State Space Reduction for Process Algebra Specifications

Hubert Garavel and Wendelin Serwe

INRIA Rhône-Alpes / VASY
655, avenue de l'Europe
F-38330 Montbonnot St Martin, France
{Hubert.Garavel,Wendelin.Serwe}@inria.fr

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 children processes inherit from their parents.

1 Introduction

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

Among the various approaches to avoid state explosion, it has been known for long (e.g. [11]) 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 to distinguish between states that only differ by the values of so-called *dead variables*, i.e., variables that will no longer 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. [11]). Some languages even provide a dedicated instruction for resetting variables (e.g. [14, §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 [13], μCRL, 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, which is applicable to process algebras and which 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 proposed in [8, 10], 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 [9]). This paper presents the most recent version of the network model, which adds to the model of [8, 10] 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 other process algebras than LOTOS.

In a second step, resets are introduced, not at the source level (process algebraic specifications), but in the intermediate model, by attaching the 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 [6], 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 [7] in order to introduce variable resets everywhere it would be possible, including in cyclic processes. A key idea in [7] was the computation of variable resets by means of classical data-flow analysis techniques (precisely, dead variable analysis), such as those used in optimising compilers for

[1] For the "rel/REL" reliable atomic multicast protocol, CÆSAR 4.3 generated 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 [12] about the poor efficiency of such a variable-length state representation scheme.

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 [12] and formalised in [3, 4], the main point of which is the proof that reduction based on dead variable analysis preserves strong bisimulation. Compared to [7], [3, 4] target at the SDL language rather than the LOTOS process algebra, and, instead of the network model, consider a set of communicating automata with state variables that are consulted and assigned by the automata transitions. The main differences between the model of [3, 4] and the network model are twofold.

As regards system architecture, the network model allows concurrent processes to be nested one in another at 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 the LOTOS behaviour "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 model of [3, 4] lacks any form of process hierarchy by allowing 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; additionally, concurrent processes may share variables inherited from their parent process(es) — as in the LOTOS behaviour "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 children processes cannot modify them. On the contrary, the model of [3, 4] relies on FIFO message queues and shared variables that can be arbitrarily read/written by all the processes. As regards shared variables, [3, 4] 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). Although 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 [5] do not reset shared variables actually — we believe that the communicating automata model used by [3, 4] 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.

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

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 of [7] 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 about experimental results and Sect. 7 gives concluding remarks.

2 Presentation of the Network Model

The network model presented here is based on the definitions of [8, 10], 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 note $domain(S)$ 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, noted "$\perp$". 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 note $eval(C, V)$ 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* (noted "!") and/or *receptions* (noted "?"). We define a relation noted "$[C, O] \xrightarrow{o} [C', v_1 \ldots v_n]$" 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 $\texttt{?}X\texttt{:}S$ binds X to

the received value(s)). For given pair $[C, O]$ there might be one or several pairs $[C', v_1 \dots v_n]$ such that $[C, O] \xrightarrow{o} [C', v_1 \dots v_n]$, since a reception $?X:S$ generates as many pairs as there are ground values in $domain(S)$.

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

Actions. Actions are terms of the form:

$A ::=$	**none**	*(empty action)*
	| **when** V	*(condition)*
	| **for** X **among** S	*(iteration)*
	| $X_0, \dots, X_{n\geq 0} := V_0, \dots, V_n$	*(vectorial assignment)*
	| **reset** $X_0, \dots, X_{n\geq 0}$	*(variable reset)*
	| $A_1 ; A_2$	*(sequential composition)*
	| $A_1 \& A_2$	*(collateral composition)*

We define a relation noted "$[C, A] \xrightarrow{c} C'$" expressing that successful execution of action A in context C yields a new context C'. For 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 $domain(S)$.

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

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

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

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: "τ", the usual notation for the internal steps of a process, and "ε", a powerful artefact (see [8, 10]) allowing the compositional construction of networks for a large class of LOTOS behaviours such as "B_1 `[]` `(` B_2 `|||` B_3 `)`". Although ε deserves a special semantic treatment, this has no influence on the approach proposed in this paper; thus, we do not distinguish ε from "ordinary" gates.

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* (we note $in(T) \triangleq Q_i$), $Q_o \subseteq \mathcal{Q}$ is a set of *output places* (we note $out(T) \triangleq Q_o$), 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

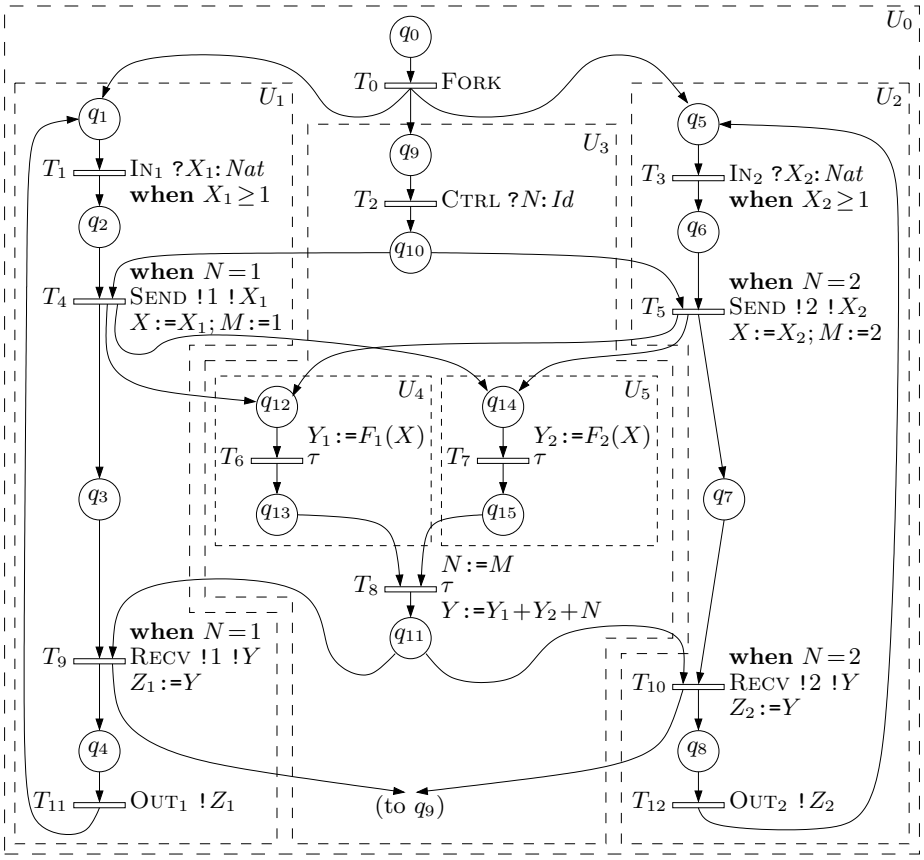

Fig. 1. Example of a network

place may contain at most one token — this is due to the so-called *static control contraints* [2, 8, 10], which only allow a statically bounded dynamic creation of processes (for instance, the following behaviour "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 noted "$[M, T] \xrightarrow{m} M'$" meaning that transition T can be fired from marking M, leading to a new marking M'. Classically, $[M, T] \xrightarrow{m} M'$ holds iff $in(T) \subseteq M$ (i.e., all input places of T have a token) and $M' = (M \setminus in(T)) \cup out(T)$ (i.e., tokens move from input to output places).

Units. Contrary to standard Petri nets, which consists 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 noted $\mathcal{U}$ 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 noted $places(U)$, such that all

sets of proper places $\{places(U) \mid U \in \mathcal{U}\}$ 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 $card\big(M \cap places(U)\big) \leq 1$, i.e., there is at most one token among the proper places of U, meaning that each unit models a (possibly inactive) sequential behaviour. 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-behaviours. There exists a *root unit* containing all other units. We note "$U' \sqsubseteq U$" the fact 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 note $places^*(U) = \bigcup_{U' \sqsubseteq U} places(U')$ the set of places transitively contained in U. For some marking M reachable from M_0, one may have $card\big(M \cap places^*(U)\big) > 1$ in case of concurrency between the sub-units of U; yet, for all units U and $U' \sqsubseteq U$, one has $\big(M \cap places(U) = \emptyset\big) \vee \big(M \cap places(U') = \emptyset\big)$, 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 note $unit(X)$ 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 $unit(X)$. In a first approximation, we will say that variable X is *shared* between two units U_1 and U_2 iff $\big(U_1 \sqsubseteq unit(X)\big) \wedge \big(U_2 \sqsubseteq unit(X)\big) \wedge (U_1 \not\sqsubseteq U_2) \wedge (U_2 \not\sqsubseteq U_1)$; this definition will be sufficient until Sect. 5, where a refined definition will be given.

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 Σ 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 Σ consists of a pair $\langle M, C \rangle$, with M a marking and C a context. The initial state σ_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 (σ_1, L, σ_2) belongs to the "$\rightarrow$" relation, a fact which we note "$\sigma_1 \xrightarrow{L} \sigma_2$", 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, the execution of each must succeed: 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 ε-transitions from the LTS; as mentioned before, we do not detail these rules here.

Example. Figure 1 gives an example of a network. According to Petri Net graphical conventions, places and transitions are represented by circles and rectangles.

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.

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 [7] 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:

- $use_v(V)$ (*resp.* $use_o(O)$, $use_a(A)$) denotes the set of variables consulted in value expression V (*resp.* offer O, action A).
- $def_o(O)$ (*resp.* $def_a(A)$) denotes the set of variables assigned a defined value by offer O (*resp.* action A).
- $und_a(A)$ denotes the set of variables assigned an undefined value (i.e., reset) by action A.
- $use\_before\_def_a(A)$ 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).

$use_v(X) \triangleq \{X\}$ $use_v\big(F(V_1,\ldots,V_n)\big) \triangleq \bigcup_{i=1}^{n} use_v(V_i)$	$use_o(\texttt{!}V) \triangleq use_v(V)$ $use_o(\texttt{?}X\texttt{:}S) \triangleq \emptyset$ $use_o(O_1 \ldots O_n) \triangleq \bigcup_{i=1}^{n} use_o(O_i)$
$und_a(\textbf{reset}\ X_0,\ldots,X_n) \triangleq \{X_0,\ldots,X_n\}$ $und_a(A_1\,\texttt{;}\,A_2) \triangleq \big(und_a(A_1) \setminus def_a(A_2)\big) \cup und_a(A_2)$ $und_a(A_1\texttt{\&}A_2) \triangleq und_a(A_1) \cup und_a(A_2)$ otherwise : $und_a(A) \triangleq \emptyset$	$def_o(\texttt{!}V) \triangleq \emptyset$ $def_o(\texttt{?}X\texttt{:}S) \triangleq \{X\}$ $def_o(O_1 \ldots O_n) \triangleq \bigcup_{i=1}^{n} def_o(O_i)$
$def_a\big(X_0,\ldots,X_n\texttt{:=}V_0,\ldots,V_n\big) \triangleq \{X_0,\ldots,X_n\}$ $def_a(\textbf{for}\ X\ \textbf{among}\ S) \triangleq \{X\}$ $def_a(A_1\,\texttt{;}\,A_2) \triangleq \big(def_a(A_1) \setminus und_a(A_2)\big) \cup def_a(A_2)$ $def_a(A_1\texttt{\&}A_2) \triangleq def_a(A_1) \cup def_a(A_2)$ otherwise : $def_a(A) \triangleq \emptyset$	$use_a(\textbf{when}\ V) \triangleq use_v(V)$ $use_a\big(X_0 \ldots \texttt{:=}V_0 \ldots\big) \triangleq \bigcup_{i=0}^{n} use_v(V_i)$ $use_a(A_1\,\texttt{;}\,A_2) \triangleq use_a(A_1) \cup use_a(A_2)$ $use_a(A_1\texttt{\&}A_2) \triangleq use_a(A_1) \cup use_a(A_2)$ otherwise : $use_a(A) \triangleq \emptyset$
$use\_before\_def_a(A_1\,\texttt{;}\,A_2) \triangleq use\_before\_def_a(A_1) \cup \big(use\_before\_def_a(A_2) \setminus def_a(A_1)\big)$ $use\_before\_def_a(A_1\texttt{\&}A_2) \triangleq use\_before\_def_a(A_1) \cup use\_before\_def_a(A_2)$ otherwise : $use\_before\_def_a(A) \triangleq use_a(A)$	

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:

– $use(T, X)$ holds iff X is consulted during the execution of T.
– $def(T, X)$ 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.
– $use\_before\_def(T, X)$ holds iff if 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:

$$use(T, X) \triangleq X \in use_a(A) \cup use_o(O) \cup use_a(W) \cup use_a(R)$$

$$def(T, X) \triangleq X \in \big(\big(def_a(A) \cup def_o(O)\big) \setminus und_a(R)\big) \cup def_a(R)$$

$$use\_before\_def(T, X) \triangleq X \in \left(\begin{array}{l} use\_before\_def_a(A) \ \cup \ \big(use_o(O) \setminus def_a(A)\big) \ \cup \\ \big(use\_before\_def_a(W\,;R) \setminus \big(def_a(A) \cup def_o(O)\big)\big) \end{array} \right)$$

Example. For the variable N in the network of Fig. 1, we have: $use(T, N)$ for $T \in \{T_4, T_5, T_8, T_9, T_{10}\}$, $def(T, N)$ for $T \in \{T_2, T_8\}$, and $use\_before\_def(T, N)$ 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 $reset(T, X)$ 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 $reset(T, X)$; this issue will be dealt with in Sect. 5; for now, we focus on computing $reset(T, X)$.

For sequential programs, the classical approach to global data-flow analysis (e.g. [1]) 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 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 [7], 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, as state explosion might occur.

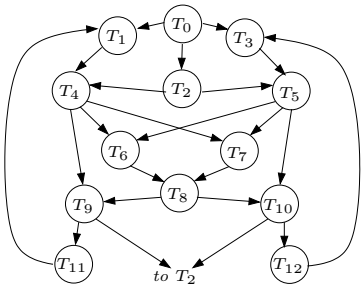

Fig. 2. CFG for Fig. 1

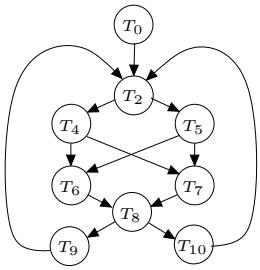

Fig. 3. CFG_N for Fig. 1

Hence, we choose a stronger abstraction by defining the control-flow graph as the directed graph $\mathrm{CFG} = \langle \mathcal{T}, \rightarrow \rangle$, 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. The CFG corresponding to the network of Fig. 1 is shown in Fig. 2.

Instead of constructing a unique CFG valid for all variables, [7] suggests to build, for each variable X, a dedicated control-flow graph CFG_X, which is a subset of CFG containing only the execution paths relevant to X (nowadays, this would be called "slicing"). According to [7, § 4.3.3], such a restricted control-flow graph increases the algorithmic efficiency; by our accounts, it also gives more precise data-flow results.

To define 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. Following the simple, "syntactic" approximation of $\mathit{scope}(X)$ given in [7], we define $\mathit{scope}(X) \triangleq \mathit{places}^*\big(\mathit{unit}(X)\big)$ as the set of all places in the unit to which X is attached.

We now define CFG_X as the directed graph $\langle \mathcal{T}_X, \rightarrow_X \rangle$ with the set of vertices $\mathcal{T}_X \triangleq \mathit{trans}\big(\mathit{unit}(X)\big)$ and such that there is an arc $T_1 \rightarrow_X T_2$ iff $\mathit{out}(T_1) \cap \mathit{in}(T_2) \cap \mathit{scope}(X) \neq \emptyset$. For $T \in \mathcal{T}_X$, we note $\mathit{succ}_X(T) \triangleq \{T' \in \mathcal{T}_X \mid T \rightarrow_X T'\}$ and $\mathit{pred}_X(T) \triangleq \{T' \in \mathcal{T}_X \mid T' \rightarrow_X T\}$.

Example. Figure 3 shows CFG_N for the network of Fig. 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. [1, pages 631–632]), we define, for $T \in \mathcal{T}_X$, the following predicate:

$\mathit{live}(T, X) \triangleq \bigvee_{T' \in \mathit{succ}_X(T)} \mathit{use\_before\_def}(T', X) \vee \big(\mathit{live}(T', X) \wedge \neg \mathit{def}(T', X)\big)$

that holds iff after T it is possible, by following the arcs of CFG_X, to reach a transition T' that uses X before any modification of X. For a given X, the set $\{T \in \mathcal{T}_X \mid \mathit{live}(T, X)\}$ is computed as a backward least fixed point.

We could now, as in [3, 4], define $\mathit{reset}(T, X) \triangleq \neg \mathit{live}(T, X)$. 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:

$\mathit{available}(T, X) \triangleq \mathit{def}(T, X) \vee \Big(\bigvee_{T' \in \mathit{pred}_X(T)} \big(\mathit{live}(T', X) \wedge \mathit{available}(T', X)\big)\Big)$

that holds iff T can be reached from some transition that assigns X a defined value, by following the arcs of CFG_X and ensuring that X remains alive all along the path. For a given X, the set $\{T \in \mathcal{T}_X \mid \mathit{available}(T, X)\}$ is computed as a forward least fixed point. [7] uses a similar definition without the $\mathit{live}(T', X)$ condition and, thus, only avoids resetting uninitialised variables.

Finally, we define $\mathit{reset}(T, X) \stackrel{\Delta}{=} \mathit{available}(T, X) \wedge \neg \mathit{live}(T, X)$, expressing that a variable can be reset where it is both available and dead.

Example. For the network of Fig. 1 and variable N, we have $\{T \mid \mathit{live}(T, N)\} = \{T_2, T_8\}$ and $\{T \mid \mathit{available}(T, N)\} = \{T_2, T_4, T_5, T_8, T_9, T_{10}\}$. Thus, we can insert "**reset** N" at the end of T_4, T_5, T_9, and T_{10}. Using the definition of [3, 4], one would insert superfluous "**reset** N" at the end of T_0, T_6, and T_7. Using the definition of [7], one would insert superfluous "**reset** N" at the end of T_6 and T_7. Using CFG instead of CFG_N would give $\{T \mid \mathit{live}(T, N)\} = \{T_0 \ldots T_5, T_8 \ldots T_{12}\}$ and $\{T \mid \mathit{available}(T, N)\} = \{T_1 \ldots T_5, T_8 \ldots T_{12}\}$, 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 [7], we noticed that systematic insertion of a "**reset** X" at the end of every transition T such that $\mathit{reset}(T, X)$ 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. In the network of Fig. 1, there exists a fireable sequence of transitions $T_0, T_1, T_2, T_4, T_6, T_7$. Although $\mathit{reset}(T_6, X)$ 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 [5]). Unfortunately, opportunities for valuable state space reduction are missed by doing so.

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

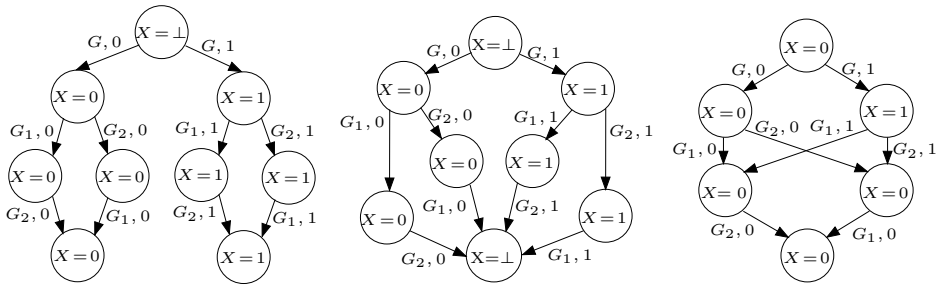

Fig. 4. LTS (a) without reset, (b) with correct resets, and (c) with incorrect resets

fined in Sect. 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 behaviours "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 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 behaviours B_1 and B_2, duplication consists in replacing, in one behaviour (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. In the previous example, duplicating X in B_2 yields the LOTOS behaviour "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 on Fig. 4 (b). Note that it is not necessary to duplicate X in B_1.

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 of creating a new variable X' of the same sort as X attached to U (whereas X is attached to $\mathit{unit}(X)$), replacing all occurrences of X in the transitions of $\mathit{trans}(U)$ by X' and 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.

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 concur-

rent behaviours, $(n-1)$ 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.

Concurrency Relation between Units. We now formalise the notion of "concurrent units". Ideally, two units U_i and U_j are concurrent if it 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.* In the LOTOS behaviour "`(`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 B_3 and B_4 also, 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 noted "$U_i \,\|\, U_j$" that is an upper-approximation of the ideal definition above, i.e., U_i and U_j concurrent implies $U_i \,\|\, U_j$. We base our definition on an abstraction function $\alpha : \mathcal{Q} \to \{1, \ldots, N\}$ (N being the number of units in the network) that maps all the proper places of each unit to the same number: $\forall Q \in \mathit{places}(U_i)$: $\alpha(Q) \triangleq i$. We extend α to sets of places by defining $\widehat{\alpha} : \wp(\mathcal{Q}) \to \wp\big(\{1, \ldots, N\}\big)$ such that $\widehat{\alpha}\big(\{Q_1, \ldots, Q_n\}\big) \triangleq \{\alpha(Q_1), \ldots, \alpha(Q_n)\}$. We then use α 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 $\mathcal{M}$ of all reachable markings for the quotient Petri net. Finally, we define $U_i \,\|\, U_j$ iff there exists $M \in \mathcal{M}$ 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 $U_i \,\|\, U_j$ implies $U_i \neq U_j$, $U_i \not\sqsubseteq U_j$, and $U_j \not\sqsubseteq U_i$.

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 according to given values of predicates *use* and *reset*, we define the predicate:

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

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 CFG_X and computing *reset* — based on the observation that $\mathit{link}(U_i, U_j) \subseteq \mathit{trans}\big(\mathit{unit}(X)\big)$.

```
1. compute the relation U_i ∥ U_j and link(U_i, U_j) (cf. Sect. 5)
2. VARS := 𝒳
3. while VARS ≠ ∅ do
4. begin
5.   X := one_of(VARS)
6.   VARS := VARS \ {X}
7.   repeat
8.     forall T ∈ trans(unit(X)) do
9.       compute use(T, X), def(T, X), and use_before_def(T, X) (cf Sect. 3)
10.    forall T ∈ trans(unit(X)) do
11.      compute reset(T, X) (cf Sect. 4)
12.    compute conflict(U_i, U_j, X, use, reset) and UCG_X (cf Sect. 5)
13.    compute U := best_of(UCG_X) (cf. Sect. 5)
14.    if U ≠ ⊥ then
15.    begin
16.      duplicate X in U by creating a new variable X'
17.      VARS := VARS ∪ {X'}
18.    end
19.  until U = ⊥
20.  -- at this point, there is no more conflict on X
21.  forall T ∈ trans(unit(X)) such_that reset(T, X) do
22.    insert "reset X" at the end of T (cf. Sect. 4)
23. end
```

Fig. 5. Complete algorithm

We then define, for given values of predicates *use* and *reset*, the *unit conflict graph for variable* X, noted UCG_X, as the undirected graph whose vertices are the units of $unit(X)$ and such that there is an edge between U_i and U_j iff $conflict(U_i, U_j, X, use, reset)$.

Complete Algorithm. The algorithm shown in Fig. 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 UCG_X. If UCG_X has no edge, X needs not be duplicated and "**reset** X" can be inserted at the end of every transition $T \in trans(unit(X))$ such that $reset(T, X)$. 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, data-flow analysis for X and UCG_X are recomputed, as duplication modifies the network by removing occurrences (definitions, uses, and resets) of X and by adding new assignments of the form $X' := X$.

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) X will be duplicated. Based on the observation that duplicating X in some unit U removes from 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 UCG_X. To select the unit (noted $best\_of(UCG_X)$) in which X should be duplicated first, we adopt a combination of top-down and greedy strategies by choosing, among the units of UCG_X having at least one edge, outermost ones; if there are several such units, we then choose one having a maximal number of edges; if UCG_X has no edges, $best\_of(UCG_X)$ returns $\perp$.

For a given variable X, the "**repeat**" loop (line 7) terminates because of fixed point convergence of global data-flow analysis and because the cardinal of $U_X = \{U \sqsubseteq unit(X) \mid \exists T \in trans(U) \setminus entry(U)$ such that $use(T, X)\}$ strictly decreases at each duplication (line 16). Indeed, U_X is the set of units U in which variable X is used within a transition T that does not "enter" U (i.e., $in(T) \cap places^*(U) \neq \emptyset$). After duplicating X in U, there are no remaining occurrences of X in the transitions of $trans(U) \setminus entry(U)$ and, thus, U is no longer in U_X. While duplication might add assignments $X' := X$ (and consequently, new uses of X) to the transitions T of $entry(U)$ (in fact, only those T such that $live(T, X)$) and, therefore, might add to U_X some new unit(s) U' such that $U \sqsubseteq U' \sqsubseteq unit(X)$, this is not the case actually, as all such units U' already belong to U_X (since $U \in U_X$ and $U \sqsubseteq U' \sqsubseteq unit(X)$ implies $U' \in U_X$).

The outermost "**while**" loop (line 3), which removes one variable X from $VARS$ 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 $L = \max\{\delta(U) \mid U \in \mathcal{U}\}$ be the maximal nesting depth, and let $\Delta(VARS)$ be the vector $(n_0, \ldots, n_L)$ such that $\forall i$, $n_i = card\{X \in VARS \mid \delta\big(unit(X)\big) = i\}$. At each iteration of the outermost loop, $\Delta(VARS)$ 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 $unit(X)$, i.e., $\delta\big(unit(X')\big) < \delta\big(unit(X)\big)$.

6 Experimental Results

We implemented our approach in a prototype version of CÆSAR and compared this prototype with the "standard" version of CÆSAR, which, as mentioned in Sect. 1, already resets variables but in a more limited way (upon process termination only). We performed all our measurements on a Sun Sparc Station 100 with 1.6 GB RAM.

We considered 279 LOTOS specifications (many of which are derived from "real world" applications) for which the entire state space could be generated with the "standard" version of CÆSAR. For 112 examples out of 279 (40%), our approach reduced the state space (still preserving strong bisimulation) by a mean factor of 9.7 (with a maximum of 220) as regards the number of states

and a mean factor of 13 (with a maximum of 360) as regards the number of transitions. For none of the other 167 examples did our prototype increase the state space.

Then, we considered 3 new "realistic" 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 (i.e., a reduction factor greater than 10^4).

We then extended our set of 279+3 examples with 17 new, large examples for which our prototype is still unable to generate the entire state space. On these 299 examples, variable duplication occurred in only 28 cases (9.4%) for which it increased the memory size needed to represent a state by 60% on average (most of the increase being due to 2 particular examples; for the 26 remaining ones, the increase is only of 10% on average). 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 outweighed by the global reduction in the number of states.

As regards execution time, we observed that our approach divides by a factor of 4 the total execution time needed to generate all LTSs corresponding to the 279 examples mentioned above. The cost in time of our algorithm is small (about 4%) and more than outweighed by the resulting state space reductions.

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, which can be consulted and modified by actions attached to the transitions.

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 [7] 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 points 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 (20–40% of examples) and can

reach several orders of magnitude (e.g. 10^4). Additionally, state space reduction makes CÆSAR four times faster when processing the complete set of examples.

As regards future work, we can mention some open issues (not addressed in this paper, since they are beyond the scope of the CÆSAR compiler for LOTOS), especially 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).

References

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

[2] G. Ailloud. Verification in ECRINS of LOTOS Programs. In *Towards Practical Verification of* LOTOS *specifications*, Universiteit Twente, 1986. Technical Report ESPRIT/SEDOS/C2/N48.1.

[3] M. Bozga, J.-C. Fernandez, and L. Ghirvu. State Space Reduction based on Live Variables Analysis. In *SAS'99*, LNCS 1694, pages 164–178, Sept. 1999. Springer.

[4] M. Bozga, J.-C. Fernandez, and L. Ghirvu. State Space Reduction based on Live Variables Analysis. *Science of Computer Programming*, 47(2–3):203–220, 2003.

[5] M. Bozga, J.-C. Fernandez, L. Ghirvu, S. Graf, J.-P. Krimm, and L. Mounier. IF: An Intermediate Representation and Validation Environment for Timed Asynchronous Systems. In *FM'99*. Springer, Sept. 1999.

[6] Y. Dong and C. R. Ramakrishnan. An Optimizing Compiler for Efficient Model Checking. In *FORTE'99*, pages 241–256, Beijing, Oct. 1999. Kluwer.

[7] 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. Supervised by Hubert Garavel and defended on June 22, 1993 before the jury composed of Hubert Garavel, Farid Ouabdesselam, Claude Puech, and Jacques Voiron.

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

[9] H. Garavel, F. Lang, and R. Mateescu. An Overview of CADP 2001. *EASST Newsletter*, 4:13–24, Aug. 2002. Also INRIA Technical Report RT-0254.

[10] H. Garavel and J. Sifakis. Compilation and Verification of LOTOS Specifications. In 10^{th} *International Symposium on Protocol Specification, Testing and Verification*, pages 379–394. IFIP, June 1990.

[11] S. Graf, J.-L. Richier, C. Rodríguez, and J. Voiron. What are the Limits of Model Checking Methods for the Verification of Real Life Protocols? In 1^{st} *Workshop on Automatic Verification Methods for Finite State Systems*, LNCS 407, pages 275–285, June 1989.

[12] G. J. Holzmann. The Engineering of a Model Checker: The Gnu i-Protocol Case Study Revisited. In 6^{th} *SPIN Workshop*, LNCS 1680, pages 232–244, 1999.

[13] ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, ISO, Genève, Sept. 1989.

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