Available online at www.sciencedirect.com

SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science 137 (2005) 25–46

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

An Input/Output Semantics for Distributed Program Equivalence Reasoning

Miquel Bertran[1] Francesc-Xavier Babot[2] August Climent[3]

Informatica La Salle, Universitat Ramon Llull, Barcelona

Abstract

A new notion of input/output equivalence of distributed imperative programs, with synchronous communications, is introduced. It preserves the input/output relation, encompassing both, initial/final state and communication channel values. For its mathematical justification, the semantic framework of Manna and Pnueli, based on finite transition systems and reduced behaviors, is extended with the notion of input/output behavior. A set of laws for the equivalence is overviewed. A deduction rule for the substitution of references to input/output equivalent procedures is defined and justified in the new semantics. The rule is applied to decompose *distributed program simplification* proofs, introduced in a prior work, which use the laws to establish the equivalence between a sequential and a parallel communicating program. They include communication elimination as one of their steps. An outline of one of such proofs, for a pipelined processor model, is included.

Keywords: Distributed programs, parallel programs, input/output equivalence, equivalence preserving transformations, verification, program simplification, synchronous communications, laws of distributed programs.

1 Introduction

Imperative languages with explicit parallelism and communication statements provide an intuitive, explicit, and complete framework to express distributed and concurrent programs and system models, with the clarity required for verification. OCCAM [11,12,13], the *simple programming language*, SPL, of Manna and Pnueli [15,16], PROMELA of the SPIN model checker [10], and

[1] Email: miqbe@salleURL.edu
[2] Email: fbabot@salleURL.edu
[3] Email: augc@salleURL.edu

doi:10.1016/j.entcs.2005.01.038

the *shared-variable language*$^{++}$, SVL^{++}, in [7] are representatives of these imperative notations.

Equivalence reasoning is heavily used for numbers, matrices, and other fields. However, for imperative programs, although having the potentiality of being a very intuitive verification activity, it has not been explored in concurrency and distribution. Simplification via internal communication elimination [2] is an equivalence proof that, since it decreases the size of the state vector, could complement other proof methods, such as model checking [10,17,5] and interactive verification [3,14]. It is based on the application of a set of laws, suitable for that purpose, as reductions to a program. The laws depend on the notion of equivalence and on the fairness assumptions [15].

A set of laws for OCCAM was given in [18]. Rather than simplification via communication elimination, the focus there was to obtain normal forms and to define the semantics of the notation. Some laws for SPL are given in [15], with an SPL semantics based on fair transition systems (FTS), but communication elimination laws are not given there. In the framework of SVL^{++}, some laws are given in [7] but they do not suffice for communication elimination.

The first set of relations suitable for communication elimination was given and proved sound in [2], showing the necessity of avoiding strong fairness. A communication elimination proof of a distributed fast Fourier transform was outlined there as well. In this earlier work, the notion of equivalence was assimilated to congruence, a very strong equivalence. This had the drawback of limiting the formulation of most communication elimination laws to unidirectional refinements. Clearly, the need of working with a weaker equivalence, where all laws could be formulated with it, avoiding the asymmetry of refinement relations, was outstanding.

This paper introduces a notion of equivalence, *input/output equivalence*, weaker than congruence, but strong enough to preserve the input/output relation of the programs and to lead to laws for communication elimination. It is justified in a semantics which extends the Manna-Pnueli framework as the main contribution of the paper, where each statement denotes a set of *input/output behaviors*. They extend the notion of *reduced behavior* given in [15]. In order to capture the complete input/output relation, the former adds to the latter a recording of the values traversing synchronous channels, in addition to the usual data state variables. This reflects the fact that values may be input or output via channels, as well as via proper variables. In this context, the grounding work with streams introduced in [4] is related, but in our concrete imperative program context we needed a new model where both channel and variable values are taken into account. The work on compatibility of components [6] is also related.

An important ingredient of equivalence reasoning is substitution of procedure reference statements of two equivalent procedures. It would allow proof decomposition. Conditions for the validity of such substitutions are also given and their justification outlined. This establishes the necessary base theory for formal input/output equivalence reasoning with distributed programs.

The paper is organized as follows. After a section on the notation, including modular procedures, and some notions needed later in the paper, the concept of input/output behavior is introduced. Composition rules to obtain io-behaviors of sequential, parallel and selection compositions are detailed, in preparation for the justification of the substitution rule. Input/output equivalence of procedures is covered next, with the substitution rule and its justification. These sections contain the main contributions of the paper. A summary of the laws and of distributed program simplification proofs follows, together with their application to a pipelined processor model. This is an overview without proofs for illustrative purposes only. A brief section on conclusions and further work ends the paper.

2 Programming Notation

2.1 Syntax of the Basic Notation

Programs will be written in a reduced version of SPL, which is general enough to express any practical program. Its syntax is presented now. The basic statements are **Skip**, **Nil**, **Stop**, the assignment $u{:=}\ e$, send $\alpha \Leftarrow e$, and receive $\alpha \Rightarrow u$. We limit our work 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. Communication statements will be referred to more simply as *communications*. The skip statement involves a transition in the underlying fair transition system, but without any effect on the data variables. The nil statement makes its pre and post control locations equivalent, involving no transition. The stop statement has neither the transition nor the label equivalence relation. Both channels and variables are declared globally before their usage. The rest of the notation is defined recursively.

Concatenation is n-ary: $[S_1; \cdots; S_n]$. The iterations are $[\textbf{while } c \textbf{ do } S]$, where c is a boolean expression, and $[\textbf{loop forever do } S]$, which is defined as $[\textbf{while } \textit{true} \textbf{ do } S]$. The *cooperation* statement is also n-ary: $[S_1|| \cdots ||S_n]$. Its substatements S_j are the *top parallel statements* of the cooperation statement, which is the *minimal common ancestor* of them. It will be assumed throughout the paper 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 *regular selection* and the *communication 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*.

2.2 Modular Procedures

This notion was introduced in [2] combining the notions of SPL *module* [15,8] and *procedure*. As modules, modular procedures can be composed in parallel, but may be invoked by *procedure reference* statements, which make explicit the names of all the interface channels and variables. Common variables are prohibited. The notation $r ::= P(p)$ will be used for a procedure reference, where r and p stand for the result and parameter lists of the interface, and P is the procedure name. Modular procedures will be referred to more simply as *procedures*. An example of procedure is given below. Its *procedure reference* stands at the left, and the *procedure body* at the right.

$$(r, cr) ::= Pc(p, cp) :: \left[\begin{array}{l} \textbf{out } r : \textbf{integer} \\ \textbf{out } cr : \textbf{channel of integer} \\ \textbf{external in } p : \textbf{integer} \\ \textbf{external in } cp : \textbf{channel of integer} \\ \textbf{local } a1, a2 : \textbf{integer} \\ \textbf{local } c : \textbf{channel of integer} \\ \quad \left[\begin{array}{l} cp \Rightarrow a1; \\ a1 := a1 + p; \\ c \Leftarrow a1; \\ \textbf{skip} \end{array}\right] \| \left[\begin{array}{l} cp \Rightarrow a2; \\ c \Rightarrow r; \\ a2 := r + a2; \\ cr \Leftarrow a2 \end{array}\right] \end{array}\right]$$

Notice that r and p are variables whereas cr and cp are channels. r and cr are the *results*, and p and cp are the *parameters* of the procedure. The exact meaning of modes **out** and **external in** is not important here, since processes are disjoint and communication is point to point and half-duplex. The declaration at the head of a procedure body will be omitted often in this work. No common variables or channels are allowed.

Semantics of the reference statement It is unchanged with the *replacement* of the reference by the procedure body, with a renaming of variables and channels when necessary. It has to be consistent with the reverse operation of *encapsulation* of a part of a program within a procedure.

The set $\mathcal{O}$ of observed variables of a procedure Contains all proper variables in the interface, and an auxiliary *channel variable* for each channel in the interface. The set $\mathcal{O}$ is also referred to as *interface set*.

A channel variable records, as a triplet, the value passed at a communication event, a count reflecting the order of the channel event, and an input/output mark (i,o). When the event is internal, a dot replaces the input/output mark. For the above procedure, this set is $\mathcal{O} : \{r, p, cr, cp\}$, where cr and cp are the auxiliary variables associated to the channels.

2.3 Basic Notions for the Semantics

The semantics of the specific SPL variant which we use follows the style of Manna and Pnueli, based on fair transition systems (FTS) [15,16]. In the following, some of its elements are summarized. A full account is in [2].

A *computation* is a sequence of states starting at an initial state with a transition taking any state to its successor. A *reduced behavior*, with respect to a set $\mathcal{O}$ of *observed variables*, is a computation where both its components of variables outside the observed set and stuttering steps (i.e. idling transitions) are deleted. The set $\mathcal{O}$ contains only proper variables. Transitions correspond to atomic actions, which are associated to statements. A *program context* $P[\ \_\ ]$ is a program P one of whose statements corresponds to a hole to be filled-in with an arbitrary statement S. With some abuse of notation $P[S]$ will denote a program context, where S denotes the arbitrary statement placed in the hole. Some laws are congruence relations between statements. Statement S_1 *refines* S_2, written $S_1 \sqsubseteq_{\mathcal{O}} S_2$, when for any program context $P[\cdot]$, any reduced behavior of $P[S_1]$ is also a reduced behavior of $P[S_2]$. S_1 is *congruent* to S_2, written $S_1 \approx_{\mathcal{O}} S_2$, when $S_1 \sqsubseteq_{\mathcal{O}} S_2$ and $S_2 \sqsubseteq_{\mathcal{O}} S_1$. These relations are defined with respect to a set $\mathcal{O}$ of observed variables.

Some extended notions, needed in the paper are introduced next. An *input/output computation* (*io-computation*) records the value histories of both the variables and the channels of the procedure body during an execution. It has a row for each value change and a column for each variable or channel.

An io-computation adds to a computation a column for each channel. Whereas a computation is a sequence of states only, an io-computation is a sequence of states where the values crossing channels are also recorded. Groups of computations will be represented as *schemas*, which have *value variables*. Computations have just values (integers, booleans, etc...). A triplet (*value, count, i/o indication*) is associated to each new value of a channel variable.

The following is an *io-computation schema* of the procedure above.

		r	p	cr	cp	$a1$	$a2$	c
0	*initial*	x	$p1$	x_T	x_T	x	x	x_T
1	$cp \Rightarrow a1$	x	$p1$	x_T	$cp1, 1, i$	$cp1$	x	x_T
2	$cp \Rightarrow a2$	x	$p1$	x_T	$cp2, 2, i$	$cp1$	$cp2$	x_T
3	$a1 := a1 + p$	x	$p1$	x_T	$cp2, 2, i$	$cp1 + p1$	$cp2$	x_T
4	$c \Leftarrow a1 \| \| c \Rightarrow r$	$cp1 + p1$	$p1$	x_T	$cp2, 2, i$	$cp1 + p1$	$cp2$	$cp1 + p1, 1, \cdot$
5	$a2 := r + a2$	$cp1 + p1$	$p1$	x_T	$cp2, 2, i$	$cp1 + p1$	$cp1 + p1 + cp2$	$cp1 + p1, 1, \cdot$
6	$cr \Leftarrow a2$	$cp1 + p1$	$p1$	$cp1 + p1 + cp2, 1, o$	$cp2, 2, i$	$cp1 + p1$	$cp1 + p1 + cp2$	$cp1 + p1, 1, \cdot$

x denotes any value and x_T any triplet. $p1$, $cp1$, $cp2$, etc ... are value variables, whereas $a1$, $a2$, r, and p are program variables. cp, cr, and c are auxiliary channel variables. Giving integer values to $p1$, $cp1$, and $cp2$, specific io-computations would be obtained. Leaving aside the initial row, each row corresponds to the state resulting from the transition of the statement at the second column. The transition of row 4 is the joint transition of the synchronous communication over channel c. All computations are of infinite length. Thus the last row corresponds to a *terminal* state, repeating itself implicitly by idle transition firings. A computation schema could be obtained by deleting the cr, cp, c, and the two left columns, and then deleting, as in [15], any row which equals its predecessor but not all of its successors.

3 Input/Output Behaviors

3.1 Basic Notions

An input/output behavior is a procedure execution trace seen from its outside.

Definition 3.1 (Input/Output behavior of a procedure) An input/output behavior of a procedure, also referred to as *io-behavior*, is the result of deleting from an io-computation all columns of variables not belonging to $\mathcal{O}$, and then deleting any row which equals its predecessor but not all of its successors.

The condition in the last deletion is necessary since the infinite implicit repetitions of the last row should not be deleted. Due to event counters, consecutive events are not deleted when their values are equal. This should be so since they may correspond to two inputs of the procedure function. Thus all channel events are represented in a io-behavior by at least one row.

An io-behavior has one row for each value change of a result variable $v \in \mathcal{O}$. A parameter variable never changes its value, unless it is also a result. Input and output channel variables exhibit value changes. The following io-behavior schema results from the io-computation schema above.

	r	p	cr	cp
0	x	$p1$	x_T	x_T
1	x	$p1$	x_T	$cp1, 1, i$
2	x	$p1$	x_T	$cp2, 2, i$
4	$cp1 + p1$	$p1$	x_T	$cp2, 2, i$
6	$cp1 + p1$	$p1$	$cp1 + p1 + cp2, 1, o$	$cp2, 2, i$

Rows 3 and 5 have been deleted since they are equal to their predecessors 2 and 4 respectively. Suppose now that $cp1 = cp2$, then row 2 would not be deleted due to the new value, 2, of the counter field of the cp column.

Definition 3.2 (Component of an io-behavior) An io-behavior component is the list of values, a column, corresponding to a variable of $\mathcal{O}$. But any value in the list which equals its predecessor but not all of its successors is deleted. There are both proper and channel variable components.

Definition 3.3 (Equivalence of io-behaviors) Two io-behaviors are equivalent when they share the same interface set, and the two components of the same variable of both are equal.

The order of value changes among different components is lost in io-behaviors, but not the order of changes within the same component. Equivalence only requires equality of homologous component lists.

3.2 Composition of io-behaviors

This subsection introduces operations between io-behaviors, needed later for the justification of the substitution rules.

3.2.1 Sequential composition

The io-behaviors of a sequential composition are formed by *post-coupling* an io-behavior of its second statement to an io-behavior of the first. In the binary composition $[r1 := P_1(p1)]$; $[r2 := P_2(p2)]$, in general, $\mathcal{O}_1 \neq \mathcal{O}_2$, and $\mathcal{O}_1 \cap \mathcal{O}_2$ may or may not be empty. Given schema b_1 of P_1, the schema b_2 of P_2 depends on the values of the last row of b_1. The schema $b_{1;2}$ of $P_1; P_2$ corresponding to b_1 and b_2 has as many components as variables in $\mathcal{O}_1 \cup \mathcal{O}_2$. It is formed by *post-coupling* b_2 after b_1. Informally, the components of b_2 go after the ones of b_1 , but certain values of the first row of b_2 have to equal their homologous ones in the last row of b_1. More specifically, the post-coupling is done as follows:

(i) Proper variable components. Cases:

(a) $v \in \mathcal{O}_1 \cap \mathcal{O}_2$: b_2 has to be such that the values of the first positions (*initial* row) of the components of such v's equal the last value of their homologous components of b_1. Such a b_2 can always be found, since corresponding values share the same type, when v is a parameter of P_2, and undefined values (x) can be changed to any value, when v is a result of P_2.

(b) $v \notin \mathcal{O}_1 \cap \mathcal{O}_2$ and $v \in \mathcal{O}_1$: The last values of such v components of b_1 are propagated into the future, over the b_2 selected as in (a).

(c) $v \notin \mathcal{O}_1 \cap \mathcal{O}_2$ and $v \in \mathcal{O}_2$: The first values of such v components of the b_2 above are propagated into the past, over b_1.

(ii) Channel variable components. Cases:

(a) $c \in \mathcal{O}_1 \cap \mathcal{O}_2$: Any undefined initial triplets of such c components of b_2 are set equal to the last triplets of their homologous components of b_1. The counts of the rest of the triplets are increased accordingly.

(b) $c \notin \mathcal{O}_1 \cap \mathcal{O}_2$ and $c \in \mathcal{O}_1$: The last triplets of such c components of b_1 are repeated into the b_2 portion, in other words into the future.

(c) $c \notin \mathcal{O}_1 \cap \mathcal{O}_2$ and $c \in \mathcal{O}_2$: The b_1 portions of such c components are filled in with undefined triplets.

Example 3.4 In the composition $[cr1, r := P_1(cp1, p)]$; $[cr2, p := P_2(cp2, r)]$, a result variable r of the first is a parameter of the second, and a parameter p of the first is a result of the second. The respective interface sets are $\mathcal{O}_1$: $\{cr1, r, cp1, p\}$ and $\mathcal{O}_2$: $\{cr2, p, cp2, r\}$. No channel is shared. In addition, $\mathcal{O}_1 \cap \mathcal{O}_2 = \{r, p\}$ and we assume that $\mathcal{O} : \mathcal{O}_1 \cup \mathcal{O}_2 : \{cr1, r, cp1, p, cr2, cp2\}$. The following schemas, where the v_{ij}'s are program variables, are meant to be the io-behaviors b_1 of P_1, b_2 of P_2, and their post-coupling $b_1; b_2$.

$\mathbf{b_1}$

		$cr1$	r	$cp1$	p
0	*initial*	x_T	x	x_T	x_2
1	$cp1 \Rightarrow v_{11}$	x_T	x	$x_3, 1, i$	x_2
2	$cr1 \Leftarrow v_{21}$	$x_4, 1, o$	x	$x_3, 1, i$	x_2
3	$r := v_{31}$	$x_4, 1, o$	x_5	$x_3, 1, i$	x_2

$\mathbf{b_2}$

		$cr2$	p	$cp2$	r
0	*initial*	x_T	x	x_T	y_2
1	$p := v_{12}$	x_T	y_3	x_T	y_2
2	$cp2 \Rightarrow v_{22}$	x_T	y_3	$y_4, 1, i$	y_2
3	$cr2 \Leftarrow v_{32}$	$y_5, 1, o$	y_3	$y_4, 1, i$	y_2

$\mathbf{b_1; b_2}$

		$cr1$	r	$cp1$	p	$cr2$	$cp2$
0	*initial*	x_T	x	x_T	x_2	x_T	x_T
1	$cp1 \Rightarrow v_{11}$	x_T	x	$x_3, 1, i$	x_2	x_T	x_T
2	$cr1 \Leftarrow v_{21}$	$x_4, 1, o$	x	$x_3, 1, i$	x_2	x_T	x_T
3	$r := v_{31}$; *initial*	$x_4, 1, o$	$(y_2 :=) x_5$	$x_3, 1, i$	$(x :=) x_2$	x_T	x_T
4	$p := v_{12}$	$x_4, 1, o$	x_5	$x_3, 1, i$	y_3	x_T	x_T
5	$cp2 \Rightarrow v_{22}$	$x_4, 1, o$	x_5	$x_3, 1, i$	y_3	x_T	$y_4, 1, i$
6	$cr2 \Leftarrow v_{32}$	$x_4, 1, o$	x_5	$x_3, 1, i$	y_3	$y_5, 1, o$	$y_4, 1, i$

Within $b_1; b_2$, the y_2 of b_2 becomes x_5, and the x of b_2 is coerced to x_2.

3.2.2 Parallel Composition

The io-behaviors of parallel compositions are formed by *side-coupling* io-behaviors of their component statements. In the following parallel composition $[r1 := P_1(p1)] \mid\mid [r2 := P_2(p2)]$, in general $\mathcal{O}_1 \neq \mathcal{O}_2$ and $\mathcal{O}_1 \cap \mathcal{O}_2$ may or may not be empty. Given schemas b_1 of P_1 and b_2 of P_2 , the io-behavior schema $b_{1||2}$ of $P_1||P_2$ corresponding to b_1 and b_2 has as many components as variables in $\mathcal{P} : (\mathcal{O}_1 \cup \mathcal{O}_2) - \mathcal{O}_I$ where $\mathcal{O}_I \subseteq (\mathcal{O}_1 \cup \mathcal{O}_2)$ is the set of proper and channel variables declared as internal, non-observable, in the composition. Channels in $\mathcal{O}_I$ give rise to internal communication events. We assume disjointness of P_1 and P_2 and deadlock-freeness of their parallel composition. In this work, deadlock-freeness of P means *internal* deadlock-freeness, disregarding interaction with any environment where P may be embedded.

The io-behavior $b_{1||2}$, resulting from side-coupling, is constructed as follows:

(i) Selection of matching behaviors. Since P_1 and P_2 are disjoint, the selection of b_1 and b_2, the io-behaviors of P_1 and P_2 respectively, is determined by the internal channels. They are chosen so that the value components of the two triplets, one in each io-behavior, giving rise to each internal communication event are equal. This will be always possible since we assume deadlock-freeness, which in our context means that any internal communication in P_1 has a matching communication in P_2, and vice versa. Furthermore, under this assumption, the counts of corresponding triplets can also be made equal, and one of them will have an i mark and the other one an o mark, but not necessarily always in the same side (io-behavior). We say that such corresponding triplets are *matching.*

(ii) Construction of the intermediate form $\bar{b}_{1||2}$.
   (a) Its number of components equals the number of variables in $\mathcal{O}_1 \cup \mathcal{O}_2$.
   (b) Its rows are separated in sublists by its internal communication event rows, constructed first as follows: their variable components are filled in with the values of the corresponding variables of the matching rows of the two io-behaviors, with the exception of the internal communication event triplet, constructed with the value and the count of the two matching triplets. A dot will be placed in its third component, replacing the i and the o. This assumes deadlock freeness.
   (c) The rows of the sublists of b_1 and b_2, separated by communication event rows, are interleaved in $\bar{b}_{1||2}$. Any interleaving is possible.

(iii) $b_{1||2}$ is constructed by deleting the components of $\bar{b}_{1||2}$ not in $\mathcal{P}$, and any row of the result which equals its predecessor but not all of its successors.

Example 3.5 Consider the composition $[r1, cr1 := S(p1, cp1)] \mid\mid [r2, cp1 := A(p2, cr1)]$ of two disjoint processes, with set of internal channels $\mathcal{O}_I : \{cr1, cp1\}$.

The interface sets of S and A are $\mathcal{O}_s : \{r1, cr1, p1, cp1\}$ and $\mathcal{O} : \{r2, cp1, p2, cr1\}$. Assume that the interface set of the composition is $\mathcal{P} : (\mathcal{O}_S \cup \mathcal{O}) - \mathcal{O}_I : \{r1, p1, r2, p2\}$. The following schemas correspond to the io-behaviors b_s of S, b_a of A, and of their side-coupling intermediate form $\bar{b}_{s||a}$.

$\mathbf{b_s}$

		$r1$	$p1$	$cr1$	$cp1$
0	*initial*	x	x_2	x_T	x_T
1	$cr1 \Leftarrow v_{21}$	x	x_2	$x_3, 1, o$	x_T
2	$cp1 \Rightarrow v_{11}$	x	x_2	$x_3, 1, o$	$x_4, 1, i$
3	$r1 := v_{31}$	x_5	x_2	$x_3, 1, o$	$x_4, 1, i$

$\mathbf{b_a}$

		$cr1$	$cp1$	$r2$	$p2$
0	*initial*	x_T	x_T	x	y_2
1	$r2 := v_{12}$	x_T	x_T	y_3	y_2
2	$cr1 \Rightarrow v_{22}$	$y_4, 1, i$	x_T	y_3	y_2
3	$cp1 \Leftarrow v_{32}$	$y_4, 1, i$	$y_5, 1, o$	y_3	y_2

$\bar{\mathbf{b}}_{\mathbf{s||a}}$

		$r1$	$p1$	$cr1$	$cp1$	$r2$	$p2$
0	*initial*	x	x_2	x_T	x_T	x	y_2
1	$r2 := v_{12}$	x	x_2	x_T	x_T	y_3	y_2
2	$cr1 \Leftarrow v_{21} \| cr1 \Rightarrow v_{22}$	x	x_2	$(y_4 :=) x_3, 1, \cdot$	x_T	y_3	y_2
3	$cp1 \Rightarrow v_{11} \| cp1 \Leftarrow v_{32}$	x	x_2	$x_3, 1, \cdot$	$(y_5 :=) x_4, 1, \cdot$	y_3	y_2
4	$r1 := v_{31}$	x_5	x_2	$x_3, 1, \cdot$	$x_4, 1, \cdot$	y_3	y_2

The rows of matching internal communications, over channels $cr1$ and $cp1$, and of internal communication evens have been isolated.

3.3 Selection Composition

For selection composition, $[b_1, c_1; [r1 := P_1(p1)]$ **or** $\cdots$ **or** $b_n, c_n; [rn := P_n(pn)]]$, the set of io-behaviors is the *union* of the io-behaviors contributed by each of its alternatives A_k. Each of them contributes with the subset of the io-behaviors of $[c_k; [rk := P_k(pk)]]$ whose first row satisfies boolean condition b_k.

Let $\mathcal{O}_k$ be the interface set of procedure P_k. Then the interface set $\mathcal{O}_{A_k}$ of alternative A_k is given by $\mathcal{O}_k \cup var(c_k) \cup chan(c_k) \cup var(b_k)$; where $var(e)$ is the set of variables of expression e, $chan(c)$ is the singleton set containing the channel variable of c. The interface set of the above selection may be any set $\mathcal{P}$ such that $\mathcal{P} \subseteq \bigcup_{k=1}^{n} \mathcal{O}_{A_k}$. All this is consistent with non-determinism.

4 Input/Output Equivalence

4.1 The Notion

Definition 4.1 (Io-equivalent procedures) Two procedures P_1 and P_2 are io-equivalent with respect to their interface set $\mathcal{O}$, written $P_1 =_{\mathcal{O}} P_2$, when any io-behavior of any of them is equivalent to an io-behavior of the other.

Io-equivalence is weaker than congruence. Congruent procedures are always equivalent but not vice versa. The relative order of value changes in distinct components is neglected in io-equivalence. Therefore, substitution of a reference to a procedure by a reference to another procedure, io-equivalent to the first, may introduce deadlock. Consider the two procedures

$$(r1, r2) ::= P1(cp1, cp2) :: \ \left[cp1 \Rightarrow r1; cp2 \Rightarrow r2\right]$$

$$(r1, r2) ::= P2(cp1, cp2) :: \ \left[cp2 \Rightarrow r2; cp1 \Rightarrow r1\right]$$

with the same interface set $\mathcal{O}$. Now $P1 \not\approx_{\mathcal{O}} P2$, since if $P1$ is parallel to a process which always offers an output via channel *cp1* before offering another output via *cp2* within a program, and we replace $P1$ by $P2$ in that program, deadlock is introduced. However, $P1 =_{\mathcal{O}} P2$.

4.2 Substitution rules

Substitution of reference statements to io-equivalent procedures is an essential step of equivalence reasoning. The three first lemmas, concerning concatenation, cooperation and selection, are given in preparation for the general rule. Only the post-concatenation case is treated, the other case would be carried out similarly.

Lemma 4.2 ***(Substitution in concatenation)*** *Let* $S; [r := A(p)]$ *be deadlock-free. Then, if* $[r := A(p)] =_{\mathcal{O}} [r := B(p)]$ *, the equivalence* $S; [r := A(p)] =_{\mathcal{P}} S; [r := B(p)]$ *holds, where* $\mathcal{P} \subseteq \mathcal{O} \cup \mathcal{O}_S$ *, and* $\mathcal{O} = \mathcal{O}_A = \mathcal{O}_B$

Justification The io-behaviors of $S; A$ and $S; B$ have the same interface set $\mathcal{P}$ by definition, and since A and B have the same interface set $\mathcal{O}$. Concerning equality of component lists, io-behaviors of the concatenation are formed by postcoupling an io-behavior of A, or of B, to an io-behavior of S. We show equality of component lists recalling the postcoupling construction rules given in subsection 3.2.

(i) Proper variable components. Cases:
   (a) $v \in \mathcal{O}_S \cap \mathcal{O}$: The initial values of this group of components of b_A (or of b_B) are equal to their corresponding last values of b_S. Since $A =_{\mathcal{O}} B$, a b_B (or a b_A) io-equivalent to b_A (or to b_B) with the same initial values can always be found. Hence, the postcouplings of both sides will give the same lists of values for each one of these v components.
   (b) $v \notin \mathcal{O}_S \cap \mathcal{O}$ and $v \in \mathcal{O}_S$: For these components, the propagation into the future only depends on b_S, the io-behavior of S, which is equal in both sides.
   (c) $v \notin \mathcal{O}_S \cap \mathcal{O}$ and $v \in \mathcal{O}$: The propagation into the past depends on b_A or on b_B, which has been selected equivalent to b_A in (a).

(ii) Channel variable components. Cases:

(a) $c \in \mathcal{O}_S \cap \mathcal{O}$: Since S is the same in both sides, the last triplets of these components of b_S will be the same in both sides. Since b_B has been selected equivalent to b_A in (i-a), it will have the same pattern of initial undefined triplets, to be changed to the last values of the c components of b_S. Hence the io-behaviors of both sides will be equivalent.

(b) $c \notin \mathcal{O}_S \cap \mathcal{O}$ and $c \in \mathcal{O}_S$: The last triplets of b_S which have to be repeated into the future, b_A or b_B region, are the same in both sides since S stands in both sides. Hence, they will give equal propagations into b_A or b_B.

(c) $c \notin \mathcal{O}_S \cap \mathcal{O}$ and $c \in \mathcal{O}$: For these components, the lists of the two sides are formed by concatenating an undefined value with the lists of b_A and b_B which are equal, since they are io-equivalent. $\square$

We study now the preservation of equivalence in a substitution within a cooperation, parallelism, statement. Let $S||[r := A(p)]$ have internal channel set I. If $\mathcal{O}$ is the interface set of $[r := A(p)]$, and $\mathcal{O}_I$ is the set of internal channel variables, corresponding to I, then $\mathcal{O}_I \subseteq \mathcal{O}$. Channel variables in $\mathcal{O} - \mathcal{O}_I$ correspond to external channels of $S||[r := A(p)]$. Similarly, $\mathcal{O}_I \subseteq \mathcal{O}_S$, where $\mathcal{O}_S$ is the interface set of S . Also, channel variables in $\mathcal{O}_S - \mathcal{O}_I$ correspond to external channels of $S||[r := A(p)]$. Actually, the interface set $\mathcal{P}$ of the parallel composition is such that $\mathcal{P} \subseteq (\mathcal{O}_S \cup \mathcal{O}) - \mathcal{O}_I$, since there may be proper variables which are not declared as external.

Lemma 4.3 ***(Substitution in parallelism)*** *Let $S||[r := A(p)]$ be deadlock-free, and $r := A(p)$ be disjoint with S . Let also $[r := A(p)] =_{\mathcal{O}} [r := B(p)]$, and $S||[r := B(p)]$ be deadlock-free. Then $[S||[r := A(p)]] =_{\mathcal{P}} [S||[r := B(p)]]$, where $\mathcal{P} \subseteq (\mathcal{O}_S \cup \mathcal{O}) - \mathcal{O}_I$.*

Justification We will show that, in the construction of io-behaviors of $S||A$ and $S||B$, the steps of side-coupling, given in subsection 3.2, can be followed so that equivalent io-behaviors result from the two statements. Deadlock-freeness is required since it has been assumed in the construction.

(i) Selection of matching behaviors. Since b_S is identical for both statements and $A =_{\mathcal{O}} B$, any io-behavior b_A matching b_S can be replaced by an equivalent b_B, which will also be matching b_S.

(ii) As a consequence, the intermediate forms $\bar{b}_{S||A}$ and $\bar{b}_{S||B}$ will be equivalent with respect to set $\mathcal{O}_S \cup \mathcal{O}$.

(iii) Hence, the operations of 3.2.2(iii), with the $\mathcal{P}$ defined above, starting from either $\bar{b}_{S||A}$ or from $\bar{b}_{S||B}$ will give io-equivalent results.

Therefore, any io-behavior of $S||A$ can be interpreted as an equivalent io-behavior of $S||B$ and vice ver

□

Lemma 4.4 ***(Substitution in selection)*** *Let* $[\ g, [r := A(p)]$ **or** $R\ ]$ *be deadlock-free, where R stands for the rest of the selection statement, and g is a boolean guard, for a selection, or both a boolean and a communication guard, for a communications selection. Let also* $[r := A(p)] =_{\mathcal{O}} [r := B(p)]$ *, and* $\mathcal{O}_R$ *be the interface set of R. Then, with* $\mathcal{P} \subseteq (\mathcal{O}_R \cup \mathcal{O})$
$[\ g, [r := A(p)]$ **or** $R\ ] \ =_{\mathcal{P}}\ [\ g, [r := B(p)]$ **or** $R\ ]$ *.*

Justification Let $\mathcal{I}_R$, $\mathcal{I}_{Ag}$, and $\mathcal{I}_{Bg}$ denote the sets of io-behaviors of R, the alternative of A and the alternative of B, respectively, and $\mathcal{I}_A$ and $\mathcal{I}_B$ be the sets of io-behaviors of A and B, respectively. If the statement is a regular selection, then $\mathcal{I}_{Ag}$ and $\mathcal{I}_{Bg}$ are formed with the io-behaviors of $\mathcal{I}_A$ and $\mathcal{I}_B$, respectively, whose first rows satisfy the boolean guard of g. However, if the statement is a communications selection, each io-behavior of $\mathcal{I}_{Ag}$ is obtained by post-coupling an io-behavior of the $\mathcal{I}_{Ag}$ of the last case to the io-behavior of to the send or receive statement in the guard g. The same is true for $\mathcal{I}_{Bg}$.

The set of io-behaviors of a selection statement is the union of the sets of io-behaviors of each of its alternatives. Hence $\mathcal{I}_R \cup \mathcal{I}_{Ag}$ and $\mathcal{I}_R \cup \mathcal{I}_{Bg}$ are the sets of io-behaviors of the l.h.s. and the r.h.s. , respectively.

Therefore, for any io-behavior b_l of the l.h.s. there is an equivalent io-behavior b_r in the r.h.s. and vice versa. This is so in the case that the io-behavior belongs to $\mathcal{I}_R$ since this set is included in both sides and equal io-behaviors are io-equivalent as well. In the remaining case, where the io-behavior is in $\mathcal{I}_{Ag}$ or in $\mathcal{I}_{Bg}$ the truth follows from the fact that $[r := A(p)] =_{\mathcal{O}} [r := B(p)]$ and the guard g is the same in both sides. □

The following result is needed for the organization of proofs around the procedures of a distributed program, making proof decomposition possible.

Lemma 4.5 ***(Equivalence deduction by procedure substitution)*** *Let* $P[\_\ ]$ *be a loop free program context,* $P[r := A(p)]$ *be deadlock-free, and* $r := A(p)$ *be disjoint with all its parallel substatements in* $P[r := A(p)]$*. Then, if* $[r := A(p)] =_{\mathcal{O}} [r := B(p)]$ *, and* $P[r := B(p)]$ *is deadlock-free, the equivalence* $P[r := B(p)] =_{\mathcal{P}} P[r := A(p)]$ *holds for any* $\mathcal{P}$*.*

Justification The result follows from lemmas 4.2, 4.3, and 4.4. The minimal ancestor of A in $P[r := A(p)]$, or in $P[r := B(p)]$, is either a concatenation, a cooperation, or a selection. Then, equivalence between these ancestor statements follows from one of the three lemmas, and the associativity laws of both cooperation and concatenation compositions, as in next section. The same reasoning can now be applied recursively to the ancestors of these ancestors until $P[r := A(p)]$ and $P[r := B(p)]$ are reached. □

5 Laws for Input/Output Equivalence

5.1 Introduction

A set of laws needed for communication elimination equivalence proofs of some distributed programs is overviewed in this section. This is necessary in order to present an example of io-equivalence reasoning. Both communication elimination and the example are overviewed later in the paper.

There are both proper elimination and auxiliary laws. The latter, although not eliminating any communication directly, are needed to transform a program to a form where a proper communication elimination law can be applied.

Some intuitive auxiliary laws are available in [2], where it is shown that many of them do not hold when strong fairness is assumed. Some of them are $\textbf{Nil}; S \approx S$, S; $\textbf{Skip} \approx S$, $S||$ $\textbf{Skip} \approx S$. In addition, both sequential and parallel composition are associative. The latter is also commutative.

5.2 Laws for communication elimination

Attention will be restricted to statements whose communications are under the scope of neither selections nor iterations. We will refer to these statements as *bounded communication* (BC) statements. The number of communication events generated by their execution is finite and constant.

The laws to be given below allow the elimination of communications from BC statements. Communication elimination for some extended forms, with indefinite iterations, will also be covered in next section. For each BC statement S we define a set I of *internal* channels, whose communications have to be eliminated. The rest of the channels involved in S are *external* in the sense that communication statements over these channels never match with other communications in S. The following are two intuitive communication elimination laws.

$$[\,\alpha \Leftarrow e \,||\, \alpha \Rightarrow u\,] \;\approx\; [u := e]$$

$$[H^l; \alpha \Leftarrow e; T^l]||[H^r; \alpha \Rightarrow u; T^r] \;\approx\; [H^l||H^r]; u := e; [T^l||T^r]$$

where H^l and H^r do not contain communication substatements over channels in I. As shown in [2], no bounded number of communication elimination laws suffices for the elimination, in a single reduction, of a pair of matching communications from a BC statement. The following schema of equivalences,

$$\begin{bmatrix} H_k^l; \\ \\ \left[\, G_k^l \;||\; P_k^l \,\right]; \\ \\ T_k^l \end{bmatrix} || \begin{bmatrix} H_k^r; \\ \\ \left[\, G_k^r \;||\; P_k^r \,\right]; \\ \\ T_k^r \end{bmatrix} =_{\mathcal{O}} \begin{bmatrix} \left[\quad H_k^l \quad || \quad H_k^r \quad\right]; \\ \\ \left[\, G_k \;||\; P_k^l \;||\; P_k^r \,\right]; \\ \\ \left[\quad T_k^l \quad || \quad T_k^r \quad\right] \end{bmatrix}$$

where $k = 0, 1, \cdots$, defines an unbounded set of laws when we identify it with

$$[\, G^l_{k+1} \mid\mid G^r_{k+1}] =_{\mathcal{O}} G_{k+1}$$

since then, the statements G^l_k, G^r_k, and G_k are defined recursively for $k = 1, 2, \cdots$ The initial condition statements G^l_0 and G^r_0 are $\alpha \Leftarrow e$, and $\alpha \Rightarrow u$, respectively. G_0 stands for $u := e$. There is a law for any finite integer k.

The former two laws are special cases for $k = 0, 1$ making some substatements equal to **Nil**. The laws hold for io-equivalence only. A law is applied as a reduction from left to right, in order to eliminate any matching pair of communication statements in a single reduction. Observe, also, in the last laws that some substatements are parallel in one side but not in the other. This disordering may introduce deadlock. Therefore, a set of suitable applicability conditions have to be checked for each law.

6 Applications to Verification

6.1 Distributed program simplification (DPS)

This is a proof procedure applying, amongst others, the laws given above. The first step is carried out by a *communication elimination* reduction algorithm, which applies automatically the laws presented in last section. When the algorithm terminates successfully, there is a guarantee that the original statement is deadlock-free. The resulting io-equivalent form has parallelism between disjoint substatements but no internal communication statements.

The following is a procedure resulting from Pc, of subsection 2.2, after elimination of internal channel c. It has the same interface set.

$$(r, cr) ::= Pnc(p, cp) :: \begin{bmatrix} [cp \Rightarrow a1 || cp \Rightarrow a2]; \\ r := a1 + p; a2 := r + a2; cr \Leftarrow a2 \end{bmatrix}$$

Each io-behavior of Pc is an io-behavior of Pnc and vice versa, so $Pc =_{\mathcal{O}} Pnc$.

The next step of DPS, *parallelism to concatenation transformation*, is carried out by applying permutation laws for transforming the parallel compositions of disjoint processes to io-equivalent sequential forms. A sequential program io-equivalent to the initial one is obtained. The third and last step of DPS is *redundant variable elimination*. State-vector reduction comes with this last step.

6.2 DPS for non-BC statements

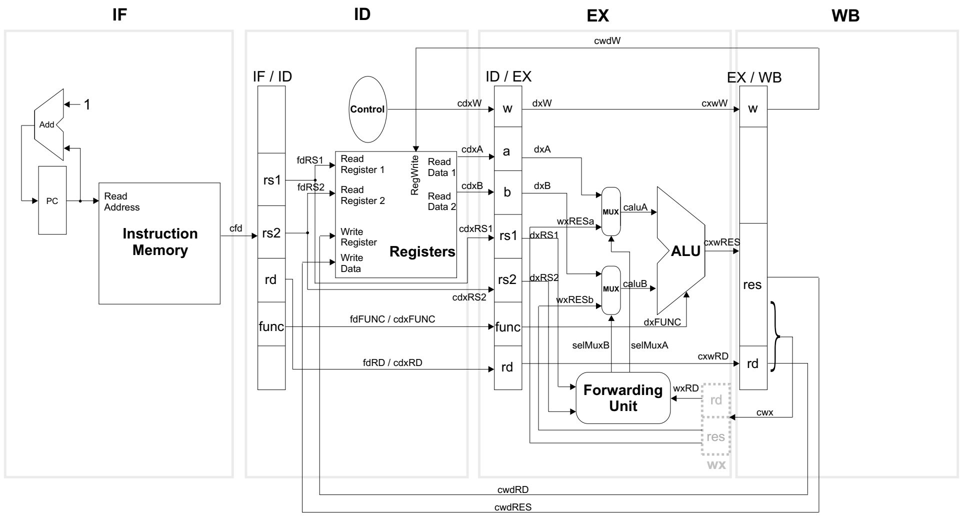
There exist more than one way to extend simplification proofs to non-BC statements, where communications appear within indefinite loops. We will center only in the following very common structure: $S = [S_1 || \cdots || S_m]$, where the S_k's are of the form $S_k =$ **loop forever do** B_k. The B_k's are BC statements. Since they have communication statements and appear within indefinite iterations, the whole statement is non-BC.

Assume that we unfold n_k times the loop of each top substatement S_k, 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 : $B_k; \cdots; B_k$.

We can apply DPS 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 both inner parallelism and 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 systems this occurs already for $n_k = 1$; $k = 1 \cdot\cdot m$.

6.3 *Global structure of an equivalence proof*

A brief account of a verification of a DLX-like [9] processor model, shown in the diagram below, will illustrate the utility of the reported results.



The proof establishes io-equivalence between a program, *pipeline2*, with two hierarchical levels of parallelism and internal communication, modeling the pipeline processor, and the following sequential program,

$$reg ::= VNCycle(reg, mem) :: \left[\begin{array}{l}\textbf{for } k := 1..n \textbf{ do} \\ \quad \left[\begin{array}{l} ir := mem(pc); \\ pc := pc + 1; \\ reg(ir.rd) := alures(ir.func, reg(ir.rs1), reg(ir.rs2)) \end{array}\right]\end{array}\right]$$

which captures the essential behavior of the pipelined processor software model. As this program makes explicit, the processor interprets programs with ALU register to register instructions only. The instruction register is ir. The destination and source register indexes are $ir.rd$, $ir.rs1$ and $ir.rs2$. Procedure *alures* gives the result of the ALU operation selected by *ir.func*. Integer n is the length of the program in *mem*.

The parallel program has four processes connected in pipeline, modeling the four stages above: IF, ID, EX, and WB. Processes ID and EX are modeled with the following procedures which encapsulate a second level of parallelism.

$(cxwW, cxwRES, cxwRD) ::= EX_{par}(cdxW, cdxA, cdxB, cdxRS1, cdxRS2, cdxFUNC, cdxRD, cwx) ::$

```
    loop forever do ( $ID/EX\ (dx)\ register$ )
        $dxA \Leftarrow dx.a;\ dxB \Leftarrow dx.b;\ dxRS1 \Leftarrow dx.rs1;\ dxRS2 \Leftarrow dx.rs2;\ dxFUNC \Leftarrow dx.func;$
        $xxw.w := dx.w;\ xxw.rd := dx.rd;$
        $cdxW \Rightarrow dx.w;\ cdxA \Rightarrow dx.a;\ cdxB \Rightarrow dx.b;\ cdxRS1 \Rightarrow dx.rs1;\ cdxRS2 \Rightarrow dx.rs2;$
        $cdxFUNC \Rightarrow dx.func;\ cdxRD \Rightarrow dx.rd;\ cxwW \Leftarrow xxw.w;\ cxwRD \Leftarrow xxw.rd$
||
    loop forever do ( $wx\ register$ )
        $cwx \Rightarrow wx;$
        $wxRESa \Leftarrow wx.res;\ wxRESb \Leftarrow wx.res;\ wxRD \Leftarrow wx.rd$
||
    loop forever do ( $Forwarding\ control$ )
        $dxRS1 \Rightarrow rs1;\ dxRS2 \Rightarrow rs2;$
        $wxRD \Rightarrow rd;$
        $selA := (rs1 = rd);\ selB := (rs2 = rd);$
        $selMuxA \Leftarrow selA;\ selMuxB \Leftarrow selB$
||
    loop forever do ( $Multiplexor\ of\ ALU\ input\ A$ )
        $dxA \Rightarrow a;\ wxRESa \Rightarrow resA;\ selMuxA \Rightarrow selA;$
        if $selA$ then $aluA := resA$ else $aluA := a;$
        $caluA \Leftarrow aluA$
||
    loop forever do ( $Multiplexor\ of\ ALU\ input\ B$ )
        $dxB \Rightarrow b;\ wxRESb \Rightarrow resB;\ selMuxA \Rightarrow selB;$
        if $selB$ then $aluB := resB$ else $aluB := b;$
        $caluB \Leftarrow aluB$
||
    loop forever do ( $ALU$ )
        $caluA \Rightarrow a; caluB \Rightarrow b;$
        $dxFUNC \Rightarrow func;$
        $xxw.res := alures(func, a, b);$
        $cxwRES \Leftarrow xxw.res$
```

$(reg, cdx\_w, cdx\_a, cdx\_b, cdx\_rs1, cdx\_rs2, cdx\_func, cdx\_rd) ::= ID_{par}(reg, cfd, cwdW, cwdRES, cwdRD) ::$

```
[ [ loop forever do ( $IF/ID\ (fd)\ register$ )
      [ $fdRS1 \Leftarrow fd.rs1$;
        $fdRS2 \Leftarrow fd.rs2$;
        $fdRD \Leftarrow fd.rd$;
        $fdFUNC \Leftarrow fd.func$;
        $cfd \Rightarrow fd$ ] ]
||
  [ loop forever do ( $Registers$ )
      [ $cwdW \Rightarrow wd.w$;
        $cwdRES \Rightarrow wd.res$;
        $cwdRD \Rightarrow wd.rd$;
        $control \Rightarrow w$;
        if $(wd.w)$ then $[reg(wd.rd) := wd.res]$ else $nil$;
        $(xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd)$
          $:=$
        $(w, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd)$;
        $fdRS1 \Rightarrow ir.rs1$;
        $fdRS2 \Rightarrow ir.rs2$;
        $fdRD \Rightarrow ir.rd$;
        $fdFUNC \Rightarrow ir.func$;
        $cdxW \Leftarrow xdx.w$;
        $cdxA \Leftarrow xdx.a$;
        $cdxB \Leftarrow xdx.b$;
        $cdxRS1 \Leftarrow xdx.rs1$;
        $cdxRS2 \Leftarrow xdx.rs2$;
        $cdxFUNC \Leftarrow xdx.func$;
        $cdxRD \Leftarrow xdx.rd$ ] ]
||
  [ loop forever do ( $Control$ )
      [ $control \Leftarrow true$ ] ] ]
```

This detail is given not to be understood by the reader, but to illustrate the non-triviality of the example. The proof establishes the io-equivalence

$$[reg ::= VNCycle(reg, mem)] =_{\mathcal{O}} [reg ::= Pipeline2(reg, mem)]$$

where $\mathcal{O} : \{reg, mem\}$. This result is proved with three DPS proofs, and the application of the io-equivalence reasoning rules introduced in this paper. Although the pipelined structure has more stages, we illustrate the partitioning of the proof only with the instruction decode (ID) and the execution (EX) stages. The two are modeled as sequential procedures also, ID_{seq} and EX_{seq}. With two DPS proofs, the following equivalences are established.

$$[reg, coutID ::= ID_{seq}(reg, cinID)] =_{\mathcal{O}} [reg, coutID ::= ID_{par}(reg, cinID)]$$

$$[coutEX ::= EX_{seq}(cinEX)] =_{\mathcal{O}} [coutEX ::= EX_{par}(cinEX)]$$

The *cout* and *cin* names denote the lists of output and input channels respectively. Procedure *Pipeline2* above has references to the procedures with inner parallelism, $Pipeline2 : Pipeline[ID_{Par}, EX_{Par}]$. *Pipeline* is a procedure with one level of parallelism and two holes for the references to *ID* and *EX*.

An auxiliary procedure, *Pipeline1*, is defined as *Pipeline* with the references to the sequential procedures, $Pipeline1 : Pipeline[ID_{Seq}, EX_{Seq}]$. It has one level of parallelism. It is the following:

$$reg ::= Pipeline_1(reg, mem) ::$$

```
local  $pc$                      : integer
local  $w$                       : boolean
local  $instr, ir$               : Typ_IR
local  $xdx, dx$                 : Typ_DX
local  $wd, wx, xxw, xw$         : Typ_XW
local  $cfd$                     : channel of Typ_IR
local  $cdx$                     : channel of Typ_DX
local  $cwd, cxw, cwx$           : channel of Typ_XW
  $pc := 1;$
  $(ir.rs1, ir.rs2, ir.rd, ir.func) := (0, 0, 0, 0);$
  $(dx.w, dx.a, dx.b, dx.rs1, dx.rs2, dx.func, dx.rd) := (false, 0, 0, 0, 0, 0, 0);$
  $(xw.w, xw.res, xw.rd) := (false, 0, 0);$
  $(wx.w, wx.res, wx.rd) := (false, 0, 0);$
    $IF ::$ loop forever do
              $instr := mem(pc);$
              $pc := pc + 1;$
              $cfd \Leftarrow instr$
    ||
    $ID_{seq} ::$ loop forever do
              $cwd \Rightarrow wd;$
              if $(wd.w)$ then $[reg(wd.rd) := wd.res]$ else $nil;$
              $(xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd)$
                :=
              $(w, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd);$
              $cfd \Rightarrow ir;$
              $cdx \Leftarrow xdx$
    ||
    $EX_{seq} ::$ loop forever do
              $cwx \Rightarrow wx;$
              if $(dx.rs1 = wx.rd)$ then $[dx.a := wx.res]$ else $nil;$
              if $(dx.rs2 = wx.rd)$ then $[dx.b := wx.res]$ else $nil;$
              $(xxw.w, xxw.res, xxw.rd) := (dx.w, alures(dx.func, dx.a, dx.b), dx.rd);$
              $cdx \Rightarrow dx;$
              $cxw \Leftarrow xxw$
    ||
    $WB ::$ loop forever do
              $cwd \Leftarrow xw;$
              $cwx \Leftarrow xw;$
              $cxw \Rightarrow xw$
```

The bodies of procedures ID_{seq} and EX_{seq} are given in it. Notice that groups of channels in the parallel versions have been reduced to a single channel in the sequential versions. A channel group hide/unhide equivalence rule, given in [1], has been used in this part of the proof. Then, the equivalence

$$[reg ::= VNCycle(reg, mem)] =_{\mathcal{O}} [reg ::= Pipeline1(reg, mem)]$$

is established with a DPS proof. Finally, $Pipeline1 =_{\mathcal{O}} Pipeline2$ by the substitution rule of lemma 4.5, and assuming that its deadlock-freeness conditions hold. These hold since the success of the communication elimination algorithm guarantees deadlock-freeness of *Pipeline1*. Deadlock-freeness of *Pipeline2* follows from deadlock-freeness of *Pipeline1* and conservation of the order of the external communication offers of the parallel and sequential versions of the EX and ID procedures.

7 Conclusions and Future Work

A new semantics for distributed imperative programs has been presented as an extension of the semantics of Manna and Pnueli. Auxiliary variables, recording the list of values crossing channels, have been added to the state variables of computations and reduced behaviors. A general formulation of input/output equivalence of procedures, integrating values communicated through both variables and synchronous channels, and a procedure reference substitution rule, have been formulated in the new semantics. A new set of laws for distributed imperative programs, and the decomposition of distributed program simplification proofs, via communication elimination, have been made possible with the new results. As an application example, a formal equivalence proof of a pipelined processor model has been summarized.

Although other equivalence proofs for distributed programs have been carried out already, this line of effort should continue for other classes of such programs. Soundness of the laws for io-equivalence was proved in a prior work. Completeness should be studied in the future.

Acknowledgement

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

References

[1] Babot, F., M. Bertran, J. Riera, R. Puig and A. Climent, *Mechanized Equivalence Proofs of Pipelined Processor Software Models*, in: *Actas de las III Jornadas de Programación y Lenguajes* (2003), pp. 91–104.

[2] Bertran, M., F. Babot, A. Climent and M. Nicolau, *Communication and Parallelism Introduction and Elimination in Imperative Concurrent Programs*, in: P. Cousot, editor, *Static Analysis. 8th International Symposium, SAS 2001*, LNCS **2126** (2001), pp. 20–39.

[3] Bjørner, N., A. Browne, B. F. M. Colón, Z. Manna, H. Sipma and T. Uribe, *Verifying Temporal Properties of Reactive Systems. A Step Tutorial*, in: *Formal Methods in System Design*, 2000, pp. 227–270.

[4] Broy, M., *A logical basis for component-based systems engineering*, in: M. Broy and R. Steinbruggen, editors, *Calculational System Design.* (1999).

[5] Clarke, E., O. Grumberg and D. Peled, "Model Checking," The MIT Press, 1999.

[6] de Alfaro, L., *Game Models for Open Systems*, in: *International Symposium on Verification (Theory and Practice)*, LNCS **2772** (2003).

[7] de Roever, W.-P., F. de Boer, U. Hanneman, Y. Lakhnech, M. Poel and J. Zwiers, "Concurrency Verification: Introduction to Compositonal and Noncompositional Methods," Cambridge University Press, 2001.

[8] Finkbeiner, B., Z. Manna and H. Sipma, *Deductive Verification of Modular Systems*, in: *In Compositionality: The Significant Difference, COMPOS'97*, LNCS **1536** (1998), pp. 239–275.

[9] Hennessy, J. L. and D. A. Patterson, "Computer Architecture: A Quantitative Approach," Morgan Kaufmann Publishers Inc., San Mateo, California, 1990.

[10] Holtzmann, G., "Design and Validation of Computer Protocols," Prentice Hall, 1991.

[11] INMOS-Limited, "Occam Programming Manual," Prentice Hall, 1985.

[12] INMOS-Limited, "Occam 2 Reference Manual," Prentice Hall, 1988.

[13] Jones, G., "Programming in Occam," Prentice Hall, 1987.

[14] Kaufmann, M. and J. S. Moore, *An Industrial Strength Theorem Prover for a Logic Based on Common Lisp*, IEEE Transactions on Software Engineering **23** (1997), pp. 203–213.

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

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

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

[18] Roscoe, A. and C. Hoare, *The laws of OCCAM programming*, Theoretical Computer Science **60** (1988), pp. 177–229.