# State Space Reduction for Process Algebra Specifications

Hubert Garavel, Wendelin Serwe

*INRIA Rhône-Alpes / VASY*
*655, avenue de l'Europe*
*F-38334 St. Ismier Cedex, France*

**Abstract**

Data-flow analysis to identify "dead" variables and reset them to an "undefined" value is an effective technique for fighting state explosion in the enumerative verification of concurrent systems. Although this technique is well-adapted to imperative languages, it is not directly applicable to value-passing process algebras, in which variables cannot be reset explicitly due to the single-assignment constraints of the functional programming style. This paper addresses this problem by performing data-flow analysis on an intermediate model (Petri nets extended with state variables) into which process algebra specifications can be translated automatically. It also addresses important issues such as avoiding the introduction of useless reset operations and handling shared read-only variables that child processes inherit from their parents.

*Key words:* Cadp – Cæsar – compositional verification – data-flow analysis – formal specification – labeled transition system – Lotos – model checking – process algebra

## 1 Introduction

We consider the verification of concurrent systems using *enumerative* (or *explicit state*) techniques, which is based on enumerating the system states reachable from the initial state.

*Email addresses:* Hubert.Garavel@inria.fr (Hubert Garavel), Wendelin.Serwe@inria.fr (Wendelin Serwe).
*URLs:* http://www.inrialpes.fr/vasy/people/Hubert.Garavel (Hubert Garavel), http://www.inrialpes.fr/vasy/people/Wendelin.Serwe (Wendelin Serwe).

Among the various approaches to avoid state explosion, it has been known for long (e.g., [1]) that a significant reduction of the state space can be achieved by *resetting* state variables as soon as their values are no longer needed. This avoids distinguishing between states that only differ by the values of so-called *dead variables*, i.e., variables that will not be used in the future before they are assigned again. Resetting these variables, as soon as they become useless, to some "undefined" value (usually, a pattern of 0-bits) allows states that would otherwise differ to be considered as identical.

When concurrent systems are described using an imperative language with explicit assignments, it is possible to reset variables by inserting zero-assignments manually in the source program (e.g., [1]). Some languages even provide a dedicated instruction for resetting variables (e.g., [2, §6]). Despite its apparent simplicity, this approach proves to be tedious and error-prone, and it obscures the source program with verification artefacts. Both its correctness and efficiency critically depend on the specifier's skills (resets have to be inserted at all the right places and only these).

Moreover, this approach does not apply to value-passing process algebras (i.e., process algebras with data values such as CCS, CSP, LOTOS [3], $\mu$CRL, etc.), which use a functional programming style in which variables are initialised only once and cannot be reassigned (thus, reset) later.

This paper addresses these two problems by presenting a general method that is applicable to process algebras, and that allows variables to be reset automatically in a fully transparent way for the specifier. This method proceeds in two steps.

In a first step, process algebra specifications are translated automatically into an intermediate model with an imperative semantics. This approach was first suggested in [4,5], which proposed a so-called *network model* consisting of a Petri net extended with state variables, the values of which are consulted and modified when the transitions are executed. This network model is used in the CÆSAR compiler for LOTOS (CÆSAR is distributed as part of the widespread CADP verification toolbox [6]). This paper presents the most recent version of the network model, which adds to the model of [4,5] the enhancements introduced since 1990 in order to allow state space reductions based on transition compaction and to support the EXEC/CÆSAR framework for rapid prototyping of LOTOS specifications. We believe that this network model is sufficiently general to be used for process algebras other than LOTOS.

In a second step, resets are introduced, not at the source level (process algebraic specifications), but in the intermediate model, by attaching resets to the transitions of the network.

Various techniques can be used to determine automatically which variables can be reset by which transitions. A simple approach consists in resetting all the variables of a process as soon as this process terminates. This approach was implemented in CÆSAR 4.3 (January 1992) and gives significant reductions [1] for terminating processes (especially at the points corresponding to the sequential composition ("`>>`") and disabling ("`[>`") operators of LOTOS, which are detected by analysing the structure of the network model), but not for cyclic (i.e., non-terminating) processes. The XMC model checker uses a similar approach [7], with two minor differences: dead variables are determined by analysing the sequential composition of processes at the source level and are removed from the representation of the state instead of being reset. [2]

A more sophisticated approach was studied in 1992–1993 by the first author and one of his MSc students [9] in order to introduce variable resets everywhere it would be possible, including in cyclic processes. A key idea in [9] was the computation of variable resets by means of classical data-flow analysis techniques (precisely, dead variable analysis), such as those used in optimizing compilers for sequential languages. An experimental version of CÆSAR implementing this idea was developed in 1993. Although it gave significant state space reductions, it also happened to produce incorrect results on certain examples, which prevented it from being integrated in the official releases of CÆSAR. The reason for these errors was unknown at that time, but is now understood and addressed in this paper.

The use of data-flow analysis for resetting dead variables was later mentioned in [8] and formalised in [10,11] and recently [12], the main point of [10,11] being the proof that reduction based on dead variable analysis preserves strong bisimulation. The main differences between [10,11], [12], and our approach are the following:

- Our work addresses value-passing process algebras, such as LOTOS. [10,11] target the SDL language, and, thus, consider a set of communicating automata with state variables that are consulted and assigned by automata transitions. [12] targets a concurrent language consisting of sequential, deterministic processes with only local variables and process algebra-like primitives for communication and synchronisation between processes.
- As regards system architecture, the network model of CÆSAR allows concurrent processes to be nested to an arbitrary depth; this is needed for a compositional translation of process algebra specifications in which parallel and sequential composition operators are intertwined arbitrarily — such as

[1] For the "rel/REL" reliable atomic multicast protocol, CÆSAR 4.3 generated (in 1992) a state space of 126,223 states and 428,766 transitions in 30 minutes on a DEC Station 5000 with 24 MB RAM, while CÆSAR 4.2 would generate a state space of 679,450 states and 1,952,843 transitions in 9 hours on the same machine.

[2] See the concerns expressed in [8] about the poor efficiency of such a variable-length state representation scheme.

the LOTOS behavior "$B_1$`>>(`$B_2$`|||`$B_3$`)>>`$B_4$" expressing that the execution of process $B_1$ is followed by the concurrent execution of two processes $B_2$ and $B_3$, which, upon termination of both, will be followed by the execution of process $B_4$. On the contrary, the models of [10,11] and [12] lack any form of process hierarchy and allow only a "flat" collection of communicating automata, all activated in the initial state.
- As regards interprocess communications, the network model implements the Hoare-style rendezvous mechanism used in process algebras by synchronised Petri net transitions, which allow data exchanges between processes. To the contrary, the model of [10,11] relies on FIFO message queues and shared variables that can be arbitrarily read/written by all the processes. The model of [12] is closer to our network model in that it uses a communication scheme based on handshaking, but less general, since in the network model concurrent processes may share variables inherited from their parent process(es) — as in the LOTOS behavior "$G$`?`$X$`:`$S$`;(`$B_1$`|||`$B_2$`)`", in which both processes $B_1$ and $B_2$ can use variable $X$ of sort $S$, whose value has been set in their parent process. These shared variables are read-only, in the sense that child processes cannot modify them.
- As regards shared variables, [10,11] propose an approach in which variable resets are computed partly at compile-time (when analysing each communicating automaton separately) and partly at run-time (when generating all reachable states of the product automaton). It is difficult to figure out how this approach can be implemented in practice, since the authors stand far from algorithmic concerns and since the most recent versions[3] of their IF tool set [13] do not actually reset shared variables. However, we believe that the communicating automata model used by [10,11] is not sufficient in itself to express resets of shared variables, so that some extra information (yet to be specified) must be passed from compile-time to run-time. In comparison, the approach presented in this paper can be performed entirely at compile-time and requires no addition to the network model.

This paper is organised as follows. Section 2 presents the network model and its operational semantics. Sections 3 and 4 respectively present the local and global data-flow analyses derived from [9] for determination of variable resets. Section 5 deals with the particular case of inherited variables, which need careful attention to avoid semantic problems caused by a "naive" insertion of resets. Section 6 reports experimental results, and Section 7 gives concluding remarks.

[3] Namely, IF 1.0 (dated November 2003) and IF 2.0 (dated March 2003 (sic)).

## 2 Presentation of the Network Model

The network model presented here is based on the definitions of [4,5], the essential characteristics of which are retained (namely, the Petri net structure with state variables); but it also contains some more recent extensions that proved to be useful.

Formally, a network is a tuple $\langle \mathcal{Q}, Q_0, \mathcal{U}, \mathcal{T}, \mathcal{G}, \mathcal{X}, \mathcal{S}, \mathcal{F} \rangle$, the components of which will be presented progressively, so as to avoid forward references. We will use the following convention consistently: elements of set $\mathcal{Q}$ (*resp.* $\mathcal{U}$, $\mathcal{T}$, $\mathcal{G}$, $\mathcal{X}$, $\mathcal{S}$, $\mathcal{F}$) are noted by the corresponding capital letter, e.g., $Q$, $Q_0$, $Q_1$, $Q'$, $Q''$, etc.

**Sorts, Functions, and Variables.** In the above definition of a network, $\mathcal{S}$ denotes a finite set of *sorts* (i.e., data types), $\mathcal{F}$ denotes a finite set of *functions*, and $\mathcal{X}$ denotes a finite set of *(state) variables*. We define *domain*$(S)$ as the (possibly infinite) set of *ground values* of sort $S$. Functions take (zero, one, or many) typed arguments and return a typed result. Variables also are typed.

**Contexts.** To represent the memory containing state variables, we define a *context* $C$ as a (partial) function mapping each variable of $\mathcal{X}$ either to its ground value or to the undefined value, written "$\perp$". We need 5 operations to handle contexts. For contexts $C_1$ and $C_2$, and variables $X_0, \ldots, X_n$, we define the contexts:

- $\{\}$: $X \mapsto \perp$ (i.e., the empty context)
- $\{X_0 \mapsto v\}$: $X \mapsto$ if $X = X_0$ then $v$ else $\perp$
- $C_1 \ominus \{X_0, \ldots, X_n\}$: $X \mapsto$ if $X \in \{X_0, \ldots, X_n\}$ then $\perp$ else $C_1(X)$
- $C_1 \oslash C_2$: $X \mapsto$ if $C_2(X) \neq \perp$ then $C_2(X)$ else $C_1(X)$
- $C_1 \oplus C_2$: $X \mapsto$ if $C_2(X) \neq \perp$ then $C_2(X)$ else $C_1(X)$
  We only use $\oplus$ on "disjoint" contexts, i.e., when $\big(C_1(X){=}\perp\big) \vee \big(C_2(X){=}\perp\big)$.

**Value Expressions.** A *value expression* is a term built using variables and functions: $V ::= X \mid F(V_1, \ldots, V_{n\geq 0})$. We define *eval*$(C, V)$ as the (unique) ground value obtained by evaluating value expression $V$ in context $C$ (after substituting variables with their ground values given by $C$ and applying functions). Because the network is generated from a LOTOS specification that is correctly typed and well-defined (i.e., each variable is initialised before used), evaluating a value expression never fails due to type errors or undefined variables.

**Offers.** An *offer* is a term of the form: $O ::= \texttt{!}V \mid \texttt{?}X\texttt{:}S \mid O_1 \ldots O_{n\geq 0}$, meaning that an offer is a (possibly empty) sequence of *emissions* (written "!") and/or *receptions* (written "?"). We define a relation "$[C,O] \xrightarrow{o} [C', v_1 \ldots v_n]$" expressing that offer $O$ evaluated in context $C$ yields a (possibly empty) list of ground values $v_1 \ldots v_n$ and a new context $C'$ ($C'$ reflects that each reception of the form "$\texttt{?}X\texttt{:}S$" binds $X$ to the received value(s)). For a given pair $[C,O]$ there might be one or several pairs $[C', v_1 \ldots v_n]$ such that $[C,O] \xrightarrow{o} [C', v_1 \ldots v_n]$, since a reception "$\texttt{?}X\texttt{:}S$" generates as many pairs as there are ground values in $domain(S)$.

$$\frac{v = eval(C,V)}{[C, \texttt{!}V] \xrightarrow{o} [\{\}, v]} \quad \frac{v \in domain(S)}{[C, \texttt{?}X\texttt{:}S] \xrightarrow{o} [\{X \mapsto v\}, v]} \quad \frac{(\forall i \in \{1,\ldots,n\})\ [C,O_i] \xrightarrow{o} [C_i, v_i]}{[C, O_1 \ldots O_n] \xrightarrow{o} \left[\bigoplus_{i=1}^{n} C_i, v_1 \ldots v_n\right]}$$

The use of $\oplus$ in the definition of $\xrightarrow{o}$ is possible, since LOTOS ensures that all variables $X_i$ used to receive inputs in an offer are pairwise distinct.

**Actions.** Actions are terms of the form:

$$\begin{array}{rll} A ::= & \textbf{none} & \textit{(empty action)} \\ \mid & \textbf{when } V & \textit{(condition)} \\ \mid & \textbf{for } X \textbf{ among } S & \textit{(iteration)} \\ \mid & X_0, \ldots, X_{n\geq 0} \texttt{:=} V_0, \ldots, V_n & \textit{(vector assignment)} \\ \mid & \textbf{reset } X_0, \ldots, X_{n\geq 0} & \textit{(variable reset)} \\ \mid & A_1 \,\texttt{;}\, A_2 & \textit{(sequential composition)} \\ \mid & A_1 \texttt{\&} A_2 & \textit{(collateral composition)} \end{array}$$

We define a relation "$[C,A] \xrightarrow{c} C'$" expressing that successful execution of action $A$ in context $C$ yields a new context $C'$. For a given pair $[C,A]$ there might be zero, one, or several $C'$ such that $[C,A] \xrightarrow{c} C'$, since a "**when** $V$" condition may block the execution if $V$ evaluates to false, whereas a "**for** $X$ **among** $S$" iteration triggers as many executions as there are ground values in $domain(S)$.

$$\frac{}{[C, \textbf{none}] \xrightarrow{c} C} \quad \frac{eval(C,V) = \mathsf{true}}{[C, \textbf{when } V] \xrightarrow{c} C} \quad \frac{v \in domain(S) \quad [C, X \texttt{:=} v] \xrightarrow{c} C'}{[C, \textbf{for } X \textbf{ among } S] \xrightarrow{c} C'}$$

$$\frac{C' = C \oslash \bigoplus_{i=0}^{n} \{X_i \mapsto eval(C, V_i)\}}{[C, X_0, \ldots, X_n \texttt{:=} V_0, \ldots, V_n] \xrightarrow{c} C'} \quad \frac{C' = C \ominus \{X_0, \ldots, X_n\}}{[C, \textbf{reset } X_0, \ldots, X_n] \xrightarrow{c} C'}$$

$$\frac{[C, A_1] \xrightarrow{c} C' \qquad [C', A_2] \xrightarrow{c} C''}{[C, A_1 \,\texttt{;}\, A_2] \xrightarrow{c} C''} \quad \frac{[C, A_1 \,\texttt{;}\, A_2] \xrightarrow{c} C'' \qquad [C, A_2 \,\texttt{;}\, A_1] \xrightarrow{c} C''}{[C, A_1 \texttt{\&} A_2] \xrightarrow{c} C''}$$

**Gates.** In the above definition of a network, $\mathcal{G}$ denotes a finite set of *gates* (i.e., names for communication points). There are two special gates: "$\tau$", the usual notation for the internal steps of a process, and "$\varepsilon$", which does not exist in the structured operational semantics of Lotos [3], but is introduced in Cæsar's translation algorithms [4,5] to allow a compositional construction of networks for a large class of Lotos behaviors such as "$B_1$ `[]` `(`$B_2$ `|||` $B_3$`)`". Although $\varepsilon$ deserves a special semantic treatment, namely the computation of an "$\varepsilon$-closure", this has no influence on the approach proposed in this paper; thus, we do not distinguish $\varepsilon$ from "ordinary" gates here.

**Places and Transitions.** In the above definition of a network, $\mathcal{Q}$ denotes a finite set of *places*, $Q_0 \in \mathcal{Q}$ is the *initial place* of the network, and $\mathcal{T}$ denotes a finite set of *transitions*. Each transition $T$ is a tuple $\langle Q_i, Q_o, A, G, O, W, R \rangle$, where $Q_i \subseteq \mathcal{Q}$ is a set of *input places* (written $\mathit{in}(T) \triangleq Q_i$), $Q_o \subseteq \mathcal{Q}$ is a set of *output places* (written $\mathit{out}(T) \triangleq Q_o$), $A$ is an action, $G$ is a gate, $O$ is a (possibly empty) offer, $W$ is a *when-guard* (i.e., a restricted form of action constructed only with "**none**", "**when**", "`;`", and "`&`"), and $R$ is a *reaction* (i.e., a restricted form of action constructed only with "**none**", "`:=`", "**reset**", "`;`", and "`&`").

**Markings.** As regards the firing of transitions, the network model obeys the standard rules of Petri nets with the particularity that it is one-safe, i.e., each place may contain at most one token. This is due to the so-called *static control contraints* [14,4,5], which only allow a statically bounded dynamic creation of processes. For instance, the following behavior "$B_1$`>>(`$B_2$`|||`$B_3$`)>>`$B_4$" is permitted, whereas recursion through parallel composition is prohibited.

Therefore, we can define a *marking* $M$ as a subset of the places of the network (i.e., $M \subseteq \mathcal{Q}$). We define the *initial marking* $M_0 \triangleq \{Q_0\}$, which expresses that, initially, only the initial place of the network has one token. We define a relation "$[M, T] \xrightarrow{m} M'$" meaning that transition $T$ can be fired from marking $M$, leading to a new marking $M'$. Classically, $[M, T] \xrightarrow{m} M'$ holds iff $\mathit{in}(T) \subseteq M$ (i.e., all input places of $T$ have a token) and $M' = (M \setminus \mathit{in}(T)) \cup \mathit{out}(T)$ (i.e., tokens move from input to output places).

**Units.** Contrary to standard Petri nets, which consist of "flat" sets of places and transitions, the places of a network are properly structured using a tree-shaped hierarchy of *units*. The set of units, which is finite, is written $\mathcal{U}$ in the above definition of a network. To each unit $U$ is associated a non-empty, finite set of places, called the *proper places of* $U$ and written $\mathit{places}(U)$, such that all sets of proper places $\{\mathit{places}(U) \mid U \in \mathcal{U}\}$ form a partition of $\mathcal{Q}$. Although units play no part in the transition relation "$[M, T] \xrightarrow{m} M'$" between markings, they satisfy an important invariant: for each marking $M$ reachable from the initial marking $M_0$ and for each unit $U$, one has $\mathit{card}\big(M \cap \mathit{places}(U)\big) \leq 1$, i.e., there is at most one token among the proper places of $U$, meaning that each unit models a (possibly inactive) sequential behavior. This invariant serves both for correctness proofs and compact memory representation of markings.

Units can be nested recursively: each unit $U$ may contain zero, one, or several units, called the *sub-units* of $U$; this is used to encapsulate sequential or concurrent sub-behaviors. There exists a *root unit* containing all other units. We define the relation "$U' \sqsubseteq U$" expressing that $U'$ is equal to $U$ or transitively contained in $U$; this relation is a complete partial order, the maximum of which is the root unit. We define $\mathit{places}^*(U) = \bigcup_{U' \sqsubseteq U} \mathit{places}(U')$ as the set of places transitively contained in $U$. For some marking $M$ reachable from $M_0$, one may have $\mathit{card}\Big(M \cap \mathit{places}^*(U)\Big) > 1$ in case of concurrency between the sub-units of $U$. Yet, for all units $U$ and $U' \sqsubseteq U$, one has $\Big(M \cap \mathit{places}(U) = \emptyset\Big) \vee \Big(M \cap \mathit{places}(U') = \emptyset\Big)$, meaning that the proper places of a unit are mutually exclusive with those of its sub-units.

Variables may be *global*, or *local* to a given unit. We define $\mathit{unit}(X)$ as the unit to which variable $X$ is attached (global variables are attached to the root unit). A variable $X$ is said to be *inherited* in all sub-units of $\mathit{unit}(X)$. To a first approximation, we will say that variable $X$ is *shared* between two units $U_1$ and $U_2$ iff $\Big(U_1 \sqsubseteq \mathit{unit}(X)\Big) \wedge \Big(U_2 \sqsubseteq \mathit{unit}(X)\Big) \wedge (U_1 \not\sqsubseteq U_2) \wedge (U_2 \not\sqsubseteq U_1)$.

**Labelled Transition Systems.** Finally, the operational semantics of the network model is defined as a *Labelled Transition System* (LTS), i.e., a tuple $\langle \Sigma, \sigma_0, \mathcal{L}, \rightarrow \rangle$ where $\Sigma$ is a set of *states*, $\sigma_0 \in \Sigma$ is the *initial state*, $\mathcal{L}$ is the set of *labels* and $\rightarrow \subseteq \Sigma \times \mathcal{L} \times \Sigma$ is the *transition relation*.

The LTS is constructed as follows. Each state of $\Sigma$ consists of a pair $\langle M, C \rangle$, with $M$ a marking and $C$ a context. The initial state $\sigma_0$ is the pair $\Big\langle M_0, \{\} \Big\rangle$, i.e., one token is in the initial place and all variables are undefined initially. Each label of $\mathcal{L}$ consists of a list $G\ v_1 \ldots v_n$, with $G$ a gate and $v_1 \ldots v_n$ a (possibly empty) list of ground values resulting from the evaluation of an offer. A transition $(\sigma_1, L, \sigma_2)$ belongs to the "$\rightarrow$" relation, which is written "$\sigma_1 \xrightarrow{L} \sigma_2$", iff

$$\frac{[M, T] \xrightarrow{m} M' \quad [C, A] \xrightarrow{c} C' \quad [C', O] \xrightarrow{o} [C'', v_1 \ldots v_n] \quad [C'', (W\,;R)] \xrightarrow{c} C'''}{\langle M, C \rangle \xrightarrow{G\ v_1 \ldots v_n} \langle M', C''' \rangle}$$

The above definition expresses that firing a transition involves several steps, each of which must execute successfully: the action is executed first, then the offer is evaluated, then the when-guard is checked, and the reaction is executed finally. In fact, the actual definition of the transition relation is more complex because there are rules to eliminate $\varepsilon$-transitions from the LTS; as mentioned before, we do not detail these rules here.

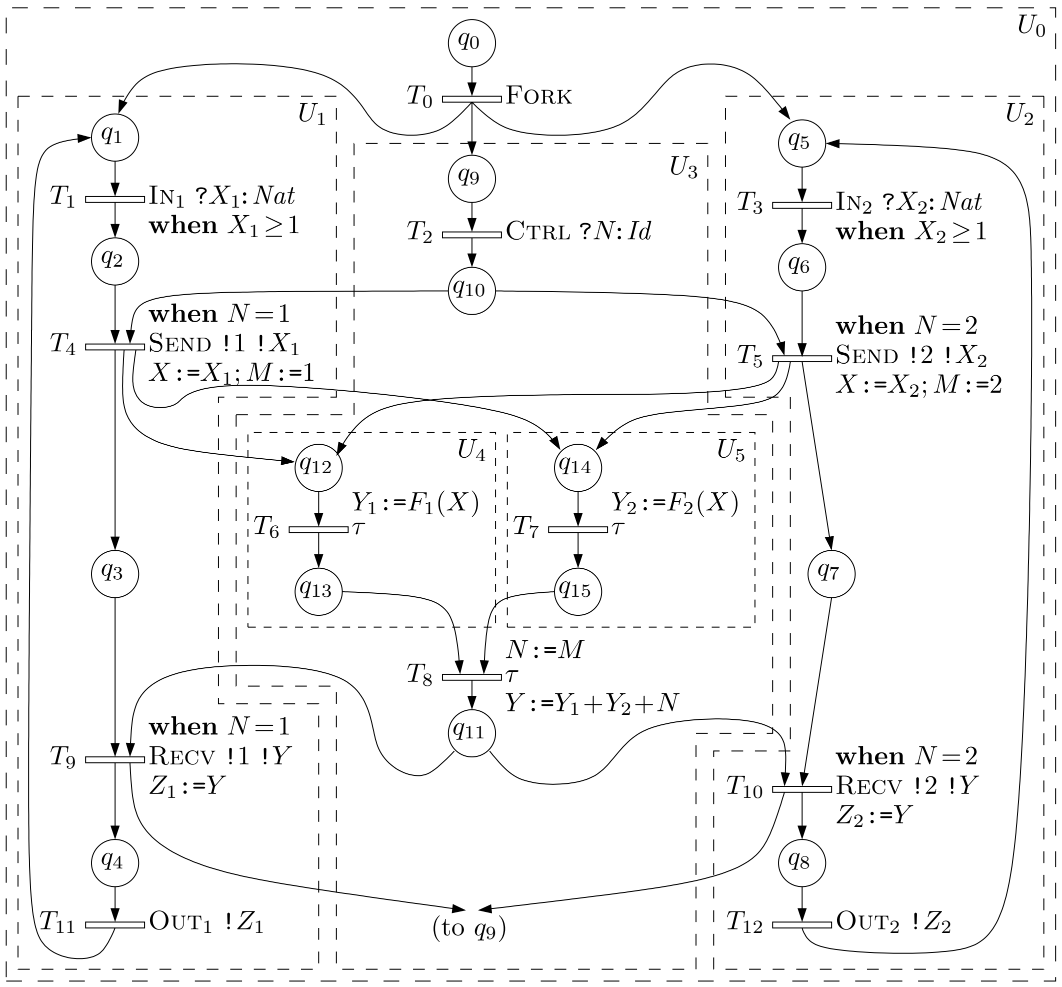

Fig. 1. Example of a network

**An Example.** Figure 1 gives an example of a network. According to Petri net graphical conventions, places and transitions are represented by circles and rectangles respectively. Dashed boxes are used to represent units. For each transition, the corresponding action, gate and offer, when-guard, and reaction are displayed (in that order) from top to bottom on the right. We omit every action, when-guard, or reaction that is equal to **none**. The variables attached to $U_1$ are $X_1$ and $Z_1$; those attached to $U_2$ are $X_2$ and $Z_2$; those attached to $U_3$ are $M$, $N$, $X$, $Y$, $Y_1$, and $Y_2$. Variable $X$ inherited from $U_3$ is shared between $U_4$ and $U_5$. Note that, contrary to a place or a variable, a transition is not attached to a particular unit, which reflects that the input and output places of a transition may belong to different units. Thus, the fact that a variable is assigned or consulted in a transition with input or output places belonging to different units does not mean that this variable is attached to each of these units, nor to the unit containing all these units (e.g., $X$ is assigned in transition $T_4$ that has input places in $U_1$ and $U_3$, but $X$ is attached to $U3$, not to $U_1$ nor $U_0$).

## 3 Local Data-Flow Analysis

In the network model, transitions constitute the equivalent of the "basic blocks" used for data-flow analysis of sequential programs. We first analyse the flow of data within each transition taken individually to characterise which variables are accessed by this transition. Our definitions are based on [9], with adaptations to take into account the latest extensions of the network model and to handle networks that already contain "**reset**" actions. We define the following sets by structural induction over the syntax of value expressions, offers, and actions:

- $\mathit{use}_v(V)$ (*resp.* $\mathit{use}_o(O)$, $\mathit{use}_a(A)$) denotes the set of variables consulted in value expression $V$ (*resp.* offer $O$, action $A$).
- $\mathit{def}_o(O)$ (*resp.* $\mathit{def}_a(A)$) denotes the set of variables assigned a defined value by offer $O$ (*resp.* action $A$).
- $\mathit{und}_a(A)$ denotes the set of variables assigned an undefined value (i.e., reset) by action $A$.
- $\mathit{use\_before\_def}_a(A)$ denotes the set of variables consulted by action $A$ and possibly modified by $A$ later (modifications, if present, should only occur after the variables have been consulted at least once).

$$\mathit{use}_v(X) \triangleq \{X\}$$
$$\mathit{use}_v\big(F(V_1,\ldots,V_n)\big) \triangleq \bigcup_{i=1}^{n} \mathit{use}_v(V_i)$$

$$\mathit{use}_o(\texttt{!}\,V) \triangleq \mathit{use}_v(V)$$
$$\mathit{use}_o(\texttt{?}X\texttt{:}S) \triangleq \emptyset$$
$$\mathit{use}_o(O_1\ldots O_n) \triangleq \bigcup_{i=1}^{n} \mathit{use}_o(O_i)$$

$$\mathit{und}_a(\textbf{reset}\ X_0,\ldots,X_n) \triangleq \{X_0,\ldots,X_n\}$$
$$\mathit{und}_a(A_1;A_2) \triangleq \big(\mathit{und}_a(A_1)\setminus \mathit{def}_a(A_2)\big) \cup \mathit{und}_a(A_2)$$
$$\mathit{und}_a(A_1\&A_2) \triangleq \mathit{und}_a(A_1) \cup \mathit{und}_a(A_2)$$
$$\text{otherwise : } \mathit{und}_a(A) \triangleq \emptyset$$

$$\mathit{def}_o(\texttt{!}\,V) \triangleq \emptyset$$
$$\mathit{def}_o(\texttt{?}X\texttt{:}S) \triangleq \{X\}$$
$$\mathit{def}_o(O_1\ldots O_n) \triangleq \bigcup_{i=1}^{n} \mathit{def}_o(O_i)$$

$$\mathit{def}_a\big(X_0,\ldots,X_n\texttt{:=}V_0,\ldots,V_n\big) \triangleq \{X_0,\ldots,X_n\}$$
$$\mathit{def}_a(\textbf{for}\ X\ \textbf{among}\ S) \triangleq \{X\}$$
$$\mathit{def}_a(A_1;A_2) \triangleq \big(\mathit{def}_a(A_1)\setminus \mathit{und}_a(A_2)\big) \cup \mathit{def}_a(A_2)$$
$$\mathit{def}_a(A_1\&A_2) \triangleq \mathit{def}_a(A_1) \cup \mathit{def}_a(A_2)$$
$$\text{otherwise : } \mathit{def}_a(A) \triangleq \emptyset$$

$$\mathit{use}_a(\textbf{when}\ V) \triangleq \mathit{use}_v(V)$$
$$\mathit{use}_a\big(X_0\ldots\texttt{:=}V_0\ldots\big) \triangleq \bigcup_{i=0}^{n} \mathit{use}_v(V_i)$$
$$\mathit{use}_a(A_1\,;A_2) \triangleq \mathit{use}_a(A_1) \cup \mathit{use}_a(A_2)$$
$$\mathit{use}_a(A_1\&A_2) \triangleq \mathit{use}_a(A_1) \cup \mathit{use}_a(A_2)$$
$$\text{otherwise : } \mathit{use}_a(A) \triangleq \emptyset$$

$$\mathit{use\_before\_def}_a(A_1\,;A_2) \triangleq \mathit{use\_before\_def}_a(A_1) \cup \big(\mathit{use\_before\_def}_a(A_2) \setminus \mathit{def}_a(A_1)\big)$$
$$\mathit{use\_before\_def}_a(A_1\&A_2) \triangleq \mathit{use\_before\_def}_a(A_1) \cup \mathit{use\_before\_def}_a(A_2)$$
$$\text{otherwise : } \mathit{use\_before\_def}_a(A) \triangleq \mathit{use}_a(A)$$

Finally, for a transition $T = \langle Q_i, Q_o, A, G, O, W, R \rangle$ and a variable $X$, we define three predicates, which will be the only local data-flow results used in subsequent analysis steps:

- $\mathit{use}(T, X)$ holds iff $X$ is consulted during the execution of $T$.
- $\mathit{def}(T, X)$ holds iff $X$ is assigned a defined value by the execution of $T$, i.e., if $X$ is defined by $A$, $O$ or $R$, and not subsequently reset.
- $\mathit{use\_before\_def}(T, X)$ holds iff $X$ is consulted during the execution of $T$ and possibly modified later (modification, if present, should only occur after $X$ has been consulted at least once).

Formally:

$$\mathit{use}(T, X) \triangleq X \in \mathit{use}_a(A) \cup \mathit{use}_o(O) \cup \mathit{use}_a(W) \cup \mathit{use}_a(R)$$

$$\mathit{def}(T, X) \triangleq X \in \Big(\big(\mathit{def}_a(A) \cup \mathit{def}_o(O)\big) \setminus \mathit{und}_a(R)\Big) \cup \mathit{def}_a(R)$$

$$\mathit{use\_before\_def}(T, X) \triangleq X \in \begin{pmatrix} \mathit{use\_before\_def}_a(A) \ \cup \ \big(\mathit{use}_o(O) \setminus \mathit{def}_a(A)\big) \ \cup \\ \Big(\mathit{use\_before\_def}_a(W\,;R) \setminus \big(\mathit{def}_a(A) \cup \mathit{def}_o(O)\big)\Big) \end{pmatrix}$$

**Example 1** For the variable $N$ in the network of Figure 1, we have: $\mathit{use}(T, N)$ for $T \in \{T_4, T_5, T_8, T_9, T_{10}\}$, $\mathit{def}(T, N)$ for $T \in \{T_2, T_8\}$, and $\mathit{use\_before\_def}(T, N)$ for $T \in \{T_4, T_5, T_9, T_{10}\}$.

## 4 Global Data-Flow Analysis

Based on local (intra-transition) data-flow predicates, we now perform global (inter-transition) data-flow analysis, the goal being to compute, for each transition $T = \langle Q_i, Q_o, A, G, O, W, R \rangle$ and for each variable $X$, a predicate $\mathit{reset}(T, X)$ expressing that it is possible to *reset variable $X$ at the end of transition $T$* (i.e., to append "**reset** $X$" at the end of $A$ if $X$ is neither defined in $O$ nor used in $O$, $W$, and $R$; or else to append "**reset** $X$" at the end of $R$). To be exact, if $X$ is an inherited shared variable, it is not always possible to insert "**reset** $X$" at the end of every transition $T$ such that $\mathit{reset}(T, X)$; this issue will be dealt with in Section 5; for now, we focus on computing $\mathit{reset}(T, X)$.

For sequential programs, the classical approach to global data-flow analysis (e.g., [15]) consists in constructing a *control-flow graph* on which boolean predicates will then be evaluated using fixed point computations. The vertices of the control-flow graph are usually the basic blocks connected by arcs expressing that two basic blocks can be executed in sequence. Since the control-flow

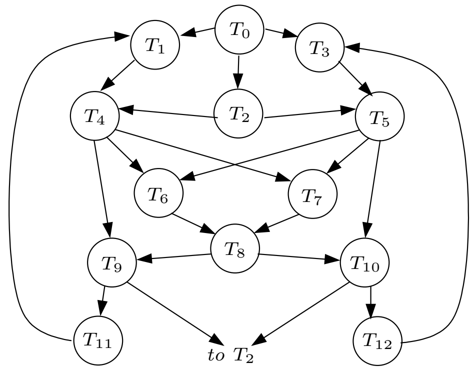

Fig. 2. CFG for Figure 1

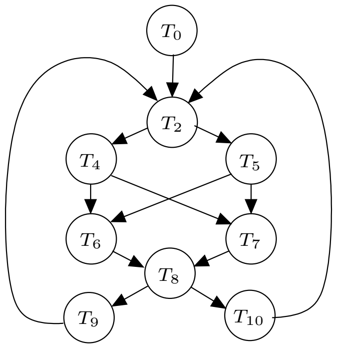

Fig. 3. $\text{CFG}_N$ for Figure 1

graph is a data-independent abstraction, it represents a superset of the possible execution paths, i.e., some paths of the control-flow graph might not exist in actual executions of the sequential program.

A significant difference between sequential programs and our setting is that networks feature concurrency. One could devise a "true concurrency" extension of data-flow analysis by evaluating the boolean predicates, not on control-flow graphs, but directly on Petri nets. Instead, following [9], we adopt an "interleaving semantics" approach that maps concurrency onto a standard control-flow graph, on which the boolean predicates can be evaluated as usual.

To abstract away concurrency from the network model, various possibilities exist, leading to different control-flow graphs. One possibility would be to base the analysis on the graph of reachable markings of the underlying Petri net; this would be accurate but costly to compute since state explosion might occur. Hence, we choose a stronger abstraction by defining the control-flow graph as the directed graph $\text{CFG} = \langle \mathcal{T}, \rightarrow \rangle$, the vertices of which correspond to the transitions of the network and such that there is an arc $T_1 \rightarrow T_2$ iff $\mathit{out}(T_1) \cap \mathit{in}(T_2) \neq \emptyset$.

**Example 2** The CFG corresponding to the network of Figure 1 is shown in Figure 2.

Instead of constructing a unique CFG valid for all variables, [9] suggests to build, for each variable $X$, a dedicated control-flow graph $\text{CFG}_X$, which is a subset of CFG containing only the execution paths relevant to $X$ (nowadays, this would be called "slicing"). According to [9, § 4.3.3], such a restricted control-flow graph increases the algorithmic efficiency; in our experience, it also gives more precise data-flow results.

To define $\text{CFG}_X$ formally, we need two auxiliary definitions. Let $\mathit{trans}(U) \triangleq \Big\{T \mid \Big(\mathit{in}(T) \cup \mathit{out}(T)\Big) \cap \mathit{places}^*(U) \neq \emptyset\Big\}$ be the set of transitions with an input or an output place in unit $U$. Let $\mathit{scope}(X)$ be (an upper-approximation of) the set of places through which the data-flow for variable $X$ passes. Initially, we define $scope(X)$ as $places^*\Big(unit(X)\Big)$, which is the set of all places in the unit to which $X$ is attached; we will see later that some places might be removed from $scope(X)$ during the analysis.

We now define $\text{CFG}_X$ as the directed graph $\langle \mathcal{T}_X, \rightarrow_X \rangle$ with the set of vertices $\mathcal{T}_X \triangleq trans\Big(unit(X)\Big)$ and such that there exists an arc $T_1 \rightarrow_X T_2$ between $T_1$ and $T_2$ iff $out(T_1) \cap in(T_2) \cap scope(X) \neq \emptyset$. For $T \in \mathcal{T}_X$, we define $succ_X(T) \triangleq \{T' \in \mathcal{T}_X \mid T \rightarrow_X T'\}$ and $pred_X(T) \triangleq \{T' \in \mathcal{T}_X \mid T' \rightarrow_X T\}$.

**Example 3** Figure 3 shows $\text{CFG}_N$ for the network of Figure 1 and variable $N$; notice that $T_4 \rightarrow T_9$, but not $T_4 \rightarrow_N T_9$.

Following the classical definition of "live" variables (e.g., [15, pages 631–632]), we define, for $T \in \mathcal{T}_X$, the following predicate:

$$live(T, X) \triangleq \bigvee_{T' \in succ_X(T)} use\_before\_def(T', X) \vee \Big(live(T', X) \wedge \neg def(T', X)\Big)$$

that holds iff after $T$ it is possible, by following the arcs of $\text{CFG}_X$, to reach a transition $T'$ that uses $X$ before any modification of $X$. Notice that the definition above could also be expressed by the following CTL formula: "$EX(E\ \neg def\ U\ use\_before\_def)$" where "$EX\varphi$" stands for "exists next" and "$E\varphi_1 U \varphi_2$" for "exists until", and where $def$ (respectively $use\_before\_def$) holds for a transition $T$ iff $def(T, X)$ (respectively $use\_before\_def(T, X)$) holds. Since we are interested in the truth value of $live$ at the end of transitions, the CTL formula does not require that $use\_before\_def$ holds in the first "state" (i.e., transition). For a given $X$, the set $\{T \in \mathcal{T}_X \mid live(T, X)\}$ is computed as a backward least fixed point.

We could now, as in [10–12], define $reset(T, X) \triangleq \neg live(T, X)$. Unfortunately, this simple approach inserts superfluous resets, e.g., before a variable is initialised or at places where a variable has already been reset. For this reason, one needs an additional predicate:

$$available(T, X) \triangleq def(T, X) \vee \Big(\bigvee_{T' \in pred_X(T)} \Big(live(T', X) \wedge available(T', X)\Big)\Big)$$

that holds iff $T$ can be reached from some transition that assigns $X$ a defined value, by following the arcs of $\text{CFG}_X$ and ensuring that $X$ remains alive all along the path. Notice that the definition above could also be expressed by the following CTL formula: "$E\ live\ S\ def$" where "$E\varphi_1 S \varphi_2$" stands for "exists since" and where $def$ (respectively $live$) holds for a transition $T$ iff $def(T, X)$ (respectively $live(T, X)$) holds. Contrary to the CTL formula for $live$, we do not need an equivalent of the outermost "$EX$" modality. [9] uses a similar definition without the $live(T', X)$ condition, and thus introduces useless resets for variables that are already reset. For a given $X$, the set $\{T \in \mathcal{T}_X \mid available(T, X)\}$ is computed as a forward least fixed point.

Finally, we define

$$reset(T, X) \triangleq available(T, X) \wedge \neg live(T, X)$$

expressing that a variable can be reset where it is both available and dead.

**Example 4** Considering the network of Figure 1 and focusing on its variable $N$, we have $\{T \mid live(T, N)\} = \{T_2, T_8\}$ and $\{T \mid available(T, N)\} = \{T_2, T_4, T_5, T_8, T_9, T_{10}\}$. Thus, we can insert "**reset** $N$" at the end of $T_4$, $T_5$, $T_9$, and $T_{10}$. Using the definition of [10,11], one would insert a superfluous "**reset** $N$" at the end of $T_0$, $T_6$, and $T_7$. Using the definition of [9], one would insert a superfluous "**reset** $N$" at the end of $T_6$ and $T_7$. Using CFG instead of CFG$_N$ would give $\{T \mid live(T, N)\} = \{T_0 \ldots T_5, T_8 \ldots T_{12}\}$ and $\{T \mid available(T, N)\} = \{T_1 \ldots T_5, T_8 \ldots T_{12}\}$, so that no "**reset** $N$" at all would be inserted.

## 5 Treatment of Inherited Shared Variables

**Issues when Resetting Shared Variables.** Experimenting with the approach of [9], we noticed that systematic insertion of a "**reset** $X$" at the end of every transition $T$ such that $reset(T, X)$ could produce either incorrect results (i.e., an LTS which is not strongly bisimilar to the original specification) or run-time errors while generating the LTS (i.e., accessing a variable that has been reset).

**Example 5** In the network of Figure 1, there exists a fireable sequence of transitions $T_0, T_1, T_2, T_4, T_6, T_7$. Although $reset(T_6, X)$ is true, one should not reset $X$ at the end of $T_6$, because $X$ is used just after in $T_7$. Clearly, the problem is that $T_6$ and $T_7$ are two "concurrent" transitions sharing the same variable $X$. This was no problem as long as $X$ was only read by both transitions, but as soon as one transition (here, $T_6$) tries to reset $X$, it affects the other transition (here, $T_7$).

So, insertion of resets turns a read-only shared variable into a read/write shared variable, possibly creating read/write conflicts as in a standard reader-writer problem. The sole difference is that resets do not provoke write/write conflicts (concurrent resets assign a variable the same undefined value).

To avoid the problem, a simple solution consists in never resetting inherited shared variables (as in the IF tool set [13]). Unfortunately, opportunities for valuable state space reduction are missed by doing so.

**Example 6** As shown in Figure 4(a) and 4(b), the LTS generated for the LOTOS behavior "$G?X:\mathbf{bit};(G_1!X;\mathbf{stop}|||G_2!X;\mathbf{stop})$" has 9 states if the inherited shared variable $X$ is not reset, and only 8 states if $X$ is reset

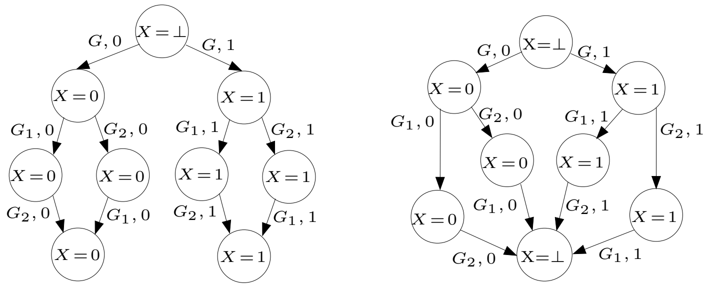

Fig. 4. Lts (a) without reset, (b) with correct resets, and (c) with incorrect resets

after firing transitions $G_1\,!\,X$ and $G_2\,!\,X$. State space reduction would be more substantial if both occurrences of "**stop**" were replaced by two complex behaviors $B_1$ and $B_2$ in which the value of $X$ is not used. Figure 4(c) shows the incorrect Lts obtained by resetting $X$ to 0 after each transition $G_1\,!\,X$ and $G_2\,!\,X$.

**Duplication of Variables.** The deep reason behind the issues when resetting inherited shared variables is that the control-flow graphs Cfg and Cfg$_X$ defined in Section 4 are nothing but approximations. Their definitions follow the place-transition paths in the network, which has the effect of handling similarly nondeterministic choice (i.e., a place with several outgoing transitions) and asynchronous concurrency (i.e., a transition with several output places). Indeed, both Lotos behaviors "$G$;($B_1$|||$B_2$)" and "$G$;($B_1$[]$B_2$)" have the same Cfg. These approximations produce compact control-flow graphs, but are only correct in the absence of data dependencies (caused by inherited shared variables) between "concurrent" transitions.

To address the problem, we introduce the notion of *variable duplication*. For an inherited variable $X$ shared between two concurrent behaviors $B_1$ and $B_2$, duplication consists in replacing in one behavior (say, $B_2$) all occurrences of $X$ with a local copy $X'$ initialised to $X$ at the beginning of $B_2$. This new variable $X'$ can be safely reset in $B_2$ without creating read/write conflicts with $B_1$. A proper application of duplication can remove all data dependencies between "concurrent" transitions, hence ensuring correctness of our global data-flow analysis approximations. It also enables the desired state space reductions.

**Example 7** In Example 6, duplicating $X$ in "$G_2\,!\,X$;**stop**" yields the Lotos behavior "$G$?$X$:**bit**;**let** $X'$:**bit**=$X$ **in** ($G_1\,!\,X$;**stop**|||$G_2\,!\,X'$;**stop**)", in which it is possible to reset $X$ after the $G_1\,!\,X$ transition and $X'$ after the $G_2\,!\,X'$ transition; this precisely gives the optimal Lts shown in Figure 4(b). Note that it is not necessary to duplicate $X$ in "$G_1\,!\,X$;**stop**".

Instead of duplicating variables at the Lotos source level, as in the above example, we prefer duplicating them in the network model, the complexity of which has already been reduced by detecting constants, removing unused variables, identifying variables local to a transition, etc. Taking into account that concurrent processes are represented by units, we define the *duplication of a variable $X$ in a unit $U$*, with $U \sqsubseteq \mathit{unit}(X)$ and $U \neq \mathit{unit}(X)$, as the operation consisting of the following steps:

- creating a new variable $X'$ of the same sort as $X$,
- attaching $X'$ to $U$ (whereas $X$ is attached to $\mathit{unit}(X)$),
- initializing $\mathit{scope}(X')$ to $\mathit{places}^*(U)$,
- replacing all occurrences of $X$ in the transitions of $\mathit{trans}(U)$ by $X'$,
- adding an assignment "$X'$`:=`$X$" at the end of all transitions $T \in \mathit{entry}(U)$ such that $\mathit{live}(T, X)$, where $\mathit{entry}(U) \triangleq \{T \in \mathit{trans}(U) \mid \mathit{in}(T) \cap \mathit{places}^*(U) = \emptyset\}$ is the set of transitions "entering" $U$, and
- removing from $\mathit{scope}(X)$ all places of $U$, i.e., $\mathit{places}^*(U)$, since after duplication, it is no longer needed to examine the data-flow for $X$ in $U$.

In general, several duplications may be needed to remove all read/write conflicts on a shared variable $X$. On the one hand, if $X$ is shared between $n$ concurrent behaviors, $(n-1)$ duplications of $X$ may be necessary. On the other hand, each new variable $X'$ duplicating $X$ might itself be shared between concurrent sub-units, so that duplications of $X'$ may also be required.

For compactness, we do not implement $\mathit{scope}(X)$ as a decreasing set of places, but rather as an increasing set of units $\{U_1, \ldots, U_n\}$, the places of which must be excluded from the scope of $X$. Formally, $\mathit{scope}(X) = \mathit{places}^*\big(\mathit{unit}(X)\big) \setminus \bigcup_{i=1}^{n} \mathit{places}^*(U_i)$. This set is initially empty, and each duplication of $X$ adds a new unit to it.

**Concurrency Relation between Units.** We now formalise the notion of "concurrent units". Ideally, two units $U_i$ and $U_j$ are concurrent if there exists a reachable state $\langle M, C\rangle$ in the corresponding LTS such that the two sets of places $\big(M \cap \mathit{places}^*(U_i)\big)$ and $\big(M \cap \mathit{places}^*(U_j)\big)$ are both non-empty and disjoint (meaning that $U_i$ and $U_j$ are "separate" and simultaneously "active" in marking $M$).

**Example 8** In the LOTOS behavior "`(`$B_1$`|||`$B_2$`)>>(`$B_3$`|||`$B_4$`)`", units $U_1$ and $U_2$ corresponding to $B_1$ and $B_2$ are concurrent, units $U_3$ and $U_4$ corresponding to $B_3$ and $B_4$ are also concurrent, but neither $U_1$ nor $U_2$ is concurrent with either $U_3$ or $U_4$.

Practically, to avoid enumerating all states of the LTS, we need a relation "$U_i \parallel U_j$" that is an upper-approximation of the ideal definition above, i.e., $U_i$ and $U_j$ concurrent implies $U_i \parallel U_j$. There are various ways of computing such an approximation, with different tradeoffs between computational cost and accuracy.

- Instead of basing the definition of concurrent units on reachable states (i.e., pairs $\langle M, C \rangle$ of a marking $M$ and a context $C$), one could consider reachable markings (regardless of contexts) and define "$U_i \parallel U_j$" iff there exists a reachable marking $M \in \mathcal{M}$ such that both sets $\Big(M \cap \mathit{places}^*(U_i)\Big)$ and $\Big(M \cap \mathit{places}^*(U_j)\Big)$ are both non-empty and disjoint. This relation can be computed using symbolic techniques (BDD) to represent the graph of reachable markings. However, since even the exploration of reachable markings may face state explosion, we prefer a second approach, which we present first and compare to a BDD-based approach later.
- We base our definition on an abstraction function $\alpha : \mathcal{Q} \rightarrow \{1, \ldots, N\}$ ($N$ being the number of units in the network) that maps all the proper places of each unit to the same number: $\Big(\forall Q \in \mathit{places}(U_i)\Big)\alpha(Q) \triangleq i$. We extend $\alpha$ to sets of places by defining $\widehat{\alpha} : \wp(\mathcal{Q}) \rightarrow \wp\Big(\{1, \ldots, N\}\Big)$ such that $\widehat{\alpha}\Big(\{Q_1, \ldots, Q_n\}\Big) \triangleq \{\alpha(Q_1), \ldots, \alpha(Q_n)\}$. We then use $\alpha$ and $\widehat{\alpha}$ to "quotient" the network, yielding a Petri net with $N$ places numbered from 1 to $N$, with initial place $\alpha(Q_0)$ ($Q_0$ being the initial place of the network), and which possesses, for each transition $T$ in the network, a corresponding transition $t$ such that $\mathit{in}(t) \triangleq \widehat{\alpha}\Big(\mathit{in}(T)\Big)$ and $\mathit{out}(t) \triangleq \widehat{\alpha}\Big(\mathit{out}(T)\Big)$. "Self-looping" transitions such that $\mathit{in}(t) = \mathit{out}(t)$, as well as transitions identical to another one, can be removed. As the number of units is usually small compared to the number of places, one can easily generate the set $\mathcal{M}_\alpha$ of all reachable markings for the quotient Petri net. Finally, we define $U_i \parallel U_j$ iff there exists $M \in \mathcal{M}_\alpha$ such that both sets $\Big(M \cap \widehat{\alpha}(\mathit{places}^*(U_i))\Big)$ and $\Big(M \cap \widehat{\alpha}(\mathit{places}^*(U_j))\Big)$ are not empty and disjoint.

Notice that in both cases, $U_i \parallel U_j$ implies $U_i \neq U_j$, $U_i \not\sqsubseteq U_j$, and $U_j \not\sqsubseteq U_i$.

In experiments with 561 Lotos specifications, we found only one example where the latter approach (based on the quotient network) yielded less precise results than the former approach (based on the symbolic computation of the reachable markings). In this case, the number of pairs $(U_i, U_j)$ such that $U_i \parallel U_j$ was increased by only 4%. On the other hand, while the run-time for the second approach never exceeded one second, the run-time for the first approach required up to 1 hour and 41 minutes (on a SPARC Blade 100, using the CUDD library for BDDs).

**Conflicts between Units.** For two units $U_i$ and $U_j$ such that $U_i \parallel U_j$, let $\mathit{ancestor}(U_i, U_j)$ denote the largest unit $U$ such that $U_i \sqsubseteq U$ and $U_j \not\sqsubseteq U$ and let $\mathit{link}(U_i, U_j)$ denote the set of transitions "connecting" the ancestors of $U_i$ and those of $U_j$. Formally:

$$\mathit{link}(U_i, U_j) \triangleq \mathit{trans}\Big(\mathit{ancestor}(U_i, U_j)\Big) \cap \mathit{trans}\Big(\mathit{ancestor}(U_j, U_i)\Big)$$

To characterise whether two units $U_i$ and $U_j$ are in conflict for variable $X$ in $\mathit{scope}(X)$ according to given values of predicates *use* and *reset*, we define the predicate:

$$\begin{aligned}
&\mathit{conflict}(U_i, U_j, X, \mathit{use}, \mathit{reset}) \triangleq \\
&\quad \mathit{places}(U_i) \subseteq \mathit{scope}(X) \;\wedge\; \mathit{places}(U_j) \subseteq \mathit{scope}(X) \;\wedge\; U_i \,\|\, U_j \;\wedge \\
&\quad \big(\exists T_i \in \mathit{trans}(U_i) \setminus \mathit{link}(U_i, U_j)\big) \;\; \big(\exists T_j \in \mathit{trans}(U_j) \setminus \mathit{link}(U_i, U_j)\big) \\
&\quad \big(\mathit{reset}(T_i, X) \wedge \mathit{use}(T_j, X)\big) \vee \big(\mathit{reset}(T_j, X) \wedge \mathit{use}(T_i, X)\big)
\end{aligned}$$

Intuitively, units $U_i$ and $U_j$ are in conflict for $X$ if there exist two "independent" transitions $T_i$ and $T_j$ likely to create a read/write conflict on $X$. To avoid irrelevant conflicts (and thus, unnecessary duplications), one can dismiss the transitions of $\mathit{link}(U_i, U_j)$, i.e., the transitions linking the ancestor of $U_i$ with that of $U_j$, since the potential impact of these transitions on the data-flow for $X$ has already been considered when constructing $\mathrm{C{\small FG}}_X$ and computing *reset* — based on the observation that $\mathit{link}(U_i, U_j) \subseteq \mathit{trans}\big(\mathit{unit}(X)\big)$.

We finally define, for given values of predicates *use* and *reset*, the *unit conflict graph for variable* $X$, noted $\mathrm{U{\small CG}}_X$, as the undirected graph whose vertices are the units of $\mathit{unit}(X)$ such that there is an edge between $U_i$ and $U_j$ iff $\mathit{conflict}(U_i, U_j, X, \mathit{use}, \mathit{reset})$.

**Complete Algorithm.** The algorithm shown in Figure 5 operates as follows. *VARS* denotes the set of all variables in the network, which might be extended progressively with new, duplicated variables. All the variables $X$ in *VARS* are processed individually, one at a time, in an unspecified order. For a given $X$, the algorithm performs local and global data-flow analysis, then builds $\mathrm{U{\small CG}}_X$. If $\mathrm{U{\small CG}}_X$ has no edge, $X$ needs not be duplicated and "**reset** $X$" can be inserted at the end of every transition $T \in \mathit{trans}\big(\mathit{unit}(X)\big)$ such that $\mathit{reset}(T, X)$. Otherwise, $X$ must be duplicated in one or several units to solve read/write conflicts. This adds to *VARS* one or several new variables $X'$, which will be later analysed as if they were genuine variables of the network (i.e., to insert resets for $X'$ and/or to solve read/write conflicts that may still exist for $X'$). Everytime a new variable $X'$ is created to duplicate $X$, the data-flow predicates for $X$ and then $\mathrm{U{\small CG}}_X$ are recomputed, as duplication modifies the network by removing occurrences (definitions, uses, and resets) of $X$ and adding new assignments of the form $X'$:=$X$, and restricts $\mathit{scope}(X)$, thus modifying $\rightarrow_X$ and $\mathrm{C{\small FG}}_X$.

Since each creation of a new variable $X'$ increases the size of the state representation (thus raising the memory cost of model checking), it is desirable to minimise the number of duplications by choosing carefully in which unit(s)

```
 1. compute the relation $U_i \| U_j$ and $link(U_i, U_j)$ (cf. Section 5)
 2. **forall** $X \in \mathcal{X}$ **do** initialize $scope(X)$ (cf. Section 4)
 3. $VARS := \mathcal{X}$
 4. **while** $VARS \neq \emptyset$ **do**
 5. **begin**
 6.    $X := one\_of(VARS)$
 7.    $VARS := VARS \setminus \{X\}$
 8.    **repeat**
 9.       **forall** $T \in trans\big(unit(X)\big)$ **do**
10.          compute $use(T, X)$, $def(T, X)$, and $use\_before\_def(T, X)$ (cf. Section 3)
11.       **forall** $T \in trans\big(unit(X)\big)$ **do**
12.          compute $reset(T, X)$ (cf. Section 4)
13.       compute $conflict(U_i, U_j, X, use, reset)$ and $\textsc{Ucg}_X$ (cf. Section 5)
14.       compute $U := best\_of(\textsc{Ucg}_X)$ (cf. Section 5)
15.       **if** $U \neq \perp$ **then**
16.       **begin**
17.          duplicate $X$ in $U$ by creating a new variable $X'$ (cf. Section 5)
18.          $VARS := VARS \cup \{X'\}$
19.       **end**
20.    **until** $U = \perp$
21.    – – at this point, there is no more conflict on $X$
22.    **forall** $T \in trans(X)$ **such_that** $reset(T, X)$ **do**
23.       insert "**reset** $X$" at the end of $T$ (cf. Section 4)
24. **end**
```

Fig. 5. Complete algorithm

$X$ will be duplicated. Based on the observation that duplicating $X$ in some unit $U$ removes from $\textsc{Ucg}_X$ all conflict edges connected to $U$, the problem is similar to the classical NP-complete "vertex cover problem", except that each edge removal provokes the recalculation of $\textsc{Ucg}_X$. To select the unit (written $best\_of(\textsc{Ucg}_X)$) in which $X$ should be duplicated first, we adopt a combination of top-down and greedy strategies by choosing, among the units of $\textsc{Ucg}_X$ having at least one edge, the outermost ones. If there are several such units, we then choose one having a maximal number of edges. If $\textsc{Ucg}_X$ has no edges, $best\_of(\textsc{Ucg}_X)$ returns $\perp$.

For a given variable $X$, the "**repeat**" loop (line 8) terminates because of fixed point convergence of global data-flow analysis and because each duplication of $X$ in $U$ (line 17) removes all places of $places^*(U)$ from $scope(X)$. By definition, each unit contains at least one place, and thus $scope(X)$ strictly decreases with respect to set inclusion at each iteration but the last one.[4]

[4] In an earlier formalization [16] of our algorithm (but not in its actual implementation), $scope(X)$ remained constant, with the following consequence: after duplicating

The outermost "**while**" loop (line 4), which removes one variable $X$ from $\mathit{VARS}$ but possibly inserts new variables $X'$ in this set, also terminates. Let $\delta(U)$ be the nesting depth of unit $U$ in the unit hierarchy, i.e., the number of parent units containing $U$ (the root unit having depth 0). Let $L = \max\{\delta(U) \mid U \in Us\}$ be the maximal nesting depth, and let $\Delta(\mathit{VARS})$ be the vector $(n_0, \ldots, n_L)$ such that $(\forall i) n_i = \mathit{card}\{X \in \mathit{VARS} \mid \delta\Big(\mathit{unit}(X)\Big) = i\}$. At each iteration of the outermost loop, $\Delta(\mathit{VARS})$ strictly decreases according to the lexicographic ordering on integer vectors of length $L$, as all variables $X'$ created to duplicate $X$ are attached to units strictly included in $\mathit{unit}(X)$, i.e., $\delta\Big(\mathit{unit}(X')\Big) < \delta\Big(\mathit{unit}(X)\Big)$.

## 6 Experimental Results

To assess our approach, we took as a basis the "standard" version 6.2 of Cæsar, from which we derived a "prototype" version of Cæsar implementing our algorithm. We then compared this prototype against the "standard" version itself. We performed all our measurements on a Sun SPARC Blade 100 with 1.6 GB RAM. [5]

As mentioned in Section 1, all versions of Cæsar since 1992 already reset variables, but in a more limited way (using "syntactic" techniques to identify process termination) than the approach presented in this paper. Therefore, the results below do not reflect the entire benefits of our approach, but only its improvements over the "syntactic" technique already implemented in Cæsar. For instance, if a variable of a process is alive until the process terminates, our prototype inserts reset(s) at the same point(s) as before, and thus brings no reduction with respect to the "standard" version of Cæsar.

We based our experiments on a collection of Lotos specifications accumulated over years during the development of Cadp. Many of them correspond to "real world" applications and none of them was written specifically to illustrate the effectiveness of our approach. [6]

$X$ in $U$ and creating the new variable $X'$, $X$ would be still available, but could be dead, at the points where $X'$ is initialized to $X$. Thus, at the next iteration, $X$ would appear to be resettable at these points, yet still with the same conflicts.

[5] The results presented in the present section differ from those reported in [16], which were based on an earlier version 6.1 of Cæsar. The newer version 6.2 brings speed improvement and, by default, eliminates all "dead" Petri net transitions using Binary Decision Diagrams, thus yielding simpler networks and possibly smaller labelled transition systems.

[6] The results presented in the present section also differ from those reported in [16], since we use here a larger set of Lotos specifications.

We first considered a collection of 289 value-passing LOTOS specifications for which the entire state space could be generated with the "standard" version of CÆSAR. For 130 examples out of 289 (45%), our approach reduced the state space (still preserving strong bisimulation) by a mean factor of 15.6 (with a maximum of 414) as regards the number of states, and a mean factor of 20.3 (with a maximum of 516) as regards the number of transitions. For none of the other 159 examples did our prototype increase the state space.

Then, we considered 3 new, "industrial" LOTOS specifications, for which our prototype could generate the corresponding state space entirely, whereas the "standard" version of CÆSAR would fail due to lack of memory. For one of these examples, the "standard" version stopped after producing an incomplete state space with more than 9 million states, while our prototype generated an LTS with 820 states and 1,500 transitions (leading to a reduction factor greater than $10^4$).

We then extended our set of 289+3 examples with 40 new, large examples for which our prototype is still unable to generate the entire state space. On these 332 examples, variable duplication occurred in only 27 cases (8.1%), for which it increased the memory size needed to represent a state by 28% in average. However, by refining our approach, this increase of 28% was reduced to 4%. The reason is the following. In some cases, duplication may introduce "useless" variables. For instance, consider the following LOTOS behavior "$G?X:S;(B_1 \,|||\, B_2)$" such that the inherited variable $X$ is used in $B_1$ but not in $B_2$. According to the definitions of Section 4, a "**reset** $X$" should be inserted at the very beginning of $B_2$, but this would create a read/write conflict with $B_1$. Therefore, our algorithm may duplicate $X$ as $X'$ in $B_2$, thus creating a useless variable $X'$ in $B_2$. This issue can be solved in two ways: we can add extra rules to our algorithm so as not to create a new variable $X'$ when duplicating a variable $X$ in a unit where $X$ is not used; or we can keep our algorithm unchanged and rely on other optimizations implemented in CÆSAR for removing useless variables (those created by duplication being only a particular case of a more general problem).

In principle, the insertion of reset(s) does not lead automatically to state space reduction (e.g., when the current value of the variable to be reset is already a pattern of 0-bits), and duplication may increase the state size. In practice, however, on all examples for which the "standard" version of CÆSAR could generate the LTS entirely, we measured that, on average, the increased memory cost of state representation was more than outweighed by the global reduction in the number of states so that the memory requirements were reduced by an average factor of 7.4.

As regards execution time, we observed that our approach divides by a factor of 10 the total execution time needed to generate all LTSs corresponding to the

289 examples mentioned above. Although our approach increases by 10% the cumulated time of the initial phases of Cæsar (parsing and type-checking of the Lotos specification, construction and optimization of the network model, generation and compilation of the C program that will generate the Lts), this small overhead is absolutely outweighed by the benefits of state space reduction during the last phase (i.e., when generating the Lts).

## 7 Conclusion

This paper has shown how state space reduction based on a general (i.e., not merely "syntactic") analysis of dead variables can be applied to process algebra specifications. Our approach requires two steps.

First, the process algebra specifications are compiled into an intermediate network model based on Petri nets extended with state variables that can be consulted and modified by actions attached to the transitions. The network model presented in this paper is used in the latest version of the Cæsar compiler and is, we believe, general enough to handle other process algebras than Lotos.

Then, data-flow analysis is performed on this network to determine automatically where variable resets can be inserted. This analysis generalizes the "syntactic" technique (resetting variables of a process upon its termination) implemented in Cæsar since 1992. It handles shared read-only variables inherited from parent processes, an issue which so far prevented the approach of [9] from being included into the official releases of Cæsar.

Compared to related work, our network model features a hierarchy of nested processes, where other approaches are usually restricted to a flat collection of communicating automata. Also, our data-flow analysis uses two passes (backward, then forward fixed point computations) in order to introduce no more variable resets than necessary.

Experiments conducted on several hundreds of realistic Lotos specifications indicate that state space reduction is frequent (45% of examples) and can reach several orders of magnitude (e.g., $10^4$). Additionally, state space reduction makes Cæsar ten times faster when processing the complete set of examples.

As regards future work, one may wish to study the finer analysis proposed in [12], namely "*partially dead*" instead of dead variables. To reset a variable $X$ at some point, this approach does not require $X$ to be dead on all subsequent execution paths but only on those for which a condition $C$ holds. [12] reports that, on one example among five, this approach allowed better state space reductions. It would be interesting to assess the potential gains of this approach on the comprehensive set of Lotos specifications used in Section 6 of the present paper.

Further open issues (not addressed in this paper, since they are beyond the scope of the Cæsar compiler for Lotos) are data-flow analysis in presence of dynamic creation/destruction of processes (arising from recursion through parallel composition) and data-flow analysis for shared read/write variables (in which case duplication is no longer possible).

*Acknowledgements.*

We are grateful to David Champelovier, Nicolas Descoubes, Frédéric Lang, Radu Mateescu, and the anonymous referees for helpful comments on this paper.

## References

[1] S. Graf, J.-L. Richier, C. Rodríguez, J. Voiron, What are the limits of model checking methods for the verification of real life protocols?, in: J. Sifakis (Ed.), Proceedings of the 1st Workshop on Automatic Verification Methods for Finite State Systems (Grenoble, France), Vol. 407 of Lecture Notes in Computer Science, Springer Verlag, 1989, pp. 275–285.

[2] R. Melton, D. L. Dill, Murphi Annotated Reference Manual, release 3.1, Updated by C. Norris Ip and Ulrich Stern. Available at `http://verify.stanford.edu/dill/Murphi/Murphi3.1/doc/User.Manual` (1996).

[3] 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, Geneva (Sep. 1989).

[4] H. Garavel, Compilation et vérification de programmes LOTOS, Thèse de doctorat, Université Joseph Fourier, Grenoble (Nov. 1989).

[5] H. Garavel, J. Sifakis, Compilation and verification of LOTOS specifications, in: L. Logrippo, R. L. Probert, H. Ural (Eds.), Proceedings of the 10th International Symposium on Protocol Specification, Testing and Verification (Ottawa, Canada), IFIP, North Holland Publishing Company, 1990, pp. 379–394.

[6] H. Garavel, F. Lang, R. Mateescu, An overview of CADP 2001, European Association for Software Science and Technology (EASST) Newsletter 4 (2002) 13–24, also available as INRIA Technical Report RT-0254 (December 2001).

[7] Y. Dong, C. R. Ramakrishnan, An optimizing compiler for efficient model checking, in: J. Wu, S. T. Chanson, Q. Gao (Eds.), Formal Methods for Protocol Engineering and Distributed Systems, Proceedings of the IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols FORTE XII and Protocol Specification, Testing and Verification PSTV XIX (Bejing, China), Kluwer Academic Publishers, 1999, pp. 241–256.

[8] G. J. Holzmann, The engineering of a model checker: The Gnu i-Protocol case study revisited, in: D. Dams, R. Gerth, S. Leue, M. Massink (Eds.), Theoretical and Practical Aspects of SPIN Model Checking, Proceedings of the 5th and 6th International SPIN Workshops (Trento, Italy / Toulouse, France), Vol. 1680 of Lecture Notes in Computer Science, Springer Verlag, 1999, pp. 232–244.

[9] J. Galvez Londono, Analyse de flot de données dans un système parallèle, Mémoire de DEA, Institut National Polytechnique de Grenoble and Université Joseph Fourier, Grenoble, defended before a jury composed of Hubert Garavel, Farid Ouabdesselam, Claude Puech and Jacques Voiron (Jun. 22, 1993).

[10] M. Bozga, J.-C. Fernandez, L. Ghirvu, State space reduction based on live variables analysis, in: A. Cortesi, G. Filé (Eds.), Proceedings of the 6th International Symposium on Static Analysis SAS '99 (Venice, Italy), Vol. 1694 of Lecture Notes in Computer Science, Springer Verlag, 1999, pp. 164–178.

[11] J.-C. Fernandez, M. Bozga, L. Ghirvu, State space reduction based on live variables analysis, Science of Computer Programming 47 (2–3) (2003) 203–220.

[12] K. Yorav, O. Grumberg, Static analysis for state space reductions preserving temporal logics, Formal Methods in System Design 25 (1) (2004) 67–96.

[13] M. Bozga, J.-C. Fernandez, L. Ghirvu, S. Graf, J.-P. Krimm, L. Mounier, IF: An intermediate representation and validation environment for timed asynchronous systems, in: J. Wing, J. Woodcock (Eds.), Proceedings of World Congress on Formal Methods in the Development of Computing Systems FM'99 (Toulouse, France), Springer Verlag, 1999.

[14] G. Ailloud, Verification in ECRINS of LOTOS programs, in: Towards Practical Verification of LOTOS specifications – ESPRIT/SEDOS/C2/N89.2 – Second Year Project Report, ESPRIT/SEDOS/C2/N48.1, Universiteit Twente, Enschede, 1986.

[15] A. V. Aho, R. Sethi, J. D. Ullman, Compilers: Principles, Techniques and Tools, Addison-Wesley, 1986.

[16] H. Garavel, W. Serwe, State space reduction for process algebra specifications, in: C. Rattray, S. Maharaj, C. Shankland (Eds.), Proceedings of the 10th International Conference on Algebraic Methodology and Software Technology AMAST'2004 (Stirling, Scotland, UK), Vol. 3116 of Lecture Notes in Computer Science, Springer Verlag, 2004, pp. 164–180.