldsymbol{t})}{1 + \displaystyle\sum_{(\boldsymbol{t},a)\in V} ET(I,\boldsymbol{t})} \qquad IR(I) \stackrel{\text{def}}{=} \frac{\displaystyle\sum_{(\boldsymbol{t},a)\in V} ET(I,\boldsymbol{t})}{1 + \displaystyle\sum_{(\boldsymbol{t},a)\in V} \sum_{i\in I\cap A(\boldsymbol{t})} ET(I,\boldsymbol{t}@i)}$$

Fig. 4. Hiding rate and interleaving rate of an aggregation I

Smart reduction selects the aggregation to which the metrics gives the highest value (high proportion of internal transitions and low interleaving). To avoid the combinatorial explosion of the number of aggregations, we proceed as in [11] and only consider: (1) aggregations whose size is bounded by a constant (definable by the user), and (2) aggregations that are *connected*. An aggregation is connected if for each pair of distinct LTSs P_i, P_j in the aggregation, P_i and P_j are *connected*,

which is defined recursively as follows: either there is a synchronization rule $(\boldsymbol{t}, a)$ such that $\{i, j\} \subseteq A(\boldsymbol{t})$ (i.e., P_i and P_j are synchronized) or, recursively, the aggregation contains a third (distinct) LTS P_k connected to both P_i and P_j.

Example 4. The metrics evaluate as follows on the network of Examples 1 and 3:

I	$\{1,2\}$	$\{1,3\}$	$\{2,3\}$	$\{1,2,3\}$
$HM(I)$	0.211	0	0	0.129
$IM(I)$	0.357	0.227	0.124	0.249
$CM(I)$	0.568	0.227	0.124	0.378

As expected, the combined metric gives the highest value to $\{1, 2\}$, therefore designating it as the best aggregation in the first step. Note that, more generally in this example, a comparison between the graph sizes in Figure 3 and the values in the above table shows that one aggregation is more efficient than the others whenever the combined metrics gives it a higher value.

5 Implementation

Smart reduction has been implemented in CADP (*Construction and Analysis of Distributed Processes*)[3] [17], a widely disseminated toolbox for the design of communication protocols and distributed systems. CADP offers a large set of features, ranging from step-by-step simulation to massively parallel model-checking. It is the only toolbox to offer compilers for several input formalisms (LOTOS, LOTOS NT, networks of automata, etc.) into LTSs, equivalence checking tools (minimization and comparisons modulo bisimulation relations), model-checkers for various temporal logics and μ-calculus, several verification techniques combined together (enumerative verification, on-the-fly verification, compositional aggregation, partial order reduction, distributed model checking, etc.), and a number of other tools providing advanced functionalities such as visual checking, performance evaluation, etc. The tools SVL and EXP.OPEN 2.0 have been extended to support smart reduction.

The tool SVL. SVL (*Script Verification Language*) [16] is both a scripting language that enables advanced verification scenarios to be described at a high-level of abstraction, and an associated compiler that enables automatic execution of the SVL scripts. Smart reduction is available as a new SVL operator, which can be parameterized by an equivalence relation. For example, the following script describes the smart branching reduction of the LOTOS behaviour corresponding to the network of Example 1:

```
% DEFAULT_LOTOS_FILE="processes.lotos"
% DEFAULT_SMART_LIMIT=3
"composition.bcg" = smart branching reduction of
      hide c in (P1 |[ a, b, c ]| (P2 |[ b ]| P3));
```

[3] http://vasy.inria.fr/cadp

The file "`processes.lotos`" is where the three processes `P1`, `P2` and `P3` are specified, and "`composition.bcg`" is the name of the file where the LTS resulting from their aggregation is to be stored, represented in a compact graph format called BCG (*Binary Coded Graph*). The (optional) line of the form "`% DEFAULT_SMART_LIMIT=3`" defines as 3 the maximal number of LTSs aggregated at each step. Otherwise, a default value of 4 is used in our implementation. This script aggregates the three processes in the order determined automatically by the tool, which in this case is the optimal order, `P1` and `P2`, then `P3`.

The tool EXP.OPEN 2.0. EXP.OPEN 2.0 [26] is a conservative extension of the former version 1.0, developed in 1995 by L. Mounier (Univ. Joseph Fourier, Grenoble, France). It takes as input an expression consisting of LTSs composed together using the parallel composition, label hiding, label renaming, and label cutting operators of the process algebras LOTOS [24], CCS [27], CSP [23], and μCRL [21], as well as the generalized parallel composition operator of E-LOTOS and LOTOS NT [18], and synchronization vectors in the style of MEC [1] and FC2 [3]. This expression is compiled into a network. In standard usage, the network is compiled into an implicit representation in C of the global LTS (initial state, transition function, etc.), which can be linked to various application programs available in CADP for simulation or verification purposes, following the OPEN/CÆSAR architecture [14]. In the framework of smart reduction, EXP.OPEN is invoked by SVL for computing the metrics, generating the networks $proj(N, I)$ and $agg(N, I)$, and generating the corresponding global LTSs.

6 Experimental Results

We have applied smart branching reduction to a set of case-studies and compared it with two other compositional aggregation strategies already implemented in SVL, namely *root leaf reduction*, which consists in aggregating all individual LTSs at once, and *node reduction*, which consists in aggregating the individual LTSs one after the other in a syntactical order given by the term describing the composition. The results are given in Figure 5, which provides for each strategy the largest number of transitions in the generated intermediate LTSs. The strategy named "Smart (HM)" (resp. "Smart (IM)") corresponds to the hiding (resp. interleaving) metric, whereas "Smart (CM)" corresponds to the combined metric. The smallest number of each line is written in bold type.

These experiments show that smart reduction is very often better than, and generally comparably efficient to, root leaf reduction and node reduction. Notable exceptions are the CFS and DFT IL experiments, for which root leaf reduction is noticeably more efficient. One reason is that EXP.OPEN uses partial order reductions and composing all individual LTSs at once may enable partial order reductions that cannot be applied to partial aggregations.

Although both hiding and interleaving metrics used separately may, in a few cases, yield slighty better results than the combined metric, we did not see cases where the combined metric is far worse than all other strategies, like the hiding

Experiment	Node	Root leaf	Smart (HM)	Smart (IM)	Smart (CM)
ABP 1	380	328	210	**104**	**104**
ABP 2	2,540	2,200	1,354	**504**	**504**
Cache	1,925	1,925	**1,848**	1,925	1,925
CFS	2,193,750	**486,990**	1,366,968	96,040,412	5,113,256
DES	22,544	**3,508**	**3,508**	14,205	14,205
DFT CAS	95,392	99,133	**336**	346	346
DFT HCPS	4,730	79,509	**425**	435	435
DFT IL	29,808	**316**	2,658	1,456	2,658
DFT MDCS	635,235	117,772	536	5,305	**346**
DFT NDPS	17,011	1,857	393	449	**346**
DLE 1	15,234	7,660	**7,709**	10,424	9,883
DLE 2	8,169	2,809	2,150	**1,852**	2,150
DLE 3	253,272	217,800	181,320	231,616	**175,072**
DLE 4	33,920	29,584	25,008	**8,896**	26,864
DLE 5	1,796,616	1,796,616	**1,403,936**	1,716,136	1,433,640
DLE 6	35,328	35,328	**5,328**	**5,328**	**5,328**
DLE 7	612,637	486,491	**369,810**	583,289	577,775
HAVi async	145,321	22,703	**21,645**	21,862	21,809
HAVi sync	19,339	5,021	**4,743**	**4,743**	**4,743**
NFP	199,728	1,986,768	104,960	**89,696**	**89,696**
ODP	158,318	158,318	87,936	**39,841**	87,936
RelRel 1	28,068	9,228	9,282	**5,574**	**5,574**
RelRel 2	11,610,235	**5,341,821**	**5,341,821**	**5,341,821**	**5,341,821**
SD 1	21,870	**3,482**	19,679	4,690	19,679
SD 2	6,561	11,997	3,624	**2,297**	3,192
SD 3	1,944	32,168	1,380	**896**	1,164
SD 4	**633,130**	1,208,592	975,872	789,886	975,872
TN	54,906,000	746,880	69,547,712	749,312	**709,504**

Fig. 5. Experimental results

metric is for TN and the interleaving metric for CFS. In that sense, combining both hiding and interleaving metrics seems to make the heuristic more robust.

7 Conclusion

We have presented smart reduction, an automated compositional aggregation strategy used to generate modulo an equivalence relation the LTS of a system made of processes composed in parallel, which generalizes previous work [29, 11]. It uses a heuristic based on a metric for evaluating the potential for an aggregation to avoid state explosion. The metric applies to the expressive network model, used internally by the EXP.OPEN tool, thus enabling the application of smart reduction to a large variety of operators (including synchronization vectors and operators from LOTOS, LOTOS NT, E-LOTOS, CCS, CSP, and μCRL), while avoiding the language restrictions that could otherwise make the optimal

order unavailable. We have provided an implementation and experimentation of smart reduction in the framework of the CADP toolbox. The resulting strategy yields good results, often better than systematic strategies such as node reduction (aggregating LTSs one after the other in an order given by the term describing the composition) and root leaf reduction (aggregating all LTSs at once). Most importantly, the metric that combines both the hiding and the interleaving metrics is *robust* in the sense that (for the test cases considered) it never leads to extremely bad orders. Smart reduction is well-integrated in the framework of the SVL scripting language of CADP, thus making it very easy to use. Smart reduction allows users of CADP to enjoy the benefits of compositional aggregation without having to guess or experimentally find a good composition order. Moreover, it enables the automatic verification of software systems that rely on elaborate forms of compositions, by combining automatic translations to compositional models and automatic compositional aggregation. Crucially, such verification can be used by software engineers or architects who have no background in formal methods. In future, we would like to have the metric integrate information about the amount of partial order reduction that can be expected in each aggregation, so as to provide even better aggregation strategies.

Acknowledgements. The authors are grateful to Christine McKinty, Gwen Salaün, Damien Thivolle, and Verena Wolf for their useful comments on this paper.

References

1. A. Arnold. MEC: A System for Constructing and Analysing Transition Systems. In *Proc. of Automatic Verif. Methods for Finite State Systems*, *LNCS* 407, 1989.
2. B. Berthomieu, J.-P. Bodeveix, P. Farail, M. Filali, H. Garavel, P. Gaufillet, F. Lang, F. Vernadat. FIACRE: an Intermediate Language for Model Verification in the TOPCASED Environment. In *Proc. of ERTS*, 2008.
3. A. Bouali, A. Ressouche, V. Roy, R. de Simone. The Fc2Tools set: a Toolset for the Verification of Concurrent Systems. In *Proc. of CAV*, *LNCS* 1102, 1996.
4. H. Boudali, P. Crouzen, M. Stoelinga. Compositional semantics for Dynamic Fault Trees in terms of Interactive Markov Chains. In *Proc. of ATVA*, *LNCS* 4762, 2007.
5. H. Boudali, P. Crouzen, M. Stoelinga. Dynamic fault tree analysis using input/output interactive Markov chains. In *Proc. of Dependable Systems and Networks*, IEEE, 2007.
6. H. Boudali, P. Crouzen, B. R. Haverkort, M. Kuntz, M. Stoelinga. Architectural dependability evaluation with Arcade. In *Proc. of Dependable Systems and Networks*, IEEE, 2008.
7. H. Boudali, H. Sözer, M. Stoelinga. Architectural Availability Analysis of Software Decomposition for Local Recovery. In *Proc. of Secure Software Integration and Reliability Improvement*, 2009.
8. G. Chehaibar, H. Garavel, L. Mounier, N. Tawbi, F. Zulian. Specification and Verification of the PowerScale Bus Arbitration Protocol: An Industrial Experiment with LOTOS. In *Proc. of FORTE/PSTV*, Chapman & Hall, 1996.
9. S. C. Cheung, J. Kramer. Enhancing Compositional Reachability Analysis with Context Constraints. In *Proc. of ACM SIGSOFT International Symposium on the Foundations of Software Engineering*. ACM Press, 1993.

10. N. Coste, H. Garavel, H. Hermanns, R. Hersemeule, Y. Thonnart, M. Zidouni. Quantitative Evaluation in Embedded System Design: Validation of Multiprocessor Multithreaded Architectures. In *Proc. of DATE*, 2008.
11. P. Crouzen, H. Hermanns. Aggregation Ordering for Massively Parallel Compositional Models. In *Proc. of ACSD*. IEEE, 2010.
12. J. Cubo, G. Salaün, C. Canal, E. Pimentel, P. Poizat. A Model-Based Approach to the Verification and Adaptation of WF/.NET Components. *ENTCS* 215, 2008.
13. H. Foster, S. Uchitel, J. Magee, J. Kramer. LTSA-WS: a tool for model-based verification of web service compositions and choreography. In *Proc. of ICSE*, 2006.
14. H. Garavel. OPEN/CÆSAR: An Open Software Architecture for Verification, Simulation, and Testing. In *Proc. of TACAS*, *LNCS* 1384, 1998.
15. H. Garavel, H. Hermanns. On Combining Functional Verification and Performance Evaluation using CADP. In *Proc. of FME*, *LNCS* 2391, 2002.
16. H. Garavel, F. Lang. SVL: a Scripting Language for Compositional Verification. In *Proc. of FORTE*, Kluwer, 2001.
17. H. Garavel, F. Lang, R. Mateescu, W. Serwe. CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes. In *Proc. of CAV*, *LNCS* 4590, 2007.
18. H. Garavel, M. Sighireanu. A Graphical Parallel Composition Operator for Process Algebras. In *Proc. of FORTE/PSTV*, Kluwer, 1999.
19. D. Giannakopoulou, J. Kramer, S. C. Cheung. Analysing the behaviour of distributed systems using TRACTA. *Journal of Automated Software Engineering*, 6(1):7–35, 1999.
20. S. Graf, B. Steffen, G. Lüttgen. Compositional Minimization of Finite State Systems using Interface Specifications. *Formal Aspects of Computation*, 8(5):607–616, 1996.
21. J. F. Groote, A. Ponse. The Syntax and Semantics of μCRL. In *Proc. of ACP*, Workshops in Computing Series, 1995.
22. H. Hermanns, J.-P. Katoen. Automated Compositional Markov Chain Generation for a Plain-Old Telephone System. *Science of Computer Programming*, 36:97–127, 2000.
23. C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978.
24. ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, International Organization for Standardization, Genève, 1989.
25. J.-P. Krimm, L. Mounier. Compositional State Space Generation from LOTOS Programs. In *Proc. of TACAS*, *LNCS* 1217, 1997.
26. F. Lang. EXP.OPEN 2.0: A Flexible Tool Integrating Partial Order, Compositional, and On-the-fly Verification Methods. In *Proc. of IFM*, *LNCS* 3771, 2005.
27. R. Milner. *A Calculus of Communicating Systems*, *LNCS* 92, 1980.
28. N.S. Rosa, P.R. Freire Cunha. A LOTOS Framework for Middleware Specification. In *Proc. of FORTE*, 2006.
29. K. C. Tai, V. Koppol. Hierarchy-Based Incremental Reachability Analysis of Communication Protocols. In *Proc. of Network Protocols*, IEEE, 1993.
30. G. Scollo, S. Zecchini. Architectural Unit Testing. *ENTCS* 111, 2005.
31. F. Tronel, F. Lang, H. Garavel. Compositional Verification Using CADP of the ScalAgent Deployment Protocol for Software Components. In *Proc. of FMOODS*, *LNCS* 2884, 2003.
32. A. Valmari. Compositional State Space Generation. In *Proc. of Advances in Petri Nets*, *LNCS* 674, 1993.