Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 188 (2007) 53–75
www.elsevier.com/locate/entcs

Formal Sequentialization of Distributed Systems via Program Rewriting[1]

Miquel Bertran[2] Francesc Babot[3] August Climent[4]

Informàtica La Salle
Universitat Ramon Llull
Barcelona

Abstract

Formal sequentialization is introduced as a rewriting process for the reduction of parallelism and internal communication statements of distributed imperative programs. It constructs an equivalence proof in an implicit way, via the application of equivalence laws as rewrite rules, thus generating a chain of equivalent programs. The variety of the possible sequentialization degrees which are attainable is illustrated with an example. The approach is static, thus avoiding the state explosion problem, has an impressive state-vector reduction in many cases, and could be combined, as a model simplification step, with model checking and interactive theorem proving in system verification. Prior grounding results needed in formal sequentialization are overviewed; more specifically, an algorithm for the automatic elimination of communications under the scope of sequential and parallel compositions, elimination laws which the algorithm applies, and a suitable equivalence criterion for the sequentialization process. The main contribution of this work is the extension of these results to encompass the formal elimination of both synchronous communications embedded within a subclass of selection statements, and of non-disjoint synchronous communication pairs. None of these cases has been treated in the literature before, and their solution considerably widens the application domain of formal sequentialization.

Keywords: Formal communication elimination, formal sequentialization, distributed programs, parallel programs, formal verification, laws of distributed programs, rewriting.

1 Introduction

Formal methods are increasingly being used in industry to establish the correctness of, and to find the flaws in, system models; of hardware, protocols, distributed algorithms, etc... In particular, model checking [11,19,7,21,20] does that automatically for finite-state systems. However, model checking is limited in scope due to the state explosion problem; since it works on the transition system defining the semantics of the distributed program. This transition system has often infinitely

[1] Work partially supported by the CICYT under project TIN2006-14738-C02-02, and by General Systems Development, www.gsystemsd.com

[2] Email:miqbe@salle.url.edu

[3] Email:fbabot@salle.url.edu

[4] Email:augc@salle.url.edu

doi:10.1016/j.entcs.2007.05.038

many states, and always a large size compared to the size of the program, which is always finite. Most practical system descriptions, notably those of software, are therefore not directly amenable to finite-state verification methods since they have very large or infinite state spaces. For such systems, interactive theorem proving has so far been the only viable alternative. However, its use requires manual effort and mathematical sophistication. New paradigms and methods that combine the ease of use of model checking with the power and flexibility of theorem proving are needed. Such hybrid techniques have started to emerge [18,23].

Imperative languages with explicit parallelism and communication statements provide an intuitive, explicit, and complete framework to express system models and distributed programs with perspective and clarity. This is important in verification. OCCAM [12,13,14], the *simple programming language* SPL of Manna and Pnueli [16,17], PROMELA of the SPIN model checker [11], and the *shared-variable language*$^{++}$, SVL^{++}, in [8] are representatives of these imperative notations.

The above considerations suggest that static verification approaches, avoiding the transition system, working directly on the imperative program would have less computational complexity, in principle. Following this motivation, some static analysis methods have been proposed for state reduction; for instance, as a prior step to model checking [6,15,25]. They reduce, indirectly, its complexity. Formal sequentialization, or *Distributed program sequentialization* (DPS), is a formal method which falls under this category since it attempts to obtain a simplified program equivalent, in a sense to be clarified later, to the original distributed program, inner communication-free, and with less variables. DPS is carried out within an interactive equivalence prover, and could be combined with model checking or interactive verification of the simplified program as a succeeding step, reducing overall proof complexity. In some cases, only an equivalent purely sequential program would have to be verified. Notice that partial order reduction methods [1] work on the transition system, whose complexity is exponential in the size of the program, whereas DPS works on the program, whose complexity is linear only.

Internal synchronous communication elimination is an important step of DPS, since it is a necessary prior task. It applies laws as reductions. A set of laws for OCCAM was given in [22]. Communication closed layered systems were introduced in [9], and some laws for them were given in [8] in the framework of SVL^{++}. The aim there was formal design via sequential-parallelism, iteration unfolding, and other transformations. Some laws for SPL were given in [16], also covering an SPL semantics based on fair transition systems (FTS). Communication elimination laws where proposed in [4], showing the necessity of avoiding strong fairness. An equivalence suitable for DPS was presented and studied in [5]. The mathematical justification of the laws in this equivalence, with their applicability conditions, and that of a first version of a communication elimination algorithm have been reported in [2]. A DPS equivalence proof of a pipelined processor model was covered in [5] as well. The impressive reduction of 2^{-607} in the upper bound on the size of the state vector of the model obtained with this DPS proof was reported in [2]. Other DPS proofs were given in [3,4].

These prior results applied neither when the synchronous communications to be eliminated were under the scope of selection statements nor when the communication statement pairs to be eliminated shared a communication statement, thus not being disjoint. This communication introduces a new algorithm for automatic formal communication elimination which handles these cases, together with the necessary laws, involving selections, which are applied automatically by the algorithm. It is organized as follows. Two sections cover background material and grounding results. The first one deals with notation, associated notions, auxiliary laws, interface equivalence, and bounded communication statements (BCSs), the class of statements for which DPS is analyzed first. The second one reviews proper communication elimination laws and comments on the algorithm for communication elimination of selection-free BCSs. A section follows introducing formal DPS proofs, and their extension to a very typical non-BCS. An example of the degrees of sequentialization compatible with internal communication elimination is also included. Then, grounding results for communication elimination from a subclass of BCSs with selections are given, together with the new algorithm which follows from these results. After that, the extension to automatic elimination of non-disjoint communication pairs and to the general scenario integrating both cases are presented as another contribution. A section on conclusions and further work closes the paper.

2 Notation and basic notions

2.1 Syntax and auxiliary laws

Programs are written in a reduced version of SPL, which is general enough to express any practical program. The basic statements are **skip**, **nil**, **stop**, the assignment $u{:=}\ e$, send $\alpha \Leftarrow e$, and receive $\alpha \Rightarrow u$. The work is limited to synchronous channels α, which will be referred to as *channels*. In them both the sender and the receiver wait for each other before exchanging a value and continuing execution. No intermediate message buffering is involved. Synchronous communication statements will be referred to more simply as *communications*.

The *cooperation* statement is n-ary: $[S_1||\cdots||S_n]$. It will also be referred to as *parallelism* statement. Its substatements S_j are the *top parallel statements* of the cooperation statement, which is the *least common ancestor* (LCA) of them. It will be assumed throughout that the S_j's are *disjoint*, in the sense that they only share read variables, and that they communicate values through synchronous channels only.

The *concatenation* statement is also n-ary: $[S_1;\cdots;S_n]$. The iterations are [**while** c **do** S], where c is a boolean expression, and [**loop forever do** S], which is defined as [**while** $true$ **do** S]. The notion of LCA statement applies to concatenation composition as in the cooperation statement. Two statements are ordered in the *concatenation ordering* if their LCA is a concatenation statement. This corresponds to the execution order.

The *regular* and the *communications selection* statements are non-

deterministic and have, respectively, the forms $[b_1, S_1 \textbf{ or} \cdots \textbf{or } b_n, S_n]$ and $[b_1, c_1; S_1 \textbf{ or} \cdots \textbf{or } b_n, c_n; S_n]$, where the b_i's are boolean expressions referred to as *boolean guards*, and the c_i's are synchronous communication statements referred to as *communication guards*. The S_i's are *the alternatives*. Statement labels, such as l in $l : S$, are used sometimes both to refer to statements and as control locations.

As an illustration of the use of the notation, the following procedure models a step of a stop and wait communications protocol

$$(\alpha, ack) ::= Step\ (m)\ :: \left[\begin{array}{ll} \textbf{local} & \delta,\ \eta : \quad \textbf{channel of message} \\ \textbf{local} & \gamma \quad\quad\ \textbf{channel of boolean} \\ \textbf{local} & \varepsilon : \quad\ \textbf{channel of nil} \\ & \\ & Emitter :: \left[\eta \Leftarrow m; \gamma \Rightarrow ack\right] \\ & \| \\ & DataChannel :: \left[\begin{array}{ll} \textbf{local} & d : \quad \textbf{message} \\ & \\ \eta \Rightarrow d; & \left[\begin{array}{l} true, \delta \Leftarrow d; \textbf{nil} \\ \textbf{or} \\ true, \varepsilon \Leftarrow; \textbf{nil} \end{array}\right] \end{array}\right] \\ & \| \\ & Receiver :: \left[\begin{array}{ll} \textbf{local} & r : \quad \textbf{message} \\ & \left[true, \delta \Rightarrow r; \alpha \Leftarrow r; \gamma \Leftarrow T\right] \\ & \textbf{or} \\ & \left[true, \varepsilon \Rightarrow; \gamma \Leftarrow F\right] \end{array}\right] \end{array}\right]$$

Greek letters denote channels. There are three statements connected in parallel: *Emitter*, *DataChannel*, and *Receiver*. The message to be sent is input to *Step* in variable m of global memory within *Emitter*, which sends it to *DataChannel* via η and waits on channel γ for acknowledgement. The message is delivered to *Receiver* via channel δ. A transmission error is simulated, non-deterministically, by communicating to *Receiver* through channel ϵ of type **nil**. Only the implicit synchronization suffices, no value passing is needed. The two options are the alternatives of the communications selection within *DataChannel*, matched by a communications selection within *Receiver*. After outputting the message from *Step* via channel α, *Receiver* acknowledges to *Emitter* by passing a true value through channel γ. In case of an erroneous reception, a false value is sent instead. This boolean is stored in variable *ack* of global memory as a result of *Step*.

In order to close the presentation of the notation, some intuitive auxiliary laws are given. They are proved sound in [4], where it is shown that many of them do not hold when strong fairness is assumed. These laws are needed to transform a statement to a form where a proper communication elimination law can be applied. Some of them are the congruences $\textbf{nil}; S \approx S$, $S \| \textbf{ nil} \approx S$, $S; \textbf{skip} \approx S$, $S \| \textbf{ skip} \approx S$. In addition, both sequential and parallel composition are associative. The latter is also commutative.

2.2 Interface equivalence

The proper communication elimination laws do not hold for congruence. However, they hold as equivalences in a weaker equivalence which has been introduced in [5].

It is referred to here as *interface equivalence*. A summary is given now.

Interface equivalence is grounded in the fair transition systems (FTS) semantics of Manna and Pnueli [16,17]. In this semantics, a statement S denotes a FTS with states and transitions. A *computation* is a sequence of states of the transition system, starting at an initial state with a transition of the FTS taking any state of the sequence to its successor. A computation has *components*, which are associated to variables. A component is the list of values taken by its associated variable, with independence of those of the others, within the computation. A *reduced behavior*, with respect to a set of observed variables $\mathcal{O}$ is a computation whose components of variables outside this set and whose stuttering steps have been deleted.

This semantics has been extended in [5] by adding to the set $\mathcal{O}$ an auxiliary variable, *channel variable*, for each *external* channel, to record the history of values traversing it. External channels are used in S, but they are not in I, the set of *internal channels*. These are channels communicating parallel substatements within S, and considered to be hidden from the outside. Any channel is either external or internal. The extended set $\mathcal{O}$ is referred to as the *interface set*. When S is the body of a procedure, the internal channels are not declared in its interface whereas the external channels are. *Internal communication* and *external communication* will mean communication substatements of S over internal and external channels, respectively.

An *extended computation* is the extension of the notion of computation taking also into account all channel variables. Usually, it will be referred to more simply as *computation* in the rest of this work. Similarly, an *interface behavior* is the extension of the notion of reduced behavior corresponding extended computations and to the extended set $\mathcal{O}$. It records the history of values associated to both variables and channels of the interface. The order of value changes of different variables and channels is preserved in interface behaviors. However, the weaker equivalence which is needed can neglect this relative order and still preserve some input/output relation. Then, instead of comparing whole behaviors, only components of behaviors are compared. The following formalizes this.

Two interface behaviors are *equivalent* when they share the same interface set, and for all its variables the two components of both behaviors correspond to the same list of values, whose repeated values are deleted with the exception of the last one. The relative order of value changes among different components is thus lost in this equivalence, but not the order of changes within the same component. This equivalence only requires equality of homologous component lists. Finally we define the equivalence used throughout this work.

Definition 2.1 (Interface equivalence) Two statements S_1 and S_2 are *interface equivalent with respect to an interface set* $\mathcal{O}$, written $S_1 =_{\mathcal{O}} S_2$, when any interface behavior of any of them is equivalent to an interface behavior of the other. From now on in this work, this equivalence will be referred to more simply as *equivalence*.

There are many other equivalences in the literature, within process algebras [24], in the polychrony framework [10], etc. It would be interesting to study interface

equivalence in their perspective. Nevertheless, interface equivalence was introduced since it was the minimal extension that was needed while keeping this work within the Manna-Pnueli framework.

Continuing with the example of the communications protocol, the interface set of *Step* would be $\mathcal{O} : \{\alpha, ack, m\}$. Its elements being the input and output variables m and ack, and the output channel variable α. The equivalent program resulting from a DPS proof would be the following

$$(\alpha, ack) ::= SimpleStep\ (m)\ ::\ \left[\begin{array}{l} \Big[true, [\alpha \Leftarrow m; ack := T]\Big] \\ \textbf{or} \\ \Big[true, [ack := F]\Big] \end{array}\right]$$

Thus $Step =_{\{\alpha,ack,m\}} SimpleStep$. It captures the essential behavior as seen from its interface.

2.3 Bounded communication statements

A statement S is said to be of *bounded communication* if: (a) all its parallel substatements are disjoint, and (b) any internal communication is outside iteration bodies. Consequently, execution of a BC statement generates a finite number of internal communication events.

The *communication front* of S, written *ComFront(I,S)*, is the subset of minimal elements of the set of communication statements in its concatenation ordering. Two internal communications of S, l and m, are said to form a *matching communication pair*, p, if they are parallel, one is an output and the other an input over the same channel. The set of *competing pairs* of S, written *CompPairs(I,S)* is, by definition, the set of matching pairs $p : (l, m)$ which can be formed with the communications in *ComFront*(I, S). Two matching pairs are *disjoint* if they share no communication statement.

3 Elimination from selection-free BCSs

A *selection-free* BCS is a BCS all of whose internal communications are outside the scope of both selections and communication selections. For any competing pair (l, m) of S, G^l and G^m are their corresponding embedding top statements in their LCA parallel statement. A *standard form* for the top statements is defined recursively, for $k = 1, \ldots$ as $G^x_k = H^x_{k-1}; [G^x_{k-1} || P^x_{k-1}]; T^x_{k-1}$ where G^x_0 is either one of the communications $\alpha \Leftarrow e$ and $\alpha \Rightarrow u$, belonging to the communication front of S. x denotes either l or m. G^l and G^m can be put in standard form with the insertion of nil statements, via application of auxiliary laws, for some integer $k = n_l$ or $k = n_l$, respectively. The T^x_k's, P^x_k's, and H^x_k's are BCSs. The H^x_k's have no internal communications. The simplest case of elimination law corresponds to $[\ \alpha \Leftarrow e \ ||\ \alpha \Rightarrow u\ ] \approx [u := e]$. It is identified with $[G^l_0 || G^r_0] \approx G_0$, as the base case. For the more complex forms the elimination law is defined for an arbitrary $k \geq 0$ as

$$\left[\begin{array}{c} H_k^l; \\ \left[\begin{array}{cc} G_k^l & \| \ P_k^l \end{array}\right]; \\ T_k^l \end{array}\right] \ \| \ \left[\begin{array}{c} H_k^r; \\ \left[\begin{array}{cc} G_k^r & \| \ P_k^r \end{array}\right]; \\ T_k^r \end{array}\right] \ =_{\mathcal{O}} \ \left[\begin{array}{c} \left[\begin{array}{ccc} H_k^l & \| & H_k^r \end{array}\right]; \\ \left[\begin{array}{ccccc} G_k & \| & P_k^l & \| & P_k^r \end{array}\right]; \\ \left[\begin{array}{ccc} T_k^l & \| & T_k^r \end{array}\right] \end{array}\right]$$

When this equivalence is identified with $[\ G_{k+1}^l \ \| \ G_{k+1}^r] =_{\mathcal{O}} G_{k+1}$, a recursive definition of G_k^l, G_k^r, and G_k is obtained. The law for $k = k_0$ would be constructed recursively, applying the same equivalence to the inner G_k, which stands for $[G_k^l || G_k^r]$, for $k = k_0, k_0 - 1, \cdots, 1, 0$. The last inner parallelism $[G_0^l || G_0^r]$ would be replaced by G_0, the right hand side of the basic congruence given earlier. There is a law for each integer $k = 0, 1, \cdots$ which may be applied as a reduction from left to right in order to eliminate a single communication pair. The detailed justification and the applicability conditions for this law schema are given in [2].

Remark 3.1 (Implicit parallelism reduction) *Parallelism is reduced as a side effect of the application of the above law. The statement pairs (T_k^l, P_k^r), (T_k^l, G_k^r), and their symmetrical ones, are parallel in the left side but concatenated in the right side.*

Assuming disjointness of all possible pairs, a communication elimination algorithm has been proposed and studied in [2]. It applies iteratively a procedure, *PElim* which carries out the elimination of a pair applying the above recursive law. When applicability conditions do not hold a boolean result is returned with a false value, and the algorithm terminates unsuccessfully. When the loop of *PElim* invocations terminates without failure, and there is still some communication left in *ComFront*(I, S), this means that the original program has deadlock possibility. This proof construction algorithm simulates an execution where the elimination of a pair of matching communication statements in the proof corresponds to a communication event in an execution. Generalizations of this algorithm are within procedures *DisjPairsElim* and *CompleteComsElim* given at the end of sections 5 and 6, respectively.

4 Distributed program sequentialization proofs

DPS is a proof with three types of step: (a) elimination of internal communication pairs, (b) parallelism to concatenation formal transformation, and (c) redundant variable elimination.

Usually, the two latter steps are interlaced. The first step of DPS proofs could be carried out by the *communication elimination* reduction algorithm, mentioned in section 3. When the algorithm terminates successfully, the resulting equivalent form has parallelism between disjoint substatements but no internal communication statements.

4.1 Parallelism to concatenation transformation

A DPS proof continues with a further step, *parallelism to concatenation transformation*. It is carried out applying permutation laws for transforming the parallel

compositions of disjoint processes to equivalent sequential forms. For instance, since S_1 and S_2 are disjoint $S_1||S_2 =_{\mathcal{O}} S_1; S_2$. Another class of these laws was introduced for communication closed layer systems. These systems, together with their laws, are treated in [8], with a semantics different to the one used here. But the laws also hold in the present semantics. The following is an example.

Lemma 4.1 (Communication-closed-layers) *Let the statement pairs (A_1, B_2) and (A_2, B_1) be non-communicating, and $[B_1; A_1]$ be disjoint with $[B_2; A_2]$. Then $[[B_1; A_1]||[B_2; A_2]] =_{\mathcal{O}} [[B_1||B_2]; [A_1||A_2]]$, and either both sides are deadlock-free or none of them is.*

Justification The only statements which change their concatenation order relation are the pairs which do not communicate. Therefore deadlock can not be introduced, since processes can only wait for internal communications to occur. Also, the same pairs are disjoint as a consequence of the assumptions. This guarantees that variable components of behaviors do not change. Hence, interface behaviors of both sides remain equivalent, as in subsection 2.2. □

The third and last step of DPS proofs is *redundant variable elimination*. State-vector reduction comes with this last step. Both, redundant variables and statements are eliminated. The former usually come from communication buffers, of the original distributed system, which are no longer necessary after their inner communications have been eliminated. The next subsection provides some detail about variable elimination.

4.2 Variable elimination and other sequential program laws

In the later stages of DPS proofs, the program has been already transformed to sequential form, and any sequential program proof technique may be applied. However, in order to use the same style based on reductions, sequential program transformation laws are applied. For instance, simple congruences such as **if** *true* **then** S_1**else** $S_2 \approx S_1$. The elimination of redundant variables is done with the law of the following lemma, which can be generalized to multiple variables.

Lemma 4.2 (Variable and assignment elimination) *Let e be an expression having no reference to variable v, such that $v \notin \mathcal{O}$. Let $S_1(v)$ have only read references to v, S_2 have no read reference to v, and be either the last statement within the scope of v or located just before a new assignment to v, with respect to the concatenation order of the program. Then*

$$[\, v := e; S_1(v); S_2 \,] \;\; =_{\mathcal{O}} \;\; [\, S_1(e); S_2 \,]$$

Justification The assignment of a new value to v in the left side has no effect upon any reduced behavior since v is not in the observed set $\mathcal{O}$, and, due to the conditions imposed upon S_2, no variable in the observed set can change its value in any reduced behavior. □

4.3 DPS for non-BC statements

There exist many types of non-BC statements, where communications appear within indefinite loops. We will center only in the following very common structure: $S = [S_1||\cdots||S_m]$, where $S_k =$ **loop forever do** B_k . The B_k's are BC statements. Channel declarations have been omitted.

Assume that we unfold n_k times the loop of each top substatement S_k of S, thus obtaining the statement $S_u = [B_1^{n_1}; S_1||\cdots||B_m^{n_m}; S_m]$, where the $B_k^{n_k}$'s stand for the concatenation of n_k copies of B_k : $B_k; \cdots; B_k$.

DPS can be applied to S_u partially, only considering its internal communications in the $B_k^{n_k}$ statements. Assume that we succeed and obtain $B; E$, where B has no internal communication but the ending statement E is non-BC, it may have both parallelism and inner communication. Assume also that $B; E$ is also reduced by DPS, partially as before, to $B; B; E$. Then, as a consequence of finite induction, $S =_{\mathcal{O}} [B^n; E]$ for any finite integer n, where B^n is inner communication free. In the frequent case where the first elimination yields $B; S$, i.e. $E = S$, then $S =_{\mathcal{O}}$ **loop forever do** B and the right hand side statement has no inner communication. In many practical system models this occurs already for $n_k = 1$; $k = 1 \cdot \cdot m$.

4.4 Degrees of sequentialization

In general, the elimination of internal communication statements reduces parallelism among external communication offerings. This follows from remark 3.1 of section 3; but, in addition, from parallelism to concatenation reductions as well. This introduces the possibility of deadlock within the embedding program environment. This will be illustrated with the following three register queue, with interface set $\mathcal{O} : \{cout, cin\}$.

$$(cout) ::= Queue(cin) ::$$

$$\left[\begin{array}{l} R0:: \left[\begin{array}{l}\textbf{loop forever do}\\ \left[\begin{array}{l} p0 : c0 \Rightarrow m0;\\ p1 : cout \Leftarrow m0\end{array}\right]\end{array}\right] \;||\; R1:: \left[\begin{array}{l}\textbf{loop forever do}\\ \left[\begin{array}{l} q0 : c1 \Rightarrow m1;\\ q1 : c0 \Leftarrow m1\end{array}\right]\end{array}\right] \;||\; R2:: \left[\begin{array}{l}\textbf{loop forever do}\\ \left[\begin{array}{l} r0 : cin \Rightarrow m2;\\ r1 : c1 \Leftarrow m2\end{array}\right]\end{array}\right] \end{array}\right]$$

Here, the three *ci* channels are internal and a maximum of three inputs via channel *cin* are allowed without the occurrence of any output via channel *cout*. From this state, a maximum of three *cout* outputs are allowed without the need of any further input. Other degrees of external input/output interleavings are allowed.

The maximum parallelism attainable with the sequentialization equivalence proof for this system corresponds to the following program

$$\left[\begin{array}{l} p0 : cin \Rightarrow m0;\\ \textbf{loop forever do}\\ \quad \left[\quad \left[q0 : cout \Leftarrow m0 \;||\; r0 : cin \Rightarrow m1\right] ;\; p1 : m0 := m1\right] \end{array}\right]$$

This system is interface equivalent to the original queue. Clearly, now only two consecutive inputs or two consecutive outputs are allowed, thus the number of possible interleavings has been reduced, and hence the degree of parallelization. Finally, here is a possible equivalent purely sequential form, allowing only one order

of input/output events. It corresponds to the minimum degree of parallelism.

$$\left[\begin{array}{l} l0 : cin \Rightarrow m0; \\ \textbf{loop forever do} \\ \quad \left[\begin{array}{l} l1 : cin \Rightarrow m1; \\ l2 : cout \Leftarrow m0; \\ l3 : m0 := m1 \end{array}\right] \end{array}\right]$$

5 Extension to BCSs with selections and disjoint pairs

In a certain sense, elimination under a selection can be already handled with the tools that have been overviewed so far, but for a very simple case, as pointed out in the following

Remark 5.1 (Improper elimination under a selection) When the two communications of a matching pair are located within the same alternative of a selection, the elimination algorithm overviewed in section 3, for selection-free BCSs, can be properly applied to the alternative to eliminate it.

Some new notions are needed at this point. A *program context* $P[\cdot]$ is a program P one of whose statements corresponds to a hole to be filled-in with an arbitrary statement. Similarly, a *double program context* $P[\cdot|\cdot]$ means a program two of whose statements correspond to holes to be filled-in with an arbitrary statement each; neither of them being a substatement of the other.

Two statements S_1 and S_2 are congruent, written $S_1 \approx S_2$, if, for any program context $P[\cdot]$, $P[S_1]$ and $P[S_2]$ have the same set of interface behaviors.

Throughout this section, at any point of an elimination proof, all the pairs in $CompPairs(I, S)$ are assumed to be disjoint, thus sharing no communication.

5.1 Single selection embedding BCSs

The following definitions specify the subclass of selection-free BC statements which is treated in this work. In them, the term *basic statement* includes procedure references. The notions of *initial* and *terminal* basic statement are used with their intuitive meanings.

Definition 5.2 (Concatenation chain of a statement) A *concatenation chain* of a statement is a list of some of its basic substatements, in consecutive ascending concatenation order, with parallelism and selection symbols, whose first element is an initial substatement or a selection or a parallelism symbol; and whose last element is a terminal or an exit substatement. Each of the two symbols represents the LCA selection or parallelism statement of its immediate successor.

Definition 5.3 (Single selection embedding BC statement) A BC statement is *single selection embedding BC* when all its concatenation chains contain at most one selection symbol in between any pair of internal communications, or in between the starting point and the first internal communication.

Only single selection level BC statements are considered in the rest of this work. Then, there are three possible locations for a communication c of a pair p in *CompPairs(I,S)* within a selection: (a) as a guard, (b) within an alternative $A[c]$ of a communications selection whose communication guard is an external communication *ext*, and (c) within an alternative $A[c]$ of a regular selection. The communication c has the forms $\alpha \Rightarrow v$ or $\alpha \Leftarrow v$. An example of the first case is

$$\Big[\cdots \alpha_1 \Leftarrow v_1; \cdots \;||\; \cdots [b_1, \alpha_1 \Rightarrow v_2; A_1 \;\; \textbf{or} \;\; b_2, \alpha_2 \Leftarrow v_3; A_2]; \cdots \;||\; \cdots \alpha_2 \Rightarrow v_4 \cdots\Big]$$

where the communications via channels α_1 and α_2 stand as guards of a communications selection. They belong to distinct pairs : $(\alpha_1 \Leftarrow v_1 \;,\; \alpha_1 \Rightarrow v_2)$ and $(\alpha_2 \Leftarrow v_3 \;,\; \alpha_2 \Rightarrow v_4)$. This case can be represented, in general, as $S[b, c; A \textbf{ or } R]$, where R stands for the rest of the selection statement. The following is an example of the second case.

$$\Big[\cdots \alpha_1 \Leftarrow v_1; \cdots \;||\; \cdots [b_1, cext \Rightarrow v_2; \cdots \alpha_1 \Rightarrow v_3 \cdots \;\; \textbf{or} \;\; R]; \cdots \;||\; \cdots\Big]$$

Here the pair is over channel α_1. This case can be represented, in general, as $S[b, cext; A[c] \textbf{ or } R]$. For the third case one has the statement

$$\Big[\cdots \alpha_1 \Leftarrow v_1; \cdots \;||\; \cdots [b_1, \cdots \alpha_1 \Rightarrow v_2 \cdots \;\; \textbf{or} \;\; R]; \cdots \;||\; \cdots\Big]$$

as an example, with general form $S[b, A[c] \textbf{ or } R]$.

5.2 Elimination principles

In the elimination of a communications pair with the successive applications of the the law schema, as a proof construction algorithm, overviewed in section 3, a possible execution is simulated. Elimination of a pair corresponds to a communication event of an execution. At any point of the elimination proof, $CompPairs(I, S)$ contains a pair for each possible execution at that point and the elimination selects one pair as the first one, but the resulting program $\hat{S}$ has to conserve the rest of the pairs, in order to be eliminated afterwards or in other alternatives. This depends on whether or not some communication of the pair is under a selection scope. This complete execution simulation principle is summarized in the following lemmas.

Lemma 5.4 (Conservation of parallel pairs) *The elimination of any one of the pairs* $p \in \text{CompPairs}(I, S)$ *which is outside a selection scope, has to conserve in* $\text{CompPairs}(I, \hat{S})$ *of the resulting statement* $\hat{S}$ *all the remaining pairs which are parallel to* p.

Justification The elimination has to simulate an execution. In this case the eliminated pair is parallel to the other pairs. In the current context, any execution order among the pairs will give equivalent behaviors, but the other pairs will execute later in the current behavior and, for that reason, have to be conserved after the elimination, within $\hat{S}$. □

Lemma 5.5 (Conservation of disabled pairs) *In the elimination of a pair*

*$p \in$*CompPairs*(I, S), one of whose communications is a guard, the pairs of the other guards of the selection should remain in* CompPairs*$(I, \hat{S})$ of the resulting statement $\hat{S}$ within a selection alternative R of the one where p is eliminated.*

Justification Here, in the corresponding execution the other pairs, one of whose communications is a guard in the selection, are disabled and will not execute later in the execution, or current behavior. However, they would execute in the behavior of an alternative. In order to retain this alternate execution, within other behaviors, they have to be left within an alternative of an $\hat{S}$ selection embedding the substatement resulting from the elimination of p. □

5.3 Elimination under a single selection

Corresponding to the three locations of c mentioned in subsection 5.1, the congruences of the following lemma prepare the elimination law which will give more concretion to lemma 5.5.

Lemma 5.6 (Single selection congruences) *Let $S[\cdot]$ be a program context, and c be a communication of pair p whose other communication $\bar{c}$ is outside another selection scope within $S[\cdot]$. Then*

$S[\ b, c; A$ **or** $R\ ] \approx [\ bp(b, H), S[c; A]\ ]$ **or** $[\ true, S[R]\ ]$

$S[\ b, A[c]$ **or** $R\ ] \approx [\ bp(b, H), S[A[c]]\ ]$ **or** $[\ true, S[R]\ ]$

$S[\ b, ext; A[c]$ **or** $R\ ] \approx [\ bp(b, H), S[ext; A[c]]\ ]$ **or** $[\ true, S[R]\ ]$

where bp(b,H) *denotes the backward propagation of boolean expression b within $S[\cdot]$, H standing for all the concatenated statements preceding b within $S[b, \cdots]$.*

Remark 5.7 (Interface extended backward propagation) The backward propagation *bp(b,H)* is a condition, not only upon the first values of regular variable components of interface behaviors of H, but upon all the values of its external input channel variable components as well. Recall that H has no inner communication since $c \in ComFront(I, S)$.

Justification An approach, as in [4], would be to show that the transitions associated or denoted by the statements at the two sides of the congruence symbols are the same. A different approach is the following. The sets of interface computations of the statements at the left sides of the congruence symbols are partitioned into three classes: the one involving the left alternative, the class with the rest of alternatives R, and that involving none of the alternatives.

By construction of the statements at the right sides, the set of interface computations of the first alternatives in the left and right sides of the congruence symbols are identical, since, by definition of $bp(b, H)$, the interface computations that satisfy b at its position in the left side are the same as those that satisfy $bp(b, H)$ at the first positions of variable components and at the input positions of external input channels in the right side.

Furthermore, the sets of interface computations corresponding to R, the other alternatives, and those involving no alternative are identical for both sides as well, by

construction of the statements at the right side. The class of computations involving none of the alternatives is included in the sets of the two alternatives at the right hand sides. But this apparent duplicity is supressed by the union operation, since the set of behaviors of a selection is the union of those of its alternatives.

The first relation may seem not to be a congruence due to the fact that the right side may deadlock on c whereas the left side does not, since its selection has other alternatives R. But this is never the case since c belongs to a matching pair p at the front of S. □

The two communications of pair p are now within the first alternative of the selections at the right sides. However, the other communication $\bar{c}$ matching c within p may still be in the scope of another selection within $S[c; A]$. When this is not the case, elimination can be accomplished with the laws of section 3, since then the LCA of c and $\bar{c}$ may be, at most, within the same alternative of a global selection and no alternate execution is generated. The following definition and lemma formulate this.

Definition 5.8 (Elimination principle under a single selection) Let $Elim\{p, A\}$ be the statement obtained in eliminating p from A, when applicability conditions hold. Let $S[\cdot]$ be a program context, and c be a communication of pair p whose other communication $\bar{c}$ is outside another selection scope within $S[\cdot]$. Then, from lemma 5.6, intuitively

$$Elim\{p, S[\ b, c; A \textbf{ or } R\ ]\} \stackrel{def}{=} Elim\{p, [\ bp(b, H), S[c; A]\ ] \textbf{ or } [true, S[R]]\}$$

$$Elim\{p, S[\ b, A[c] \textbf{ or } R\ ]\} \stackrel{def}{=} Elim\{p, [\ bp(b, H), S[A[c]]\ ] \textbf{ or } [true, S[R]]\}$$

$$Elim\{p, S[\ b, ext, A[c] \textbf{ or } R\ ]\} \stackrel{def}{=} Elim\{p, [bp(b, H), S[ext; A[c]]] \textbf{ or } [true, S[R]]\}$$

Lemma 5.9 (Elimination under a single selection) *With the same* Elim$\{p, A\}$*, and assuming that only c of p is under the scope of a selection within $S[c; A]$. Then*

$$Elim\{p, S[\ b, c; A \textbf{ or } R\ ]\} \approx [\ bp(b, H), Elim\{p, S[c; A]\}\ ] \textbf{ or } [true, S[R]]$$

$$Elim\{p, S[\ b, A[c] \textbf{ or } R\ ]\} \approx [\ bp(b, H), Elim\{p, S[A[c]]\}\ ] \textbf{ or } [true, S[R]]$$

$$Elim\{p, S[\ b, ext, A[c] \textbf{ or } R\ ]\} \approx [bp(b, H), Elim\{p, S[ext, A[c]]\}] \textbf{ or } [true, S[R]]$$

Justification The congruences follow from lemma 5.6, definition 5.8, the fact that elimination from a selection is the selection composition of the elimination from each of its alternatives, and that p is not in $S[R]$. Furthermore, the conservation principle of lemma 5.5, which applies to the first relation, is met since the disabled pairs some of whose communications have to be in R remain in the R alternatives in the right sides. □

Let A_c be a variable taking values from the set $\{c; A|A[c]|ext; A[c]\}$, where the vertical bar | is used as a value separator. Then the three laws of lemma 5.6, and the three laws of lemma 5.9, can be expressed, respectively, with the two laws

$$S[\ b, A_c \textbf{ or } R\ ] \approx [\ bp(b, H), S[A_c]\ ] \textbf{ or } [\ true, S[R]\ ]$$

and

$$Elim\{p, S[\ b, A_c \textbf{ or } R\ ]\} \ \approx\ [\ bp(b, H), Elim\{p, S[A_c]\}\ ] \textbf{ or } [true, S[R]].$$

Remark 5.10 (The empty alternative) Only for simplicity and uniformity of treatment within the algorithms below, the empty statement $\emptyset_{alt}$ is accepted as a possible value of R in the above congruences. Then $b, A[c]$ **or** $\emptyset_{alt}$ is a valid selection, By convention, $\emptyset_{alt}$ means *non-existent*, $S[\emptyset_{alt}] \approx \emptyset_{alt}$, and $[b, \emptyset_{alt}] \approx \emptyset_{alt}$. When $R \approx \emptyset_{alt}$, the right hand sides of the two congruences just before this remark are reduced to their first alternatives.

5.4 Elimination of two communications selection guards

When the other communication of p is also within the scope of another selection, the case of elimination of communication guards has to be treated separately from the other cases; since the matching communication guards are chosen to execute in parallel, and none of them could match any guard of the alternatives of the other selection, since all pairs in the current $CompPairs(I, S)$ have to be disjoint.

Lemma 5.11 (Congruence for matching guards of two selections) *Let $S[\cdot|\cdot]$ be a double program context and $(c1, c2) \in$ CompPairs(I, S). Then*

$$S[\ b_1, c_1; A_1 \textbf{ or } R_1 \mid b_2, c_2; A_2 \textbf{ or } R_2\ ]$$
$$\approx\ [\ bp(b_1, H_1) \wedge bp(b_2, H_2), S[c_1; A_1 | c_2; A_2]\ ] \textbf{ or } [\ true, S[R_1|R_2]\ ]$$

Justification The justification would go along the same lines as the one of lemma 5.6. The conservation principle of definition 5.5 is also met, since the disabled pairs are kept in another alternative of the selection of the right side. The assumption of disjoint pairs is crucial here, since it excludes two alternatives, out of the four which would be combinatorically possible. □

Definition 5.12 (Elimination principle for the guards of two selections) From lemma 5.11, the following is intuitive

$$Elim\{p, S[\ b_1, c_1; A_1 \textbf{ or } R_1 \mid b_2, c_2; A_2 \textbf{ or } R_2\ ]\}$$
$$\stackrel{def}{=}\ Elim\{p, [\ bp(b_1, H_1) \wedge bp(b_2, H_2), S[c_1; A_1 | c_2; A_2]\ ] \textbf{ or } [\ true, S[R_1|R_2]\ ]\}$$

Lemma 5.13 (Elimination of guards of two selections) *Let $S[\cdot|\cdot]$ be a double program context and $(c1, c2) \in$ CompPairs(I, S). Then*

$$Elim\{p, S[\ b_1, c_1; A_1 \textbf{ or } R_1 \mid b_2, c_2; A_2; \textbf{or } R_2\ ]\}$$
$$\approx\ [\ bp(b_1, H_1) \wedge bp(b_2, H_2), Elim\{p, S[b_1, c_1; A_1 | b_2, c_2; A_2]\}\ ] \textbf{ or } [\ true, S[R_1|R_2]\ ]$$

Justification The justification would go along the same lines as the one of lemma 5.9. The conservation principle of lemma 5.5 is also met. □

5.5 Elimination from the alternatives of two selections

When the two communications of the pair to be eliminated are within the proper alternatives of two selections, these communications do not affect the alternative selection criterion. This is determined by the evaluation of boolean guards only. In this case, instead of A_c, the set $\bar{A}_c : \{A[c]|ext; A[c]\}$ will be used. It has only two elements. The internal communication guard case is excluded.

Lemma 5.14 (Congruences for two proper selection alternatives) *Let $S[\cdot|\cdot]$ be a double program context and no communication of p be a guard. Then*

$$\begin{aligned} & S[\ b_1, \bar{A}_{c,1} \textbf{ or } R_1 \mid b_2, \bar{A}_{c,2} \textbf{ or } R_2\ ] \\ \approx\ & [\ bp(b_1, H_1) \wedge bp(b_2, H_2), S[\bar{A}_{c,1}|\bar{A}_{c,2}]\ ] \\ & \quad \textbf{or } [\ bp(b_1, H_1), S[\bar{A}_{c,1}|R_2] \textbf{ or } [\ bp(b_2, H_2), S[R_1|\bar{A}_{c,2}] \textbf{ or } [\ true, S[R_1|R_2]\ ] \end{aligned}$$

The justification would go along the same lines as the one of lemma 5.6. However, lemma 5.5 does not apply since there is no disabled communication guard. The selection of alternatives depends on the boolean conditions only, and all the combinatorically possible alternatives have to be kept.

Definition 5.15 (Elimination from two proper selection alternatives) From lemma 5.14, and under the same conditions, the following is intuitive

$$\begin{aligned} & Elim\{p, S[\ b_1, \bar{A}_{c,1} \textbf{ or } R_1 \mid b_2, \bar{A}_{c,2} \textbf{ or } R_2\ ]\} \\ \stackrel{def}{=}\ & Elim\{p, [\ bp(b_1, H_1) \wedge bp(b_2, H_2), S[\bar{A}_{c,1}|\bar{A}_{c,2}]\ ] \\ & \quad \textbf{or } [\ bp(b_1, H_1), S[\bar{A}_{c,1}|R_2]\ ] \textbf{or } [\ bp(b_2, H_2), S[R_1|\bar{A}_{c,2}]\ ] \textbf{or } [\ true, S[R_1|R_2]\ ]\} \end{aligned}$$

Lemma 5.16 (Elimination from two proper selection alternatives) *Let $S[\cdot|\cdot]$ be a double program context and no communication of p be a guard, then*

$$\begin{aligned} & Elim\{p, S[\ b_1, \bar{A}_{c,1} \textbf{ or } R_1 \mid b_2, \bar{A}_{c,2} \textbf{ or } R_2\ ]\} \\ \approx\ & [\ bp(b_1, H_1) \wedge bp(b_2, H_2), Elim\{p, S[\bar{A}_{c,1}|\bar{A}_{c,2}]\}\ ] \\ & \quad \textbf{or } [\ bp(b_1, H_1), S[\bar{A}_{c,1}|R_2] \textbf{ or } [\ bp(b_2, H_2), S[R_1|\bar{A}_{c,2}] \textbf{ or } [\ true, S[R_1|R_2]\ ] \end{aligned}$$

The justification can be done as in lemma 5.9, but the conservation lemma 5.5 does not apply here since there is no disabling of communication guards.

5.6 Elimination of a guard and an alternative communication

Finally, an hybrid case has to be considered, when only one of the communications of the pair p to be eliminated is a guard of a communications selection. The other communication of p is within a proper alternative. In this context, the following is important

Remark 5.17 (Parallelism within A) $A[c]$ of lemma 5.9 may have other communications of the current $ComFront(I, S)$ in addition to c, since parallelism is allowed within A.

In this situation, instead of $A_{c,1}$ and $A_{c,2}$ one has either $c_1; A_1$ and $\bar{A}_{c,2}$ or $\bar{A}_{c,1}$ and $c_2; A_2$. Since other communications of the front may be in the $\bar{A}_{c,i}$'s, these alternatives have to be combined with the rest of the communications selection of the eliminated guard.

Lemma 5.18 (Congruences for the hybrid case) *Let $S[\cdot|\cdot]$ be a double program context and only one of the communications of p be a guard. Then*

$$\begin{aligned} & S[\ b_1, c_1; A_1 \textbf{ or } R_1 \mid b_2, \bar{A}_{c,2} \textbf{ or } R_2\ ] \\ \approx\ & [\ bp(b_1, H_1) \wedge bp(b_2, H_2), S[c_1; A_1|\bar{A}_{c,2}]\ ] \textbf{or } [\ bp(b_2, H_2), S[R_1|\bar{A}_{c,2}] \textbf{ or } [\ true, S[R_1|R_2]\ ] \end{aligned}$$

The justification would go along the same lines as the one of lemma 5.6. Here, the combinatorically possible alternative $[bp(b_1, H_1), S[c_1; A_1|R_2]]$ does not appear at

the right side since the pairs in $CompPairs(I, S)$ are disjoint. c_1 only communicates with c_2 within $\bar{A}_{c,2}$.

Definition 5.19 (Elimination principle for the hybrid case) From lemma 5.18, the following is intuitive

$Elim\{p, S[\ b_1, c_1; A_1$ **or** $R_1 \mid b_2, \bar{A}_{c,2}$ **or** $R_2\ ]\}$
$\approx\ Elim\{p, [\ bp(b_1, H_1) \wedge bp(b_2, H_2), S[c_1; A_1|\bar{A}_{c,2}]\ ]$ **or** $[\ bp(b_2, H_2), S[R_1|\bar{A}_{c,2}]$
or $[\ true, S[R_1|R_2]\ ]\}$

Lemma 5.20 (Elimination from the hybrid case) *Let $S[\cdot|\cdot]$ be a double program context and only one of the communications of p be a guard. Then*

$Elim\{p, S[\ b_1, c_1; A_1$ **or** $R_1 \mid b_2, \bar{A}_{c,2}$ **or** $R_2\ ]\}$
$\approx\ [\ bp(b_1, H_1) \wedge bp(b_2, H_2), Elim\{p, S[c_1; A_1|\bar{A}_{c,2}]]\}$ **or** $[\ bp(b_2, H_2), S[R_1|\bar{A}_{c,2}]$
or $[\ true, S[R_1|R_2]\ ]$

The justification can be done as in lemma 5.9. The conservation principle of lemma 5.5 applies to the disabled pairs which have a communication guard in R_1. The second alternative of the statement to the right retains these alternate pairs. In addition the statement has the usual $S[R_1|R_2]$ alternative.

Remark 5.21 (Empty remainders) Continuing the convention of remark 5.10, all the above congruences have meaning when anyone or the two remainders, R_1 or R_2, are empty. Then, $S[\emptyset_{alt}|B] \approx S[A|\emptyset_{alt}] \approx S[\emptyset_{alt}|\emptyset_{alt}] \approx \emptyset_{alt}$. and the empty alternatives of their right side selections are canceled, since the corresponding combinatorial cases are non-existent.

5.7 General elimination procedure for a single pair

The above laws and definitions suggest the following algorithm for the elimination of a single communication pair $p : (c1, c2) \in CompPairs(I, S)$ from a general BC statement S.

$(failure,\ Sr) ::= GPElim(c1, c2, sel1, sel2, S)$ ::

begin
case
NoSelScope(sel1, sel2): **do** $(failure,Sr) := PElim((c1, c2),S)$ **od**
OneSelScope(sel1, sel2):
do $(failure,St) := PElim((c1, c2),S[sel1.Ac])$;
if $\neg failure$ **then** $Sr := [[bp(sel1.b, sel1.H), St]$ **or** $[true, S[sel1.R]]]$ **od**
TwoSelScopes(sel1, sel2):
do $(failure,St) := PElim((c1, c2),\ S[sel1.Ac|sel2.Ac])$;
if $\neg failure$ **then**
case
TwoGuards(sel1, sel2):
do $Sr := [[bp(sel1.b, sel1.H) \wedge bp(sel2.b, sel2.H), St]$
or $[true, S[sel1.R|sel2.R]]]$ **od**
NoGuard(sel1, sel2):
do $Sr := [[bp(sel1.b, sel1.H) \wedge bp(sel2.b, sel2.H), St]$
or $[bp(sel1.b, sel1.H), S[sel1.A_c|sel2.R]]$
or $[bp(sel2.b, sel2.H), S[sel1.R|sel2.A_c]]$ **or** $[true, S[sel1.R|sel2.R]]]$ **od**
OneGuard(sel1, sel2):
do $Sr := [[bp(sel1.b, sel1.H) \wedge bp(sel2.b, sel2.H), St]$
or $[bp(sel2.b, sel2.H), S[sel1.R|sel2.A_c]]$ **or** $[true, S[sel1.R|sel2.R]]]$ **od**
endcase
od
endcase
end

The boolean functions *NoSelScope*, *OneSelScope* and *TwoSelScopes* correspond to the cases introduced above. Variables *sel1* and *sel2* are assumed to contain information about the specific selection embedding communication ci, for $i = 1, 2$. They are structure typed, with fields *Scope*, *Ac*, *b*, *H*, and *R*. *Scope* is a boolean indicating whether or not the communication ci is under the scope of a selection. When these fields of the two variables are true, it is implied that the two selections are distinct, see remark 5.1. *Ac*, *b*, *H*, and *R* correspond to the A_c variables and to the b's, H's, and R's introduced in lemma 5.11 above. The three boolean selection functions of the inner case statement select the embedding possibilities of the communications to be eliminated when the pair is under two selections.

Remark 5.22 (*GPElim* processing of empty reminders) Following remarks 5.10 and 5.21, *GPElim* ignores empty alternatives when forming the selections of the Sr's in the four cases involving selections. It is assumed that, before processing a statement, $\emptyset_{alt}$ is added as the rightmost alternative of all selections.

Lemma 5.23 (General elimination of a communication pair) *The disjoint pair* $(c1, c2)$ *has been eliminated in* Sr *resulting from* GPElim *when* failure$= false$. *Then* $Sr =_{\mathcal{O}} S$, *for any* $\mathcal{O}$ *not containing inner communications* $c1$ *and* $c2$.

Justification The first case, *NoSelScope*, of the outer case statement amounts to the equivalence $PElim((c1, c2),S)=_{\mathcal{O}} S$ which has been proved in [2]. For the remaining three cases, the equivalence follows from the congruences of lemmas 5.9,

5.13, 5.16 and 5.20, monotonicity of the **or** operator, and the fact that, $St =_{\mathcal{O}} S[sel1.Ac]$, $St =_{\mathcal{O}} S[sel1.Ac|sel2.Ac]$, where St is the temporary eliminated form of the procedure. The two latter equivalences also correspond to the selection free case of [2]. □

Remark 5.24 (Limitations of the algorithm) *In general, the automatic computation of* bp *is not possible, since it may involve automatic invariant generation. In these situations the algorithm does not apply. Nevertheless, in many scenarios the backward propagation can be computed. This is so in the simple but frequent case where* $b = true$.

5.8 Elimination proof construction algorithm

In the following, $GElim\{p, S\}$ denotes the statement Sr resulting from procedure *GPElim* of last subsection assuming that applicability conditions hold, and hence $failure = false$, and that it has been possible to compute the $bp(b, H)$'s.

Lemma 5.25 (Elimination of a set of disjoint competing pairs) *Let* n_{cp} *be the cardinality of* CompPairs(I,S)*, all of whose pairs* $cp_i, i = 1, \cdots, n_{cp}$ *are disjoint. Then*

$$\begin{aligned} S &=_{\mathcal{O}} GElim\{cp_1, GElim\{cp_2, \cdots, GElim\{cp_{n_{cp}}, S\}\cdots\}\} \\ &=_{\mathcal{O}} GElim\{cp_{p(1)}, GElim\{cp_{p(2)}, \cdots, GElim\{cp_{p(n_{cp})}, S\}\cdots\}\} \end{aligned}$$

where $< p(1), \cdots, p(n_{cp}) >$ *is any permutation of* $< 1, \cdots, n_{cp} >$.

This lemma was mathematically justified for the special case of selection-free BCS's in [2]. The proof here would be very similar. The lemma says that the order of elimination of the pairs is unimportant. Based on it, and assuming that all the pairs are mutually disjoint at any point of the elimination, the following communication elimination algorithm is derived:

```
(failure,deadlock, Sr)::= DisjPairsElim(I, S) ::
 begin
  failure:= false ; deadlock:= false ; Sr:= S ;
  (existsPair,c1,c2,sel1,sel2) := ObtainCompPair(I, Sr) ;
  while ¬failure ∧ existsPair
   do (failure, Sr) := GPElim(c1,c2,sel1,sel2,Sr);
    if ¬failure then
     (existsPair,c1,c2,sel1,sel2) := ObtainCompPair(I, Sr) od;
  if ¬failure then if ComFront(I, Sr)≠ ∅ then deadlock := true
 end
```

Procedure *GPElim* has been given above. *ObtainCompPair* obtains a communication pair $(c1, c2)$ from the current communication front, as well as information, within *sel1* and *sel2*, on the selections embedding the two communications. The search for a pair within this procedure is done over a new communication front in each invocation of *ObtainCompPair*, since the previous pair has been eliminated in the loop, and this may uncover new communications. When no pair is found, *existsPair* is returned with the value *false*. For any selection embedding inner com-

munications, each alternative is processed in a different invocation of *GPElim*; the added $\emptyset_{alt}$'s, as observed in remarks 5.10, 5.21, and 5.22, indicate termination of the selection.

Theorem 5.26 (Correctness of *DisjPairsElim*) *The Sr resulting from* DisjPairsElim *when* failure=deadlock=false *is such that* $Sr =_{\mathcal{O}} S$*, and Sr has no inner communication statements. When* deadlock=true*, S is not deadlock-free.*

Justification When at some iteration *failure=true*, the procedure exits with failure. Due to lemmas 5.23 and 5.25 the equivalence of the theorem is true at the exit point of the while loop, at the last if statement, when *failure=false*. Under this condition, *existsPair* has to take the value *false* as well at the same point, indicating that no matching pair could be formed with the communication front. Therefore, when the communication front at this point has some communication, execution of S would wait forever at these communications and, correspondingly, the procedure exits with *deadlock=true*. Otherwise, since in this framework execution waits on pending communications only, there is no possibility of deadlock and, correspondingly, the procedure exits with *deadlock=false*. □

6 Elimination of non-disjoint communication pairs

The following is a statement with a non-disjoint pair

$$\left[\begin{array}{l}\cdots a1:\alpha \Leftarrow v_1;\cdots \,||\, \cdots a2:\alpha \Rightarrow v_2;\cdots \\ \qquad\qquad\qquad\qquad ||\cdots a3:\alpha \Rightarrow v_3;\cdots ||\cdots a4:\alpha \Leftarrow v_4;\cdots\end{array}\right]$$

where the communications have been labeled for easy reference. In this case,

$$CompPairs(\{\alpha\},S):\{p1:(a1,a2),p2:(a1,a3),p3:(a2,a4),p4:(a3,a4)\}.$$

Pairs $p1$ and $p2$ are non-disjoint, since they share $a1$. There are other non-disjoint pairs: those sharing $a2$, those sharing $a3$, and those sharing $a4$. Here is another example

$$\left[\begin{array}{l}\cdots a1:\alpha_1 \Leftarrow v_1;\cdots ||\cdots [b_1, a21:\alpha_1 \Rightarrow v_2; A_1 \textbf{ or } b_2, a22:\alpha_2 \Leftarrow v_3; A_2];\cdots \\ \qquad\qquad\qquad\qquad ||\cdots a3:\alpha_2 \Rightarrow v_4 \cdots ||\cdots a4:\alpha_1 \Leftarrow v_1;\cdots\end{array}\right]$$

where

$$CompPairs(\{\alpha_1,\alpha_2\},S):\{p1:(a1,a21),p2:(a21,a4),p3:(a22,a3)\}.$$

$p1$ and $p2$ are non-disjoint. No other pair is disjoint.

Lemma 6.1 (Elimination of non-disjoint pairs) *Let* $\bar{p}:\{p_1,\cdots,p_n\}$ *be the list of communication pairs,* $p_i \in$ CompPairs(I,S)*,* $i=1,\cdots,n$ *, sharing the same inner communication statement c. Then*

$$S \;=_{\mathcal{O}}\; [true, GElim\{p_1,S\} \textbf{ or } \cdots \textbf{ or } true, GElim\{p_n,S\}]$$

where GElim *eliminates a single pair with the assumptions stated in subsection 5.8. All the pairs of S in $\bar{p}$ have been eliminated at the statement to the right.*

Justification Due to lemma 5.23, $S =_{\mathcal{O}} GElim\{p_k, S\}$ for $k = 1, \cdots, n$. Also $S =_{\mathcal{O}} \mathbf{OR}_{k\in\mathcal{K}} GElim\{p_k, S\}$ for any $\mathcal{K} \subseteq \{1, \cdots, n\}$. But no behavior of S is lost in the statement to the right of the equivalence symbol, since all the pairs in $\bar{p}$ have communications in *ComFront(I,S)* and share the same communication c. Therefore, the first inner communication event corresponding to any pair in $\bar{p}$ disables the other pairs in $\bar{p}$. Hence, the set of possible interface behaviors of S can be expressed as a partition whose classes are the interface behaviors of $GElim\{p_i, S\}$ for $i = 1, \cdots, n$. The **or** statement of the lemma includes precisely all of them. □

($failure$, $Sres$)::= $NonDisjPairsElim(I, S)$ ::
begin
$failure$:= false ; ($existNDPairs$, c, $altCom$, n) := $ObtainNDPairs(I, S)$;
if $existNDPairs$ **then**
do i:=1; $selJoint$:= $DetermineSelJoint(c, S)$;
while $\neg failure \wedge i \leq n$
do $selAlt$:= $DetermineSelAlt(c, altCom(i), S)$;
($failure$, St) := $GPElim(c, altCom(i), selJoint, selAlt, S)$;
if $\neg failure$ **then**
do ($failure$, $Sr(i)$):= $NonDisjPairsElim(I, St)$; i:=i+1 **od**
od;
if $\neg failure$ **then** $Sres$:= $[true, Sr(1)$ **or** $\cdots$ **or** $true, Sr(n)]$
od
else $Sres$:= S
end

Lemma 6.1 suggests the above recursive elimination algorithm. n is the number of pairs in S sharing c, as determined by *ObtainNDPairs*, which also stores the other communication of the i-th pair in $altCom(i)$. Variables *selJoint* and *selAlt* correspond to *sel1* and *sel2* of *GPElim*. The former concerns the shared communication c, and is computed only once.

Theorem 6.2 (Correctness of *NonDisjPairsElim*) *Statement* Sres *resulting from* NonDisjPairsElim *when* failure=false *is such that* $Sres =_{\mathcal{O}} S$, *and, should its* ComFront(I, S) *have any communication pairs, they are disjoint.*

Justification The recursive algorithm terminates since the number of pairs is finite, the statement being BC. The result follows from lemma 6.1 and structural induction. □

The elimination of the remaining pairs of *Sres* can be attempted with procedure *DisjPairsElim* of section 5. An algorithm combining the two is the following.

```
(failure, deadlock, S)::= CompleteComsElim(I, S) ::
  begin
    failure := false; deadlock := false;
    while ¬failure ∧ CompPairs(I, S)≠ ∅
      do (failure,S) := NonDisjPairsElim(I, S);
       if ¬failure then do (existsPair,c1,c2,sel1,sel2) :=ObtainCompPair(I, S);
        if existsPair then (failure,S):=GPElim(c1,c2,sel,sel2,S) od
      od;
     if ¬failure then if ComFront(I, S)≠ ∅ then deadlock := true
  end
```

Theorem 6.3 (Correctness of *CompleteComsElim*) *The statement S resulting at the exit from* CompleteComsElim *when* failure=deadlock=false *is such that* $S =_{\mathcal{O}} S'$*, where* S' *is the statement S at the entry point, and no eliminable pairs remain in S. When, at the exit,* deadlock=true*,* S' *is not deadlock-free.*

Justification At the entry point of the while loop, a new iteration is started whenever there are still competing pairs. By theorem 6.2, at the if statement of the loop body, all pairs of the current front are guaranteed to be disjoint and, whenever a pair still exists it is eliminated by *GPElim*, whose precondition of disjoint pairs is fulfilled. The loop is continued while there still remain pairs to be eliminated. If, after some iteration, *failure=true*, exit with failure occurs.

At the exit point of the loop, at the last if statement, *CompPairs* is empty when *failure=false*. If then there still remain communications in the front, the program will wait forever at these inner un-matched communications and, correspondingly, the algorithm exits with *deadlock=true*. □

7 Conclusions and further work

Formal sequentialization, of distributed programs, has been introduced as an equivalence simplification proof, one of whose components is formal internal communication elimination. A new algorithm for the automatic construction of these elimination proofs has been proposed and mathematically justified. A class of programs whose inner communications may be under the scope of selections and with possibly non-disjoint communication pairs are handled by the proposed algorithm. The results apply to bounded communication statements, and are built on top of prior work which has been summarized: a suitable equivalence criterion for communication elimination laws, justification of all the laws in the new equivalence criterion, a communication elimination algorithm for BC selection free statements. Extension to a typical non-BC case has been covered as well.

Sequentialization proofs have many output possibilities. Each one of them affects the ordering of the external communication offerings of the resulting program. At one extreme one has total sequentialization, and at the other, one has a maximum parallelism of the offerings after inner communication elimination. This point has been illustrated with a communication queue example. Further work on the

control of these possibilities, on the automatic construction of other parts of sequentialization proofs, and on the extension of the class of non-selection-free BC statements which is currently handled should be undertaken.

Acknowledgement

We thank the encouragement received during the last years from Zohar Manna, Bernd Finkbeiner and Tomas Uribe.

References

[1] R. Alur, R.K. Brayton, T.A. Henzinger, S. Qadeer, and S.K. Rajamani. Partial-Order Reduction in Symbolic State-Space Exploration. *Formal Methods in System Design*, 18(2):97–116, March 2001.

[2] Francesc Babot, Miquel Bertran, and August Climent. A Static Communication Elimination Algorithm for Distributed System Verification. In Kung-Kiu Lau and Richard Banach, editors, *Formal Methods and Software Engineering. 7th International Conference on Formal Engineering Methods, ICFEM 2005*, volume 3785 of *LNCS*, pages 375–389, Manchester, England, November 2001. Springer.

[3] Francesc Babot, Miquel Bertran, Jordi Riera, Ricard Puig, and August Climent. Mechanized Equivalence Proofs of Pipelined Processor Software Models. In *Actas de las III Jornadas de Programación y Lenguajes*, pages 91–104, Alicante, November 2003. Universitat d'Alacant.

[4] Miquel Bertran, Francesc Babot, August Climent, and Miquel Nicolau. Communication and Parallelism Introduction and Elimination in Imperative Concurrent Programs. In Patrick Cousot, editor, *Static Analysis. 8th International Symposium, SAS 2001*, volume 2126 of *LNCS*, pages 20–39, Paris, France, July 2001. Springer.

[5] Miquel Bertran, Francesc-Xavier Babot, and August Climent. An Input/output Semantics for Distributed Program Equivalence Reasoning. *Electronic Notes in Theoretical Computer Science*, 137(1):25–46, July 2005.

[6] Edmund M. Clarke, Orna Grumberg, and D.E. Long. Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

[7] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. The MIT Press, 1999.

[8] Willem-Paul de Roever, Franck de Boer, Ulrich Hanneman, Yassine Lakhnech, Mannes Poel, and Job Zwiers. *Concurrency Verification: Introduction to Compositonal and Noncompositional Methods*. Cambridge University Press, 2001.

[9] Tzilla Elrad and Nissim Francez. Decomposition of Distributed Programs into Communication Closed Layers. *Science of Computer Programming*, 2:155–173, 1982.

[10] Paul Le Guernic, Jean-Pierre Talpin, and Jean-Christophe Le Lann. Polychrony for System Design. *Journal of Circuits, Systems and Computers. Application Specific Hardware Design*, August 2002.

[11] Gerald Holtzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[12] INMOS-Limited. *Occam Programming Manual*. Prentice Hall, 1985.

[13] INMOS-Limited. *Occam 2 Reference Manual*. Prentice Hall, 1988.

[14] G. Jones. *Programming in Occam*. Prentice Hall, 1987.

[15] R. Kurshan, V. Levin, M. Minea, D. Peled, and H. Yenigun. Static Partial Order Reduction. In B. Steffen, editor, *Proceedings of TACAS'98*, volume 1384 of *LNCS*, pages 335–357, Noordwijkerhout, The Netherlands, June 1-4 1998. Springer.

[16] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems. Specification*. Springer, 1991.

[17] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems. Safety*. Springer, 1995.

[18] Zohar Manna, Anca Browne, Henny Sipma, and Tomas Uribe. Visual Abstraction for Temporal Verification. In *Algebraic Methods and Software Technology, AMAST'98*, volume 1548 of *LNCS*, pages 28–41. Springer, 1998.

[19] K.L. McMillan and D.L. Dill. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic, 1993.

[20] S. Merz. Model checking: A tutorial overview. In F.Cassez, editor, *Modeling and Verification of Parallel Processes*, volume 2067 of *LNCS*, pages 3–38. Springer, 2001.

[21] M.Muller-Olm, D.A. Schmit, and B. Steffen. Model Checking: A Tutorial Introduction. In G.File A.Cortesi, editor, *Static Analysis, Proc. 6th Intl. Symp. SAS'99*, volume 1694 of *LNCS*, pages 330–354, Venice, Italy, September 1999. Springer.

[22] A.W. Roscoe and C.A.R. Hoare. The laws of OCCAM programming. *Theoretical Computer Science*, 60:177–229, 1988.

[23] Henny B. Sipma, Tomas E. Uribe, and Zohar Manna. Deductive Model Checking. *Formal Methods in System Design*, 15(1):49–74, July 1999.

[24] Rob J. van Glabbeek. *Handbook of Process Algebra, chapter The Linear Time - Branching Time Spectrum I: The semantics of Concrete, Sequential Processes*. Elsevier, 2001.

[25] Karen Yorav and Orna Grumberg. Static Analysis for State-space Reductions. *Formal Methods in System Design*, 25:67–96, 2004.