HAL
open science

Compositional Verification of Asynchronous Concurrent Systems using CADP (extended version)

Hubert Garavel, Frédéric Lang, Radu Mateescu

To cite this version:

Hubert Garavel, Frédéric Lang, Radu Mateescu. Compositional Verification of Asynchronous Concurrent Systems using CADP (extended version). [Research Report] RR-8708, INRIA Grenoble - Rhône-Alpes. 2015. hal-01138749

HAL Id: hal-01138749

https://inria.hal.science/hal-01138749v1

Submitted on 2 Apr 2015

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.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Compositional Verification of Asynchronous Concurrent Systems using CADP (extended version)

Hubert Garavel — Frédéric Lang — Radu Mateescu

N° 8708

April 2, 2015

Thème COM

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--8708--FR+ENG

Compositional Verification of Asynchronous Concurrent Systems using CADP (extended version)

Hubert Garavel , Frédéric Lang , Radu Mateescu

Thème COM — Systèmes communicants
Projet CONVECS

Rapport de recherche n° 8708 — April 2, 2015 — 65 pages

Abstract: During the last decades, concurrency theory successfully developed salient concepts to formally model and soundly reason about distributed and parallel systems. In practice, however, most attempts at analyzing large systems face severe complexity issues, especially state explosion, which prevents to exhaustively enumerate reachable state spaces. Compositionality is the most promising approach to fight state explosion. In this report, we focus on finite-state verification techniques for asynchronous message-passing systems, highlighting the existence of multiple, diverse compositional techniques such as: compositional model generation, semi-composition and projection, automatic generation of projection interfaces, formula-dependent model generation, and partial model checking. These approaches have been implemented in the framework of the CADP (*Construction and Analysis of Distributed Processes*) software toolbox and applied to large-scale, industrial systems. A key point is the ability to combine several compositional techniques, as no single technique is sufficient to address all kinds of systems.

Key-words: Bisimulation, Concurrency theory, Formal method, Labeled Transition System, Model checking, Model generation, Model minimization, Mu-calculus, Network of automata, Partial model checking, Process algebra, Projection interface, Semi-composition, Temporal logic, Verification

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

Vérification compositionnelle de systèmes concurrents asynchrones avec CADP

Résumé : Au cours des dernières décennies, la théorie de la concurrence a donné naissance à des concepts importants pour modéliser et raisonner formellement sur les systèmes parallèles et distribués. En pratique cependant, la plupart des tentatives d'analyser de gros systèmes se confronte à de sérieux problèmes de complexité, en particulier l'explosion d'états, qui empêche d'énumérer de manière exhaustive les espaces d'états atteignables. La compositionnalité est l'approche la plus prometteuse pour lutter contre l'explosion d'états. Dans ce rapport, nous nous concentrons sur les techniques de vérification d'états finis pour les systèmes asynchrones avec passage de messages, mettant en exergue l'existence de multiples et diverses techniques compositionnelles telles que : génération compositionnelle de modèle, semi-composition et projection, génération automatique d'interfaces de projection, génération de modèle dépendant de la formule, et model checking partiel. Ces approches ont été implémentées dans le contexte de la boîte à outils logiciels CADP (*Construction and Analysis of Distributed Processes*) et appliquées à des systèmes industriels de grande échelle. Un élément clé est la possibilité de combiner plusieurs techniques compositionnelles, une technique unique étant insuffisante pour traiter tous les types de systèmes.

Mots-clés : Algèbre de processus, Bisimulation, Génération de modèle, Interface de projection, Logique temporelle, Méthode formelle, Minimisation de modèle, Model checking, Model checking partiel, Mu-calcul, Réseau d'automates, Semi-composition, Système de transitions étiquetées, Théorie de la concurrence, Vérification

1 Introduction

Concurrency theory, which can be traced back to the early seventies, has produced over the years a large corpus of fundamental results for the modeling and analysis of distributed and parallel systems. Among the major theoretical successes, one can mention: *process calculi* and *process algebras* [77, 1, 107, 26, 18, 108, 109, 19], which explore the consequences of introducing parallel composition operators as first-class citizens in specification languages; *structural operational semantics* [118, 76, 1, 120, 119], which cleanly formalize the semantics of process calculi in terms of lower-level models, such as *labelled transition systems* [116]; *bisimulations* and related preorder/equivalence relations [107, 116, 108, 136, 135, 137], which enable a precise and meaningful comparison of concurrent systems; and *temporal logics* and *μ-calculus* [121, 82, 97], which provide declarative means to specify system properties.

Beyond these high-level concepts and formalisms, concurrency theory also led to concrete algorithmic advances, such as finite- and infinite-state techniques for analyzing state spaces, efficient *equivalence checking* algorithms for computing bisimulations [115, 73, 20], and powerful *model checking* algorithms for temporal logics and μ-calculus [34, 12].

Such advances have been implemented in many academic and industrial tools, which have been applied to a full range of problems, often with success. However, in most cases, the automated analysis of realistic, large-scale systems faces difficult complexity issues, especially the *state-explosion* problem that occurs when the set of reachable states of a system is too large to be analyzed, either in extension or in comprehension. Although careful specification decisions may help contain this complexity to a certain extent [113, Chapter 6][71], in general algorithmic approaches are needed to fight against such complexity while maintaining the goal of exhaustive verification; examples of such approaches are based on partial order [114, 81, 66, 133, 67, 68, 117, 54] or symmetry [78, 36, 35, 2] reductions. Probably, the most general and most promising approaches are *compositional verification* techniques, which rely on the "divide-and-conquer" paradigm to breakdown the complexity of the systems under study.

The present report focuses on compositionality in the setting of finite-state verification techniques for *asynchronous*, *action-based* systems. Concretely, this encompasses systems described using process calculi or networks of communicating finite-state automata, with underlying semantic models such as LTSs (labelled transition systems) as regards functional verification, as well as (extensions of) Markov chains as regards quantitative performance evaluation. In this context, the term *asynchronous* refers to classes of systems, the concurrent components of which may either synchronize or evolve independently, contrary to the so-called *synchronous* systems [74], in which concurrent components evolve together at periodic instants, e.g., cadenced by some clock. The term *action-based* refers to system models in which the only observable events are attached to transitions; such events are called *actions* (or *transition labels*) and may correspond, in practice, to inputs received and outputs sent by the system, as well as internal transitions performed by the system. Action-based models differ from so-called *state-based* models, in which only the internal contents of system states (namely, variables and other information stored in memory) can be observed. Although action-based and state-based can be seen as dual concepts from a mathematical point of view, they are practically different for at least two reasons: (1) action-based approaches correspond to "black box" observation while state-based approaches correspond to "white box" observation, and (2) compositional verification techniques for both kinds of approaches are quite different algorithmically.

As a matter of fact, state-based approaches are currently predominant in the scientific publications and university textbooks on model checking, whereas action-based approaches are less widespread and mostly appear in academic software tools dedicated to concurrency theory and process calculi. The present report follows this latter line, and addresses both equivalence checking and model checking, whereas state-based approaches often deal exclusively with model checking issues. It gathers numerous scientific results produced in the framework of the CADP verification toolbox [58] during the last 15–20 years, and presents a unifying overview of these results by situating them in a global, coherent landscape.

This paper is organized as follows. Section 2 gives definitions that will be used throughout the paper. Section 3 introduces *property-independent* compositional approaches; such approaches (among which *compositional model generation*, *semi-composition*, and *behavioral interfaces*) are closely related to equivalence checking and preserve a given equivalence relation, as well as all temporal logic properties adequate with respect to this equivalence relation. Section 4 presents *property-dependent* compositional approaches (among

which *property-dependent reductions* and *partial model checking*), which are specifically related to model checking and preserve the truth value of a particular set of properties (temporal logic or μ-calculus formulas) of interest. Section 5 discusses high-level strategies for effective compositional verification, among which *smart reduction* techniques. Section 6 details how the aforementioned theoretical principles are implemented within the CADP toolbox, with a particular emphasis on the SVL language that provides a powerful, user-friendly means to specify compositional verification scenarios. Section 7 presents a concrete application in which (most of) the compositional capabilities of CADP can be illustrated. Finally, Section 8 concludes the paper and lists some open issues and directions for future work. The comparison to related work is not gathered into one unique section, but done at appropriate places throughout the paper.

2 Definitions

2.1 Vector notations

Definition 1 (Vector) *A* vector $\vec{v}$ *of size n is a total function on $[1, n]$. We use the following notations:*

- *We write $|\vec{v}|$ for the size of the vector $\vec{v}$.*
- *For $i \in [1, |\vec{v}|]$, we write $\vec{v}[i]$ for $\vec{v}$ applied to i, denoting the element of $\vec{v}$ stored at index i.*
- *We write $(e_1, \ldots, e_n)$ for the vector $\vec{v}$ of size n such that $(\forall i \in [1, n])\ \vec{v}[i] = e_i$. In particular, $()$ denotes a vector of size 0.*
- *We write e^n for the vector $\vec{v}$ of size n such that $(\forall i \in [1, n])\ \vec{v}[i] = e$.*
- *Given two vectors $\vec{v}_1$ and $\vec{v}_2$, we write $\vec{v}_1 \mathbin{+\!\!+} \vec{v}_2$ for the concatenation of $\vec{v}_1$ and $\vec{v}_2$, i.e., the vector $\vec{v}$ of size $|\vec{v}_1| + |\vec{v}_2|$ such that for all $i \in [1, |\vec{v}_1|]$, $\vec{v}[i] = \vec{v}_1[i]$ and for all $i \in [|\vec{v}_1| + 1, |\vec{v}_1| + |\vec{v}_2|]$, $\vec{v}[i] = \vec{v}_2[i - |\vec{v}_1|]$.*
- *Given an element e and a vector $\vec{v}$, we also write $e :: \vec{v}$ for $(e) \mathbin{+\!\!+} \vec{v}$.*

Definition 2 (Vector projection) *Given $I = \{i_1, \ldots, i_m\} \subseteq [1, n]$ with $i_1 < \ldots < i_m$ $(0 \leq m \leq n)$, we write $\vec{v}|I$ for the projection of $\vec{v}$ on to I, defined as the vector of size m such that $(\forall j \in [1, m])\ (\vec{v}|I)[j] = \vec{v}[i_j]$. We write $\vec{v}|\overline{I}$ for $\vec{v}|([1, n] \setminus I)$.*

2.2 Labelled Transition Systems

We consider systems whose behavioural semantics can be represented using an LTS (*Labelled Transition System*).

Definition 3 (LTS) *Let $\mathcal{A}$ denote a finite set of elements called* labels*. An LTS is a tuple $(\Sigma, A, \longrightarrow, s_0)$, where:*

- *Σ is a set of states,*
- *$A \subseteq \mathcal{A}$ is a set of labels (or* actions*),*
- *$\longrightarrow \subseteq \Sigma \times A \times \Sigma$ is the (labelled) transition relation,*
- *and $s_0 \in \Sigma$ is the initial state.*

We consider a particular label written τ and called the invisible *label, which denotes* internal actions*. All labels different from τ are called the* visible *labels.*

For an LTS $S = (\Sigma, A, \longrightarrow, s_0)$, we may write $s \xrightarrow{a} s'$ instead of $(s, a, s') \in \longrightarrow$. We write $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$. Given a regular expression r built upon labels, we write $s \xrightarrow{r} s'$ if there exist $a_1, \ldots, a_m$ and $s_1, \ldots, s_m$ $(m \geq 0)$ such that $s \xrightarrow{a_1} s_1 \xrightarrow{a_2} \ldots \xrightarrow{a_m} s_m$, the sequence $a_1 \ldots a_m$ belongs to the regular language denoted by r, and $s_m = s'$.

Definition 4 (LTS inclusion) *Let $S_i = (\Sigma_i, A_i, \longrightarrow_i, s_i^0)$ for $i \in \{1,2\}$. We say that S_1 is included in S_2, written $S_1 \subseteq S_2$, if $s_1^0 = s_2^0$, $\Sigma_1 \subseteq \Sigma_2$, $A_1 \subseteq A_2$, and $\longrightarrow_1 \subseteq \longrightarrow_2$.*

Definition 5 (Equality modulo reachability) *Let $S_i = (\Sigma_i, A_i, \longrightarrow_i, s_i^0)$ for $i \in \{1,2\}$. The reachable part of S_i is the LTS $(\Sigma'_i, S'_i, \longrightarrow'_i, s_i^0)$, where:*

- $\longrightarrow'_i = \{(s, a, s') \in \longrightarrow_i \mid s_i^0 \longrightarrow_i^* s\}$
- $A'_i = \{a \in A_i \mid \exists (s, a, s') \in \longrightarrow'_i\}$
- $\Sigma'_i = \{s_i^0\} \cup \{s' \in \Sigma_i \mid \exists (s, a, s') \in \longrightarrow'_i\}$

S_1 is equal to S_2 modulo reachability, written $S_1 \stackrel{\mathsf{r}}{=} S_2$, if the reachable parts of S_1 and S_2 are equal.

2.3 Networks of LTSs

To deal with compositions of LTSs, we use a general model called *network of LTSs* (or *network* for short) [85], which is inspired from the MEC [11] and FC2 [24] synchronization vectors.

Definition 6 (Network of LTSs) *A* network of LTSs*, written N, is a pair $(\vec{S}, V)$, where $\vec{S}$ is a vector of LTSs (called* component LTSs*), and V is a set of* synchronization rules*. Each synchronization rule has the form $(\vec{t}, a)$, where $a \in \mathcal{A}$ is a label and $\vec{t}$ is a vector comprising labels and occurrences of a special symbol $\bullet \notin \mathcal{A}$, such that $|\vec{t}| = |\vec{S}|$. Each $\vec{t}$ is called a* synchronization vector*. The* size *of the network N is defined as $|\vec{S}|$. Let $\vec{S}[i] = (\Sigma_i, A_i, \longrightarrow_i, s_i^0)$ $(i \in [1,n])$. To N one can associate a "*compound*" LTS* $\mathsf{lts}\ (N)$ *that is the parallel composition of component LTSs. Each $(\vec{t}, a) \in V$ defines transitions labelled by a, obtained either by synchronization (if more than one index i is such that $\vec{t}[i] \neq \bullet$) or by interleaving (otherwise) of component LTS transitions. Formally,* $\mathsf{lts}\ (N) = (\Sigma, A, \longrightarrow, \vec{s}_0)$*, where:*

- $\Sigma = \Sigma_1 \times \ldots \times \Sigma_n$,
- $A = \{a \mid \exists (\vec{t}, a) \in V\}$,
- $\vec{s}_0 = (s_1^0, \ldots, s_n^0)$, *and*
- $\longrightarrow$ *is the relation satisfying $\vec{s} \xrightarrow{a} \vec{s}'$ if and only if there exists $(\vec{t}, a) \in V$ such that for all $i \in [1,n]$:*

$$\begin{cases} \vec{s}'[i] = \vec{s}[i] & \text{if } \vec{t}[i] = \bullet \\ \vec{s}[i] \xrightarrow{\vec{t}[i]}_i \vec{s}'[i] & \text{otherwise} \end{cases}$$

We define $\mathsf{acv}(\vec{t}) = \{i \mid i \in [1,n] \wedge \vec{t}[i] \neq \bullet\}$, the set of active *LTS indices of the synchronization vector $\vec{t}$.*

Example 1 *Let a, b, c, and d be labels, and S_1, S_2, and S_3 be the LTSs defined in Figure 1 (top), where the initial states are denoted by bold circles. Let $N = ((S_1, S_2, S_3), V)$ with $V = \{((a, a, \bullet), a), ((a, \bullet, a), a), ((b, b, b), b), ((c, c, \bullet), \tau), ((\bullet, \bullet, d), d)\}$. The first two synchronization rules of N express a nondeterministic synchronization on a between either S_1 and S_2, or S_1 and S_3. The third rule expresses a multiway synchronization on b. The fourth rule yields an internal (τ) transition. The fifth rule expresses full interleaving of transitions labelled by d. The compound LTS corresponding to N is depicted in Figure 1 (bottom).*

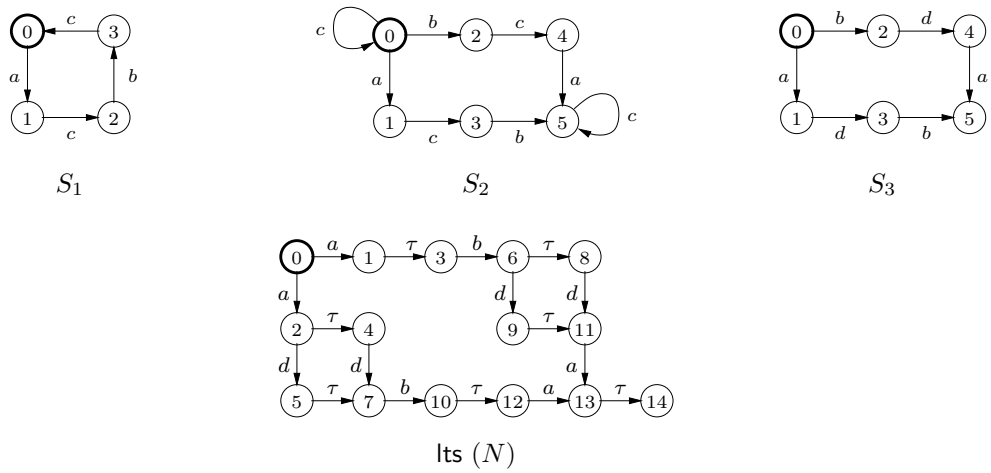


Figure 1: Compound LTS corresponding to N defined in Example 1

B	::=	S	LTS
	|	$B_1 \mid[A]\mid B_2$	
	|	$B_1 \mid\mid\mid B_2$	
	|	$B_1 \mid\mid B_2$	parallel compositions
	|	**rename** $a_1 \rightarrow a'_1, \ldots, a_n \rightarrow a'_n$ **in** B_0	rename
	|	**hide** A **in** B_0	hide
	|	**cut** A **in** B_0	cut

where $A \subseteq \mathcal{A} \setminus \{\tau\}, \{a_1, \ldots, a_n\} \subseteq \mathcal{A} \setminus \{\tau\}, (\forall i \neq j)\ a_i \neq a_j, \{a'_i, \ldots, a'_n\} \subseteq \mathcal{A}$

Figure 2: Composition expressions

2.4 Composition expressions

Alternatively to networks, which compose LTSs in a flat manner, LTSs can be composed together using higher-level algebraic operators, such as parallel composition, label hiding, label renaming, and transition cutting, which we define below. Many operators available in the literature can be considered, e.g., from μCRL [72], CCS [108], CSP [125], LOTOS [79], E-LOTOS [80], or LNT [28] without theoretical difficulty, but for the sake of brevity, this paper considers only expressions comprising LTSs composed with LOTOS-like parallel composition, hiding, renaming, and cutting operators.

Definition 7 (Composition expression) *The syntax of* composition expressions*, written $B, B_0, B_1, \ldots$, is given in Figure 2. Each composition expression B has semantics in terms of an LTS written $\mathsf{lts}\,(B)$, defined as follows. By convention, given a composition expression B_i, we write $(\Sigma_i, A_i, \longrightarrow_i, s_i^0)$ for $\mathsf{lts}\,(B_i)$.*

1. *The composition expression "S" denotes the LTS S, whose semantics are defined by $\mathsf{lts}\,(S) = S$.*
2. *The composition expression "$B_1 \mid[A]\mid B_2$" denotes the parallel composition of B_1 and B_2 with synchronization on the labels belonging to A. Its semantics are defined by $\mathsf{lts}\,(B_1 \mid[A]\mid B_2) = (\Sigma_1 \times \Sigma_2, A_1 \cup A_2, \longrightarrow, (s_1^0, s_2^0))$, where $\longrightarrow$ is the smallest relation satisfying the following SOS (*Structural Operational Semantics*) [1] rules:*

$$\frac{s_1 \xrightarrow{a}_1 s'_1 \quad a \notin A}{(s_1, s_2) \xrightarrow{a} (s'_1, s_2)} \qquad \frac{s_2 \xrightarrow{a}_2 s'_2 \quad a \notin A}{(s_1, s_2) \xrightarrow{a} (s_1, s'_2)}$$

$$\frac{s_1 \xrightarrow{a}_1 s'_1 \quad s_2 \xrightarrow{a}_2 s'_2 \quad a \in A}{(s_1, s_2) \xrightarrow{a} (s'_1, s'_2)}$$

The notations "B_1 $|||$ B_2" and "B_1 $||$ B_2" are respective shorthands for "B_1 $|[\emptyset]|$ B_2" and "B_1 $|[\mathcal{A} \setminus \{\tau\}]|$ B_2".

3. *The composition expression "***rename** $a_1 \rightarrow a'_1, \ldots, a_n \rightarrow a'_n$ **in** B_0*" denotes B_0 in which each visible action a_i ($i \in [1, n]$) is replaced by the corresponding a'_i. For convenience, we may write θ instead of $a_1 \rightarrow a'_1, \ldots, a_n \rightarrow a'_n$. To $\theta = a_1 \rightarrow a'_1, \ldots, a_n \rightarrow a'_n$ we associate a total function $\overline{\theta} : \mathcal{A} \rightarrow \mathcal{A}$ defined as follows:*

$$\overline{\theta}(a) = \begin{cases} a' & \text{if } (\exists i \in [1, n])\ a = a_i \wedge a' = a'_i \\ a & \text{otherwise} \end{cases}$$

The semantics of rename are defined by $\mathsf{lts}\,(\textbf{rename}\ \theta\ \textbf{in}\ B_0) = (\Sigma_0, (A_0 \setminus \{a_1, \ldots, a_n\}) \cup \{a'_1, \ldots, a'_n\}, \longrightarrow, s_0^0)$*, where $\theta = a_1 \rightarrow a'_1, \ldots, a_n \rightarrow a'_n$ and $\longrightarrow$ is the smallest relation satisfying the following SOS rule:*

$$\frac{s \xrightarrow{a}_0 s'}{s \xrightarrow{\overline{\theta}(a)} s'}$$

4. *The composition expression "***hide** A **in** B_0*" denotes B_0 where labels in A are renamed into the invisible label τ, i.e., turned into an unobservable action. Its semantics are defined by* $\mathsf{lts}\,(\textbf{hide}\ A\ \textbf{in}\ B_0) = (\Sigma_0, (A_0 \setminus A) \cup \{\tau\}, \longrightarrow, s_0^0)$*, where $\longrightarrow$ is the smallest relation satisfying the following SOS rules:*

$$\frac{s \xrightarrow{a}_0 s' \quad a \in A}{s \xrightarrow{\tau} s'} \qquad \frac{s \xrightarrow{a}_0 s' \quad a \notin A}{s \xrightarrow{a} s'}$$

If $A = \{a_1, \ldots, a_n\}$, then **hide** A **in** B *is equivalent to* **rename** $a_1 \rightarrow \tau, \ldots, a_n \rightarrow \tau$ **in** B.

5. *The composition expression "***cut** A **in** B_0*" denotes B_0 where every transition labelled by an element of A is deleted. Its semantics are defined by* $\mathsf{lts}\,(\textbf{cut}\ A\ \textbf{in}\ B_0) = (\Sigma_0, A_0 \setminus A, \longrightarrow, s_0^0)$*, where $\longrightarrow$ is the smallest relation satisfying the following SOS rule:*

$$\frac{s \xrightarrow{a}_0 s' \quad a \notin A}{s \xrightarrow{a} s'}$$

It follows that $\xrightarrow{a} \subseteq \xrightarrow{a}_0$.

Note that a composition expression is a syntactic algebraic expression structured as a tree, which we call the *component hierarchy*. We could further enrich composition expressions with operators borrowed from other process calculi than LOTOS.

There exists a mapping from composition expressions to networks: every composition expression B can be translated into a network of LTSs, by flattening the component hierarchy. This network is written $\mathsf{net}\,(B)$ and defined as the couple $(\mathsf{ind}\,(B), \mathsf{sync}\,(B))$, where $\mathsf{ind}\,(B)$ and $\mathsf{sync}\,(B)$ denote respectively the vector of

component LTSs and the synchronization rules corresponding to B. These functions are defined as follows:

$$
\begin{aligned}
&\mathsf{ind}\,(S) = (S)\\
&\mathsf{ind}\,(B_1 \,|[A]|\, B_2) = \mathsf{ind}\,(B_1) \mathbin{+\!\!+} \mathsf{ind}\,(B_2)\\
&\mathsf{ind}\,(\textbf{hide}\ A\ \textbf{in}\ B_0) = \mathsf{ind}\,(B_0)\\
&\mathsf{ind}\,(\textbf{cut}\ A\ \textbf{in}\ B_0) = \mathsf{ind}\,(B_0)\\
&\mathsf{ind}\,(\textbf{rename}\ \theta\ \textbf{in}\ B_0) = \mathsf{ind}\,(B_0)
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{sync}\,(S) = \{((a), a) \mid a \in A\} \text{ where } S = (\Sigma, A, \longrightarrow, s_0)\\
&\mathsf{sync}\,(B_1 \,|[A]|\, B_2) =\\
&\qquad \{(\vec{t}_1 \mathbin{+\!\!+} \bullet^{|\mathsf{ind}\,(B_2)|}, a) \mid (\vec{t}_1, a) \in \mathsf{sync}\,(B_1) \wedge a \notin A\} \cup\\
&\qquad \{(\bullet^{|\mathsf{ind}\,(B_1)|} \mathbin{+\!\!+} \vec{t}_2, a) \mid (\vec{t}_2, a) \in \mathsf{sync}\,(B_2) \wedge a \notin A\} \cup\\
&\qquad \{(\vec{t}_1 \mathbin{+\!\!+} \vec{t}_2, a) \mid (\vec{t}_1, a) \in \mathsf{sync}\,(B_1) \wedge (\vec{t}_2, a) \in \mathsf{sync}\,(B_2) \wedge a \in A\}\\
&\mathsf{sync}\,(\textbf{rename}\ \theta\ \textbf{in}\ B_0) =\\
&\qquad \{(\vec{t}_0, \overline{\theta}(a)) \mid (\vec{t}_0, a) \in \mathsf{sync}\,(B_0)\}\\
&\mathsf{sync}\,(\textbf{hide}\ A\ \textbf{in}\ B_0) =\\
&\qquad \{(\vec{t}_0, \tau) \mid (\vec{t}_0, a) \in \mathsf{sync}\,(B_0) \wedge a \in A\} \cup\\
&\qquad \{(\vec{t}_0, a) \mid (\vec{t}_0, a) \in \mathsf{sync}\,(B_0) \wedge a \notin A\}\\
&\mathsf{sync}\,(\textbf{cut}\ A\ \textbf{in}\ B_0) =\\
&\qquad \{(\vec{t}_0, a) \mid (\vec{t}_0, a) \in \mathsf{sync}\,(B_0) \wedge a \notin A\}
\end{aligned}
$$

Example 2 *The network of Example 1 corresponds to the composition expression* **hide** c **in** $(S_1 \,|[a,b,c]|\, (S_2 \,|[b]|\, S_3))$.

Networks are more general than composition expressions, as illustrated by the following example.

Example 3 *Networks enable m-among-n synchronization [61], where any m LTSs among the n LTSs of the composition synchronize on a given label. For instance, 2-among-3 synchronization on a between LTSs* S_1, S_2, *and* S_3 *can be achieved using the network* $((S_1, S_2, S_3), \{((a, a, \bullet), a), ((a, \bullet, a), a), ((\bullet, a, a), a)\})$. *This type of synchronization cannot be described using classical process algebraic operators [61].*

2.5 LTS Equivalences

Equivalence relations on LTSs characterize semantic equivalences between systems. Several relations are available in the literature, differing mainly in their treatment of invisible labels. We focus here on a few of them, namely strong bisimulation, branching bisimulation and its divergence-sensitive variant, and $\tau^*.a$ equivalence.

Definition 8 (Strong bisimulation [116]) *A* strong bisimulation *is a symmetric relation* $R \subseteq \Sigma \times \Sigma$ *such that if* $(s_1, s_2) \in R$ *then: for all* $s_1 \xrightarrow{a} s_1'$, *there exists* s_2' *such that* $s_2 \xrightarrow{a} s_2'$ *and* $(s_1', s_2') \in R$.

Two states s_1 *and* s_2 *are strongly bisimilar if there exists a strong bisimulation* R *such that* $(s_1, s_2) \in R$. *Two LTSs are strongly bisimilar if their initial states are strongly bisimilar.*

Definition 9 (Branching bisimulation [136, 137]) *A* branching bisimulation *is a symmetric relation* $R \subseteq \Sigma \times \Sigma$ *such that if* $(s_1, s_2) \in R$ *then: for all* $s_1 \xrightarrow{a} s_1'$, *either* $a = \tau$ *and* $(s_1', s_2) \in R$, *or there exists a sequence* $s_2 \xrightarrow{\tau^*} s_2' \xrightarrow{a} s_2''$ *such that* $(s_1, s_2') \in R$ *and* $(s_1', s_2'') \in R$.

Two states s_1 *and* s_2 *are branching bisimilar if there exists a branching bisimulation* R *such that* $(s_1, s_2) \in R$. *Two LTSs are branching bisimilar if their initial states are branching bisimilar.*

Branching bisimulation does not distinguish between inaction and a cycle of internal actions. Divergence-sensitive branching bisimulation is introduced to take into account cycles of internal actions.

Definition 10 (Divergence-sensitive branching bisimulation [136, 137]) *A* divergence-sensitive branching bisimulation *(or* divbranching bisimulation *for short) is a branching bisimulation R such that if $(s_1^0, s_2^0) \in R$ and there is an infinite sequence $s_1^0 \xrightarrow{\tau} s_1^1 \xrightarrow{\tau} s_1^2 \xrightarrow{\tau} \ldots$ with $(s_1^i, s_2^0) \in R$ for all $i \geq 0$, then there is an infinite sequence $s_2^0 \xrightarrow{\tau} s_2^1 \xrightarrow{\tau} s_2^2 \xrightarrow{\tau} \ldots$ such that $(s_1^i, s_2^j) \in R$ for all $i, j \geq 0$.*
Two states s_1 and s_2 are divbranching bisimilar if there exists a divbranching bisimulation R such that $(s_1, s_2) \in R$. Two LTSs are divbranching bisimilar if their initial states are divbranching bisimilar.

Definition 11 ($\tau^*.a$ equivalence [51]) *A* $\tau^*.a$ equivalence *is a symmetric relation $R \subseteq \Sigma \times \Sigma$ such that if $(s_1, s_2) \in R$ then: for each sequence $s_1 \xrightarrow{\tau^*.a} s_1'$ where a is a visible label, there exists a sequence $s_2 \xrightarrow{\tau^*.a} s_2'$ such that $(s_1', s_2') \in R$.*

Two states s_1 and s_2 are $\tau^*.a$ equivalent if there exists a $\tau^*.a$ equivalence R such that $(s_1, s_2) \in R$. Two LTSs are $\tau^*.a$ equivalent if their initial states are $\tau^*.a$ equivalent.

Other equivalence relations are defined in the literature and possibly used in our framework of composition expressions and networks, such as observation equivalence [108], safety equivalence [23], trace equivalence (also known as *language equivalence*), and weak trace equivalence [26]. For the sake of conciseness, we do not give their definitions in this paper.

Equivalence relations and their associated preorders can be used to compare the behaviour of two systems. Equivalence relations can also be used to compute a "*canonical form*" for each LTS, by identifying and merging equivalent states and by replacing equivalent transition sequences by a single, representative one. Such an operation is often called LTS *minimization*, or *reduction* if applied partially. In the case of strong and branching bisimulations, the resulting LTS is indeed the smallest (in both the number of states and the number of transitions) of its equivalence class. In the case of $\tau^*.a$ equivalence, the representative of every sequence of transitions matching $\tau^*.a$ is a single transition labelled by a, i.e., the τ-transitions are eliminated. This yields an LTS that is not necessarily minimal in the number of transitions, but always minimal in the number of states (and obviously in the number of τ-transitions). Note that there exist other equivalence relations whose canonical forms may be dramatically larger (in the number of states) than the smallest elements of their equivalence classes. This is the case of trace and weak trace equivalences, whose canonical forms require determinization. This illustrates an advantage of bisimulations and equivalence relations preserving the branching structure of LTSs over relations dealing with traces.

Definition 12 *Given an LTS equivalence relation R and an LTS S, we write $\mathsf{reduce}_R(S)$ for the LTS corresponding to S minimized modulo R. Given a network N (respectively a composition expression B) and an LTS equivalence relation R, we write $\mathsf{reduce}_R(N)$ (respectively $\mathsf{reduce}_R(B)$) as a shorthand for $\mathsf{reduce}_R(\mathsf{lts}(N))$ (respectively $\mathsf{reduce}_R(\mathsf{lts}(B))$).*

2.6 Congruence results

Definition 13 (Congruence for networks) *An LTS equivalence relation R is a congruence for networks if and only if for all networks $N = (\vec{S}, V)$, for all $i \in [1, n]$ where n is the size of N, and for all S_i equivalent to $\vec{S}[i]$ modulo R, then $\mathsf{lts}(\vec{S}, V)$ is equivalent modulo R to $\mathsf{lts}(\vec{S}', V)$ where $\vec{S}'$ is the same as $\vec{S}$ in which $\vec{S}[i]$ has been replaced by S_i'.*

Proposition 1 *Strong bisimulation and trace equivalence are congruences for networks [85].*

Proposition 2 *Branching bisimulation, divergence-sensitive branching bisimulation, observation equivalence, safety equivalence, and weak trace equivalence [26], are congruences for networks provided that the synchronization rules satisfy the following constraints [85] regarding the internal transitions of component LTSs, for all $i \in [1, n]$:*

- *Internal transitions should not be synchronized:*

$$(\vec{t}, a) \in V \wedge \vec{t}[i] = \tau \implies \mathsf{acv}(\vec{t}) = \{i\}$$

- *Internal transitions should not be renamed:*

$$(\vec{t}, a) \in V \wedge \vec{t}[i] = \tau \implies a = \tau$$

- *Internal transitions should not be cut:*

$$\xrightarrow{\tau}_i \neq \emptyset \implies (\exists(\vec{t}, \tau) \in V)\ \vec{t}[i] = \tau$$

In the sequel, we consider only networks satisfying the constraints defined in Proposition 2. Note that these constraints are natural, and that they are satisfied by the networks obtained from composition expressions by translation.

Note also that $\tau^*.a$ equivalence is neither a congruence for networks nor for composition expressions. However, it can be used compositionally as a pre-reduction for weaker congruence relations, such as safety and weak trace equivalences.

3 Property-independent compositional approaches

3.1 Basic compositional LTS generation

In its simplest forms [49, 95, 126, 140, 128, 129, 134, 125], compositional verification (also called incremental reduction [126], incremental reachability analysis [128, 129], compositional state space generation [134], or inductive compression [125]) consists in replacing each component LTS by an *abstraction*, simpler than the original component but still preserving the properties to be verified on the whole system.

The approach for compositionality presented in this section deals with the construction of a minimal (or, at least, reduced) LTS modulo a given equivalence relation such as strong, branching, or observational equivalence, starting from a composition expression or a network. The chosen abstraction is thus LTS minimization modulo an appropriate equivalence relation: instead of $\mathsf{lts}\,((S_1, \ldots, S_n), V)$, we compute $\mathsf{lts}\,((\mathsf{reduce}_R\,(S_1), \ldots, \mathsf{reduce}_R\,(S_n)), V)$. For weak equivalence relations, such as branching bisimulation, this may yield substantial reductions of the initial LTS, and consequently facilitate its subsequent analysis.

This abstraction is sound as long as the equivalence relation R is a congruence for the composition expression. Minimization can also be applied at any intermediate level in the composition, thus opening the way for various compositional minimization strategies. This will be detailed in Section 5.

Although basic compositional LTS generation has been applied successfully to some complex systems (e.g., [50, 29] in the case of the LOTOS language [79]), it may be counter-productive in some other cases: generating the LTS of each component separately may lead to state explosion, whereas the generation of the whole system at once might succeed if components constrain each other when composed in parallel. Indeed, there may be many states of a component that, although useful in a general environment, are useless (i.e., never explored) in a particular environment.

3.2 Interfaces and projections

Enhanced compositional verification approaches [69, 31, 139, 32, 33, 70, 83, 30, 63] have been proposed to generate the LTS of a component by taking into account *interface constraints* (also known as *environment constraints* or *context constraints*). These constraints express the behavioural restrictions imposed on the considered component by synchronization with its neighbour components. Taking into account the environment of a component permits local elimination of states and transitions unreachable in the LTS of the whole system.

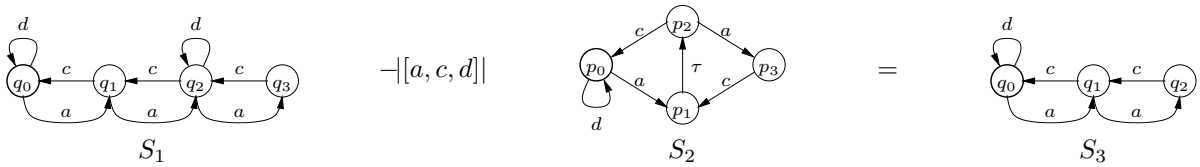
In general, interface constraints are expressed in the form of an LTS called *interface*. There are two approaches to restrict the behaviour of a component w.r.t. an interface. In the first one [30, 31, 32, 33, 63], the component is composed in parallel with the interface, which must have been transformed beforehand so that the composition does not affect the global behaviour of the system (a property known as *context transparency*).

In the second approach, the component is constrained using a specific *semi-composition* operator [69, 70, 83], which cuts the component states and transitions that cannot be reached when considering the traces of

the interface as the only possible interactions between the component and its environment. We consider the second approach. The semi-composition operator is defined as follows.

Definition 14 (Semi-composition) *Let* $S_i = (\Sigma_i, A_i, \longrightarrow_i, s_i^0)$ $(i = 1, 2)$ *be two LTSs,* A *be a set of visible labels, and* $(\Sigma_{||}, A_{||}, \longrightarrow_{||}, s_{0||}) = \mathsf{lts}\,(S_1 \,|[A]|\, S_2)$*. The* semi-composition of S_1 and S_2*, written "*$S_1 \dashv[A]|\, S_2$*", is the LTS* $(\Sigma, A_1, \longrightarrow, s_1^0)$*, where* $\longrightarrow\; = \;\longrightarrow_1 \cap \{(s_1, a, s_1') \mid (s_1^0, s_2^0) \longrightarrow_{||}^{*} (s_1, s_2) \xrightarrow{a}_{||} (s_1', s_2')\}$ *and* $\Sigma = \{s \mid (\exists a, s')\ s \xrightarrow{a} s' \vee s' \xrightarrow{a} s\}$*. A is called the* synchronization set *and the pair* (A, S_2) *is called the* interface[1]*. We say that a label* $a \in A_1$ *is* controlled *by the interface* (A, S_2) *if* $a \in A$*. We write* $S_1 \dashv|\, S_2$ *for* $S_1 \dashv[\mathcal{A} \setminus \{\tau\}]|\, S_2$*.*

Example 4 *The following holds:*



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* $S_1 \,|[a, c, d]|\, S_2$*. Therefore, state* q_3 *is unreachable in* S_3*.*

The following properties ensure the applicability of semi-composition.

Proposition 3 *Semi-composition cannot increase the size of the LTS to which it is applied:* $(\forall A, S_1, S_2)\ \mathsf{lts}\,(S_1 \dashv[A]|\, S_2) \subseteq S_1$*.*

Proposition 4 $(\forall A, S_1, S_2)\ \mathsf{lts}\,(S_1 \dashv[A]|\, S_2) = S_1$ *if the visible traces of "***hide** $\mathcal{A} \setminus A$ **in** S_1*" are included in the visible traces of "***hide** $\mathcal{A} \setminus A$ **in** S_2*".*

Proposition 5 (Laws of semi-composition [83])

$$S_1 \,|[A]|\, S_2 \;\stackrel{\mathsf{r}}{=}\; (S_1 \dashv[A]|\, S_2) \,|[A]|\, S_2 \tag{1}$$

$$(S_1 \,|[A]|\, S_3) \,|[A']|\, S_2 \;\stackrel{\mathsf{r}}{=}\; ((S_1 \dashv[A' \cap (A \cup (A_1 \setminus A_3))]|\, S_2) \,|[A]|\, S_3) \,|[A']|\, S_2 \tag{2}$$

where A_1 *is the set of labels of* S_1*, and* A_3 *is the set of labels of* S_3

$$(\textbf{hide } A \textbf{ in } S_1) \,|[A']|\, S_2 \;\stackrel{\mathsf{r}}{=}\; (\textbf{hide } A \textbf{ in } (S_1 \dashv[A' \setminus A]|\, S_2)) \,|[A']|\, S_2 \tag{3}$$

Proposition 5 defines how semi-composition can be used to reduce S_1 given an LTS S_2 in its environment, by removing the unreachable states and transitions, without losing any temporal property of the system. Note that, unlike the approach of Cheung & Kramer, 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.

Proposition 6 $S_1 \dashv[A]|\, S_2 = S_1 \dashv[A]|\, S_2'$ *if "***hide** $\mathcal{A} \setminus A$ **in** S_2*" and "***hide** $\mathcal{A} \setminus A$ in S_2'*" have the same visible traces.*

This proposition implies that the uncontrolled labels can be hidden in the interface and that the resulting interface can then be minimized modulo any relation preserving visible traces (e.g., *safety equivalence* [23]), 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 never contains more states than the input. Minimization of the interface is not mandatory but helps reduce the cost of semi-composition.

[1] This definition of semi-composition is simpler but equivalent to that given in [83].

3.3 Automatic generation of interfaces

Interfaces can be either written by the user (and possibly checked automatically [83]) 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 often undergo two main limitations: first, these techniques are specific to a given parallel composition operator and thus not directly applicable in the framework of concurrent languages featuring different and/or more general parallel composition operators; second, as already observed in [31], they may fail to capture effective interface constraints due to deficiencies in their analysis of synchronizations between components.

In this section, we propose to generate interfaces automatically from networks of LTSs, following the approach proposed in [86]. The network of LTS intermediate representation permits the derivation of effective interface constraints imposed on a given component by a subset of its neighbours automatically, independently of the component hierarchy and of the nature of the parallel composition operators. This permits combination of constraints induced by distant components, and improvement of the accuracy of interfaces by exploiting more precisely the synchronizations between components. For this reason, we qualify as *refined* the interfaces generated using this technique.

A method was previously proposed in [83] to compute automatically an exact interface in the framework of composition expressions built upon parallel composition and label hiding. In this method, defined inductively based on the semi-composition laws described in Proposition 5, two component LTSs S_1 and S_2 are selected and a synchronization set A is computed such that S_1 can be replaced by $S_1 \dashv[A]| S_2$ without changing the compound LTS. The interface (A, S_2) built using this method generally does not give the best account of environment constraints.

Here, we propose to generate automatically interfaces that give a better account of environment constraints, using *refined interface generation* [86]. This method uses the information available in the intermediate network model to compute an interface capturing the constraints imposed on a given component P in a concurrent system by one or several neighbour components. This interface can then be semi-composed with P on-the-fly, so as to restrict P's behaviour.

Definition 15 (Refined interface generation) *Given a network $N = (\vec{S}, V)$ of size n, an index $k \in [1, n]$, and a nonempty set of indices $I \subseteq [1, n] \setminus \{k\}$, the* refined interface *of $\vec{S}[k]$ capturing constraints induced by $\{\vec{S}[i] \mid i \in I\}$, written $\mathsf{refint}\,(N, k, I)$, is the couple $(\mathcal{A}, S_I)$, where:*

$$S_I = \mathsf{lts}\,(\vec{S}|I, V_I)$$

$$V_I = \{(\vec{t}|I, \vec{t}[k]) \mid (\exists a)\ (\vec{t}, a) \in V \wedge \vec{t}[k] \neq \bullet\} \cup \{(\vec{t}|I, \tau) \mid (\exists a)\ (\vec{t}, a) \in V \wedge \vec{t}[k] = \bullet\}$$

The refint *operation may create synchronization rules of the form $(\bullet^n, a)$, which induce a self-looping transition labelled by a in each state of the interface. Some of these synchronization rules can be eliminated as follows:*

- *Every synchronization rule of the form $(\bullet^n, \tau)$ can merely be removed, as τ loops do not affect the visible traces of the interface.*
- *Every synchronization rule of the form $(\bullet^n, a)$ where $a \neq \tau$ can be removed and then a removed from the synchronization set if the set of synchronization rules does not contain another rule with the same label a as right-hand side. Indeed, for all $\vec{S}$, S', A, and V in which a does not occur as a right-hand side, $S' \dashv[A]| \mathsf{lts}\,(\vec{S}, V \cup (\bullet^n, a)) = S' \dashv[A \setminus \{a\}]| \mathsf{lts}\,(\vec{S}, V)$.*

The refined interface thus consists of a product of the LTSs $\vec{S}[i]$ $(i \in I)$, synchronized by synchronization rules derived systematically from the synchronization rules of N, each rule $(\vec{t}, a)$ being transformed into a rule $(\vec{t}|I, \vec{t}[k])$ if $\vec{t}[k] \neq \bullet$, or $(\vec{t}|I, \tau)$ otherwise. Therefore, whenever a transition $\vec{q} \xrightarrow{a} \vec{q}'$ can be fired in $\mathsf{lts}\,(N)$ using a synchronization rule $(\vec{t}, a)$ with $\vec{t}[k] \neq \bullet$, then the participating transition $\vec{q}[k] \xrightarrow{\vec{t}[k]} \vec{q}'[k]$ of $\vec{S}[k]$ is also a transition of $\vec{S}[k] \dashv[A]| S_I$. Conversely, transitions of $\vec{S}[k]$ that cannot participate in any mandatory synchronization with S_I are eliminated by the semi-composition $\vec{S}[k] \dashv[A]| S_I$.

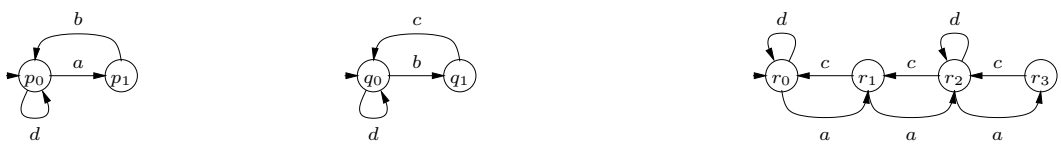
Example 5 *Consider the network N displayed at the left below, with arbitrary LTSs $S_1, \ldots, S_4$. The refined interface of S_1 capturing constraints induced by S_3 and S_4, written* $\mathsf{refint}\,(N, 1, \{3,4\})$*, has the LTS corresponding to the network displayed at the right below. Note the 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.*

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

Proposition 7 *Let N be a network of LTSs of size n, $k \in [1,n]$, $I \subseteq [1,n] \setminus \{k\}$, and $(A, S_I) = \mathsf{refint}\,(N, k, I)$. If $\vec{S}'$ is the vector of LTSs of size n defined by $(\forall i \in [1,n] \setminus \{k\})\ \vec{S}'[i] = \vec{S}[i]$ and $\vec{S}'[k] = \vec{S}[k] \dashv|[A]|\ S_I$, then $\mathsf{lts}\,(\vec{S}, V) \stackrel{\mathsf{r}}{=} \mathsf{lts}\,(\vec{S}', V)$.*

The following examples show that refined interfaces may constrain components better than the interfaces generated by the method proposed in [83].

Example 6 *Let $B = S_1\ |[a,b,d]|\ (S_2\ |[c,d]|\ S_3)$ with S_1, S_2, and S_3 defined as follows:*



S_1 S_2 S_3

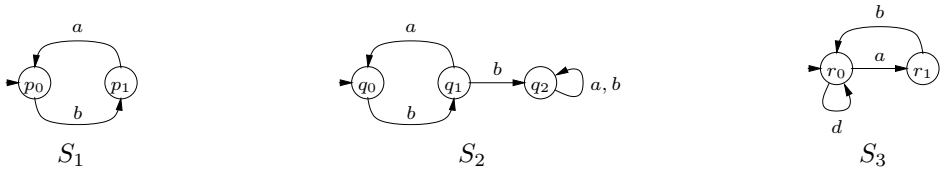
According to the semi-composition laws of Proposition 5, S_3 can be replaced in B either by $S_3 \dashv|[a,d]|\ S_1$, or by $S_3 \dashv|[c,d]|\ S_2$, but both expressions result in S_3 itself. Yet, one can see that actions a and c are executed with some alternation in B, due to the mandatory synchronization on b between S_1 and S_2. As a consequence, state r_3 is not reachable in B. 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 B containing S_1 and S_2 only (parallel composition is not associative). This is not possible using the method described in [83][2], but this can be done using refined interface generation. To this aim, the expression $B = S_1\ |[a,b,d]|\ (S_2\ |[c,d]|\ S_3)$ is translated into the network N displayed below and then S_3 may be restricted using the refined interface $\mathsf{lts}\,(N') = \mathsf{refint}\,(N, 3, \{1,2\})$ that takes simultaneously both S_1 and S_2 into account, where N' and $\mathsf{lts}\,(N')$ are displayed below. $S_3 \dashv|\ \mathsf{lts}\,(N')$, also displayed below, is strictly included in S_3 as the unreachable state r_3 and transitions $r_2 \xrightarrow{a} r_3, r_3 \xrightarrow{c} r_2$, and $r_2 \xrightarrow{d} r_2$ are deleted.

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

$\mathsf{lts}\,(N')$ $S_3 \dashv|\ \mathsf{lts}\,(N')$

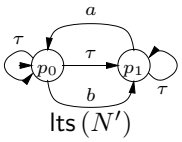
[2]This limitation holds similarly for the approach described in [31].

Example 7 *Let $B = S_1 \,|[a,b]|\, (S_2 \,|[a]|\, S_3)$ with S_1, S_2, and S_3 defined as follows:*



According to the semi-composition laws of Proposition 5, S_2 can be replaced by $S_2 \dashv|[a]|\, S_1$, but this expression yields S_2 itself. Yet, it is clear from S_1 and the synchronizations in B that state q_2 of S_2 is unreachable in B, 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. Such an interface can be obtained using refined interface generation. To this aim, the expression $B = S_1 \,|[a,b]|\, (S_2 \,|[a]|\, S_3)$ is translated into the network N displayed below and then S_2 may be restricted using the refined interface $\mathsf{lts}\,(N') = \mathsf{refint}\,(N, 2, \{1\})$ that takes S_1 into account, where N' and $\mathsf{lts}\,(N')$ are displayed below. In practice, $\mathsf{lts}\,(N')$ can be minimized modulo safety equivalence, yielding an LTS with 2 states and 3 transitions. $S_2 \dashv|\, \mathsf{lts}\,(N')$ is isomorphic to S_1.

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



This example shows that 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 visible traces of the interface from a^* with b uncontrolled using the method of [83] to $a^* + b + (ba^+)^*$ using refined interface generation. 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.*

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 component LTSs participating in the interface modulo safety equivalence. This is correct, because safety equivalence is a congruence for networks. In addition, well-known partial order reductions preserving visible traces can be used to further reduce interfaces on-the-fly during their construction.

So far, refined interface generation required that each component of the concurrent system under verification (usually written in a high-level language) was replaced by its LTS. This apparently contradicts the claim that refined interfaces can be used to restrict processes on the fly. However, one can see in Definition 15 that the states and transitions of the component to be restricted ($\vec{S}[k]$) are not needed for interface generation. In practice, only the visible labels of $\vec{S}[k]$ are needed to compute the synchronization rules of the network from higher-level operators. To do so, $\vec{S}[k]$ can be replaced by an abstraction consisting of an arbitrary (and much smaller) LTS containing the same set of labels. In fact, the method remains correct if the abstraction contains a superset of $\vec{S}[k]$'s labels, although the reduction obtained on $\vec{S}[k]$ by semi-composition generally increases while the set of labels of the abstraction gets closer to the exact set of labels of $\vec{S}[k]$. In principle, the abstraction can be generated automatically by examining the gates (or channels) occurring in the high-level specification and the types of their data, and to enumerate data of this type appropriately.

4 Property-dependent compositional approaches

There are two ways of expressing and checking properties on a given concurrent system: one (*equivalence checking*) is based on behavioural relations and the other (*model checking*) is based on properties expressed, e.g., as formulas of some temporal logic.

The approach for compositionality presented in the previous section was based on equivalences, and independent from any particular set of properties, in the sense that it preserves all properties compatible with this equivalence. In this section, we consider alternative property-based approaches, which exploit compositionality while preserving a given set of temporal logic formulas to be verified. We develop this idea along two complementary ways:

- The first way aims at checking a set of temporal formulas on a (monolithic) LTS after reducing it as much as possible w.r.t. each formula. This is done by identifying the maximal set of actions that do not influence the truth value of a formula, hiding these actions in the LTS, and minimizing the LTS modulo an equivalence relation preserving the formula. The minimization can be carried out using the compositional equivalence-based approach of Section 3, which therefore becomes specialized for a given formula.
- The second way aims at checking one temporal formula on the parallel composition of several LTSs by taking them into account individually following the *partial model checking* method [8]. This is done by combining the formula with one of the LTSs and then checking the resulting formula recursively on the parallel composition of the remaining LTSs, until all of them have been taken into account. At intermediate stages, simplifications must be applied to the formula to keep it as small as possible.

These two ways of verification can be combined naturally by applying maximal hiding before starting partial model checking, and thus cumulating their benefits. One of our achievements is the reformulation of partial model checking in terms of graph manipulations, which paves the way towards the connection between compositional model checking and compositional state space construction.

We first revisit briefly the definitions of the temporal logics considered and the associated on-the-fly model checking methods. Then, we present the property-preserving reductions based on maximal hiding, and the partial model checking approach on networks of LTSs.

4.1 Temporal logics

Temporal logics [121, 122] were introduced more than three decades ago as a means for reasoning about the behaviour of concurrent systems. These formalisms consist of a small set of temporal operators expressing the logical precedence of states or events over time. Since the early proposals, a plethora of temporal logics emerged in the literature (see [48, 110, 25] for surveys), having different interpretation models, expressiveness, conciseness, and complexity of model checking. Regarding the interpretation models, two orthogonal criteria are most often used for classifying temporal logics: state-based (resp. action-based) logics specify properties of states (resp. actions, or events) of the system under study, and linear-time (resp. branching-time) logics are interpreted over execution sequences (resp. trees) of the system. Traditionally, state-based logics are interpreted on Kripke structures (KSs), in which states are labelled with atomic propositions, and action-based logics are interpreted on LTSs. The table below shows the most representative temporal logics in each of the four classes induced by the two criteria.

	Linear	Branching
State	LTL [122]	CTL [44]
Action	ALTL [64]	ACTL [111]

Besides these "pure" temporal logics, many other temporal logics combining linear-time and branching-time operators have been proposed, e.g., CTL* [45] and ACTL* [112] in the state- and action-based setting, respectively. Variants of these logics combining state- and action-based properties were also defined, having as interpretation models Kripke transition systems (KTSs) [110], i.e., state spaces containing relevant information on both states and transitions. A class of very expressive temporal logics are those comprising fixed point operators, such as the modal μ-calculus ($L\mu$) [82], which subsumes virtually all other temporal logic operators, and therefore can be seen as an assembly language for reasoning on LTSs.

Finally, to increase their expressiveness and facilitate their usage for value-passing systems, temporal logics were extended with various constructs: regular expressions over sequences (e.g., ELTL [138], RCTL [16], PDL-Δ [127]), automata (e.g., BRTL [75], ECTL* [131]), or data manipulation (e.g., extended

μ-calculi [42, 124], FORSPEC [9]). Also, extensions with both regular expressions and data handling are proposed in the state-based, linear-time setting (e.g., EAGLE [14]) and the action-based, branching-time setting (e.g., MCL [105]).

Given that we work in the compositionality framework underpinned by process algebras, we focus on action-based, branching-time temporal logics, which are suitable for LTSs and adequate w.r.t. bisimulation relations. More precisely, we use the dataless fragment of MCL (*Model Checking Language*) [105], defined in Figure 3. Action formulas (denoted by α) are built over the set of actions by using Boolean connectors in a way similar to ACTL [112], which is a slight extension w.r.t. the original definition of $L\mu$ [82]. Derived action operators can be defined as usual: $\mathsf{true} = \neg\mathsf{false}$, $\alpha_1 \wedge \alpha_2 = \neg(\neg\alpha_1 \vee \neg\alpha_2)$, etc. Regular formulas (denoted by β) are built from action formulas by using the testing (?), concatenation (.), choice (|), and transitive reflexive closure ($*$) operators. Derived regular operators can be defined as usual: $\varepsilon = \mathsf{false}^*$ is the empty sequence operator, $\beta^+ = \beta.\beta^*$ is the transitive closure operator, etc. State formulas (denoted by φ) are built from Boolean connectors, the possibility modality ($\langle\ \rangle$) and the infinite looping operator ($\langle\ \rangle\,@$) containing regular formulas as in PDL [53] and PDL-Δ [127], and the minimal fixed point operator (μ) defined over propositional variables X belonging to a set $\mathcal{X}$. Derived state operators can be defined as usual: $\mathsf{true} = \neg\mathsf{false}$, $\varphi_1 \wedge \varphi_2 = \neg(\neg\varphi_1 \vee \neg\varphi_2)$, $[\beta]\,\varphi = \neg\,\langle\beta\rangle\,\neg\varphi$ is the necessity modality, and $[\beta]\,\dashv = \neg\,\langle\beta\rangle\,@$ is the saturation operator. $\nu X.\varphi = \neg\mu X.\neg\varphi[\neg X/X]$ is the maximal fixed point operator ($\varphi[\neg X/X]$ stands for φ in which all free occurrences of X, i.e., not bound by a fixed point operator, have been negated). State formulas are assumed to be syntactically monotonic [82], i.e., each free occurrence of a propositional variable X inside a formula $\mu X.\varphi$ or $\nu X.\varphi$ must be preceded by an even number of negations.

Action formulas:

$$
\begin{array}{lll}
\alpha ::= b & & [\![b]\!]_A = \{b\} \\
\quad\mid\ \mathsf{false} & & [\![\mathsf{false}]\!]_A = \emptyset \\
\quad\mid\ \neg\alpha_1 & & [\![\neg\alpha_1]\!]_A = A \setminus [\![\alpha_1]\!]_A \\
\quad\mid\ \alpha_1 \vee \alpha_2 & & [\![\alpha_1 \vee \alpha_2]\!]_A = [\![\alpha_1]\!]_A \cup [\![\alpha_2]\!]_A
\end{array}
$$

Regular formulas:

$$
\begin{array}{lll}
\beta ::= \alpha & & \|\alpha\|_A = \{(s,s') \in \Sigma \times \Sigma \mid (\exists b \in A)\ s \xrightarrow{b} s' \wedge b \in [\![\alpha]\!]_A\} \\
\quad\mid\ \varphi? & & \|\varphi?\|_A = \{(s,s) \in \Sigma \times \Sigma \mid s \in [\![\varphi]\!]_M\} \\
\quad\mid\ \beta_1.\beta_2 & & \|\beta_1.\beta_2\|_A = \|\beta_1\|_A \circ \|\beta_2\|_A \\
\quad\mid\ \beta_1|\beta_2 & & \|\beta_1|\beta_2\|_A = \|\beta_1\|_A \cup \|\beta_2\|_A \\
\quad\mid\ \beta_1^* & & \|\beta_1^*\|_A = \|\beta_1\|_A^*
\end{array}
$$

State formulas:

$$
\begin{array}{lll}
\varphi ::= \mathsf{false} & & [\![\mathsf{false}]\!]_M\,\rho = \emptyset \\
\quad\mid\ \neg\varphi_1 & & [\![\neg\varphi_1]\!]_M\,\rho = \Sigma \setminus [\![\varphi_1]\!]_M\,\rho \\
\quad\mid\ \varphi_1 \vee \varphi_2 & & [\![\varphi_1 \vee \varphi_2]\!]_M\,\rho = [\![\varphi_1]\!]_M\,\rho \cup [\![\varphi_2]\!]_M\,\rho \\
\quad\mid\ \langle\beta\rangle\,\varphi_1 & & [\![\langle\beta\rangle\,\varphi_1]\!]_M\,\rho = \{s \in \Sigma \mid (\exists s' \in \Sigma)\ (s,s') \in \|\beta\|_A \wedge s' \in [\![\varphi_1]\!]_M\,\rho\} \\
\quad\mid\ \langle\beta\rangle\,@ & & [\![\langle\beta\rangle\,@]\!]_M = \{s \in \Sigma \mid (\forall k \geq 0)\ (\exists s' \in \Sigma)\ (s,s') \in \|\beta\|_A^k\} \\
\quad\mid\ X & & [\![X]\!]_M\,\rho = \rho(X) \\
\quad\mid\ \mu X.\varphi_1 & & [\![\mu X.\varphi_1]\!]_M\,\rho = \bigcap\{U \subseteq \Sigma \mid [\![\varphi_1]\!]_M\,(\rho \oslash [U/X]) \subseteq U\}
\end{array}
$$

Figure 3: Syntax and semantics of dataless MCL

The interpretation $[\![\alpha]\!]_A$ of an action formula on the set of actions of an LTS $M = (\Sigma, A, T, s_0)$ denotes the subset of actions satisfying α. An action b satisfies a formula α (also denoted by $b \models_A \alpha$) if and only if $b \in [\![\alpha]\!]_A$. A transition $s_1 \xrightarrow{b} s_2$ such that $b \models_A \alpha$ is called an α-transition. The interpretation $\|\beta\|_A$ of a regular formula on an LTS denotes a relation between the states that are source and target of transition sequences whose concatenated actions form a word belonging to the regular language defined by β. The testing operator specifies state formulas that must hold in the intermediate states of a transition sequence. A propositional context $\rho : \mathcal{X} \to 2^\Sigma$ is a partial function mapping propositional variables to subsets of states. The notation $\rho \oslash [U/X]$ stands for a propositional context identical to ρ except for variable X, which is mapped to the state subset U. The interpretation $[\![\varphi]\!]_M\,\rho$ of a state formula on an LTS M and

a propositional context ρ (which assigns a set of states to each propositional variable occurring free in φ) denotes the subset of states satisfying φ in that context. The Boolean connectors are interpreted as usual in terms of set operations. The possibility modality $\langle\beta\rangle\,\varphi_1$ (resp. the necessity modality $[\beta]\,\varphi_1$) denotes the states for which some (resp. all) of their outgoing transition sequences satisfying β lead to states satisfying φ_1. The infinite looping operator $\langle\beta\rangle\,@$ (resp. the saturation operator $[\beta]\,\dashv$) denotes the states having some (resp. no) outgoing transition sequence consisting of an infinite concatenation of sub-sequences satisfying β. The minimal fixed point operator $\mu X.\varphi_1$ (resp. the maximal fixed point operator $\nu X.\varphi_1$) denotes the least (resp. greatest) solution of the equation $X = \varphi_1$ interpreted over the complete lattice $(2^{\Sigma}, \emptyset, \Sigma, \cap, \cup, \subseteq)$. A state s satisfies a closed formula φ, denoted by $s \models_M \varphi$, if and only if $s \in [\![\varphi]\!]_M$ (the propositional context ρ can be omitted since φ does not contain free variables). An LTS $M = (\Sigma, A, T, s_0)$ satisfies a closed formula φ, denoted by $M \models \varphi$, if and only if $s_0 \models_M \varphi$.

Regular operators can be eliminated by translating possibility modalities and infinite looping operators into plain $L\mu$ by applying the identities below [46] (dual identities hold for necessity modalities and saturation operators):

$$\langle\varphi'?\rangle\,\varphi = \varphi' \wedge \varphi \qquad \langle\beta_1.\beta_2\rangle\,\varphi = \langle\beta_1\rangle\,\langle\beta_2\rangle\,\varphi$$
$$\langle\beta_1|\beta_2\rangle\,\varphi = \langle\beta_1\rangle\,\varphi \vee \langle\beta_2\rangle\,\varphi \qquad \langle\beta^*\rangle\,\varphi = \mu X.(\varphi \vee \langle\beta\rangle\,X)$$
$$\langle\beta\rangle\,@ = \nu X.\,\langle\beta\rangle\,X$$

Although regular formulas do not increase expressiveness, they make possible a much more concise and intuitive description of properties than their fixed point counterparts. For example, the *fair reachability* [123] (i.e., by skipping cycles) of a reception *rcv* after each emission *snd* possibly followed by a finite number of transmission errors *err*, can be specified in MCL as follows:

$$[\mathsf{true}^*.snd.((\neg rcv)^*.err)^*]\,\langle\mathsf{true}^*.rcv\rangle\,\mathsf{true}$$

whereas in plain $L\mu$ a more verbose formula is needed:

$$\nu X.([snd]\,\nu Y.(\mu Z.(\langle rcv\rangle\,\mathsf{true} \vee \langle\mathsf{true}\rangle\,Z) \wedge \nu W.([err]\,Y \wedge [\neg rcv]\,W)) \wedge [\mathsf{true}]\,X)$$

To obtain efficient model checking algorithms, i.e., linear in the size of the LTS (number of states and transitions) and the size of the formula (number of operators) we consider only alternation-free [46] fixed point formulas, i.e., without mutually recursive minimal and maximal fixed point operators. Note that infinite looping operators whose regular formulas contain star operators yield after translation fixed point formulas of alternation depth 2, such as $\langle a^*.b\rangle\,@ = \nu X.\mu Y.(\langle b\rangle\,X \vee \langle a\rangle\,Y)$, which contains two mutually recursive variables X and Y of different fixed point sign. Although the $L\mu_2$ fragment of alternation depth 2 has in general a model checking complexity quadratic in the size of the LTS [46], the particular formulas obtained from infinite looping operators can be checked with a linear complexity, as shown in the next subsection.

4.2 Model checking

Given an LTS $M = (\Sigma, A, T, s_0)$ and a closed state formula φ, the model checking problem consists in determining if $M \models \varphi$, which amounts to verify if the initial state s_0 satisfies φ, i.e., $s_0 \in [\![\varphi]\!]$. In the explicit-state setting, there are basically two approaches to solve this problem: the *global* (or enumerative) approach evaluates the interpretation of φ on M, i.e., the subset of states satisfying φ, and checks if s_0 belongs to it, whereas the *local* (or on-the-fly) approach evaluates the truth value of φ on s_0 based on the truth values of its subformulas on the descendant states of s_0. The global approach requires the entire construction of the LTS before carrying out verification, whereas the local approach explores it in a demand-driven way during verification. In practice, local model checking is suitable for the early phases of the design process, when errors are likely to occur more frequently and are detected quickly by exploring relatively small parts of the LTS; at later phases, when the model becomes stable and no errors are detected anymore, global model checking is more appropriate for verifying invariant properties since the underlying algorithms are slightly more efficient.

The alternation-free μ-calculus fragment (consisting of formulas without mutually recursive variables of different fixed point signs) is equipped with global [38] and local [3] model checking algorithms having a

linear complexity $O(|\varphi| \cdot (|\Sigma| + |T|))$. For checking MCL formulas, we adopt the on-the-fly approach proposed in [3], based on Boolean Equation Systems (BESs), and we generalize it to deal efficiently with formulas of particular shapes frequently encountered in practice [102]. A BES is a tuple $B = (X, M_1, ..., M_n)$, where $X \in \mathcal{X}$ is a Boolean variable and M_i are equation blocks ($i \in [1, n]$). Each block $M_i = \{X_j \stackrel{\sigma_i}{=} op_j \boldsymbol{X}_j\}_{j \in [1, m_i]}$ is a set of minimal (resp. maximal) fixed point equations of sign $\sigma_i = \mu$ (resp. $\sigma_i = \nu$). The right-hand side of each equation j is a pure disjunctive or conjunctive formula obtained by applying a Boolean operator $op_j \in \{\vee, \wedge\}$ to a set of variables $\boldsymbol{X}_j \subseteq \mathcal{X}$. The Boolean constants F and T abbreviate the empty disjunction $\vee\emptyset$ and the empty conjunction $\wedge\emptyset$.

The *main* variable X must be defined in block M_1. A variable X_j depends upon a variable X_l if $X_l \in \boldsymbol{X}_j$. A block M_i depends upon a block M_k if some variable of M_i depends upon a variable defined in M_k. A block is *closed* if it does not depend upon any other blocks. A BES is *alternation-free* if there are no cyclic dependencies between its blocks; in this case, the blocks are assumed to be sorted topologically such that a block M_i only depends upon blocks M_k with $k > i$.

The semantics of a formula $op_i\{X_1, ..., X_k\}$ w.r.t. $\mathbf{Bool} = \{\mathsf{F}, \mathsf{T}\}$ and a context $\delta : \mathcal{X} \rightarrow \mathbf{Bool}$, which must initialize all variables X_1, ..., X_k, is the boolean value defined as follows:

$$[\![op_i\{X_1, ..., X_k\}]\!]\, \delta = \delta(X_1)\ op_i\ ...\ op_i\ \delta(X_k)$$

The semantics of an equation block $M_i = \{X_j \stackrel{\sigma_i}{=} op_j \boldsymbol{X}_j\}_{j \in [1, m_i]}$ w.r.t. a context δ is the σ_i-fixed point of a vectorial functional $\Phi_{i\delta} : \mathbf{Bool}^{m_i} \rightarrow \mathbf{Bool}^{m_i}$:

$$\left[\!\left[\{X_j \stackrel{\sigma_i}{=} op_j \boldsymbol{X}_j\}_{j \in [1, m_i]}\right]\!\right] \delta = \sigma_i \Phi_{i\delta}$$

where

$$\Phi_{i\delta}(b_1, ..., b_{m_i}) = ([\![op_j \boldsymbol{X}_j]\!]\ (\delta \oslash [b_1/X_1, ..., b_{m_i}/X_{m_i}]))_{j \in [1, m_i]}$$

The notation $\delta \oslash [b_1/X_1, ..., b_n/X_n]$ stands for a context identical to δ except for variables $X_1, ..., X_n$, which are assigned values $b_1, ..., b_n$, respectively.

The semantics of an alternation-free BES is the value of its main variable X given by the solution of M_1, i.e., $\delta_1(X)$, where the contexts δ_i are calculated as follows:

$$\begin{array}{rcl} \delta_n & = & [\![M_n]\!]\ [] \\ \delta_i & = & ([\![M_i]\!]\ \delta_{i+1}) \oslash \delta_{i+1} \text{ for } i \in [1, n-1] \end{array}$$

Note that the context for interpreting M_n is empty (since M_n is closed) and a block M_i is interpreted in the context of all blocks M_k with $k > i$.

A block M_i is *acyclic* if there are no cyclic dependencies between the variables defined in M_i. A variable X_j is called *disjunctive* (resp. *conjunctive*) if $op_j = \vee$ (resp. $op_j = \wedge$). A block M_i is disjunctive (resp. conjunctive) if each of its variables either is disjunctive (resp. conjunctive), or it depends upon at most one variable defined in M_i, all its other dependencies being constants or variables defined in other blocks.

The on-the-fly resolution of an alternation-free BES $B = (X, M_1, ..., M_n)$ consists in computing the value of X by exploring the right-hand sides of the equations in a demand-driven way, without explicitly constructing the blocks. To each block M_i is associated a resolution routine R_i responsible for computing the values of the variables defined in M_i. When a variable X_j defined in M_i is computed by a call $R_i(X_j)$, the values of other variables X_l defined in other blocks M_k may be needed; these values are computed by calls $R_k(X_l)$ of the routine associated to M_k. This process always terminates because there are no cyclic dependencies between blocks (the call stack of resolution routines has a size bounded by the depth of the dependency graph between blocks). Since a variable X_j defined in M_i may be required several times during the resolution process, the computation results must be kept persistent between subsequent calls of R_i in order to obtain a linear-time overall resolution. Compared to other on-the-fly resolution algorithms like LMC [43], which consists of a single monolithic routine handling the whole BES, the above scheme yields simpler resolution algorithms (dedicated to blocks of equations having the same fixed point sign), and is easier to optimize (by designing more efficient algorithms for blocks with particular structure) [102].

The on-the-fly model checking procedure for MCL formulas is described in detail in [104, 105]. We briefly illustrate it here for the dataless MCL fragment using the example on Figure 4. The LTS shown on

Figure 4(a) models the behaviour of a two-place buffer made by connecting two one-place cells in sequence. States are represented by numbers, the initial state being 0. The buffer accepts as input an infinite stream containing two kinds of messages, denoted by 0 or 1 (actions put_0 and put_1) and delivers it as output (actions get_0 and get_1) in the same order. The move of a message from the first to the second cell of the buffer is modeled by a τ-transition.

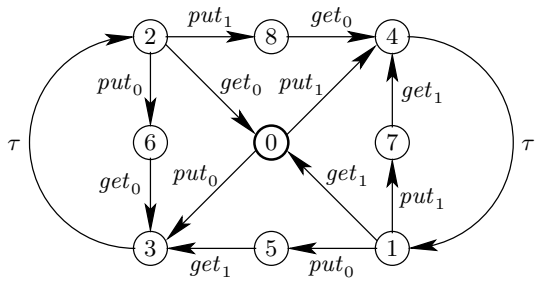
On this LTS, we want to check that after every message 0 accepted by the buffer, it is possible that the buffer cyclically accepts a message 1 and delivers it. This property is expressed by the MCL formula on top of Figure 4(b), which contains a PDL box modality followed by an infinite looping operator of alternation depth 2. The subsequent lines in Figure 4(b) illustrate the various translation steps that bring this formula into an equational form suitable for evaluation using BESs: (i) the formula is first translated into PDL with recursion [104], which consists of two blocks of fixed point equations defining propositional variables using PDL formulas; (ii) the PDL modalities in the equations are split to eliminate concatenation operators in the regular formulas, and the equation block corresponding to the infinite looping operator is transformed specifically by adding an extra equation and flipping its fixed point sign; (iii) the star operators in the regular formulas are replaced by fixed point equations, yielding a specification in HML with recursion [92], which contains only single-step modal formulas; (iv) the interpretation of this specification on the LTS is encoded as a BES made of two corresponding equation blocks M_1 and M_2 defining Boolean variables of the form X_s (resp. Y_s, Z_s, W_s), which are true if and only if the state s satisfies the propositional variable X (resp. Y, Z, W). In this way, the local model checking of the formula on the initial state of the LTS amounts to solve the variable X_0 of block M_1.

This evaluation is carried out by using local resolution algorithms that explore the Boolean graph [3] associated to the BES in a forward way, as shown by the excerpt on Figure 4(c). Since the block M_1 is conjunctive and M_2 is disjunctive, they are solved in a memory-efficient manner by the algorithms proposed in [102] and [105], respectively. These algorithms, based on a depth-first search of the Boolean graph, store only Boolean variables and not dependencies between them, and hence, only the states and not the transitions of the LTS. When exploring the Boolean graph of M_2, each cycle containing a variable Y_s is interpreted as a constant `true`, since it corresponds to a sequence satisfying the infinite looping operator in the initial formula. This is done in linear-time by computing the strongly connected components in the Boolean graph of M_2 and keeping track of those containing variables Y_s, which are similar to the marked states in Büchi automata. The thick arrows in Figure 4(c) define the diagnostic of the BES resolution, which is a minimal Boolean subgraph (w.r.t. graph inclusion) containing the variable of interest and whose variables have the same solution as those of the BES [101]. On this example, the LTS satisfies the MCL formula, and the positive diagnostic (witness), illustrated partially on Figure 4(c), contains an execution sequence leading to every put_0-transition in the LTS, followed by a cycle passing through put_1- and get_1-transitions.

4.3 Property-dependent reductions

A property-based specification of a system consists usually of several properties expressed as temporal logic formulas, each one describing a certain aspect of the system. This kind of specification has naturally two benefits [96]: it is *abstract*, in the sense that it may characterize a family of systems sharing the same observable behaviour (e.g., the abstract properties of mutual exclusion and inevitable progress towards the critical section must be satisfied by all mutual exclusion protocols, regardless their implementation details); and it is *modular*, meaning that one can add, eliminate, or replace a property in the specification without changing the other properties and their interpretation on the system under study.

In practice, since the model checking complexity grows with the size of the formula (number of operators), to increase efficiency it is desirable to verify formulas as small as possible. In our framework, a given dataless MCL formula φ should be first decomposed into a set of smaller "temporal" formulas (i.e., dominated by a modal or fixed point operator), each one being checked separately on the system. A simple way to achieve this is by transforming the formula in conjunctive normal form (CNF), by considering as literals the maximal temporal subformulas of φ; then, for each conjunct, each of its disjuncts is verified in turn (by stopping the process when some disjunct is satisfied by the system), the whole process terminating when all conjuncts have been checked to hold on the system, or some conjunct not satisfied by the system is encountered. This decomposition and evaluation process can be easily automated. However, in practice a property-based



(a)

MCL formula:
$[\mathsf{true}^*.put_0]\,\langle \mathsf{true}^*.put_1.\mathsf{true}^*.get_1\rangle\,@$

Translation to PDL with recursion:

$$\left\{\begin{array}{lll} X & =_\nu & [\mathsf{true}^*.put_0]\,Y \\ Y & =_\nu & \langle \mathsf{true}^*.put_1.\mathsf{true}^*.get_1\rangle\,Y \end{array}\right\}$$

Conversion of infinite looping block:

$$\left\{\begin{array}{lll} X & =_\nu & [\mathsf{true}^*]\,[put_0]\,Y \\ \left\{\begin{array}{lll} Y & =_\mu & Z \\ Z & =_\mu & \langle \mathsf{true}^*\rangle\,\langle put_1\rangle\,\langle \mathsf{true}^*\rangle\,\langle get_1\rangle\,Y \end{array}\right\} \end{array}\right.$$

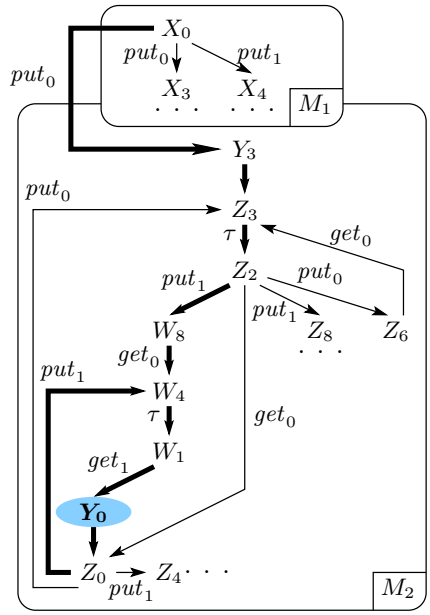
Translation to HML with recursion:

$$\left\{\begin{array}{lll} X & =_\nu & [put_0]\,Y \wedge [\mathsf{true}]\,X \\ \left\{\begin{array}{lll} Y & =_\mu & Z \\ Z & =_\mu & \langle put_1\rangle\,W \vee \langle \mathsf{true}\rangle\,Z \\ W & =_\mu & \langle get_1\rangle\,Y \vee \langle \mathsf{true}\rangle\,W \end{array}\right\} \end{array}\right.$$

Model checking encoding as BES:

$$\left\{\begin{array}{lll} X_s & =_\nu & \wedge_{s \xrightarrow{put_0} s'} Y_{s'} \wedge \wedge_{s \longrightarrow s''} X_{s''} \\ \left\{\begin{array}{lll} Y_s & =_\mu & Z_s \\ Z_s & =_\mu & \vee_{s \xrightarrow{put_1} s'} W_{s'} \vee \vee_{s \longrightarrow s''} Z_{s''} \\ W_s & =_\mu & \vee_{s \xrightarrow{get_1} s'} Y_{s'} \vee \vee_{s \longrightarrow s''} W_{s''} \end{array}\right\}_{s \in S} \end{array}\right\}_{s \in S}$$

(b)



(c)

Figure 4: Example of on-the-fly model checking using BESs. (a) LTS of a two-place buffer accepting and delivering messages of two kinds, denoted by 0 and 1. (b) MCL formula and the translation of its evaluation on the LTS in terms of a BES resolution. (c) Local resolution of the resulting BES by forward exploration of its Boolean graph (excerpt).

specification is often a list of temporal formulas that are supposed to hold on the system (which corresponds to a CNF formula with monome conjuncts), and this is the case we focus on here, by considering the verification of a single temporal formula.

Given an LTS and a temporal formula, the performance of verification can be improved by reducing the LTS before checking the formula, modulo a relation that must preserve the truth value of the formula. The standard approach is to reduce the LTS modulo an equivalence relation adequate w.r.t. the temporal logic to which belongs the formula. Of course, to improve the overall performance of the verification process (LTS generation, reduction, and model checking), the computational complexity of the LTS reduction should be smaller than that of checking the formula on the initial LTS; this can be achieved by using the compositional LTS construction and minimization described in Section 3. This approach relies on existing adequacy results between behavioural equivalences and temporal logics (see, e.g., [48] for a survey of adequacy in the action-based setting). In our framework, a useful result is the adequacy of ACTL-X (Action-based CTL without the next state operator) [112] w.r.t. divergence-sensitive branching bisimulation. This result also holds for μACTL-X [47], which extends ACTL-X with fixed point operators.

Nevertheless, the reduction of the LTS modulo an equivalence relation preserves *all* temporal formulas adequate w.r.t. that relation, whereas in practice one is interested in preserving only the properties forming the specification of the system. A better solution would be to specialize the reduction for each temporal formula considered: given that each formula references specific aspects of the system, the other aspects (referenced by the other properties in the specification) could be abstracted away when checking that formula, thus yielding more potential for reduction. An early attempt for property-dependent reduction in the action-based setting was made with the selective μ-calculus [13], which synthesizes an equivalence relation preserving a given formula. However, this attempt has two practical limitations: it imposes a particular formulation of properties (using special modalities indexed by action sets), and the synthesized equivalence relations are variants of the $\tau^*.a$ equivalence (which is not a congruence w.r.t. parallel composition and hence not appropriate for compositional reasoning, and it preserves only safety and weak liveness properties).

Another, more recent, attempt for property-dependent reduction [106] takes a modal μ-calculus formula and synthesizes the maximal set of actions which can be hidden (i.e., renamed into τ) in the LTS without disturbing the interpretation of the formula. This approach keeps in the LTS only the relevant information determining the truth value of the formula, therefore yielding a high potential for reduction. After applying this maximal hiding, the LTS can be reduced modulo an adequate equivalence relation before checking the formula.

Definition 16 (Hiding set) *Let α be an action formula interpreted over a set of actions A. The* hiding set *of α w.r.t. A is defined as follows:*

$$h_A(\alpha) = \begin{cases} [\![\alpha]\!]_A & \text{if } \tau \models_A \alpha \\ A \setminus [\![\alpha]\!]_A & \text{if } \tau \not\models_A \alpha \end{cases}$$

The hiding set of a state formula φ w.r.t. A, denoted by $h_A(\varphi)$, is defined as the intersection of $h_A(\alpha)$ for all action subformulas α of φ.

Definition 17 (Hiding) *Let A be a set of actions and $H \subseteq A$. The* hiding *of an action $b \in A$ w.r.t. H is defined as follows:*

$$hide_H(b) = \begin{cases} b & \text{if } b \notin H \\ \tau & \text{if } b \in H \end{cases}$$

The hiding of an LTS M w.r.t. H is defined as follows:

$$hide_H(M) = \mathsf{lts}\,(\textbf{hide } H \textbf{ in } M)$$

The following lemma [106] states that, given an action formula α, the fact of hiding an action w.r.t. the hiding set of α does not disturb the interpretation of α on that action.

Lemma 1 *Let α be an action formula interpreted over a set of actions A. A subset $H \subseteq A$ does not disturb the interpretation of α on the actions of A after hiding them w.r.t. H if the following property holds:*

$$\forall b \in A.(b \models_A \alpha \Leftrightarrow hide_H(b) \models_A \alpha)$$

Then: a subset $H \subseteq A$ satisfies this property if and only if $H \subseteq h_A(\alpha)$.

Lemma 1 ensures that, for an action formula α, its hiding set $h_A(\alpha)$ is the maximal set of actions that can be hidden in the LTS without disturbing the interpretation of α. To make possible LTS reductions prior to (or simultaneously with) the verification of a state formula φ, it is desirable to hide as many actions as possible in the LTS, i.e., all actions in $h_A(\varphi)$. The following proposition [106], which makes use of Lemma 1, guarantees that this hiding preserves the interpretation of φ.

Proposition 8 (Maximal Hiding) *Let $M = (\Sigma, A, \longrightarrow, s_0)$ be an LTS, φ be a state formula of $L\mu$, and $H \subseteq h_A(\varphi)$. Then:*

$$[\![\varphi]\!]_M \, \rho = [\![\varphi]\!]_{hide_H(M)} \, \rho$$

for any propositional context ρ.

For instance, consider the MCL formula below, expressing the inevitable execution of a *recv* action after every *send* action:

$$\varphi = [\mathsf{true}^*.send] \, \mu X.(\langle \mathsf{true} \rangle \, \mathsf{true} \wedge [\neg recv] \, X)$$

When checking φ on an LTS, one can hide all actions in $h_A(\varphi) = h_A(send) \cap h_A(\neg recv) = (A \setminus [\![send]\!]_A) \cap [\![\neg recv]\!]_A = (A \setminus \{send\}) \cap (A \setminus \{recv\}) = A \setminus \{send, recv\}$, i.e., all actions other than *send* and *recv*, without changing the interpretation of the formula.

For a given formula φ, after hiding all actions in $h_A(\varphi)$, one can improve the performance of verification by reducing the LTS modulo an equivalence relation that preserves the interpretation of φ. Two such relations are considered in [106]: strong bisimulation, which preserves all $L\mu$ formulas, and divergence-sensitive branching bisimulation (divbranching for short), which is adequate w.r.t. μACTL-X [47] and also with the L_μ^{dsbr} fragment of μ-calculus proposed in [106]. Strong bisimulation minimization, although it has a lower potential for reduction compared to weak equivalences (since strong bisimulation does not give a special treatment to τ-transitions), can nevertheless improve the performance of verification when the LTS obtained after hiding contains a high percentage of τ-transitions (execution time divided by 4 and memory consumption divided by 2 [106]). Regarding divbranching bisimulation, the performances can be improved significantly either by minimization (time divided by 20 and memory divided by 5), or by on-the-fly τ-confluence reduction (time divided by 10) [106].

More generally, maximal hiding for a given formula can be applied in conjunction with the compositional LTS construction and minimization approach described in Section 3. This specialization is likely to yield better results (especially w.r.t. the peak memory consumption of the whole verification process) than the compositional reduction modulo an equivalence relation, as shown in Section 7.

4.4 Partial model checking

Partial model checking is a dual approach to compositional model generation, where one needs to check a formula on a network of automata. Partial model checking was first proposed by Andersen [8, 4, 5] for concurrent components running asynchronously and composed using CCS parallel composition and restriction operators. For a modal μ-calculus formula φ and a composition expression $S_1 \mid\mid \ldots \mid\mid S_n$, Andersen uses an operation $\varphi /\!\!/ S_1$ called *quotienting* of the formula φ w.r.t. the component S_1, so that $S_1 \mid\mid \ldots \mid\mid S_n$ satisfies φ if and only if the smaller composition $S_2 \mid\mid \ldots \mid\mid S_n$ satisfies $\varphi /\!\!/ S_1$. In addition, simplifications can (and must) be applied to $\varphi /\!\!/ S_1$ to reduce its size. Partial model checking is the incremental application of quotienting and simplifications, so that state explosion is avoided if the size of intermediate formulas can be kept sufficiently small.

Partial model checking has been adapted and used successfully in various contexts, such as state-based models [7, 6], synchronous state/event systems [22], and timed systems [17, 27, 90, 91, 93]. It has also been specialised for security properties [99]. More recently, it has been generalised to the full CCS process algebra, with an application to the verification of parameterized systems [15]. These various developments of partial model checking, although successful, were relatively scarce, which may be explained by the complexity of the method: obtaining a fully operational partial model checker requires a significant implementation effort and extensive experiments for fine-tuning and optimization.

In this section, we focus on partial model checking of the modal μ-calculus applied to (untimed) concurrent asynchronous components [87, 88]. By considering only binary associative parallel composition operators

(such as CCS and CSP parallel compositions), previous works [4, 5, 15] are not directly applicable to more general operators, such as m-among-n synchronization and parallel composition by synchronization interfaces (where all components containing a given action in their synchronization interface must synchronize on that action) [61], present in the E-LOTOS standard and variants [28, 80]. Our first contribution in this section is thus a generalisation of partial model checking to networks of LTSs, which subsumes all the above-mentioned parallel composition operators. Regarding the communication of data values, our approach is applicable to classical (i.e., with static communication) value-passing process algebras equipped with early operational semantics. This framework encompasses a significant fragment of the π-calculus (containing channel mobility and bounded process creation), which can be translated into classical value-passing process algebras [103].

In realistic cases, partial model checking handles huge formulas and components, thus requiring efficient implementations. Our second contribution is a reformulation of quotienting as a synchronous product (which can itself be represented in the network model) between a graph representing the formula (called a *formula graph*) and the behaviour graph of a component, thus enabling efficient implementation using existing tools dedicated to graph manipulations. Our third contribution is the reformulation of formula simplifications as a combination of graph reductions (including minimisations modulo equivalence relations and bisimulations) and partial evaluation of the formula graph using a BES (*Boolean Equation System*) [3].

In this section, we consider plain $L\mu$ formulas in disjunctive form, i.e., expressed using negation $\neg$, variables, and the disjunctive operators false, $\langle _ \rangle$, $\vee$, and μ. We also suppose that the formulas are *well-formed block-labelled*, as defined below.

Definition 18 (Block-labelled formula) *A* block-labelled *μ-calculus formula φ is a formula in which each propositional variable X is labelled by a natural number k, called its* block number*.*

Given a block-labelled μ-calculus formula φ, we write $\mathsf{fv}(\varphi)$ for the set of variables free in φ, and $\mathsf{bv}(\varphi)$ for the set of variables bound in φ. We call a closed formula *any formula φ such that $\mathsf{fv}(\varphi) = \emptyset$. We assume that all bound variables have distinct names, and for $X^k \in \mathsf{bv}(\varphi)$, we write $\varphi[X^k]$ for the (unique) sub-formula of φ of the form $\mu X^k.\varphi_0$. Given φ_1 and φ_2, we write $\varphi_1[\varphi_2/X^k]$ for substituting all free occurrences of X^k in φ_1 by φ_2 (while implicitly applying α-conversion[3] to maintain the unicity of bound variables).*

Intuitively, a block-labelling is well-formed *if the μ-calculus formula can be converted into an equivalent set of μ-calculus equations partitioned into blocks, so that all variables having the same block number are defined in the same block and if $k < k'$ then the equations within block number k occur before the equations within block number k'. The well-formedness conditions are the following:*

1. *All occurrences of a given variable are labelled by the same block number.*
2. *All variables sharing the same block number have the same fixed point sign[4].*
3. *For all $X^k \in \mathsf{bv}(\varphi)$ and $Y^{k'} \in \mathsf{fv}(\varphi[X^k])$ it holds that $k' \leq k$.*

By convention, we assume without loss of generality that the even block numbers are associated to variables of sign μ and odd block numbers are associated to variables of sign ν.

Initially, every unlabelled formula φ can be turned into the well-formed block-labelled formula $\mathsf{bl}(\varphi, \mathsf{true}, 0, [])$, where $\mathsf{bl}(\psi, b, k, \gamma)$ is defined as follows, b denoting a boolean that is true only if ψ is in the context of an even number of negations, k the current block number, and γ a mapping from variables to block numbers:

$$
\begin{array}{rcl}
\mathsf{bl}(\mathsf{false}, b, k, \gamma) & = & \mathsf{false} \\
\mathsf{bl}(X, b, k, \gamma) & = & X^{\gamma(X)} \\
\mathsf{bl}(\neg\varphi_0, b, k, \gamma) & = & \neg\mathsf{bl}(\varphi_0, \neg b, k, \gamma) \\
\mathsf{bl}(\varphi_1 \vee \varphi_2, b, k, \gamma) & = & \mathsf{bl}(\varphi_1, b, k, \gamma) \vee \mathsf{bl}(\varphi_2, b, k, \gamma) \\
\mathsf{bl}(\langle a \rangle \varphi_0, b, k, \gamma) & = & \langle a \rangle \mathsf{bl}(\varphi_0, b, k, \gamma) \\
\mathsf{bl}(\mu X.\varphi_0, b, k, \gamma) & = & \begin{cases} \mu X^k.\mathsf{bl}(\varphi_0, \mathsf{true}, k, \gamma \oslash [X \mapsto k]) & \textit{if } b = \mathsf{true} \\ \mu X^{k+1}.\mathsf{bl}(\varphi_0, \mathsf{true}, k+1, \gamma \oslash [X \mapsto k+1]) & \textit{otherwise} \end{cases}
\end{array}
$$

[3] α-conversion is a well-known equivalence relation between terms that differ only in variable names, originally defined in the framework of the λ-calculus and applied implicitly during substitution to avoid variable capture.

[4] Considering a formula φ in disjunctive form, the sign of variable X^k is μ if $\varphi[X^k]$ occurs in φ in the context of an even number of negations, and ν otherwise.

We write $\mathsf{blocks}(\varphi)$ *for the set of block numbers occurring in* φ*. A block-labelled formula* φ *is* alternation-free *if* $k' = k$ *for all* $X^k \in \mathsf{bv}(\varphi)$ *and* $Y^{k'} \in \mathsf{fv}(\varphi[X^k])$*.*

To check a closed formula φ on a network $N = (\vec{S}, V)$, one can choose a component LTS $\vec{S}[i]$, compute the quotient of the formula φ with respect to $\vec{S}[i]$, and check the resulting quotient formula on the smaller (at least in the number of component LTSs, but also hopefully in the compound LTS size) network $N_{|\overline{\{i\}}}$.

Definition 19 (Quotient formula) *$N = (\vec{S}, V)$ being a network of size n, we assume a function $\alpha(\vec{t}, a)$ that assigns a unique unused label to each $(\vec{t}, a) \in V$. Given φ a closed formula and $i \in [1, n]$, the quotient formula is written $\varphi /\!\!/_i^{\emptyset}\ s_i^0$ and defined as follows:*

$$
\begin{array}{rcl}
\mathsf{false} /\!\!/_i^B\ s & = & \mathsf{false} \\
X^k /\!\!/_i^B\ s & = & \varphi[X^k] /\!\!/_i^B\ s \\
(\neg\varphi_0) /\!\!/_i^B\ s & = & \neg(\varphi_0 /\!\!/_i^B\ s) \\
(\varphi_1 \vee \varphi_2) /\!\!/_i^B\ s & = & (\varphi_1 /\!\!/_i^B\ s) \vee (\varphi_2 /\!\!/_i^B\ s) \\
(\mu X^k.\varphi_0) /\!\!/_i^B\ s & = & \begin{cases} X_s^k & \textit{if } X_s^k \in B \\ \mu X_s^k.(\varphi_0 /\!\!/_i^{B \cup \{X_s^k\}}\ s) & \textit{otherwise} \end{cases} \\
(\langle a \rangle\, \varphi_0) /\!\!/_i^B\ s & = & \bigvee_{(\vec{t},a) \in V} \left(\begin{array}{lllll} (\quad i \notin \mathsf{acv}(\vec{t}) & \wedge & \langle a \rangle & (\varphi_0 /\!\!/_i^B\ s)\ ) & \vee \\ (\{i\} \subsetneq \mathsf{acv}(\vec{t}) & \wedge \bigvee_{s \xrightarrow{\vec{t}[i]}_i s'} & \langle \alpha(\vec{t}, a) \rangle & (\varphi_0 /\!\!/_i^B\ s')) & \vee \\ (\{i\} = \mathsf{acv}(\vec{t}) & \wedge \bigvee_{s \xrightarrow{\vec{t}[i]}_i s'} & & (\varphi_0 /\!\!/_i^B\ s')) & \end{array} \right)
\end{array}
$$

This definition follows and generalises Andersen's [4] (specialised for CCS) to networks. The main difference is the definition of $(\langle a \rangle\, \varphi_0) /\!\!/_i^B\ s$, CCS composition corresponding to vectors $((a, \bullet), a)$, $((\bullet, a), a)$, or $((a, \overline{a}), \tau)$, a and $\overline{a}$ being an action and its CCS *co-action*, making the use of special labels $\alpha(\vec{t}, a)$ not necessary. A minor difference is that we use μ-calculus terms instead of equations[5]. Any sub-formula produced by quotienting has the same block number as the original sub-formula, reflecting the order of equation blocks in Andersen's definition. The set B keeps track of new variables already introduced in the quotient formula. Quotienting is well-defined, because formulas are finite, every $\varphi[X^k]$ has the form $\mu X^k.\varphi_0$ (the formula is in disjunctive form), and the size of the set B is bounded by $|\,\mathsf{bv}\,(\varphi)| \times |\Sigma_i|$. Note that well-formedness of the block-labelling is preserved by quotienting, because for every variable $X_s^k \in \mathsf{bv}\,(\varphi /\!\!/_i^{\emptyset}\ s_0)$ we have $X^k \in \mathsf{bv}\,(\varphi)$ and for every variable $Y_{s'}^{k'} \in \mathsf{fv}\,((\varphi /\!\!/_i^{\emptyset}\ s_0)[X_s^k])$ we have $Y^{k'} \in \mathsf{fv}\,(\varphi[X^k])$, and therefore $k' \leq k$.

Example 8 *The μ-calculus formula $\mu X^0.\, \langle a \rangle\, \mathsf{true} \vee \langle b \rangle\, X^0$ (existence of a path of zero or more b leading to an a) can be rewritten to disjunctive form as $\mu X^0.\, \langle a \rangle\, \neg\mathsf{false} \vee \langle b \rangle\, X^0$. Quotienting of this formula with respect to S_3 in the network N introduced in Example 1 yields the formula $\mu X_0^0.\, \langle a \rangle\, \neg\mathsf{false} \vee \langle \alpha_a \rangle\, \neg\mathsf{false} \vee \langle \alpha_b \rangle\, \mu X_2^0.\, \langle a \rangle\, \neg\mathsf{false} \vee \mathsf{false}$. In other words, an action a can be reached after a (possibly empty) sequence of b actions in the network N if and only if an action a, or an action α_a, or an action α_b followed by an action a, can be reached immediately in $N_{|\overline{\{3\}}}$, given the behaviour of S_3 depicted in Figure 1.*

Interestingly, quotienting can be implemented as a network that realises a product between an LTS encoding the formula (called a *formula graph*) and a component LTS of the network under verification. In practice, the EXP.OPEN tool of CADP can be used for that purpose.

We now introduce the notion of *formula graph*, and then we define the network that implements quotienting.

Definition 20 (Circuit) *Let $S = (\Sigma, A, \longrightarrow, s_0)$ be an LTS and $T \subseteq \longrightarrow$ be a subset of its transitions. The* states *of T are defined as the set $\mathrm{st}\,(T) = \{s, s' \in \Sigma \mid (s, a, s') \in T\}$. T is a* circuit *of S if for all $s, s' \in \mathrm{st}\,(T)$ there is a sequence of transitions belonging to T from s to s'. A state $s \in \mathrm{st}\,(T)$ is a* root *of the circuit T if there is a sequence of transitions from s_0 to s that does not traverse any transition of T.*

[5]Note that terms will be compiled into graphs, thus enabling the sharing of sub-formulas that is also possible using equations.

Definition 21 (Formula graph) *A* formula graph *is an LTS* $(\Sigma, A, \longrightarrow, s_0)$ *such that:*

1. *Every label* $\sigma \in A$ *has either form* $\vee$, $\neg$, $\langle a \rangle$ *(for some* a *belonging to a fixed set of action names), or* μ^k *(for some* $k \in \mathbb{N}$*). To avoid confusion, in the sequel, we will use* σ *to denote the label of a formula graph and* a *to denote a label of the network on which the formula is to be checked.*

2. *If* $s_0 \xrightarrow{\delta} s \xrightarrow{\mu^k} s'$ *for some* $\delta \in A^*$ *and* $k \in \mathbb{N}$*, then* k *is even if and only if* δ *contains an even number of occurrences of the label* $\neg$*.*

3. *If* $s \in \Sigma$ *is a root of a circuit then (a) the circuit contains a* μ^k*-transition and (b) if the first* μ^k*-transition traversed on the circuit starting in* s *has block number* k' *then every* μ^k*-transition belonging to the circuit satisfies* $k \geq k'$*.*

Every formula graph can be decoded into a closed formula as follows.

Definition 22 (Decoding a formula graph) *A formula graph* $P = (\Sigma, A, \longrightarrow, s_0)$ *encodes the modal* μ*-calculus formula* $\mathsf{dec_s}\,(P, s_0, \emptyset)$*, where* $\mathsf{dec_s}\,(P, s, E)$ *is defined as follows (*$E \subseteq \Sigma$*). In our decoding every variable is uniquely identified by the source state* s *and the block number* k *of the* μ*-transition, and written* $\overline{s}^k$*.*

$$\mathsf{dec_s}\,(P, s, E) = \bigvee_{s \xrightarrow{\sigma} s' \in P} \mathsf{dec_t}\,(P, s \xrightarrow{\sigma} s', E)$$

where

$$\begin{array}{rcl}
\mathsf{dec_t}\,(P, s \xrightarrow{\vee} s', E) & = & \mathsf{dec_s}\,(P, s', E) \\
\mathsf{dec_t}\,(P, s \xrightarrow{\neg} s', E) & = & \neg\, \mathsf{dec_s}\,(P, s', E) \\
\mathsf{dec_t}\,(P, s \xrightarrow{\langle a \rangle} s', E) & = & \langle a \rangle\, \mathsf{dec_s}\,(P, s', E) \\
\mathsf{dec_t}\,(P, s \xrightarrow{\mu^k} s', E) & = & \begin{cases} \overline{s}^k & \textit{if } s \in E \\ \mu \overline{s}^k.\, \mathsf{dec_s}\,(P, s', E \cup \{s\}) & \textit{otherwise} \end{cases}
\end{array}$$

This definition implies that a deadlock state decodes as false (empty disjunction). Function $\mathsf{dec_s}$ is well-defined. In particular, it terminates because every cyclic path contains a label of the form μ^k. By recording in the set E the source states of traversed μ^k-transitions, we thus avoid infinite traversals of cycles. In practice, formula graphs need not be decoded.

Definition 23 (Encoding a formula into a formula graph) *The formula graph corresponding to a well-formed block-labelled formula* φ *in disjunctive form is an LTS written* $\mathsf{enc}\,(\varphi)$*, whose states are identified with sub-formulas of* φ*. The initial state of the formula graph is* φ*,* false *is a deadlock state, and each sub-formula has transitions as follows:*

$$\begin{array}{ccc}
X^k \xrightarrow{\vee} \varphi[X^k] & \neg\varphi_0 \xrightarrow{\neg} \varphi_0 & \langle a \rangle\, \varphi_0 \xrightarrow{\langle a \rangle} \varphi_0 \\
\varphi_1 \vee \varphi_2 \xrightarrow{\vee} \varphi_1 & \varphi_1 \vee \varphi_2 \xrightarrow{\vee} \varphi_2 & \mu X^k.\varphi_0 \xrightarrow{\mu^k} \varphi_0
\end{array}$$

Although the states of a formula graph are identified by formulas, only the transition labels are required for decoding. In figures, states will be simply identified by numbers.

Note that the formula graph obtained by encoding a formula satisfies the conditions given in Definition 21. Condition (2) is a direct consequence of the block-labelling convention stated in Definition 18. Condition (3) comes from the fact that the roots of the circuits are the states associated to formulas of the form $\mu X^k.\psi$ such that X^k occurs free in ψ. In particular, subcondition (b) is a consequence on the third well-formedness condition given in Definition 18.

Example 9 *The formula graph corresponding to the formula* $\mu X^0.(\langle a \rangle\, \mathsf{true}) \vee \langle b \rangle\, X^0$ *introduced in Example 8 is depicted in Figure 5 (a).*

Using this encoding, the quotient of a formula with respect to the ith LTS of a network can be computed as a synchronous product using a network called *quotient formula network*.

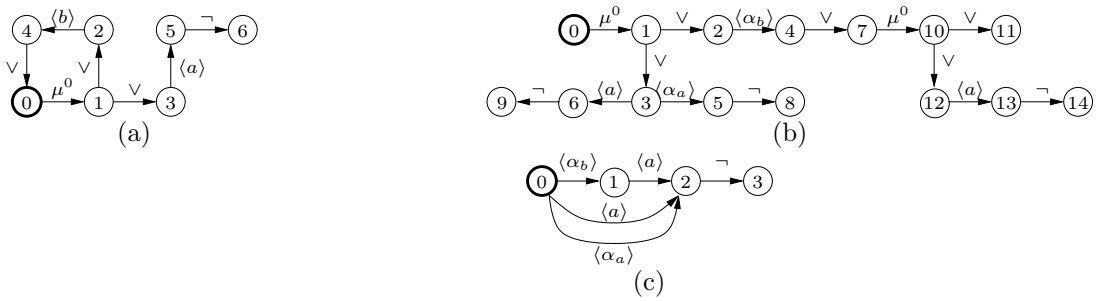


Figure 5: Examples of formula graphs

Definition 24 (Quotient formula network) *Let φ be a modal μ-calculus formula in disjunctive form, $N = (\vec{S}, V)$ be a network of size n, and $i \in [1, n]$. The* quotient formula network *of φ with respect to $\vec{S}[i]$ is defined as the network $((\mathsf{enc}\,(\varphi), \vec{S}[i]), V/\!\!/_i)$, where $V/\!\!/_i$ denotes the following set of rules:*

$$\begin{array}{llll} \{\ ((\sigma, & \bullet), & \sigma) & \mid \sigma \in \{\neg, \vee\} \cup \{\mu^k \mid k \in \mathsf{blocks}(\varphi)\}\ \} \quad \cup \\ \{\ ((\langle a \rangle, & \bullet), & \langle a \rangle) & \mid (\vec{t}, a) \in V \wedge i \notin \mathsf{acv}(\vec{t})\ \} \quad \cup \\ \{\ ((\langle a \rangle, & \vec{t}[i]), & \langle \alpha\,(\vec{t}, a) \rangle) & \mid (\vec{t}, a) \in V \wedge \{i\} \subsetneq \mathsf{acv}(\vec{t})\ \} \quad \cup \\ \{\ ((\langle a \rangle, & \vec{t}[i]), & \vee) & \mid (\vec{t}, a) \in V \wedge \{i\} = \mathsf{acv}(\vec{t})\ \} \end{array}$$

Note that the LTS corresponding to the quotient formula network is a formula graph. We have proven [88] that our encoding of closed formulas into formula graphs is sound, so that the formula can be recovered from the formula graph into which the formula is encoded (Proposition 9 below), and that the LTS corresponding to the quotient formula network indeed encodes the quotient correctly (Proposition 10 below).

Proposition 9 (Soundness of formula encoding) *If φ is a closed formula in disjunctive form, then $\mathsf{dec_s}\,(\mathsf{enc}\,(\varphi), \varphi, \emptyset)$ is equivalent to φ.*

Proposition 10 (Correctness of the quotient formula network) *The LTS corresponding to the quotient formula network of φ with respect to $\vec{S}[i]$ encodes a formula that is equivalent to $\varphi /\!\!/_i^{\emptyset}\ s_i^0$, where s_i^0 is the initial state of $\vec{S}[i]$.*

Example 10 *Consider the network N of Example 1 and the formula of Example 9. Quotienting of the formula with respect to S_3 involves the following set of rules:*

$$\{((\neg, \bullet), \neg), ((\vee, \bullet), \vee), ((\mu^0, \bullet), \mu^0), ((\langle a \rangle, \bullet), \langle a \rangle), ((\langle a \rangle, a), \langle \alpha_a \rangle), ((\langle b \rangle, b), \langle \alpha_b \rangle)\}$$

It yields the formula graph depicted in Figure 5 (b). This graph encodes as expected the quotient formula of Example 8, which can be evaluated on $N_{|\overline{\{3\}}}$.

Working with formulas in disjunctive form is crucial: branches in the formula graph denote disjunctions between sub-formulas (*or-nodes*). During composition between the formula graph and a component LTS, the impossibility to synchronize on a modality $\langle a \rangle$ (no transition labelled by $\vec{t}[i]$ in the current state of the component LTS) denotes invalidation of the corresponding sub-formula, which merely disappears, in conformance with the equality $\mathsf{false} \vee \varphi_0 = \varphi_0$.

The quotient of a formula graph with n states with respect to an LTS with m states may have up to $n \times m$ states. Hence, as observed by Andersen [4], simplifications are needed to keep intermediate quotiented formulas at a reasonable size. We present in Figure 6 several simplifications applying to formula graphs, as conditional rules of the form "$l \rightsquigarrow r$ (*cond*)" where l and r are transition relations and *cond* is a Boolean condition. l, r, and *cond* are expressed using variables representing either states (written $s, s_1, s_2, \ldots$) or labels (written $\sigma, \sigma_1, \sigma_2, \ldots$), such that every variable occurring in r or in *cond* must also occur in l. It means that all transitions matching the left-hand side so that *cond* is satisfied can be replaced by the transitions of the right-hand side.

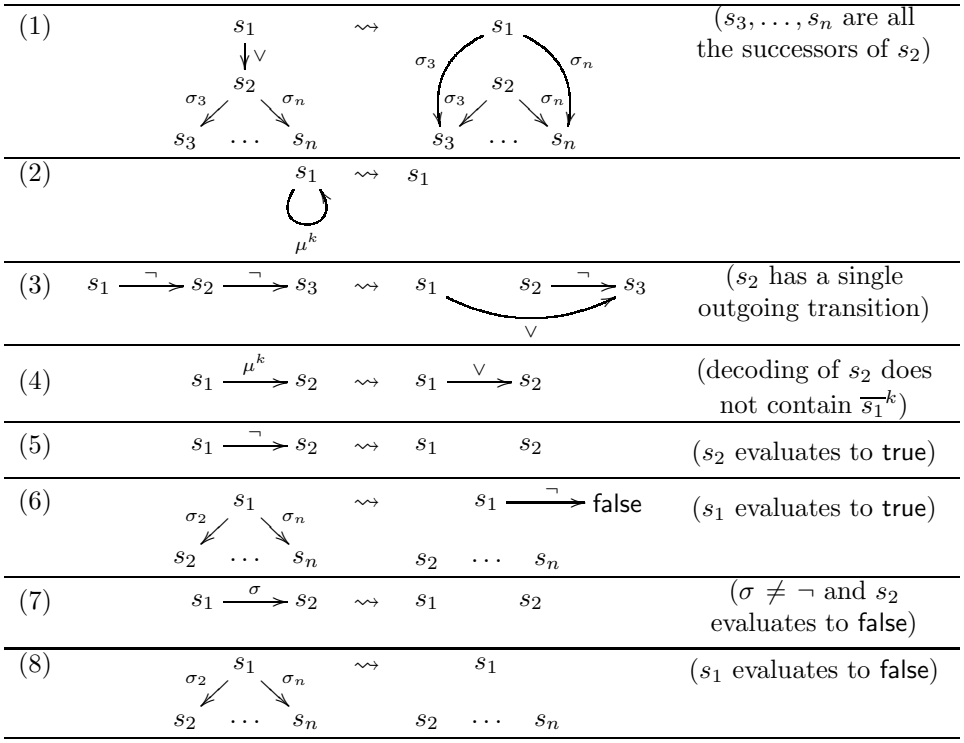


Figure 6: Simplification rules applying to formula graphs

Elimination of $\vee$-transitions (1). This rule allows transitions generated by synchronization rules of the form $((\langle a\rangle, \vec{t}[i]), \vee)$ in the quotient formula network to be eliminated. This elimination can be achieved efficiently by applying reduction modulo $\tau^*.a$ equivalence [51], $\vee$-transitions being interpreted as internal (τ) transitions.

Elimination of unguarded variables (2). When combined with the previous rule, this rule allows unguarded variable occurrences to be eliminated. Indeed, an unguarded variable is characterized by a (possibly empty) sequence of $\vee$-transitions connecting the target and source of a μ-transition. The elimination of this sequence of $\vee$ transitions then produces a self-looping transition labelled by μ, which can be thereafter eliminated using the current rule.

Elimination of double-negations (3). This rule can be used to simplify formulas of the form $\neg\neg\varphi$, which often occur in quotient formulas. For instance, a double-negation is introduced in the quotient of the formula $\neg\langle a\rangle\neg\varphi'$ (which is the disjunctive form of $[a]\varphi'$) with respect to an LTS that offers an action synchronizing with a (thus having the modality disappear if the synchronization is binary).

Elimination of μ-transitions (4). In this rule, the transition from s_1 to s_2 denotes the binder of a propositional variable $\overline{s_1}^k$. If this variable does not occur free in the sub-formula denoted by state s_2, then the μ-transition can be replaced by an $\vee$-transition, which can be subsequently eliminated using rule (1). Determining whether $\overline{s_1}^k$ occurs free would require to decode the formula graph, which should be avoided in practice. For this reason, we only consider the following sufficient conditions, which can be checked in linear-time:

- s_1 and s_2 are not in the same strongly connected component (i.e., there is no path from s_2 to s_1), or
- s_1 is not the initial state and has a single predecessor p, and either (1) p has a single outgoing transition (which necessarily goes to s_1) and this transition is labelled by $\mu^{k'}$, or (2) p satisfies the same condition

as s_1, recursively (this recursion is well-founded as long as it is applied to states reachable from the initial state)

Evaluation of constant sub-formulas (5–8). These four rules apply when some state denotes a sub-formula that evaluates to a constant in any context. This can be determined by using the following BES, which implements partial evaluation of the formula. This BES consists of blocks T^k and F^k ($k \in [0, n]$) of respective signs μ and ν, n being the greatest block number in the formula graph. Blocks are ordered so that $k < k'$ implies T^k (resp. F^k) occurs before $T^{k'}$ (resp. $F^{k'}$):

$$\begin{array}{lll} T^k: & \{\ T^k_s & =_\mu \ \bigvee_{s \xrightarrow{\vee} s'} T^k_{s'} \vee \bigvee_{s \xrightarrow{\neg} s'} F^k_{s'} \vee \bigvee_{s \xrightarrow{\mu^{k'}} s'} T^{k'}_{s'}\ \}_{s \in \Sigma} \\ F^k: & \{\ F^k_s & =_\nu \ \bigwedge_{s \xrightarrow{\vee} s'} F^k_{s'} \wedge \bigwedge_{s \xrightarrow{\langle\beta\rangle} s'} F^k_{s'} \wedge \bigwedge_{s \xrightarrow{\neg} s'} T^k_{s'} \wedge \bigwedge_{s \xrightarrow{\mu^{k'}} s'} F^{k'}_{s'}\ \}_{s \in \Sigma} \end{array}$$

We consider only the variables reachable from $T^0_{s_0}$ or $F^0_{s_0}$, s_0 being the initial state of the formula graph. A state s denotes true (resp. false) if the Boolean variables T^k_s (resp. F^k_s) evaluate to true in all (reachable) blocks k. Due to the presence of modalities, there may be states s and blocks k such that T^k_s and F^k_s are both false, indicating that the corresponding sub-formula is not constant. Intuitively, T^k_s expresses that s evaluates to true in block k (i.e., $\overline{s}^k$ evaluates to true) if one of its successors following a transition labelled by $\vee$ or $\mu^{k'}$ evaluates to true, or one of its successors following a transition labelled by $\neg$ evaluates to false. Variable F^k_s expresses that state s evaluates to false in block k (i.e., $\overline{s}^k$ evaluates to false) if all its successors following transitions labelled by $\vee$, $\mu^{k'}$, or modalities (by applying the identity $\langle a\rangle$ false = false) evaluate to false and all its successors following transitions labelled by $\neg$ evaluate to true. Regarding fixed point signs, observe that for the formula $\mu X^k.X^k$ (which is equivalent to the constant false), $F^k_{\mu X^k.X^k}$ and $T^k_{\mu X^k.X^k}$ are defined respectively by the greatest fixed point equation $F^k_{\mu X^k.X^k} =_\nu F^k_{\mu X^k.X^k}$ and the least fixed point equation $T^k_{\mu X^k.X^k} =_\mu T^k_{\mu X^k.X^k}$. This BES has the solution $F^k_{\mu X^k.X^k} = \mathsf{true}, T^k_{\mu X^k.X^k} = \mathsf{false}$, reflecting that $\mu X^k.X^k$ is constantly false, as expected.

Repeated application of quotienting progressively eliminates modalities, until none of them remains in the formula graph, which then necessarily evaluates to a constant equal to the result of evaluating the formula on the whole network.

Sharing of equivalent sub-formulas. In addition to the above eight rules, reducing a formula graph modulo strong bisimulation does not change its semantics. Strong bisimulation reduction can thus decrease the size of intermediate formula graphs.

Example 11 *After applying the above simplifications to the formula graph of Example 10, we obtain the (smaller) formula graph depicted in Figure 5 (c), which corresponds to the formula* $(\langle a\rangle\,\mathsf{true}) \vee (\langle\alpha_a\rangle\,\mathsf{true}) \vee (\langle\alpha_b\rangle\,\langle a\rangle\,\mathsf{true})$.

Example 12 *The graph corresponding to* $\mu X^0.(\langle a\rangle\,\mu Y^0.\,\langle b\rangle\,X^0) \vee \langle c\rangle\,X^0$ *reduces as expected to a deadlock state representing the constant* false *(left as an exercise).*

Note that the simplification of a formula graph produces a formula graph. In particular, the parity of the number of occurrences of the label $\neg$ on paths leading to a μ^k-transition is not changed by any rule, including rule (3) which eliminates negations by pair. Also, the simplifications do not create new circuits and every μ^k-transition eliminated by rule (4) cannot be the first μ^k-transition occurring on any circuit.

All the simplifications that we propose in this paper correspond more or less to simplifications already proposed by Andersen [4], but we apply them directly on formula graphs instead of systems of μ-calculus equations.

The paper [88] presents a specialization of this technique to the alternation-free μ-calculus formulas (using alternation-free BES), and how this specialisation can be again generalised to handle also useful fairness operators of alternation 2 (in particular the infinite looping operator $\langle_\rangle\,@$) in linear time without developing the complex machinery to evaluate general alternation-2 μ-calculus formulas.

5 High-level strategies and smart reduction

The efficiency of compositional model generation (including formula graph generation in the case of partial model checking) depends on the order in which the components are gathered and minimized. In practice, the order can be specified by the user, but since it is not always possible to predict 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 burden can be relieved using pre-defined strategies, which can be either based on the component hierarchy, or on heuristics to automatically determine efficient composition orders, based on the interactions between components.

5.1 Pre-defined strategies based on component hierarchy

We present three strategies for compositional model generation modulo an equivalence relation R, namely *leaf reduction*, *root leaf reduction*, and *node reduction*.

Definition 25 (Leaf and root leaf reductions) *Given a composition expression B and an equivalence relation R that is a congruence for composition expressions,* leaf reduction *computes another composition expression* $\mathsf{leaf_reduce}_R\,(B)$ *which has the same hierarchy as B, such that:*

- *hiding and cutting operators are propagated as far as possible in the composition expression, and*
- *the LTSs occurring in* $\mathsf{leaf_reduce}_R\,(B)$ *are minimized modulo the equivalence relation R after hiding and cutting.*

Formally, $\mathsf{leaf_reduce}_R\,(B)$ *is a shorthand for* $\mathsf{leaf_reduce}_R\,(B, \emptyset, \emptyset)$*, where* $\mathsf{leaf_reduce}_R\,(B, A_h, A_c)$ *is defined as follows, A_h denoting a set of labels to be hidden in B and A_c denoting a set of labels to be cut in B. The sets A_h and A_c satisfy the invariant $A_h \cap A_c = \emptyset$.*

$$
\begin{array}{l}
\mathsf{leaf_reduce}_R\,(S, A_h, A_c) = \mathsf{reduce}_R\,(\textbf{hide } A_h \textbf{ in cut } A_c \textbf{ in } S) \\
\mathsf{leaf_reduce}_R\,(B_1 \,|[A]|\, B_2, A_h, A_c) = \\
\quad \textbf{hide } A_h \cap A \textbf{ in} \\
\qquad \mathsf{leaf_reduce}_R\,(B_1, A_h \setminus A, A_c) \\
\qquad |[A]| \\
\qquad \mathsf{leaf_reduce}_R\,(B_2, A_h \setminus A, A_c) \\
\mathsf{leaf_reduce}_R\,(\textbf{rename } \theta \textbf{ in } B_0, A_h, A_c) = \\
\quad \textbf{rename } \theta \textbf{ in } \mathsf{leaf_reduce}_R\,(B_0, \overline{\theta}^{-1}(A_h), \overline{\theta}^{-1}(A_c)) \\
\quad \textit{where } \overline{\theta}^{-1}(A) = \{a \mid \overline{\theta}(a) \in A\} \\
\mathsf{leaf_reduce}_R\,(\textbf{hide } A \textbf{ in } B_0, A_h, A_c) = \\
\quad \mathsf{leaf_reduce}_R\,(B_0, A_h \cup A, A_c \setminus A) \\
\mathsf{leaf_reduce}_R\,(\textbf{cut } A \textbf{ in } B_0, A_h, A_c) = \\
\quad \mathsf{leaf_reduce}_R\,(B_0, A_h \setminus A, A_c \cup A)
\end{array}
$$

The root leaf reduction *is the same as leaf reduction, except that the LTS corresponding to the composition expression B' is finally generated and minimized. Therefore, the final result of root leaf reduction is a (minimal) LTS instead of a composition expression:*

$$\mathsf{root_leaf_reduce}_R\,(B) = \mathsf{reduce}_R\,(\mathsf{leaf_reduce}_R\,(B))$$

Proposition 11 *If R is a congruence for composition expressions, then* $\mathsf{leaf_reduce}_R\,(B)$ *and* $\mathsf{root_leaf_reduce}_R\,(B)$ *are both equivalent to B modulo R.*

Note however that B, $\mathsf{leaf_reduce}_R\,(B)$, and $\mathsf{root_leaf_reduce}_R\,(B)$ are not the same *operationally*: the sizes of intermediate LTSs differ in both expressions.

Proposition 12 *For strong, branching, and divbranching bisimulations, the largest intermediate LTS generated to compute* $\mathsf{root_leaf_reduce}_R\,(B)$ *is not larger than the largest intermediate LTS generated to compute* B*.*

The *node reduction* is another strategy, which makes more intermediate steps than *leaf* and *root leaf reduction*: an intermediate LTS is generated and minimized after each parallel composition.

Definition 26 (Node reduction) *Given a composition expression* B *and an equivalence relation* R *that is a congruence for composition expressions,* node reduction *computes an LTS* $\mathsf{node_reduce}_R\,(B)$*. Formally,* $\mathsf{node_reduce}_R\,(B)$ *is a shorthand for* $\mathsf{node_reduce}_R\,(B, \emptyset, \emptyset)$*, where* $\mathsf{node_reduce}_R\,(B, A_h, A_c)$ *is defined as follows,* A_h *denoting a set of labels to be hidden in* B *and* A_c *denoting a set of labels to be cut in* B*. The sets* A_h *and* A_c *satisfy the invariant* $A_h \cap A_c = \emptyset$*.*

$$
\begin{array}{l}
\mathsf{node_reduce}_R\,(S, A_h, A_c) = \mathsf{reduce}_R\,(\textbf{hide } A_h \textbf{ in cut } A_c \textbf{ in } S) \\
\mathsf{node_reduce}_R\,(B_1 \,|[A]|\, B_2, A_h, A_c) = \\
\quad \mathsf{reduce}_R\,(\textbf{hide } A_h \cap A \textbf{ in} \\
\qquad \mathsf{node_reduce}_R\,(B_1, A_h \setminus A, A_c) \\
\qquad |[A]| \\
\qquad \mathsf{node_reduce}_R\,(B_2, A_h \setminus A, A_c)) \\
\mathsf{node_reduce}_R\,(\textbf{rename } \theta \textbf{ in } B_0, A_h, A_c) = \\
\quad \mathsf{reduce}_R\,(\textbf{rename } \theta \textbf{ in } \mathsf{node_reduce}_R\,(B_0, \overline{\theta}^{-1}(A_h), \overline{\theta}^{-1}(A_c))) \\
\quad \textit{where } \overline{\theta}^{-1}(A) = \{a \mid \overline{\theta}(a) \in A\} \\
\mathsf{node_reduce}_R\,(\textbf{hide } A \textbf{ in } B_0, A_h, A_c) = \\
\quad \mathsf{node_reduce}_R\,(B_0, A_h \cup A, A_c \setminus A) \\
\mathsf{node_reduce}_R\,(\textbf{cut } A \textbf{ in } B_0, A_h, A_c) = \\
\quad \mathsf{node_reduce}_R\,(B_0, A_h \setminus A, A_c \cup A)
\end{array}
$$

Proposition 13 *If* R *is a congruence for composition expressions, then* $\mathsf{node_reduce}_R\,(B)$ *is equivalent to* B *modulo* R*.*

Unlike root leaf reduction (which does not compute intermediate compositions before computing the LTS corresponding to B), there is no guarantee that computing the node reduction of B is less expensive (in time or memory) than computing the LTS corresponding to B. Indeed, the LTSs corresponding to the intermediate compositions may be larger (even much larger) than the LTS corresponding to B. However, there exist cases where node reduction is efficient.

5.2 Heuristic-based strategy: Smart reduction

Heuristics to automatically determine efficient composition orders, based on the component interactions, have been proposed in [128] for concurrent finite state machines communicating via named channels. More recently, such heuristics have been refined and implemented in a prototype tool for LTSs synchronizing on their common alphabets [40]. In both works, the components 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 restricted composition operators, which are generally insufficient to capture the semantics of state-of-the-art software description languages: The parallel composition operators used in [128] do not support multiway synchronization (more than two components synchronizing all together), and none of the parallel composition operators used in [128, 40] enable nondeterministic synchronization (e.g., two clients competing to synchronize with a server on some label).

In this section, we present a refinement of the compositional model generation techniques of [128, 40], called *smart reduction* [41], which relies on networks of LTSs. The first advantage is that it makes compositional model generation applicable to a wider variety of operators, provided that they can be compiled into networks. The second advantage is that networks enable any composition order, which is not in general the

case in process algebraic models, where some composition orders, possibly including the optimal order, may not be represented using the available algebraic operators.

The LTS corresponding to a network can be generated incrementally, by alternating compositions of well-chosen subsets of the components and minimizations modulo an equivalence relation. We call *aggregation* a composition followed by a minimization of the result, and *compositional aggregation* this incremental technique. Contrary to composition expressions, networks enable all possible aggregations as they are not constrained by a component hierarchy.

Definition 27 (Aggregation) *As in Definition 19, we assume a function $\alpha(\vec{t}, a)$ that associates to each $(\vec{t}, a) \in V$ a unique label distinct from all others. Given a network N of size n and a set of indices $I \subseteq [1, n]$, we define the network* $\mathsf{agg}\,(N, I)$ *obtained by* aggregation *of components inside I as follows:*

$$
\begin{array}{rrcl}
 & \mathsf{agg}\,(N, I) & = & (\mathsf{reduce}_R\,(N_{|I}) :: \vec{S}|\overline{I}, V_{\mathsf{agg}}) \\
\textit{where} & N_{|I} & = & (\vec{S}|I, V_{|I}) \\
 & V_{\mathsf{agg}} & = & \{ \ (a :: \vec{t}|\overline{I}, a) \quad | \ (\vec{t}, a) \in V \wedge \mathsf{acv}(\vec{t}) \subseteq I \ \} \cup \\
 & & & \{ \ (\alpha(\vec{t}, a) :: \vec{t}|\overline{I}, a) \quad | \ (\vec{t}, a) \in V \wedge \emptyset \subsetneq (I \cap \mathsf{acv}(\vec{t})) \subsetneq \mathsf{acv}(\vec{t}) \ \} \cup \\
 & & & \{ \ (\bullet :: \vec{t}|\overline{I}, a) \quad | \ (\vec{t}, a) \in V \wedge (I \cap \mathsf{acv}(\vec{t})) = \emptyset \ \} \\
\textit{and} & V_{|I} & = & \{ \ (\vec{t}|I, a) \quad | \ (\vec{t}, a) \in V \wedge \mathsf{acv}(\vec{t}) \subseteq I \ \} \cup \\
 & & & \{ \ (\vec{t}|I, \alpha(\vec{t}, a)) \quad | \ (\vec{t}, a) \in V \wedge \emptyset \subsetneq (I \cap \mathsf{acv}(\vec{t})) \subsetneq \mathsf{acv}(\vec{t}) \ \}
\end{array}
$$

The components inside I ($\vec{S}|I$) are aggregated into $\mathsf{reduce}_R\,(N_{|I})$ *and the components outside I ($\vec{S}|\overline{I}$) are kept non-aggregated. The synchronization rules $V_{|I}$ of the auxiliary network $N_{|I}$ are obtained by projection of V on to I, whereas the synchronization rules V_{agg} are obtained by synchronization of the labels in $V_{|I}$ with the projection of V on to $\overline{I}$.*

For convenience, since I denotes the set of components to be aggregated, we call I an aggregation.

The rules of $V_{|I}$ and V_{agg} are organized in three subsets:

- The first subset, containing those rules of the form $(\vec{t}|I, a)$ in $V_{|I}$ and $(a :: \vec{t}|\overline{I}, a)$ in V_{agg}, represents the synchronization rules that involve only components inside I. In this case, $\vec{t}|\overline{I}$ is a vector of $\bullet$, which expresses that transitions of $\mathsf{reduce}_R\,(N_{|I})$ obtained by synchronization of components inside I do not need to synchronize with components outside I.

- The second subset, containing those rules of the form $(\vec{t}|I, \alpha(\vec{t}, a))$ in $V_{|I}$ and $(\alpha(\vec{t}, a) :: \vec{t}|\overline{I}, a)$ in V_{agg}, represents the synchronization rules that involve both components inside I and components outside I. This expresses that transitions of $\mathsf{reduce}_R\,(N_{|I})$ obtained by synchronization of components inside I still need to synchronize with components outside I. The special label $\alpha(\vec{t}, a)$ is an intermediate label used for this synchronization. Note that in general, a cannot be used instead of $\alpha(\vec{t}, a)$ because (1) a can be the internal label τ (even though $\vec{t}$ 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 synchronization is nondeterministic involved) and using a could create unexpected synchronizations.

- The third subset, containing those rules of the form $(\bullet :: \vec{t}|\overline{I}, a)$ in V_{agg}, represents the synchronization rules that involve only components outside I. These rules do not impose synchronization constraints on components inside I, thus explaining why $V_{|I}$ has no rule in the third subset.

Proposition 14 *If R is a congruence for networks, then* $\mathsf{lts}\,(\mathsf{agg}\,(N, I))$ *is equivalent modulo R to* $\mathsf{lts}\,(N)$.

Definition 28 (Compositional aggregation algorithm) *A compositional aggregation algorithm to generate and minimize modulo R the LTS corresponding to a network of LTSs $((S_1, \ldots, S_n), V)$ of size n is defined iteratively as follows:*

1. $N := (\mathsf{reduce}_R\,(S_1), \ldots, \mathsf{reduce}_R\,(S_n), V)$

2. *Select a set I of indices containing at least those of two of the component LTSs of N. A strategy for selection will be addressed in the next section.*

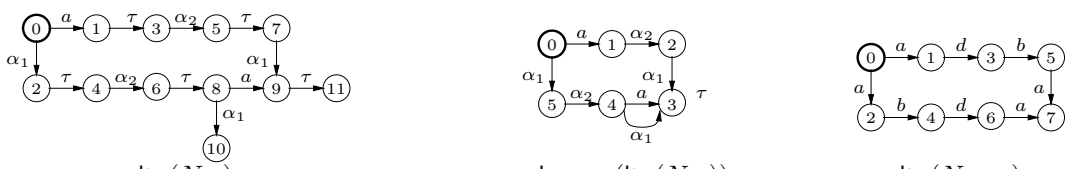
3. $N := \mathsf{agg}\,(N, I)$.

4. *If N still contains more than two LTSs, then continue in step 2. Otherwise return* $\mathsf{reduce}_R\,(N)$.

Proposition 15 *The compositional aggregation algorithm terminates and if R is a congruence for networks, then it returns an LTS that is equivalent to its input network modulo R.*

Equivalence modulo R follows from Proposition 14, which ensures that $\mathsf{lts}\,(N)$ remains invariantly equivalent to the input modulo R until the end of the algorithm. Termination follows from the fact that the size of $\mathsf{agg}\,(N, I)$ is the size of N plus 1 minus the size of I (which is at least 2). Since N is substituted by $\mathsf{agg}\,(N, I)$ at each step, this guarantees that the size of N strictly decreases at each step. These results establish the correctness of compositional aggregation.

Example 13 *We write* $\mathsf{reduce}_{\mathsf{bra}}\,(S)$ *for minimization of S modulo branching bisimulation. The compound LTS corresponding to the network of Example 1, whose components S_1, S_2, and S_3 are already minimal, can be generated modulo branching bisimulation by first composing S_1 and S_2, then S_3 as follows. First, build $N_{12} = N_{|\{1,2\}} = ((S_1, S_2), V_{12})$ for the aggregation of S_1 and S_2, where V_{12} is the set of rules $\{((a,a),a),((a,\bullet),\alpha_1),((b,b),\alpha_2),((c,c),\tau)\}$, α_1 is the label $\alpha((a,\bullet,a),a)$, and α_2 is the label $\alpha((b,b,b),b)$. Compute the intermediate LTS $S_{12} = \mathsf{reduce}_{\mathsf{bra}}\,(\mathsf{lts}\,(N_{12}))$ (see below). Second, build $N_{(12)3} = \mathsf{agg}\,(N, \{1,2\}) = ((S_{12}, S_3), V_{(12)3})$, where $V_{(12)3}$ is the set $\{((a,\bullet),a),((\alpha_1,a),a),((\alpha_2,b),b),((\tau,\bullet),\tau),((\bullet,d),d)\}$. Return* $\mathsf{reduce}_{\mathsf{bra}}\,(\mathsf{lts}\,(N_{(12)3}))$ *(see* $\mathsf{lts}\,(N_{(12)3})$ *below), which is branching equivalent to* $\mathsf{lts}\,(N)$ *(see Figure 1). Note that both LTSs* $\mathsf{lts}\,(N_{12})$ *and* $\mathsf{lts}\,(N_{(12)3})$ *are smaller than* $\mathsf{lts}\,(N)$.



$\mathsf{lts}\,(N_{12})$ $\mathsf{reduce}_{\mathsf{bra}}\,(\mathsf{lts}\,(N_{12}))$ $\mathsf{lts}\,(N_{(12)3})$

Other aggregation orders are possible, yielding different intermediate graphs. Figure 7 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 states and 12 transitions).

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. We present *smart reduction*, which corresponds to compositional aggregation using a heuristic based on metrics evaluated against possible aggregations. The metrics use an estimate of the number of compound transitions in I, defined as follows.

Definition 29 (Estimate number of compound transitions) *The estimate number of compound transitions in I generated by synchronization vector $\vec{t}$, written $ET(I,\vec{t})$, is defined below:*

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

Informally, $ET(I,\vec{t})$ counts, for vector $\vec{t}$, the number of transitions going out of every product state of I, including unreachable states. This count equals 0 if $\vec{t}$ does not involve components inside I (first line). Otherwise, $N_{|I}$ has a rule of the form $(\vec{t}|I, b)$. This rule generates a transition in a state of $N_{|I}$ without condition on the states of the components $\vec{S}[i]$ such that $i \in I \setminus \mathsf{acv}(\vec{t})$ — thus justifying the first

Order 1 : (S_1, S_2) then S_3	states	transitions
$\mathsf{comp}(S_1, S_2)$	**12**	**12**
$\mathsf{reduce_{bra}}\,(\mathsf{comp}(S_1, S_2))$	6	7
$\mathsf{comp}(\mathsf{reduce_{bra}}\,(\mathsf{comp}(S_1, S_2)), S_3)$	8	8
$\mathsf{reduce_{bra}}\,(\mathsf{comp}(\mathsf{reduce_{bra}}\,(\mathsf{comp}(S_1, S_2)), S_3))$	7	7
Order 2 : (S_1, S_3) then S_2	states	transitions
$\mathsf{comp}(S_1, S_3)$	**19**	**23**
$\mathsf{reduce_{bra}}\,(\mathsf{comp}(S_1, S_3))$	17	22
$\mathsf{comp}(\mathsf{reduce_{bra}}\,(\mathsf{comp}(S_1, S_3)), S_2)$	15	17
$\mathsf{reduce_{bra}}\,(\mathsf{comp}(\mathsf{reduce_{bra}}\,(\mathsf{comp}(S_1, S_3)), S_2))$	7	7
Order 3 : (S_2, S_3) then S_1	states	transitions
$\mathsf{comp}(S_2, S_3)$	**18**	**34**
$\mathsf{reduce_{bra}}\,(\mathsf{comp}(S_2, S_3))$	**18**	**34**
$\mathsf{comp}(\mathsf{reduce_{bra}}\,(\mathsf{comp}(S_2, S_3)), S_1)$	15	17
$\mathsf{reduce_{bra}}\,(\mathsf{comp}(\mathsf{reduce_{bra}}\,(\mathsf{comp}(S_2, S_3)), S_1))$	7	7
Order 4 : (S_1, S_2, S_3)	states	transitions
$\mathsf{comp}(S_1, S_2, S_3)$	**15**	**17**
$\mathsf{reduce_{bra}}\,(\mathsf{comp}(S_1, S_2, S_3))$	7	7

Figure 7: Sizes of intermediate graphs for all aggregation orders; given $S_1, \ldots, S_m$ some components of the current network N and I the set of their indices, $\mathsf{comp}(S_1, \ldots, S_m)$ denotes $\mathsf{lts}\,(N_{|I})$

product in the definition of $ET(I, \vec{t})$ — and provided that the states of the components $\vec{S}[i]$ such that $i \in I \cap \mathsf{acv}(\vec{t})$ have a transition labelled $\vec{t}[i]$ — thus justifying the second product. In general, the exact number of compound transitions in I generated by synchronization vector $\vec{t}$ is below this count since some states may be unreachable.

We now define our metrics on networks, which is the sum of two terms, namely the *hiding metrics* and the *interleaving metrics*, defined below.

Definition 30 (Hiding metrics) *The* hiding metrics *is defined by $HM(I) = HR(I)/|I|$, where $HR(I)$ (the* hiding rate*) is defined as follows:*

$$HR(I) = \frac{\displaystyle\sum_{(\vec{t},\tau) \in V \wedge \mathsf{acv}(\vec{t}) \subseteq I} ET(I, \vec{t})}{1 + \displaystyle\sum_{(\vec{t},a) \in V} ET(I, \vec{t})}$$

Informally, $HR(I)$ represents an estimate of the proportion of transitions in $N_{|I}$ that are internal. Those transitions are necessarily created by rules involving only components 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 metrics will be combined with the *interleaving metrics* defined as follows.

Definition 31 (Interleaving metrics) *Let $\vec{t}@i$ denote the synchronization vector of size n defined by $(\vec{t}@i)[i] = \vec{t}[i]$ and $(\forall j \in [1, n] \setminus \{i\})\ (\vec{t}@i)[j] = \bullet$.*

The interleaving metrics *is defined by $IM(I) = (1 - IR(I))/|I|$, where the* interleaving rate *$IR(I)$ is defined as follows:*

$$IR(I) = \frac{\sum_{(\vec{t},a)\in V} ET(I,\vec{t})}{1 + \sum_{(\vec{t},a)\in V} \sum_{i\in I\cap \mathsf{acv}(\vec{t})} ET(I,\vec{t}@i)}$$

Intuitively, the value $IR(I)$ is the quotient between an estimate of the number of compound transitions of I and the number of compound transitions that I would have if all components were fully interleaving. For an aggregation I of fully interleaving components, we thus have $IR(I)$ very close to 1. It is a refinement of the *interleaving count* defined in [40], which uses the proportion of fully interleaving (i.e., non-synchronized) component LTS transitions out of the total number of component 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.

Definition 32 (Combined metrics) *Taking into account both the hiding rate and the interleaving rate, we use here the* combined metrics $CM(I) = HM(I) + IM(I)$.

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 [40] 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 components S_i, S_j in the aggregation, S_i and S_j are *connected*, which is defined recursively as follows: either there is a synchronization rule $(\vec{t}, a)$ such that $\{i, j\} \subseteq \mathsf{acv}(\vec{t})$ (i.e., S_i and S_j are synchronized) or, recursively, the aggregation contains a third (distinct) LTS S_k connected to both S_i and S_j.

Example 14 *The metrics evaluate as follows on the network of Examples 1 and 13:*

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 metrics 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 7 and the values in the above table shows that one aggregation is more efficient than another whenever the combined metrics gives it a higher value.*

5.3 Combining smart reduction and partial model checking

The order in which the components of the network are composed and minimized has also an influence on the efficiency of partial model checking. The only difference is that we aim to generate compositionally the LTS corresponding to the quotient formula network obtained by composition of the formula graph with the network of LTSs under verification. Therefore, the smart reduction heuristics can be used to select the next aggregation to be computed. The reduction that has to be applied at each step depends whether the selected aggregation contains the formula graph (corresponding to the computation of a quotient) or not (corresponding to a simple compositional model generation step). If the formula graph belongs to the aggregation, then the simplifications designed for partial model checking must be applied. Otherwise, the aggregation can be minimized modulo any appropriate equivalence relation that is adequate with the formula under verification.

Currently, we use a simple strategy: the heuristics always selects an aggregation limited to two components, which always includes the formula graph, so that the formula graph simplifications are applied at each step. More elaborate strategies could be designed in the future.

6 Implementation in the CADP toolbox

All the above concepts and algorithms for compositional verification have been implemented in the framework of CADP [58]. CADP (*Construction and Analysis of Distributed Processes*) is a tool platform implementing salient results of concurrency theory, with a particular focus on applicability and scalability of the proposed approaches. Related tool platforms for asynchronous, action-based systems are, e.g., the *Edinburgh Concurrency Workbench* [37] and *Concurrency Workbench of the New Century*[6] for CCS, the *Failures-Divergences Refinement* tools [10, 65] for CSP, the *micro Common Representation Language*[7] toolset [72] for ACP, and LTSmin[8], which supports several input languages [21].

The CADP[9] toolbox is one of the pioneering platforms in concurrency theory, as its development was undertaken in the mid-80s. Since then, the toolbox has grown to include nearly 50 tools; moreover, CADP tries as much as possible to interface with tools developed by other research teams, so as to provide comprehensive solutions for analyzing complex concurrent models.

Many tools of CADP address compositional verification issues, or are needed to do so. In the present section, we give an orderly presentation of these tools sorted according to their functionalities.

6.1 Labelled transition systems and their generation

CADP implements the theoretical concept of LTS in two different ways:

- For *explicit* LTSs that are defined *in extension* by giving the list of their states and transitions, CADP provides a file format named BCG (*Binary-Coded Graphs*). This format enables large LTSs to be stored on disk in a compact manner (typically, two bytes per transition). It is equipped with programming libraries for reading and writing BCG files, as well as a collection of tools that query and transform BCG files, e.g., to compute statistics such as the branching factor, to display the list of labels attached to transitions, to relabel transitions using string substitutions defined by regular expressions, etc. Notice that CADP supports other LTS formats (such as AUT, FC2, and SEQ), which are textual and, thus, readable by humans; however, BCG remains the format of choice for large explicit LTSs.

- For *implicit* LTSs that are defined *in comprehension* by giving their initial state and their transition function, CADP provides the OPEN/CAESAR Application Programming Interface [56]. This interface enables algorithms to be written for on-the-fly LTS exploration, and provides sufficient abstraction capabilities to make such algorithms independent from the details of the particular source language used to specify these LTSs. Above the OPEN/CAESAR interface, many tools have been developed, some of them will be described in this section.

These LTSs can be produced in different ways using the CADP tools. They can be generated from specifications written in a data-passing process calculus such as LOTOS (using the CAESAR.ADT and CAESAR compilers [55, 60]) or from the parallel composition of several LTSs (using the EXP.OPEN tool [85], which implements the parallel, hiding, and renaming operators of all mainstream process calculi, and supports the Arnold-Nivat synchronization vectors as well). These LTSs can also be generated from other concurrent languages, such as LNT [28], FSP [94, 89], CHP [98, 59] that are automatically translated to LOTOS and/or EXP.OPEN input formalism.

6.2 Equivalence checking: Minimization and comparison of labelled transition systems

The CADP toolbox also provides tools for minimizing (i.e., computing the quotient) of an LTS modulo various behavioral equivalence relations — especially bisimulations — and to compare two LTSs with each other modulo various behavioral equivalence or preorder relations. The development of such tools has been a

[6] `http://www.cs.sunysb.edu/~cwb`
[7] `mcrl2.org`
[8] `fmt.cs.utwente.nl/tools/ltsmin`
[9] `http://cadp.inria.fr`

continuous effort since the early days of CADP. The ALDEBARAN tool [52] was developed for this purpose, and was later replaced by version 1 of another tool named BCG_MIN.

At present, CADP provides three equivalence-checking tools: version 2 of BCG_MIN [39], which can minimize an explicit LTS encoded in the BCG format, BCG_CMP, which can compare two explicit LTSs encoded in the BCG format, and BISIMULATOR [102], which can compare an implicit LTS that implements the OPEN/CAESAR interface against an explicit LTS encoded in the BCG format. On powerful machines, these tools have been able to successfully handle LTS with one billion explicit states and several billion transitions. The comparison of LTSs can generate diagnostics, i.e., discrimininating sequences of transitions (or more generally, graph fragments) that explain why two LTSs are not equivalent or included one in the other.

6.3 Semi-composition and interfaces

To implement the concept of semi-composition, CADP provides the PROJECTOR tool, which takes as input an implicit LTS (represented using the OPEN/CAESAR Application Programming Interface) and an interface (encoded in the BCG format), and produces as output a BCG file containing the semi-composition of the LTS and the interface. The PROJECTOR tool has been continuously enhanced: over the years, three successive versions of the tool have been developed. The current version is PROJECTOR 3. PROJECTOR was used successfully in the verification of an industrial protocol [132].

6.4 Model checking: Local and partial evaluation

There are many model checkers in the world, with many academic prototypes and a few industrial-strength tools. Most model checkers follow the *state-based approach*, in which temporal logic formulas are built upon atomic properties that can query the values of variables contained in each state of the system, e.g., computing the set of states in which a given variable X is positive. The state-based approach relies on a "white-box" assumption, i.e., the assumption that the contents of all variables of the system under verification are observable.

In concurrency theory, however, such a "white-box" assumption does not hold in general, and an *action-based approach* is used instead: the mainstream semantic models of concurrency theory (e.g., the concepts of LTS or Markov chains) follow a "black box" assumption, in which all information is attached to transitions and no information is attached to states, except the difference between the initial state(s) and other states. Practically, in order to remain compatible with equivalence checking (especially, bisimulations), the atomic properties of the temporal logic used must refer to actions (i.e., transition labels) rather than state contents. Such design constraint is materialized, e.g., by the difference that exists between the *modal* μ-calculus [25], which is action-based, and the *propositional* μ-calculus [82], which is state-based.

The CADP toolbox provides advanced tools for model checking, but does it in a way that remains compatible with the foundations of concurrency theory. Namely, the CADP tools follow the action-based approach rather than the state-based one. All the model checking tools of CADP work on the fly, on implicit LTSs rather than explicit LTSs, thus enabling to halt state-space generation as soon as the property under evaluation is found to be false or true. Moreover, all the model checking tools of CADP can produce diagnostics (also called witnesses or counterexamples), i.e., (explicit) fragments from the implicit LTS in order to explain why a property is true or false.

The first model checking tools added to CADP in the 90s were specialized for simple classes of properties. For instance, the EXHIBITOR tool searches for traces matching a given regular expression, and the TERMINATOR tool searches for deadlocks (i.e., states with no outgoing transition) using Holzmann's *bitstate hashing* algorithm.

Later, more generic model checkers have been added to CADP. The EVALUATOR 3 model checker [104] is based on the alternation-free modal μ-calculus, supplemented for convenience with regular expressions on action names and execution sequences. To produce its verdicts, EVALUATOR 3 relies on the CAESAR_SOLVE engine [102], a library for solving Boolean equation systems on the fly and generating appropriate diagnostics. For many years, EVALUATOR 3 has been intensively used within CADP and applied to a number of large case studies, many of which in an industrial context.

Recently, EVALUATOR 4 [105] has been introduced as the new default model checker of CADP. EVALUATOR 4 generalizes EVALUATOR 3 by accepting a more expressive logic named MCL, in which one can specify properties and dependencies on the typed data values contained in action names. Using MCL, one can express meaningful and practically relevant properties (such as: "on any execution path, the sequence number attached to each message is constantly increasing") that cannot be expressed using conventional logics, which consider actions as members of some finite alphabet, rather than complex data structures containing both communication channel names as well as typed value exchanged between concurrent processes. To evaluate the rich formula language of MCL, the concept of *Parameterized Boolean Equation Systems* [100] has been invented and implemented in the EVALUATOR 4 tool.

Finally, in order to support partial model checking, the PMC[10] tool has been developed and made available as a CADP companion tool.

6.5 The SVL language for compositional verification

Although compositional verification is firmly grounded in strong theoretical results and although it is likely to bring significant reductions in complexity, this is not sufficient to make it usable on a practical scale, especially by novice users. Indeed, applying compositional verification to any non-trivial system raises a number of difficult, low-level, yet unavoidable issues, among which: invoking all the required CADP tools in the proper order with appropriate command-line options, splitting models into components and logic formulas into subformulas, recombining system components and subformulas into larger agglomerates, and managing tens (and often hundreds) of intermediate files, which need to be archived, given unique names, and to be deleted as soon as they are no longer useful.

To address these issues, the CADP toolbox is equipped with a unique solution named SVL (*Script Verification Language*)[11] [57, 84]. SVL is both a high-level scripting language proposed to CADP end-users, as well as a compiler that translates SVL scripts into Bourne shell scripts, which can be directly executed on any POSIX system and perfom all low-level tasks of invoking the CADP tools and managing intermediate files. SVL enables the description of systems consisting of:

- Explicit LTSs encoded in one of the formats supported by CADP, namely, AUT, BCG, FC2, or SEQ;
- Implicit LTSs represented using the OPEN/CAESAR Application Programming Interface and generated from high-level languages such as LOTOS, LNT, FSP, or any concurrent language that can be translated to these;
- Explicit LTSs obtained from implicit LTSs by exhaustively enumerating all their reachable states and transitions;
- Parallel composition of LTSs combined together using the parallel operators of mainstream process calculi (e.g., ACP/μCRL [72], CCS [108], CSP [125], LOTOS [79], E-LOTOS [80], LNT [28]) or Arnold-Nivat synchronization vectors;
- Semi-composition of an LTS projected over another LTS playing the role of an interface;
- Modification of an LTS by applying operations that hide or rename given labels, or that cut transitions labelled with given labels (i.e., labels that match specified regular expressions);
- Minimization (total or partial) of an LTS modulo some behavioural equivalence, e.g., bisimulation relations.

Additionally, SVL provides language constructs to perform the following verification tasks:

- Equivalence checking: comparison of two systems to decide whether they are equivalent or included one in the other modulo behaviour equivalence or preorder relations;

[10] `http://convecs.inria.fr/software/pmc`
[11] `http://cadp.inria.fr/man/svl.html`

	TFTP A		TFTP B	
Scenario	read	write	read	write
A		✓		
B	✓			
C		✓		✓
D	✓			✓
E	✓		✓	

Table 1: The five scenarios of the TFTP/UDP case study

- Model checking: search for deadlocks or livelocks in a system and, more generally, evaluation of a temporal logic formula on a system;
- Compositional verification: SVL has so-called *meta-operators* that recursively apply (total or partial) minimizations to the sub-components of a system, using various compositional reduction strategies, among which smart reduction.

Finally, SVL implements various expert strategies to make verification tractable. For instance, the LTSs representing interfaces are automatically minimized modulo safety equivalence before being used for semi-composition; also, when branching minimization fails to handle a large LTS, then SVL first tries to apply strong minimization before restarting branching bisimulation, and so on. Examples of SVL scripts are provided in Section 7.

7 Applications and experimental results

We now illustrate how the various compositional verification techniques presented in this paper and available in CADP can be applied to a common case study, an airplane-ground communication protocol studied by Airbus and based upon TFTP/UDP (*Trivial File Transfer Protocol/User Datagram Protocol*) [62]. The modelled system consists of two TFTP entities (A and B), connected by an UDP link that can be modelled as a lossy FIFO channel.

Since the state space of this case study is very large, five scenarios named A to E are considered [62], depending on whether each TFTP entity may write and/or read a file (see Table 1). We used the same five scenarios in our study. All of them are specified in LNT [28], in files named `SCENARIO_A.lnt` to `SCENARIO_E.lnt`, as the parallel composition of eight concurrent components named TFTP_A, TFTP_B, MEDIUM_A, MEDIUM_B, RCV_A, RCV_B, SND_A, and SND_B. All experiments were done on a 64-bit computer with 148 gigabytes of memory.

7.1 Generation of minimized LTS

Direct generation of the compound LTS. Our aim is to generate a compound LTS that is as small as possible, either for strong or for (div)branching bisimulation. A first way to obtain this LTS is to generate directly and then minimize the compound LTS. For scenario A and divbranching equivalence, we use the following SVL script:

"SCENARIO_A.bcg" = **divbranching reduction of**
generation of "SCENARIO_A.lnt"

The LTS is small enough to be generated directly, and contains $12,885,069$ states and $51,305,563$ transitions. Its reduction modulo divbranching bisimulation yields an LTS with $1,620,754$ states and $7,060,163$ transitions, i.e., between 7 and 8 times less states and transitions, but the reduction already requires more than 1 gigabyte of memory. For the other scenarios, the LTSs obtained are much larger, which justifies resorting to compositional methods.

	Non-minimized		Minimized	
	States	Trans.	States	Trans.
TFTP_A	30,865	263,986	704	4,542
TFTP_B	24,846	213,118	504	3,421
MEDIUM_{A,B}	4,642	9,296	801	5,440
SND_A, RCV_B	1	4	1	4
SND_B, RCV_A	1	4	1	4

Table 2: LTS sizes for scenario A, before and after minimization modulo strong, branching, or divbranching bisimulations (component LTSs do not contain internal actions)

	Scenario B		Scenario C		Scenario D		Scenario E	
	States	Trans.	States	Trans.	States	Trans.	States	Trans.
TFTP_A	719	4,610	704	4,542	719	4,610	719	4,610
TFTP_B	504	3,421	1,058	7,164	1,058	7,164	1,079	7,274
MEDIUM_{A,B}	801	5,440	801	5,440	801	5,440	801	5,440
SND_A, RCV_B	1	4	1	7	1	5	1	6
SND_B, RCV_A	1	3	1	7	1	6	1	6

Table 3: Component LTS sizes (states and transitions) once minimized for strong, branching, or divbranching bisimulation (which coincide on this example) for scenarios B to E

Component LTS generation. The first step consists in generating and reducing the LTSs of the components. We use the LNT.OPEN and GENERATOR tools of CADP to generate the LTSs of the components, and BCG_MIN to minimize them. This can be done easily using SVL statements of the following form:

"TFTP_A.bcg" = **divbranching reduction of**
generation of "SCENARIO_A.lnt" : TFTP_A

or equivalently:

```
% DEFAULT_PROCESS_FILE="SCENARIO_A.lnt"
"TFTP_A.bcg" = divbranching reduction of generation of TFTP_A
```

For scenario A, we obtain the LTS sizes of Table 2. Left columns report LTS sizes before reduction and right columns report LTS sizes after reduction modulo strong bisimulation — which coincides with (div)branching bisimulation since components do not have internal actions. For the 4 other scenarios, Table 3 provides the sizes after strong reduction of the LTSs corresponding to the eight components.

Using an automatically generated interface. Although the component LTSs are not very large, an interface can be used to constrain the largest component, namely TFTP_A, for the sake of illustration. In SCENARIO_A, the refined interface that takes into account constraints imposed on TFTP_A by SND_A, generated using EXP.OPEN, has the same number of states and transitions as SND_A. On-the-fly semi-composition of TFTP_A with this interface yields an LTS containing 119 states and 320 transitions, which avoids generating the LTS containing $30,865$ states and $263,986$ transitions. By being two orders of magnitude smaller, this constrained LTS can be minimized more efficiently. Note, however, that this does not decrease the size of the compound LTS, which remains unchanged as expected.

Generation and minimization of the global compound LTS using root leaf reduction. Once component LTSs are generated and minimized, they can be composed, yielding a non-minimal (but partially reduced) LTS that can then be further minimized modulo an equivalence relation. This is done by root leaf reduction, the simplest strategy proposed in CADP. All these steps can be achieved using a single SVL statement, in which the LTS composition expression is the same as the composition of processes defined in

	non-minimized		strong bisimulation		(div)branching bisimulation	
Scen.	States	Transitions	States	Transitions	States	Transitions
A	1,905,716	7,950,203	1,785,841	7,695,534	1,620,754	7,060,163
B	851,171	3,503,318	827,396	3,548,292	754,112	3,262,499
C	34,003,384	151,810,170	33,430,548	144,216,908	29,278,308	128,562,334
D	39,908,375	177,531,725	33,205,455	152,203,388	29,140,913	135,963,503
E	19,016,593	77,698,491	18,322,082	78,628,709	15,895,904	69,420,304

Table 4: Size of the LTS corresponding to the whole system for each scenario (component LTSs minimized), non-minimized, once minimized modulo strong bisimulation, and once minimized modulo branching or divbranching bisimulation (which coincide on this example)

	Generation		Strong minimization		(Div)branching minimization	
Scenario	time	memory	time	memory	time	memory
A	19	36	33	167	34	170
B	8	23	12	73	15	74
C	670	434	1,060	3,138	1,138	3,204
D	805	503	1,193	3,766	1,237	3,841
E	326	246	501	1,716	493	1,743

Table 5: Generation and minimization: time (in seconds) and memory (in megabytes) for the LTS corresponding to each scenario

file `SCENARIO_A.lnt`:

```
% DEFAULT_PROCESS_FILE="SCENARIO_A.lnt"
"SCENARIO_A.bcg" = root leaf divbranching reduction of
    par
      SEND_A, RECEIVE_A → hide GET_A, PUT_A in TFTP_A end hide
    || SEND_B, RECEIVE_B → hide GET_B, PUT_B in TFTP_B end hide
    || SEND_A, RECEIVE_B → MEDIUM_A
    || SEND_B, RECEIVE_A → MEDIUM_B
    || RECEIVE_A → RECEIVE_A
    || RECEIVE_B → RECEIVE_B
    || SEND_A → SEND_A
    || SEND_B → SEND_B
    end par
```

Table 4 provides LTS sizes after composition, non-minimized and minimized modulo strong, branching, and divbranching bisimulation. Note that branching and divbranching bisimulations coincide, because the global compound LTS does not contain cycles of internal transitions. LTS reductions are not dramatic here, essentially because only a few labels are hidden. Table 5 provides the time and memory used to generate the global compound LTS from the component LTSs, and to minimize it modulo strong bisimulation and (div)branching bisimulation.

Compositional generation of the state space using other strategies. More elaborate strategies (namely node reduction and smart reduction) consist in composing the minimized LTSs in a certain order, and minimizing the intermediate LTSs after hiding synchronized labels. For each of these strategies, Table 6 reports both the memory peak and the size of the largest LTS generated for scenario *A*. These strategies can be easily implemented by replacing the `root leaf` keywords in the above SVL script by either `node` or `smart`. Both node and smart reductions are good strategies for this example. The largest LTS generated by smart reduction has the same size as for node reduction, although the composition order differs slightly. In both cases, memory peak (156 megabytes instead of one gigabyte) and largest intermediate LTS size (less

	Largest LTS size		Memory peak
	States	Trans.	(megabytes)
root leaf reduction	1,905,716	7,950,203	170
node reduction	1,783,372	7,827,859	156
smart reduction	1,783,372	7,827,859	156

Table 6: Compositional strategies for Scenario A (branching bisimulation)

Scenario	Largest LTS	Time	Memory
A	1,810,925	50	158
B	811,893	25	70
C	32,574,552	1,605	4,492
D	38,567,594	1,907	996
E	17,986,105	755	1,812

Table 7: Smart divbranching reduction (without label hiding) results for the TFTP/UDP case study: largest LTS sizes (in the number of states), CPU time (in seconds), and memory peak (in megabytes)

than 2 million states instead of more than 12 millions) are about six times lower than for direct generation. Table 7 reports largest LTS size (in the number of states), CPU time (in seconds), and peak of memory (in megabytes) used to minimize the LTSs corresponding to the five scenarios modulo divbranching equivalence using smart reduction.

7.2 Model checking

We consider the MCL properties named $A01$ to $A28$, studied in [62], as well as an additional alternation-2 fairness property $A29$ not checked in [62]. These properties are reported in Appendix A (page 55).

Standard model checking. We first model check the properties on the BCG graphs corresponding to the five scenarios, minimized modulo divbranching equivalence using the methods presented in Section 7.1. This is done using the tool BCG_OPEN and the model checker EVALUATOR 3 [104] of CADP. These tools are automatically invoked by SVL script fragments of the following form, where in the formula part (i.e., between symbols "$\models$" and ";"), double quotes enclose character strings denoting a label, such as "REINIT_A", and single quotes enclose regular expressions denoting a label, such as 'RECEIVE_A.*'.

```
property A06
    An internal error must cause the transfer to abort. The following formula
    ensures that there is no sequence of transitions in which a reception or a
    transmission can occur after an internal error but before a reinitialisation
    and/or the transmission of an error.
is
    "SCENARIO.bcg" ⊨
        [
            true* .
            "INTERNAL_ERROR_A" .
            not ("REINIT_A" or "SEND_A !ERROR")* .
            'RECEIVE_A.*' or ('SEND_A.*' and not "SEND_A !ERROR")
        ] false;
    expected true
end property
```

Table 8 reports time and memory results for model checking. Some properties being irrelevant to some scenarios (e.g., they concern a read or write operation absent in the corresponding scenario), they are not

	Scenario A		Scenario B		Scenario C		Scenario D		Scenario E	
Prop	time	memory	time	memory	time	memory	time	memory	time	memory
$A01$	13	106	5	49	700	1,803	714	1,807	330	946
$A02$	16	110	5	50	840	1,900	867	1,908	386	4
$A03$	11	100	4	45	582	1,700	599	1,714	257	877
$A04$	13	106	5	49	700	1,803	711	1,807	330	946
$A05$	3	34	1	16	52	663	55	693	28	361
$A06$	12	101	4	47	635	1,734	661	1,755	283	898
$A07$	12	101	4	47	633	1,734	658	1,755	282	898
$A08$	11	100	4	46	587	1,704	603	1,718	260	880
$A09a$							662	1,777	309	927
$A09b$					692	1,805				
$A10$					1,018	1,998			410	1,015
$A11$					884	1,924	1,432	2,239	425	1,032
$A12$					69	861	57	710	29	393
$A13$							1,169	2,242	511	1,176
$A14$	30	135			1,488	2,301			636	1,218
$A15$			6	46	67	827	55	715	314	921
$A16$									39	427
$A17$					171	941	133	923		
$A18$			4	46	86	856	75	808	265	877
$A19$			32	96	3,789	3,441	4,925	4,020	1,262	1,687
$A20$	3	40			143	999			43	465
$A21$	86	188			2,303	2,782			1,094	1,571
$A22$			1	16			321	1,191	45	441
$A23$			22	88			3,062	3,239	1,157	1,660
$A24$	3	34			120	859				
$A25$	79	192			2,718	3,028				
$A26$	13	105			651	1,760			306	921
$A27$	20	105			1,158	2,091			503	1,112
$A28$			7	56	1,259	2,147	1,133	2,087	495	1,088
$A29$	13	106	5	49	698	1,803	714	1,807	331	946

Table 8: Standard model checking results for the TFTP/UDP case study: CPU time (in seconds) and memory (in megabytes)

checked, which explains the shaded cells. As can be seen from the results, a TRUE/FALSE verdict is returned for all verifications, because the model on which they are applied is not too large. However, some of them use much time and memory, e.g., model checking property $A19$ on scenario D takes 1 hour and 20 minutes and uses more than 4 gigabytes of memory.

On-the-fly model checking. We then model check the properties on the fly, i.e., without generating the compound LTS but directly on the composition expression, the transitions being fired on demand by the model checker. This is done still by using EVALUATOR 3, but replacing BCG_OPEN by EXP.OPEN. These tools are automatically invoked using an SVL fragment similar to the above, but replacing "SCENARIO.bcg" by "SCENARIO.exp", which stores the corresponding composition expression. The time and memory results are reported in Table 9. Note that most often, on-the-fly model checking takes more time and memory than standard model checking, because it includes the resources needed by EXP.OPEN to compute fireable transitions, which is more expensive than enumerating the already computed transitions of a BCG graph. However, in a few cases, on-the-fly model checking is faster and uses less memory, in particular when the portion of the LTS to be explored is smaller than the entire compound LTS. For instance, the time and memory used to check property $A15$ on scenario D decreases from 55 seconds and 715 megabytes using standard model checking down to 15 seconds and 156 megabytes using on-the-fly model checking.

Model checking using maximal hiding and smart reduction. Our next experiment concerns the combination of maximal hiding with smart reduction. For formulas $A08$, $A09a$, $A09b$, $A14$, and $A16$, the maximal hiding set is empty, and hence we do not consider them in this experiment. For each of the relevant formulas and for each scenario, the maximal hiding set is generated (using an ad-hoc option of EVALUATOR 4) and then used to minimize the system modulo divbranching bisimulation with smart reduction, and the property is verified on the minimized LTS using EVALUATOR. Maximal hiding is not yet integrated in the SVL scripting language, but this is planned on a short term. Table 10 reports the time

	Scenario A		Scenario B		Scenario C		Scenario D		Scenario E	
Prop	time	memory	time	memory	time	memory	time	memory	time	memory
$A01$	28	199	10	89	1,324	2,947	1,590	3,351	772	1,530
$A02$	31	207	12	93	1,640	3,156	2,010	3,631	883	1,612
$A03$	22	182	8	80	1,210	2,737	1,365	3,162	668	1,386
$A04$	26	199	10	89	1,400	2,947	1,598	3,351	770	1,530
$A05$	1	10	1	7	1	7	1	7	1	10
$A06$	23	187	9	85	1,306	2,808	1,540	3,249	667	1,428
$A07$	23	187	9	85	1,299	2,808	1,687	3,249	674	1,428
$A08$	22	186	8	80	1,220	2,745	1,620	3,170	625	1,390
$A09a$							1,679	3,290	695	1,488
$A09b$					1,415	2,955				
$A10$					2,112	3,354			929	1,674
$A11$					1,722	3,206	3,583	4,444	997	1,711
$A12$					76	620	8	133	6	101
$A13$							3,297	4,499	1,446	2,094
$A14$	54	267			2,681	3,988			1,443	2,107
$A15$			11	118	55	521	15	156	705	1,524
$A16$									40	186
$A17$					315	667	217	569		
$A18$			9	85	86	476	35	255	599	1,391
$A19$			53	207	6,159	6,352	9,393	8,753	2,697	3,104
$A20$	1	31			224	837			39	261
$A21$	131	374			4,004	4,958			2,293	2,817
$A22$			1	35			147	427	43	191
$A23$			39	170			5,605	6,909	2,345	3,039
$A24$	1	41			148	427				
$A25$	133	391			4,163	5,480				
$A26$	25	195			1,383	2,857			687	1,477
$A27$	38	228			2,323	3,534			1,196	1,871
$A28$			15	102	2,538	3,654	2,615	4,032	1,277	1,821
$A29$	26	198	11	88	1,524	2,942	1,738	3,350	700	1,525

Table 9: On-the-fly model checking results for the TFTP/UDP case study: CPU time (in seconds) and memory (in megabytes)

(in seconds) and memory peak (in megabytes). Table 11 reports the largest intermediate LTS size (in the number of states) reached during each experiment. The percentages (columns labelled with %) indicate the ratio between the size of the largest LTS generated during this experiment and the size given in Table 7 of the largest LTS generated during smart divbranching reduction without label hiding. All experiments show a gain, sometimes impressive. For instance, the number of states of the largest LTS generated to check property $A12$ on scenario D represents only 3 % of the maximum number of states generated by smart divbranching reduction without label hiding. The gain in CPU time and memory peak is also significant.

Partial model checking. As the last experiment, we apply the partial model checking approach described in Section 5.2. Table 12 gives, for each scenario and each property, the time in seconds and the peak of memory in megabytes used by partial model checking. The symbol "$\star$" corresponds to verifications that either take too long (exceeding a 12 hour limit) and/or use too much memory (more than 16 gigabytes). Note that most of the time and memory are used by formula simplifications, as compared to the rather low complexity of the synchronous product operation used for quotienting. Partial model checking is not yet integrated in the SVL scripting language, so that tools have to be invoked manually at the moment.

These results confirm that partial model checking may be much more efficient (up to several hundred times less memory in this example) than both standard and on-the-fly model checking. This is particularly true for some formulas having the form "$[R]\,\mathsf{false}$" and "$\langle R\rangle\,\mathsf{true}$", where R is a regular expression; Such formulas denote the absence (respectively the existence) of a sequence of transitions matching R. In these examples, the quotient evaluates to true (in the case of formulas of the form "$[R]\,\mathsf{false}$") or false (in the case of formulas of the form "$\langle R\rangle\,\mathsf{true}$") before all component LTSs have been taken into account in the quotient, because it is possible to determine that none of the paths possible in the parts of the system already explored may yield a path satisfying R in the compound system. As an illustration, Table 13 gives details on the verification of formula $A09b$ on Scenario C. This formula has the form "$[R]\,\mathsf{false}$" and evaluates to true after the partial model checking steps reported in Table 13.

	Scenario A		Scenario B		Scenario C		Scenario D		Scenario E	
Prop	time	mem.	time	mem.	time	mem.	time	mem.	time	mem.
$A01$	16	101	14	39	247	1,381	301	1,565	134	701
$A02$	16	75	10	35	245	1,036	165	765	136	570
$A03$	6	16	6	6	70	299	160	74	38	159
$A04$	16	100	10	42	247	1,231	157	996	133	724
$A05$	11	52	10	20	130	678	83	481	71	377
$A06$	14	52	12	20	183	690	116	488	100	390
$A07$	14	52	12	20	183	690	116	488	100	390
$A10$					272	857			84	588
$A11$					153	996	227	372	129	439
$A12$					71	306	169	78	39	164
$A13$							125	537	103	439
$A15$			14	45	304	1,199	325	1,334	166	575
$A17$					125	744	310	489		
$A18$			9	25	159	1,418	136	508	91	1,040
$A19$			10	26	303	1,388	243	451	128	1,056
$A20$	10	70			166	1,973			83	633
$A21$	13	70			237	1,973			122	633
$A22$			11	26			229	513	93	653
$A23$			12	26			267	513	117	653
$A24$	12	84			164	1,110				
$A25$	15	84			211	1,110				
$A26$	13	95			175	1,405			97	762
$A27$	17	66			1,001	4,492			626	2,262
$A28$			7	8	1,186	3,953	270	771	454	1,365
$A29$	15	100	9	42	247	1,231	157	996	133	724

Table 10: Maximal hiding results for the TFTP/UDP case study: time (in seconds) and memory peak (in megabytes)

	Scenario A		Scenario B		Scenario C		Scenario D		Scenario E	
	size	%	size	%	size	%	size	%	size	%
$A01$	440,679	24 %	152,392	18 %	7,446,443	22 %	9,910,844	25 %	4,207,603	23 %
$A02$	436,352	24 %	213,665	26 %	7,446,443	22 %	4,379,978	11 %	4,207,603	23 %
$A03$	293,563	16 %	29,192	3 %	4,793,248	14 %	1,260,400	3 %	2,694,634	14 %
$A04$	436,352	24 %	213,665	26 %	7,446,443	22 %	4,379,978	11 %	4,207,603	23 %
$A05$	285,256	15 %	103,739	12 %	4,845,485	14 %	2,798,880	7 %	2,740,106	15 %
$A06$	341,953	18 %	122,465	15 %	5,561,875	17 %	3,104,457	8 %	3,174,438	17 %
$A07$	341,953	18 %	122,465	15 %	5,561,875	17 %	3,104,457	8 %	3,174,438	17 %
$A10$					4,793,248	14 %			2,694,634	14 %
$A11$					4,793,248	14 %	1,260,400	3 %	2,694,634	14 %
$A12$					4,819,226	14 %	1,319,036	3 %	2,740,106	15 %
$A13$							3,201,847	8 %	3,219,910	17 %
$A15$			305,590	37 %	8,678,547	26 %	8,591,743	22 %	4,097,150	22 %
$A17$					3,728,837	11 %	2,333,189	6 %		
$A18$			94,094	11 %	4,793,248	14 %	1,863,667	4 %	2,694,634	14 %
$A19$			96,945	11 %	4,793,248	14 %	1,863,667	4 %	2,694,634	14 %
$A20$	293,563	16 %			4,793,248	14 %			2,694,634	14 %
$A21$	293,563	16 %			4,793,248	14 %			2,694,634	14 %
$A22$			115,924	14 %			1,474,645	3 %	3,174,438	17 %
$A23$			115,924	14 %			1,474,645	3 %	3,174,438	17 %
$A24$	351,927	19 %			5,561,875	17 %				
$A25$	351,927	19 %			5,561,875	17 %				
$A26$	351,927	19 %			5,561,875	17 %			3,174,438	17 %
$A27$	293,563	16 %			4,793,248	14 %			2,694,634	14 %
$A28$			37,012	4 %	4,793,248	14 %	1,721,736	4 %	2,694,634	14 %
$A29$	436,352	24 %	213,665	26 %	7,446,443	22 %	4,379,978	11 %	4,207,603	23 %

Table 11: Maximal hiding results for the TFTP/UDP case study: largest LTS size (in the number of states) and percentage w.r.t. the largest LTS size generated by smart reduction without label hiding (see Table 7)

The fairness formula $A29$ also evaluates more efficiently using partial model checking. This formula, specified in MCL as "$\langle \mathsf{true}^*.A_1.(\neg(A_1 \vee A_2))^*.A_3.(\neg A_1)^*.A_2\rangle$ @", denotes the existence of a cyclic sequence of transitions matching the regular expression "$\mathsf{true}^*.A_1.(\neg(A_1 \vee A_2))^*.A_3.(\neg A_1)^*.A_2$", where A_1, A_2, and A_3 are particular actions. It evaluates to false on all scenarios. The first steps of partial model checking for this formula on Scenario E are detailed in Table 14.

	Scenario A		Scenario B		Scenario C		Scenario D		Scenario E	
Prop	time	memory	time	memory	time	memory	time	memory	time	memory
$A01$	2	6	3	6	3	24	2	27	3	23
$A02$	3	6	3	6	6	25	7	28	6	10
$A03$	1	6	1	6	1	6	1	6	1	6
$A04$	3	6	3	6	3	6	3	29	3	7
$A05$	5	6	5	6	5	6	5	6	5	10
$A06$	3	6	3	6	3	6	3	7	3	6
$A07$	3	6	3	6	3	6	3	6	3	6
$A08$	3	6	3	6	3	6	3	6	3	6
$A09a$							7	28	3	6
$A09b$					8	6				
$A10$					3	6			3	6
$A11$					3	6	1	7	3	6
$A12$					$\star$	$\star$	$\star$	$\star$	$\star$	$\star$
$A13$							$\star$	$\star$	$\star$	$\star$
$A14$	3	6			3	23			3	15
$A15$			5	15	$\star$	$\star$	$\star$	$\star$	7	59
$A16$									1	8
$A17$					$\star$	$\star$	$\star$	$\star$		
$A18$			1	6	7	11	3	6	1	6
$A19$			3	6	3	90	3	13	3	55
$A20$	3	9			6	21			6	25
$A21$	3	6			3	25			3	25
$A22$			12	7			2,712	1,271	1,007	650
$A23$			3	6			9	9	6	40
$A24$	13	9			3,189	1,786				
$A25$	3	6			6	40				
$A26$	3	6			3	15			3	10
$A27$	3	6			3	6			3	6
$A28$			3	6	3	22	3	6	3	6
$A29$	2	7	2	7	6	9	3	7	5	9

Table 12: Partial model checking results for the TFTP/UDP case study: time (in seconds) and memory peak (in megabytes)

Step	States	Transitions
Initial formula graph	13	62
Simplification & reduction	7	56
Quotient wrt. TFTP_A	125	1,964
Simplification & reduction	60	1,512
Quotient wrt. TFTP_B	9,166	69,490
Simplification & reduction	5,308	50,799
Quotient wrt. MEDIUM_B (encodes true)	2	1

Table 13: Successive steps for the partial model checking of property $A09b$

Step	States	Transitions
Initial formula graph	19	151
Simplification & reduction	7	139
Quotient wrt. TFTP_B	903	20,388
Simplification & reduction	896	20,099
Quotient wrt. TFTP_A	26,369	197,480
Simplification & reduction (encodes false)	1	0

Table 14: Successive steps for the partial model checking of property $A29$

In a few other cases, partial model checking leads to combinatorial explosion (properties $A12$, $A13$, $A15$, and $A17$) while other model checking techniques perform efficiently. We illustrate this with the verification

Step	Time	Memory	States	Transitions
Initial formula graph			8	56
Simplification	0	4	8	56
Reduction	0	66	4	52
Quotient wrt. TFTP_A	0	66	210	5,687
Simplification	0	4	136	3,665
Reduction	0	66	134	3,587
Quotient wrt. TFTP_B	0	66	21,172	168,172
Simplification	0	6	21,015	168,172
Reduction	1	66	14,042	119,789
Quotient wrt. MEDIUM_B	14	66	1,648,096	10,327,294
Simplification	35	267	1,648,089	10,327,294
Reduction	72	234	1,551,338	14,773,975
Quotient wrt. MEDIUM_A	686	540	40,572,824	229,050,227

...

Table 15: First successive steps for the partial model checking of property $A12$ (time in seconds and memory in megabytes)

of formula $A12$ on scenario C. This formula has the form "$\langle R \rangle \mathsf{true}$" and evaluates to true. The first steps of partial model checking are detailed in Table 15, which provides the time and memory used to complete each step. The reduction step includes both pre-reduction modulo $\tau^*.a$ equivalence (i.e., elimination of τ-transitions) and minimization modulo strong bisimulation. Note that this may produce a graph that is not minimal in the number of transitions, although always minimal in the number of states. This explosion seems inherent to the structure of the system and the formula, as intermediate quotients need to capture a large part of the behaviour before the truth value of the formula can be determined. These experiments show that partial model checking is complementary to, but does not replace, other model checking techniques.

8 Conclusion

This report has given a comprehensive panorama of compositional verification techniques that can be used in an asynchronous, action-based modeling setting. Although such techniques are inherently complex and require significant implementation efforts, they often achieve impressive state-space reduction, successfully tackling verification problems that could not be addressed otherwise. All the approaches presented in this report have been implemented, a task for which the CADP toolbox proved to be a suitable experimentation platform.

It is worth noticing that most of the results presented here are firmly rooted in concurrency theory. They build upon major advances of concurrency theory, namely: (1) formally-defined parallel composition operators that use transition labels as the criterion to decide when concurrent processes must synchronize and when they can interleave; (2) formally-defined abstraction operators that can hide or rename transition labels; (3) preorder and equivalence relations, such as bisimulations, that can be used to compare the behaviors of system models, that can be efficiently computed, and that satisfy the key property of congruence, meaning that such relations are "compatible" with the parallel composition operators; and (4) the existence of modal μ-calculus and action-based temporal logics that, under certain conditions, are preserved by these equivalence relations and exhibit suitable properties with respect to parallel composition operators. Although implementing compositional approaches is intrinsically difficult, all these results give concurrency theory a unique advantage to fight state-explosion issues on the long term.

Regarding future work, there are two grand challenges to be met: (i) as the size of models under verification steadily increases, the efficiency of compositional approaches (i.e., the state-space reductions that they allow) should progress as well, and (ii) automation of compositional verification should increase, making it easily available to non-expert users. The latter point is particularly difficult, as there are multiple approaches

to compositional verification, as evidenced in the present report. Given the multiple sources of complexity, not a single compositional approach is sufficient in itself, as complexity can only be mastered by combining different approaches and strategies. In this respect, action-based approaches to compositional verification are probably easier to automate than state-based ones, because the former benefit from the existence of equivalence relations that preserve full classes of properties, whereas the latter must take into account (especially with the *assume-guarantee* paradigm) each particular property of the system under verification.

Among the open issues, one may wonder whether the principles of semi-composition and interfaces (as described in Section 3) could be transposed to formula-dependent approaches (as described in Section 4). Such a combination would be promising, but remains to be investigated. Also, it is worth noticing that all the approaches presented in the present report are based upon the decomposition of a system into concurrent components, according to parallel composition operators used to model that system; orthogonal decompositions could also be considered, in particular those taking into account data structures, especially for systems that are architected around large amounts of shared data.

References

[1] L. Aceto, W. Fokkink, and C. Verhoef. *Structural Operational Semantics*. In *Handbook of Process Algebra*, chapter 3, pages 197–292. North-Holland, 2001.

[2] K. Ajami, S. Haddad, and J.-M. Ilié. Exploiting Symmetry in Linear Time Temporal Logic Model Checking: One Step Beyond. In *Proceedings of Tools and Algorithms for Construction and Analysis of Systems TACAS'98*, volume 1384 of *Lecture Notes in Computer Science*, pages 52–67. Springer, 1998.

[3] H. R. Andersen. Model checking and boolean graphs. *Theoretical Computer Science*, 126(1):3–30, 1994.

[4] H. R. Andersen. Partial Model Checking. In *Proceedings of Logic in Computer Science LICS'95*, pages 398–407. IEEE, 1995.

[5] H. R. Andersen and J. Lind-Nielsen. Partial Model Checking of Modal Equations: A Survey. *Journal on Software Tools for Technology Transfer (STTT)*, 2:242–259, 1999.

[6] H. R. Andersen, J. Staunstrup, and N. Maretti. A Comparison of Modular Verification Techniques. In *Proceedings of CAAP/FASE'97*, volume 1214 of *Lecture Notes in Computer Science*. Springer, 1997.

[7] H. R. Andersen, J. Staunstrup, and N. Maretti. Partial Model Checking with ROBDDs. In *Proceedings of Tools and Algorithms for Construction and Analysis of Systems TACAS'97*, volume 1217 of *Lecture Notes in Computer Science*. Springer, 1997.

[8] H. R. Andersen and G. Winskel. Compositional Checking of Satisfaction. In *Proceedings of Computer Aided Verification CAV'91*, volume 575 of *Lecture Notes in Computer Science*, pages 24–36. Springer, 1991.

[9] R. Armoni, L. Fix, A. Flaisher, R. Gerth, B. Ginsburg, T. Kanza, A. Landver, S. Mador-Haim, E. Singerman, A. Tiemeyer, M. Y. Vardi, and Y. Zbar. The ForSpec Temporal Logic: A New Temporal Property-Specification Language. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems TACAS'02*, volume 2280 of *Lecture Notes in Computer Science*, pages 296–211. Springer, 2002.

[10] P. J. Armstrong, M. Goldsmith, G. Lowe, J. Ouaknine, H. Palikareva, A. W. Roscoe, and J. Worrell. Recent Developments in FDR. In *Proceedings of Computer Aided Verification CAV'12*, volume 7358 of *Lecture Notes in Computer Science*, pages 699–704. Springer, 2012.

[11] A. Arnold. MEC: A System for Constructing and Analysing Transition Systems. In *Proceedings of Automatic Verification Methods for Finite State Systems CAV'89*, volume 407 of *Lecture Notes in Computer Science*, pages 117–132. Springer, 1989.

[12] C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2008.

[13] R. Barbuti, N. de Francesco, A. Santone, and G. Vaglini. Selective Mu-Calculus and Formula-Based Equivalence of Transition Systems. *Journal of Computer and System Sciences*, 59(3):537–556, 1999.

[14] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-Based Runtime Verification. In *Proceedings of Verification, Model Checking, and Abstract Interpretation VMCAI'04*, volume 2937 of *Lecture Notes in Computer Science*, pages 44–57. Springer, 2004.

[15] S. Basu and C.R. Ramakrishnan. Compositional Analysis for Verification of Parameterized Systems. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems TACAS'03*, volume 2619 of *Lecture Notes in Computer Science*, pages 315–330. Springer, 2003.

[16] I. Beer, S. Ben-David, and A. Landver. On-the-Fly Model Checking of RCTL Formulas. In *Proceedings of Computer Aided Verification CAV'98*, volume 1427 of *Lecture Notes in Computer Science*, pages 184–194. Springer, 1998.

[17] B. Berard and F. Laroussinie. Verification compositionnelle des p-automates. Technical Report Lot 4.1, Réseau National des Technologies Logicielles, projet AVERROES, 2003.

[18] J. A. Bergstra and J. W. Klop. Algebra of Communicating Processes with Abstraction. *Theoretical Computer Science*, 37:77–121, 1985.

[19] J. A. Bergstra, A. Ponse, and S. A. Smolka, editors. *Handbook of Process Algebra*. Elsevier, 2001.

[20] S. Blom and S. Orzan. Distributed state space minimization. *Software Tools for Technology Transfer*, 7(3):280–291, 2005.

[21] S. Blom, J. van de Pol, and M. Weber. LTSmin: Distributed and Symbolic Reachability. In *Proceedings of Computer Aided Verification CAV'10*, volume 6174 of *Lecture Notes in Computer Science*, pages 354–359. Springer, 2010.

[22] N. Bodentien, J. Vestergaard, J. Friis, K. Kristoffersen, and K. G. Larsen. Verification of State/Event Systems by Quotienting. Technical Report RS-99-41, Basic Research in Computer Science, 1999.

[23] A. Bouajjani, J.-C. Fernandez, S. Graf, C. Rodríguez, and J. Sifakis. Safety for Branching Time Semantics. In *Proceedings of ICALP*. Springer, 1991.

[24] A. Bouali, A. Ressouche, V. Roy, and R. de Simone. The Fc2Tools set: a Toolset for the Verification of Concurrent Systems. In *Proceedings of Computer-Aided Verification CAV'96*, volume 1102 of *Lecture Notes in Computer Science*. Springer, 1996.

[25] J. C. Bradfield and C. Stirling. *Modal Logics and Mu-Calculi: An Introduction*. In *Handbook of Process Algebra*, chapter 4, pages 293–330. Elsevier, 2001.

[26] 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.

[27] F. Cassez and F. Laroussinie. Model-checking for hybrid systems by quotienting and constraints solving. In *Proceedings of Computer Aided Verification CAV'00*, volume 1855 of *Lecture Notes in Computer Science*. Springer, 2000.

[28] D. Champelovier, X. Clerc, H. Garavel, Y. Guerte, C. McKinty, V. Powazny, F. Lang, W. Serwe, and G. Smeding. Reference Manual of the LNT to LOTOS Translator (Version 6.1). INRIA/VASY and INRIA/CONVECS, 131 pages, 2014.

[29] 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 *Proceedings of Formal Description Techniques for Distributed Systems and Communication Protocols / Protocol Specification, Testing, and Verification FORTE/PSTV'96*, pages 435–450. IFIP, Chapman & Hall, 1996.

[30] K. H. Cheung. *Compositional Analysis of Complex Distributed Systems*. PhD thesis, Department of Computer Science, Hong Kong University of Science and Technology, Hong Kong, 1998.

[31] S. C. Cheung and J. Kramer. Enhancing Compositional Reachability Analysis with Context Constraints. In *Proceedings of Foundations of Software Engineering*, pages 115–125. ACM Press, 1993.

[32] S. C. Cheung and J. Kramer. Compositional Reachability Analysis of Finite-State Distributed Systems with User-Specified Constraints. In *Proceedings of Foundations of Software Engineering*, pages 140–150. ACM Press, 1995.

[33] S. C. Cheung and J. Kramer. Context Constraints for Compositional Reachability. *ACM Transactions on Software Engineering Methodology TOSEM*, 5(4):334–377, 1996.

[34] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[35] E. M. Clarke, E. A. Emerson, S. Jha, and A. Prasad Sistla. Symmetry Reductions in Model Checking. In *Proceedings of Computer Aided Verification CAV'98*, volume 1427 of *Lecture Notes in Computer Science*, pages 147–158. Springer, 1998.

[36] E. M. Clarke, S. Jha, R. Enders, and T. Filkorn. Exploiting Symmetry in Temporal Logic Model Checking. *Formal Methods in System Design*, 9(1/2):77–104, 1996.

[37] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench. In *Proceedings of Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 24–37. Springer, 1989.

[38] R. Cleaveland and B. Steffen. A Linear-Time Model-Checking Algorithm for the Alternation-Free Modal Mu-Calculus. *Formal Methods in System Design*, 2(2):121–147, 1993.

[39] N. Coste, H. Garavel, H. Hermanns, F. Lang, R. Mateescu, and W. Serwe. Ten Years of Performance Evaluation for Concurrent Systems Using CADP. In *Proceedings of Leveraging Applications of Formal Methods, Verification and Validation ISoLA'10*, volume 6416 of *Lecture Notes in Computer Science*, pages 128–142. Springer, 2010.

[40] P. Crouzen and H. Hermanns. Aggregation Ordering for Massively Parallel Compositional Models. In *Proceedings of Application of Concurrency to System Design ACSD'10*. IEEE, 2010.

[41] P. Crouzen and F. Lang. Smart Reduction. In *Proceedings of Fundamental Approaches to Software Engineering FASE'2011*, volume 6603 of *Lecture Notes in Computer Science*, pages 111–126. Springer, 2011.

[42] M. Dam. Model Checking Mobile Processes. In *Proceedings of Concurrency Theory CONCUR'93*, volume 715 of *Lecture Notes in Computer Science*, pages 22–36. Springer, 1993.

[43] X. Du, S. A. Smolka, and R. Cleaveland. Local Model Checking and Protocol Analysis. *Journal on Software Tools for Technology Transfer STTT*, 2(3):219–241, 1999.

[44] E. A. Emerson and E. M. Clarke. Using Branching Time Logic to Synthesize Synchronization Skeletons. *Science of Computer Programming*, 2:241–266, 1982.

[45] E. A. Emerson and J. Y. Halpern. "Sometimes" and "Not Never" Revisited: On Branching versus Linear Time Temporal Logic. *Journal of the ACM*, 33(1):151–178, 1986.

[46] E. A. Emerson and C.-L. Lei. Efficient Model Checking in Fragments of the Propositional Mu-Calculus. In *Proceedings of Logic in Computer Science LICS'86*, pages 267–278, 1986.

[47] A. Fantechi, S. Gnesi, and G. Ristori. From ACTL to Mu-Calculus. In *Proceedings of Theory and Practice in Verification*, pages 3–10. IEI-CNR, 1992.

[48] A. Fantechi, S. Gnesi, and G. Ristori. Model Checking for Action-Based Logics. *Formal Methods in System Design*, 4:187–203, 1994.

[49] J.-C. Fernandez. *ALDEBARAN : un système de vérification par réduction de processus communicants*. Thèse de Doctorat, Université Joseph Fourier (Grenoble), 1988.

[50] J.-C. Fernandez, H. Garavel, L. Mounier, A. Rasse, C. Rodríguez, and J. Sifakis. A Toolbox for the Verification of LOTOS Programs. In *Proceedings of Software Engineering ICSE'14*, pages 246–259. ACM, 1992.

[51] J.-C. Fernandez and L. Mounier. "On the Fly" Verification of Behavioural Equivalences and Preorders. In *Proceedings of Computer-Aided Verification CAV'91*, volume 575 of *Lecture Notes in Computer Science*, pages 181–191. Springer, 1991.

[52] J.-C. Fernandez and L. Mounier. A Tool Set for Deciding Behavioral Equivalences. In *Proceedings of CONCUR'91*, 1991.

[53] M. J. Fischer and R. E. Ladner. Propositional Dynamic Logic of Regular Programs. *Journal of Computer and System Sciences*, 18(2):194–211, 1979.

[54] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of Principles of Programming Languages POPL'05*. ACM Press, 2005.

[55] H. Garavel. Compilation of LOTOS Abstract Data Types. In *Proceedings of Formal Description Techniques FORTE'89*, pages 147–162. North-Holland, 1989.

[56] H. Garavel. OPEN/CÆSAR: An Open Software Architecture for Verification, Simulation, and Testing. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems TACAS'98*, volume 1384 of *Lecture Notes in Computer Science*, pages 68–84. Springer, 1998.

[57] H. Garavel and F. Lang. SVL: a Scripting Language for Compositional Verification. In *Proceedings of Formal Techniques for Networked and Distributed Systems FORTE'2001*, pages 377–392. IFIP, Kluwer Academic Publishers, 2001.

[58] H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems TACAS'2011*, volume 6605 of *Lecture Notes in Computer Science*, pages 372–387, 2011.

[59] H. Garavel, G. Salaün, and W. Serwe. On the Semantics of Communicating Hardware Processes and their Translation into LOTOS for the Verification of Asynchronous Circuits with CADP. *Science of Computer Programming*, 74(3):100–127, 2009.

[60] H. Garavel and J. Sifakis. Compilation and Verification of LOTOS Specifications. In *Proceedings of Protocol Specification, Testing and Verification PSTV'90*, pages 379–394. IFIP, North-Holland, 1990.

[61] H. Garavel and M. Sighireanu. A Graphical Parallel Composition Operator for Process Algebras. In *Proceedings of Formal Description Techniques for Distributed Systems and Communication Protocols / Protocol Specification, Testing, and Verification FORTE/PSTV'99*, pages 185–202. IFIP, Kluwer Academic Publishers, 1999.

[62] H. Garavel and D. Thivolle. Verification of GALS Systems by Combining Synchronous Languages and Process Calculi. In *Proceedings of Model Checking Software SPIN'2009*, volume 5578 of *Lecture Notes in Computer Science*, pages 241–260. Springer, 2009.

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

[64] D. Giannakopoulou and J. Magee. Fluent Model Checking for Event-Based Systems. In *Proceedings of Foundations of Software Engineering ESEC/FSE'2003*, pages 257–266. ACM, 2003.

[65] T. Gibson-Robinson, P. J. Armstrong, A. Boulgakov, and A. W. Roscoe. FDR3 – A Modern Refinement Checker for CSP. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems TACAS'14*, volume 8413 of *Lecture Notes in Computer Science*, pages 187–201. Springer, 2014.

[66] P. Godefroid. Using Partial Orders to Improve Automatic Verification Methods. In *Proceedings of Computer-Aided Verification CAV'90*, volume 3 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 321–340. AMS-ACM, 1990.

[67] P. Godefroid and P. Wolper. A Partial Approach to Model Checking. In *Proceedings of Logic in Computer Science LICS'91*. IEEE, 1991.

[68] P. Godefroid and P. Wolper. Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties. In *Proceedings of Computer-Aided Verification CAV'91*, volume 575 of *Lecture Notes in Computer Science*. Springer, 1991.

[69] S. Graf and B. Steffen. Compositional Minimization of Finite State Systems. In *Proceedings of Computer-Aided Verification CAV'90*, volume 531 of *Lecture Notes in Computer Science*, pages 186–196. Springer, 1990.

[70] 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.

[71] J. F. Groote, T. W. D. M. Kouters, and A. A. H. Osaiweran. Specification guidelines to avoid the state space explosion problem. *Journal on Software Testing, Verification and Reliability*, 2014. Published online.

[72] J. F. Groote and A. Ponse. The Syntax and Semantics of μCRL. In *Proceedings of Workshop on the Algebra of Communicating Processes ACP'94*, Workshops in Computing Series, pages 26–62. Springer, 1995.

[73] J. F. Groote and F. Vaandrager. An Efficient Algorithm for Branching Bisimulation and Stuttering Equivalence. In *Proceedings of ICALP'90*, volume 443 of *Lecture Notes in Computer Science*, pages 626–638. Springer, 1990.

[74] N. Halbwachs. *Synchronous Pogramming of Reactive Systems*. Kluwer, 1993.

[75] K. Hamaguchi, H. Hiraishi, and S. Yajima. Branching Time Regular Temporal Logic for Model Checking with Linear Time Complexity. In *Proceedings of Computer Aided Verification CAV'90*, volume 531 of *Lecture Notes in Computer Science*, pages 253–262. Springer, 1990.

[76] M. Hennessy. *The Semantics of Programming Languages: an Elementary Introduction using Structural Operational Semantics*. John Wiley and Sons, 1990.

[77] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978.

[78] C. N. Ip and D. L. Dill. Better Verification Through Symmetry. *Formal Methods in System Design*, 9(1/2):41–75, 1996.

[79] 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, 1989.

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

[81] S. Katz and D. Peled. An Efficient Verification Method for Parallel and Distributed Programs. In *Proceedings of Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *Lecture Notes in Computer Science*, pages 489–507. Springer, 1988.

[82] D. Kozen. Results on the Propositional μ-Calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[83] J.-P. Krimm and L. Mounier. Compositional State Space Generation from LOTOS Programs. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems TACAS'97*, volume 1217 of *Lecture Notes in Computer Science*. Springer, 1997.

[84] F. Lang. Compositional Verification using SVL Scripts. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems TACAS'2002*, volume 2280 of *Lecture Notes in Computer Science*, pages 465–469. Springer, 2002.

[85] F. Lang. EXP.OPEN 2.0: A Flexible Tool Integrating Partial Order, Compositional, and On-the-fly Verification Methods. In *Proceedings of Integrated Formal Methods IFM'2005*, volume 3771 of *Lecture Notes in Computer Science*, pages 70–88. Springer, 2005.

[86] F. Lang. Refined Interfaces for Compositional Verification. In *Proceedings of Formal Techniques for Networked and Distributed Systems FORTE'2006*, volume 4229 of *Lecture Notes in Computer Science*, pages 159–174. Springer, 2006.

[87] F. Lang and R. Mateescu. Partial Model Checking using Networks of Labelled Transition Systems and Boolean Equation Systems. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems TACAS'2012*, volume 7214 of *Lecture Notes in Computer Science*, pages 141–156. Springer, 2012.

[88] F. Lang and R. Mateescu. Partial Model Checking using Networks of Labelled Transition Systems and Boolean Equation Systems. *Logical Methods in Computer Science*, 9(4):1–32, 2013.

[89] F. Lang, G. Salaün, R. Hérilier, J. Kramer, and J. Magee. Translating FSP into LOTOS and Networks of Automata. *Formal Aspects of Computing*, 22(6):681–711, 2010.

[90] F. Laroussinie and K. G. Larsen. Compositional Model Checking of Real Time Systems. In *Proceedings of Concurrency Theory CONCUR'95*, volume 962 of *Lecture Notes in Computer Science*. Springer, 1995.

[91] F. Laroussinie and K. G. Larsen. CMC: A Tool for Compositional Model Checking of Real-Time Systems. In *Proceedings of Formal Description Techniques for Distributed Systems and Communication Protocols / Protocol Specification, Testing and Verification FORTE/PSTV'98*, volume 135 of *IFIP Conference Proceedings*. Kluwer, 1998.

[92] K. G. Larsen. Proof Systems for Hennessy-Milner logic with Recursion. In *Proceedings of Trees in Algebra and Programming CAAP'88*, volume 299 of *Lecture Notes in Computer Science*, pages 215–230. Springer, 1988.

[93] K. G. Larsen, P. Pettersson, and W. Yi. Compositional and Symbolic Model Checking of Real-Time Systems. In *Proceedings of Real-Time Systems*. IEEE, 1995.

[94] J. Magee and J. Kramer. *Concurrency: State Models and Java Programs*. Wiley, 2006 edition, 2006.

[95] J. Malhotra, S. A. Smolka, A. Giacalone, and R. Shapiro. A Tool for Hierarchical Design and Simulation of Concurrent Systems. In *Proceedings of Specification and Verification of Concurrent Systems*, pages 140–152. British Computer Society, 1988.

[96] Z. Manna and A. Pnueli. A Hierarchy of Temporal Properties. In *Proceedings of Principles of Distributed Computing PODC'90*, pages 377–408, 1990.

[97] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*, volume I (Specification). Springer, 1992.

[98] A. J. Martin. Compiling Communicating Processes into Delay-Insensitive VLSI Circuits. *Distributed Computing*, 1(4):226–234, 1986.

[99] F. Martinelli. Symbolic Partial Model Checking for Security Analysis. In *Proceedings of Mathematical Methods, Models, and Architectures for Computer Network Security MMM-ACNS*, volume 2776 of *Lecture Notes in Computer Science*. Springer, 2003.

[100] R. Mateescu. Local Model-Checking of an Alternation-Free Value-Based Modal Mu-Calculus. In *Proceedings ofVerification, Model Checking and Abstract Interpretation VMCAI'98*. University Ca' Foscari of Venice, 1998.

[101] R. Mateescu. Efficient Diagnostic Generation for Boolean Equation Systems. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems TACAS'2000*, volume 1785 of *Lecture Notes in Computer Science*, pages 251–265. Springer, 2000.

[102] R. Mateescu. CAESAR_SOLVE: A Generic Library for On-the-Fly Resolution of Alternation-Free Boolean Equation Systems. *Journal on Software Tools for Technology Transfer STTT*, 8(1):37–56, 2006.

[103] R. Mateescu and G. Salaün. Translating Pi-Calculus into LOTOS NT. In *Proceedings of Integrated Formal Methods IFM'2010*, volume 6396 of *Lecture Notes in Computer Science*, pages 229–244. Springer, 2010.

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

[105] R. Mateescu and D. Thivolle. A Model Checking Language for Concurrent Value-Passing Systems. In *Proceedings of Formal Methods FM'08*, volume 5014 of *Lecture Notes in Computer Science*, pages 148–164. Springer, 2008.

[106] R. Mateescu and A. Wijs. Property-Dependent Reductions Adequate with Divergence-Sensitive Branching Bisimilarity. *Science of Computer Programming*, 96(3):354–376, 2014.

[107] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[108] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[109] R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999.

[110] M. Müller-Olm, D. Schmidt, and B. Steffen. Model-Checking: A Tutorial Introduction. In *Proceedings of Static Analysis Symposium SAS'99*, volume 1694 of *Lecture Notes in Computer Science*, pages 330–354. Springer, 1999.

[111] R. De Nicola, A. Fantechi, S. Gnesi, and G. Ristori. An Action-Based Framework for Verifying Logical and Behavioural Properties of Concurrent Systems. In *Proceedings of Computer Aided Verification CAV'91*, volume 575 of *Lecture Notes in Computer Science*, pages 37–47. Springer, 1991.

[112] R. De Nicola and F. W. Vaandrager. *Action versus State Based Logics for Transition Systems*. In *Semantics of Concurrency*, volume 469 of *Lecture Notes in Computer Science*, pages 407–419. Springer, 1990.

[113] A. A. H. Osaiweran. *Formal Development of Control Software in the Medical Systems Domain*. PhD thesis, Eindhoven University of Technology, 2012.

[114] W. T. Overman and S. D. Crocker. Verification of Concurrent Systems: Function and Timing. In *Proceedings of Protocol Specification, Testing and Verification PSTV'82*, pages 401–409. North-Holland, 1982.

[115] R. Paige and R. E. Tarjan. Three Partition Refinement Algorithms. *SIAM Journal of Computing*, 16(6):973–989, 1987.

[116] D. Park. Concurrency and Automata on Infinite Sequences. In *Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer, 1981.

[117] D. Peled. All from One, One for All: on Model Checking Using Representatives. In *Proceedings of Computer Aided Verification CAV'93*, volume 697 of *Lecture Notes in Computer Science*, pages 409–423. Springer, 1993.

[118] G. D. Plotkin. A structural approach to operational semantics. Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[119] G. D. Plotkin. A Structural Approach to Operational Semantics. *Journal of Logic and Algebraic Programming*, 60–61:17–139, 2004.

[120] G. D. Plotkin. The Origins of Structural Operational Semantics. *Journal of Logic and Algebraic Programming*, 60–61:3–15, 2004.

[121] A. Pnueli. The Temporal Logic of Programs. In *Proceedings of Foundations of Computer Science*, pages 46–57. IEEE, 1977.

[122] A. Pnueli. A Temporal Logic of Concurrent Programs. *Theoretical Computer Science*, 13:45–60, 1981.

[123] J.-P. Queille and J. Sifakis. Fairness and Related Properties in Transition Systems — A Temporal Logic to Deal with Fairness. *Acta Informatica*, 19:195–220, 1983.

[124] J. Rathke and M. Hennessy. Local Model Checking for a Value-Based Modal μ-Calculus. Report 5/96, School of Cognitive and Computing Sciences, University of Sussex, 1996.

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

[126] 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.

[127] R. Streett. Propositional Dynamic Logic of Looping and Converse. *Information and Control*, (54):121–141, 1982.

[128] K. C. Tai and V. Koppol. Hierarchy-Based Incremental Reachability Analysis of Communication Protocols. In *Proceedings of Network Protocols*, pages 318–325. IEEE, 1993.

[129] K. C. Tai and V. Koppol. An Incremental Approach to Reachability Analysis of Distributed Programs. In *Proceedings of Software Specification and Design*, pages 141–150. IEEE Press, 1993.

[130] D. Thivolle. *Langages modernes pour la vérification des systèmes asynchrones.* Thèse de Doctorat, Université Joseph Fourier (Grenoble, France) and Universitatea Politehnica din Bucuresti (Bucharest, Romania), 2011.

[131] W. Thomas. *Computation Tree Logic and Regular ω-Languages.* In *Linear time, branching time and partial order in logics and models of concurrency*, volume 354 of *Lecture Notes in Computer Science*, pages 690–713. 1989.

[132] F. Tronel, F. Lang, and H. Garavel. Compositional Verification Using CADP of the ScalAgent Deployment Protocol for Software Components. In *Proceedings of Formal Methods for Open Object-based Distributed Systems FMOODS'2003*, volume 2884 of *Lecture Notes in Computer Science*, pages 244–260. Springer, 2003.

[133] A. Valmari. A Stubborn Attack on State Explosion. In *Proceedings of Computer-Aided Verification CAV'90*, volume 3 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 25–42. AMS-ACM, 1990.

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

[135] R. J. van Glabbeek. *The Linear Time – Branching Time Spectrum I. The Semantics of Concrete, Sequential Processes*. In *Handbook of Process Algebra*, chapter 1, pages 3–99. North-Holland, 2001.

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

[137] R. J. van Glabbeek and W. P. Weijland. Branching-Time and Abstraction in Bisimulation Semantics. *Journal of the ACM*, 43(3):555–600, 1996.

[138] P. Wolper. Temporal Logic Can Be More Expressive. *Information and Control*, 56(1/2):72–99, 1983.

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

[140] W. J. Yeh and M. Young. Compositional Reachability Analysis Using Process Algebra. In *Proceedings of Testing, Analysis, and Verification SIGSOFT'91*, pages 49–59. ACM Press, 1991.

A Temporal logic specifications for the TFTP case study

This appendix presents the temporal logic specifications used in this paper for the TFTP case study. Properties $A01$ to $A28$ were taken from [130]. Property $A29$ was taken from [88]. The 29 properties are given here using the SVL property notation. In the formula part (i.e., between symbols "$\models$" and ";"), double quotes enclose character strings denoting a label (or part of a label), such as "REINIT_A" in property $A06$; single quotes enclose regular expressions denoting a label, such as $'$RECEIVE_A.$^{*}{}'$ in property $A06$; and braces enclose typed expression tuples denoting a label by specifying its various components, such as {RECEIVE_A !"ACK" $?N$: Nat} in property $A14$.

The following constants are stored in a file named `macros.mcl`, which is automatically included in all the properties below.

```
macro FILE_SIZE_A ()          = 2 of Nat end_macro
macro FILE_SIZE_B ()          = 2 of Nat end_macro
macro MAX_RETRIES_A ()        = 2 of Nat end_macro
macro MAX_RETRIES_B ()        = 4 of Nat end_macro
macro MIN_RETRIES_AB ()       = 2 of Nat end_macro
```

property A01
The TFTP automaton has two output ports, ARM_TIMER *and* STOP_TIMER*, that respectively start and stop the timer used to decide when an incoming message should be considered as lost. The following formula ensures that between two consecutive* STOP_TIMER *actions, there must be an* ARM_TIMER *action. Otherwise said, it states that, from every reachable state (i.e., after firing any sequence* true* *of transitions from the initial state), there exists no sequence of transitions containing two* STOP_TIMER *actions with no* ARM_TIMER *action in between.*
is

```
"SCENARIO.bcg" ⊨
    [
        true* .
        "STOP_TIMER_A" .
        not ("ARM_TIMER_A")* .
        "STOP_TIMER_A"
    ] false;
expected true
end property
```

property A02
Between two consecutive ARM_TIMER *actions, there must be a* STOP_TIMER *action, a timeout, or a reception. The following formula ensures this by guaranteeing that there exists no sequence of transitions containing two* ARM_TIMER *actions without a* STOP_TIMER*, a timeout, or a reception in between.*
is

```
"SCENARIO.bcg" ⊨
    [
        true* .
        "ARM_TIMER_A" .
        not ("STOP_TIMER_A" or "TIMEOUT_A" or 'RECEIVE_A.*')*.
        "ARM_TIMER_A"
    ] false;
expected true
```

end property

property A03
The timer cannot be active between two transfers. The following formula ensures there is no ACTIVE_TIMER_BETWEEN_TRANSFERS *action reachable.*
is

```
"SCENARIO.bcg" ⊨
    [ true* . "ACTIVE_TIMER_BETWEEN_TRANSFERS_A" ] false;
expected true
```

end property

property A04
A timeout cannot occur between a STOP_TIMER *action and an* ARM_TIMER *action. The following formula states that there is no sequence of transitions in which a timeout can follow a* STOP_TIMER *action without an* ARM_TIMER *action in between.*
is

```
"SCENARIO.bcg" ⊨
    [
        true* .
        "STOP_TIMER_A" .
        not ("ARM_TIMER_A")* .
        "TIMEOUT_A"
    ] false;
expected true
```

end property

property A05
A timeout cannot occur before the first message is sent. The following formula ensures that there is no sequence of transitions in which a timeout occurs before the first send action.
is

```
"SCENARIO.bcg" ⊨
    [ not ('SEND_A.*')* . "TIMEOUT_A" ] false;
expected true
```

end property

property A06

An internal error must cause the transfer to abort. The following formula ensures that there is no sequence of transitions in which a reception or a transmission can occur after an internal error but before a reinitialisation and/or the transmission of an error.

is

```
"SCENARIO.bcg" ⊨
    [
        true* .
        "INTERNAL_ERROR_A" .
        not ("REINIT_A" or "SEND_A !ERROR")* .
        'RECEIVE_A.*' or ('SEND_A.*' and not "SEND_A !ERROR")
    ] false;
expected true
end property
```

property A07

An invalid packet must cause the transfer to abort. The following formula ensures that there is no sequence of transitions in which a reception or a transmission can occur after an invalid packet was received but before a reinitialisation and/or the transmission of an error.

is

```
"SCENARIO.bcg" ⊨
    [
        true* .
        "INVALID_PACKET_A" .
        not ("REINIT_A" or "SEND_A !ERROR")* .
        'RECEIVE_A.*' or ('SEND_A.*' and not "SEND_A !ERROR")
    ] false;
expected true
end property
```

property A08

When a TFTP protocol entity receives an error, it must abort the current transfer. The original formula ensured that receiving an error cannot be followed by sending an error. It stated that there exists no sequence of transitions in which TFTP entity A can send an error after receiving one, without performing any other action in between. Actually, we used a simpler formula (given below) which, in the TFTP verification setting, is equivalent to the former.

is

```
"SCENARIO.bcg" ⊨
    [ true* . "RECEIVE_A !ERROR" . "SEND_A !ERROR" ] false;
expected true
end property
```

property A09a

If both protocol entities initiate a transfer at the same time, they must abort upon receiving the other protocol entity's request, in particular, when a transfer request (either read or write) is received after sending a read request. The following formula ensures that there exists no sequence of transitions in which sending a read request and receiving a request can be followed by the transmission of a message until there has been a reinitialisation (transfer succeeded or aborted).

is

```
"SCENARIO.bcg" ⊨
    [
```

```
            true* .
            'SEND_A !RRQ.*' .
            true . (* ARM_TIMER_A *)
            'RECEIVE_A !RRQ.*' or 'RECEIVE_A !WRQ.*' .
            not ("REINIT_A")* .
            'SEND_A.*')
        ] false;
    expected true
end property
```

property A09b

If both protocol entities initiate a transfer at the same time, they must abort upon receiving the other protocol entity's request, in particular, when a transfer request (either read or write) is received after sending a write request. The following formula ensures that there exists no sequence of transitions in which sending a read request and receiving a request can be followed by the transmission of a message until there has been a reinitialisation (transfer succeeded or aborted).

is

```
    "SCENARIO.bcg" ⊨
        [
            true* .
            'SEND_A !WRQ.*' .
            true . (* ARM_TIMER_A *)
            'RECEIVE_A !RRQ.*' or 'RECEIVE_A !WRQ.*' .
            not ("REINIT_A")* .
            'SEND_A.*')
        ] false;
    expected true
end property
```

property A10

A process cannot switch from sending data fragments to sending acknowledgements without having received a write request or sent a read request. The following formula ensures that there is no sequence of transitions in which sending a data fragment can be followed by sending an acknowledgement without first either sending a read request or receiving a write request in between.

is

```
    "SCENARIO.bcg" ⊨
        [
            true* .
            'SEND_A !DATA.*' .
            not ('SEND_A !RRQ.*' or 'RECEIVE_A !WRQ.*')* .
            'SEND_A !ACK.*'
        ] false;
    expected true
end property
```

property A11

A process cannot switch from sending acknowledgements to sending data fragments without having received a read request or sent a write request. The following formula ensures that there is no sequence of transitions in which sending an acknowledgement can be followed by sending a data fragment without first either sending a write request or receiving a read request in between.

```
is
    "SCENARIO.bcg" ⊨
        [
            true* .
            'SEND_A !ACK.*' .
            not ('SEND_A !WRQ.*' or 'RECEIVE_A !RRQ.*')* .
            'SEND_A !DATA.*'
        ] false;
    expected true
end property
```

property A12

In the case where TFTP protocol entity A is transferring a file, it must be possible for its transfer to finish.

```
is
    "SCENARIO.bcg" ⊨
        ⟨ true* . "SUCCESS_A" ⟩ true;
    expected true
end property
```

property A13

During the dallying phase, it is possible to begin a new transfer upon receiving a read or write request. The following formula states that for each of the two phases (corresponding to waiting for two timeouts to occur after sending the final acknowledgement), the reception of a request before the timeout occurs can be answered.

```
is
    "SCENARIO.bcg" ⊨
        forall X : Nat among {0...1} .
            ⟨
                true* .
                {RECEIVE_A !"DATA" ?N : Nat ?any !TRUE} .
                not ({SEND_A !"ACK" !N})* .
                {SEND_A !"ACK" !N} .
                (not (TIMEOUT_A or REINIT_A)* . TIMEOUT_A) {X} .
                not (TIMEOUT_A or REINIT_A)*
            ⟩
                (
                    ⟨
                        {RECEIVE_A !"WRQ" ?any} .
                        not {RECEIVE_A ...}* .
                        {SEND_A !"ACK" !0 of Nat}
                    ⟩ true
                    or
                    ⟨
                        {RECEIVE_A !"RRQ" ?any} .
                        not {RECEIVE_A ...}* .
                        {SEND_A !"DATA" !1 of Nat ...}
                    ⟩ true
                );
    expected true
end property
```

property A14

In order to avoid the Sorcerer's Apprentice bug, all resent acknowledgements must be ignored. The following formula ensures that there exists no sequence of transitions in which the same acknowledgement is answered twice without an intervening reinitialisation.

is

```
"SCENARIO.bcg" ⊨
    [
        true* .
        {RECEIVE_A !"ACK" ?N : Nat} .
        {SEND_A !"DATA" !N + 1 ...} .
        not (REINIT_A or {RECEIVE_A !"ACK" !N})* .
        {RECEIVE_A !"ACK" !N} .
        {SEND_A !"DATA" !N + 1 ...}
    ] false;
expected true
```

end property

property A15

Re-sent data fragment can be acknowledged, to the limit set by the value of the maximum number of retries. The following formula states that after a data fragment is received, it can be received and acknowledged again (only ARM_TIMER_A *and* STOP_TIMER_A *actions can be performed by TFTP entity A in the meantime) up to* MIN_RETRIES_AB *times (where* MIN_RETRIES_AB *is the minimum of* MAX_RETRIES_A *and* MAX_RETRIES_B*).*

is

```
"SCENARIO.bcg" ⊨
    forall N : Nat among {1...FILE_SIZE_A()} .
        ⟨
            true* .
            {RECEIVE_A !"DATA" !N ...} .
            (not ('.*_A.*') or '.*TIMER_A.*')* .
            {SEND_A !"ACK" !N} .
            (
                (not ('.*_A.*') or '.*TIMER_A.*')* .
                {RECEIVE_A !"DATA" !N ...} .
                (not ('.*_A.*') or '.*TIMER_A.*')* .
                {SEND_A !"ACK" !N}
            ) {MIN_RETRIES_AB()}
        ⟩ true;
expected true
```

end property

property A16

Every resent read request must be answered, to the limit set by the value of the maximum number of retries. The following formula states that after a read request is received for the first time, it can be received and answered again up to MIN_RETRIES_AB *times.*

is

```
"SCENARIO.bcg" ⊨
    [
        not {RECEIVE_A !"RRQ" ...}* .
        {RECEIVE_A !"RRQ" ?N : Nat} .
        not {RECEIVE_A ...}* .
        {SEND_A !"DATA" !1 of Nat ...}
    ]
        ⟨
```

```
                (
                    not (REINIT_A or {RECEIVE_A !"RRQ" !N})* .
                    {RECEIVE_A !"RRQ" !N} .
                    {SEND_A !"DATA" !1 of Nat ...}
                ) {MIN_RETRIES_AB()}
            ⟩ true;
    expected true
end property


property A17
    Every write request must be acknowledged, to the limit set by the value of
    the maximum number of retries. The following formula states that after a
    write request is received for the first time, it can be received and answered
    again up to MIN_RETRIES_AB times.
is
    "SCENARIO.bcg" ⊨
        [
            not {RECEIVE_A !"WRQ" ...}* .
            {RECEIVE_A !"WRQ" ?N : Nat} .
            not ({RECEIVE_A ...})* .
            {SEND_A !"ACK" !0 of Nat}
        ]
            ⟨
                (
                    not (REINIT_A or {RECEIVE_A !"WRQ" !N})* .
                    {RECEIVE_A !"WRQ" !N} .
                    {SEND_A !"ACK" !0 of Nat}
                ) {MIN_RETRIES_AB()}
            ⟩ true;
    expected true
end property


property A18
    An acknowledgement can be resent as many times as allowed by the value of
    the maximum number of retries. The following formula states that after an
    acknowledgement is sent for the first time, it can be sent again (regardless
    of the reason) up to MAX_RETRIES_A times within the same transfer.
is
    "SCENARIO.bcg" ⊨
        forall N : Nat among {0 ... FILE_SIZE_A()} .
            [
                not ({SEND_A !"ACK" !N})* .
                {SEND_A !"ACK" !N}
            ]
                ⟨
                    (
                        not ('.*[WR]RQ.*' or {SEND_A !"ACK" !N})* .
                        {SEND_A !"ACK" !N}
                    ) {MAX_RETRIES_A()}
                ⟩ true;
    expected true
end property


property A19
```

An acknowledgement cannot be resent more times than allowed by the value of the maximum number of retries. The following formula states that there is no sequence of transitions in which sending the same acknowledgement can occur more than MAX_RETRIES_A *times within the same transfer.*

```
is
    "SCENARIO.bcg" ⊨
        forall N : Nat among {0 ... FILE_SIZE_A()} .
            [
                true* .
                {SEND_A !"ACK" !N} .
                (
                    not ('.*[WR]RQ.*' or {SEND_A !"ACK" !N})* .
                    {SEND_A !"ACK" !N}
                ) {MAX_RETRIES_A() + 1}
            ] false;
    expected true
end property
```

property A20

A data fragment can be resent as many times as allowed by the value of the maximum number of retries. The following formula states that after a data fragment is sent for the first time, it can be sent again (regardless of the reason) up to MAX_RETRIES_A *times within the same transfer.*

```
is
    "SCENARIO.bcg" ⊨
        forall N : Nat among {1 ... FILE_SIZE_A()} .
            [
                not ({SEND_A !"DATA" !N ...})* .
                {SEND_A !"DATA" !N ...}
            ]
                ⟨
                    (
                        not ('.*[WR]RQ.*' or {SEND_A !"DATA" !N ...})* .
                        {SEND_A !"DATA" !N ...}
                    ) {MAX_RETRIES_A()}
                ⟩ true;
    expected true
end property
```

property A21

A data fragment cannot be resent more times than allowed by the value of the maximum number of retries. The following formula states that there is no sequence of transitions in which sending the same data fragment can occur more than MAX_RETRIES_A *times within the same transfer.*

```
is
    "SCENARIO.bcg" ⊨
        forall N : Nat among {1 ... FILE_SIZE_A()} .
            [
                true* .
                {SEND_A !"DATA" !N ...} .
                (
                    not ('.*[WR]RQ.*' or {SEND_A !"DATA" !N ...})* .
                    {SEND_A !"DATA" !N ...}
                ) {MAX_RETRIES_A() + 1}
            ] false;
    expected true
end property
```

property A22

A read request can be resent as many times as allowed by the value of the maximum number of retries. The following formula states that after a read request is sent for the first time, it can be sent again (regardless of the reason) up to MAX_RETRIES_A *times within the same transfer.*

is

```
"SCENARIO.bcg" ⊨
    [
        not ({SEND_A !"RRQ" ...})* .
        {SEND_A !"RRQ" ?N : Nat}
    ]
        ⟨
            (
                not (REINIT_A or {SEND_A !"RRQ" !N})* .
                {SEND_A !"RRQ" !N}
            ) {MAX_RETRIES_A()}
        ⟩ true;
expected true
end property
```

property A23

A read request cannot be resent more times than allowed by the value of the maximum number of retries. The following formula states that there is no sequence of transitions in which sending the same read request can occur more than MAX_RETRIES_A *without a reinitialisation.*

is

```
"SCENARIO.bcg" ⊨
    [
        true* .
        {SEND_A !"RRQ" ?N : Nat} .
        (
            not (REINIT_A or {SEND_A !"RRQ" !N})* .
            {SEND_A !"RRQ" !N}
        ) {MAX_RETRIES_A() + 1}
    ] false;
expected true
end property
```

property A24

A write request can be resent as many times as allowed by the value of the maximum retries. The following formula states that after a write request is sent for the first time, it can be sent again (regardless of the reason) up to MAX_RETRIES_A *times within the same transfer.*

is

```
"SCENARIO.bcg" ⊨
    [ not ({SEND_A !"WRQ" ...})* . {SEND_A !"WRQ" ?N : Nat} ]
        ⟨
            (
                not (REINIT_A or {SEND_A !"WRQ" !N})* .
                {SEND_A !"WRQ" !N}
            ) {MAX_RETRIES_A()}
        ⟩ true;
expected true
end property
```

property A25

A write request cannot be resent more times than allowed by the value of the maximum number of retries. The following formula states that there is no sequence of transitions in which sending the same write request can occur more than MAX_RETRIES_A *without a reinitialisation.*

is

```
"SCENARIO.bcg" ⊨
    [
        true* .
        {SEND_A !"WRQ" ?N : Nat} .
        (
            not (REINIT_A or {SEND_A !"WRQ" !N})* .
            {SEND_A !"WRQ" !N}
        ) {MAX_RETRIES_A() + 1}
    ] false;
expected true
end property
```

property A26

Data fragments must be sent in proper order. The following formula states that any data fragment numbered X cannot be followed by a data fragment numbered Y, where $Y < X$, unless there has been a reinitialisation in between.

is

```
"SCENARIO.bcg" ⊨
    [
        true* .
        {SEND_A !"DATA" ?X : Nat ...} .
        not (REINIT_A)* .
        {SEND_A !"DATA" ?Y : Nat ... where Y < X}
    ] false;
expected true
end property
```

property A27

Between the transmission of two successive data fragments, there must be the reception of the corresponding acknowledgement. The following formula states that there is no sequence of transitions in which the transmission of a data fragment numbered X can be followed by the transmission of a data fragment numbered $X + 1$ without the reception of the acknowledgement numbered X in between.

is

```
"SCENARIO.bcg" ⊨
    [
        true* .
        {SEND_A !"DATA" ?X : Nat ?any !FALSE} .
        not ({RECEIVE_A !"ACK" !X})* .
        {SEND_A !"DATA" !X + 1 ...}
    ] false;
expected true
end property
```

property A28

Between the transmission of two successive acknowledgements, there must be the reception of the corresponding data fragment. The following formula states that there is no sequence of transitions in which the transmission of an acknowledgement numbered X can be followed by the transmission of an acknowledgement numbered $X + 1$ without the reception of the data fragment numbered $X + 1$ in between.

is

```
"SCENARIO.bcg" ⊨
    [
        true* .
        {SEND_A !"ACK" ?X : Nat} .
        not ({RECEIVE_A !"DATA" !X + 1 ...})* .
        {SEND_A !"ACK" !X + 1}
    ] false;
expected true
end property
```

property A29

There exists a cycle of transitions satisfying the regular expression written in between angles below.

is

```
"SCENARIO.bcg" ⊨
    ⟨
        true* .
        "STOP_TIMER_A" .
        not ("STOP_TIMER_A" or "ARM_TIMER_A")* .
        "TIMEOUT_A" .
        not ("STOP_TIMER_A")* .
        "ARM_TIMER_A"
    ⟩ @;
expected false
end property
```

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

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

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