Translating FSP into LOTOS and Networks of Automata

Frédéric Lang[1], Gwen Salaün[2,1], Rémi Hérilier[1], Jeff Kramer[3], and Jeff Magee[3]

[1]Vasy project-team, Inria Grenoble Rhône-Alpes/Lig, Montbonnot, France
[2]Grenoble Institut National Polytechnique, Grenoble, France
[3]Department of Computing, Imperial College, London, UK

Abstract

Many process calculi have been proposed since Robin Milner and Tony Hoare opened the way more than 25 years ago. Although they are based on the same kernel of operators, most of them are incompatible in practice. We aim at reducing the gap between process calculi, and especially making possible the joint use of underlying tool support. FSP is a widely-used calculus equipped with Ltsa, a graphical and user-friendly tool. Lotos is the only process calculus that has led to an international standard, and is supported by the Cadp verification toolbox. We propose a translation of FSP sequential processes into Lotos. Since FSP composite processes (i.e., parallel compositions of processes) are hard to encode directly in Lotos, they are translated into networks of automata which are another input language accepted by Cadp. Hence, it is possible to use jointly Ltsa and Cadp to validate FSP specifications. Our approach is completely automated by a translator tool.

1 Introduction

Process calculi (or process algebras) are abstract description languages to specify concurrent systems. The process algebra community has been working on this topic for 25 years and many different calculi have been proposed. Meanwhile, several toolboxes have been implemented to support the design and verification of systems specified with process calculi. However, although they are based on the same kernel of operators, most of them are incompatible in practice. In addition, there are very few bridges between existing verification tools. Our goal is to reduce the gap between the different formalisms, and to propose some bridges between existing tools to make their joint use possible.

We focus here on the process calculi FSP and Lotos. FSP [MK06] is an easy to learn process calculus conceived to make specifications easy to write and concise. FSP is supported by Ltsa, a user-friendly tool that compiles FSP specifications into finite state machines known as Ltss (*Labeled Transition Systems*), visualising and animating Ltss through graphical interfaces, and verifying LTL properties. FSP/Ltsa are quite widely-used: Magee and Kramer's book on Concurrency [MK06], which presents FSP and Ltsa, has sold over $15,000$ copies, courses using FSP/Ltsa are taught at numerous universities worldwide, and a considerable number of research groups are using FSP/Ltsa in their research (589 citations in Google Scholar as of January 2009).

On the other hand, Lotos is an ISO standard [ISO89], which has been applied successfully to many application domains. Lotos is more structured than FSP, and then adequate to specify complex systems possibly involving data types. Lotos is equipped with Cadp [GLMS07], a verification toolbox for asynchronous concurrent systems distributed worldwide, which allows very large state spaces to be handled, and implements various verification techniques such as model checking, compositional verification, equivalence checking, distributed model checking, etc.

To sum up, the simplicity of FSP makes it more accessible to "*newcomers*" than Lotos, which requires a better level of expertise. In addition, Cadp is a rich and efficient verification toolbox that can complement basic analysis possible with Ltsa. We propose to translate FSP specifications into Lotos to enable FSP users to access the verification techniques available in the Cadp toolbox. Since some FSP constructs for composite processes are difficult to encode into Lotos (for instance synchronisations between complex labels or priorities), they have been encoded into another input format of Cadp named Exp.Open 2.0 [Lan05] (simply written Exp.Open in the sequel). Exp.Open allows networks of automata to be described using general parallel composition operators, but also supports renaming, hiding and priorities.

Our goal is not to replace Ltsa, since Ltsa is convenient to debug and visualise graphically simple examples, but to complement it with supplementary verification techniques such as those mentioned before. Furthermore, we choose a high-level translation between process calculi, as most as possible, instead of low-level connections with Cadp (through the Open/Cæsar application programming interface [Gar98] for instance) because:

- We preferred to keep the expressiveness of the specification and then make the translation of most behavioural operators easier;
- High-level models are necessary to use some verification techniques available in Cadp, such as compositional verification [GL01, Lan02, Lan05, Lan06];
- Verification of the generated Lotos code can benefit from the numerous optimisations implemented in the Cæsar.adt and Cæsar [Gar89a, Gar89b, Gar90, GS06] compilers for Lotos available in Cadp, which would be too expensive to re-implement for FSP.

We implemented the translation from FSP to Lotos/Exp.Open in a completely automated tool named Fsp2Lotos (about 25,000 lines of code). This tool was validated on many examples (more than 10,000 lines of FSP) to ensure that the translation is reliable. As regards semantics, our translation preserves strong equivalence between processes.

The remainder of this article is organised as follows. Section 2 gives short introductions to Ltss, FSP, Lotos, and Exp.Open. Section 3 presents formally the translation rules from FSP sequential processes into Lotos and from FSP composite processes into Exp.Open. Section 4 presents the Fsp2Lotos tool and its validation. Section 5 illustrates how Ltsa and Cadp can be used jointly on a simple system. Section 6 presents some related work. Section 7 provides concluding remarks.

2 Background

In this section, we present the underlying semantic model used in this work, namely *Labeled Transition Systems* (Ltss) as well as the source language FSP and the target languages Lotos and Exp.Open of our translator.

2.1 Labeled Transition Systems and Bisimulations

An Lts is a graph defined as a quadruple "$(Q, A, \rightarrow, q_0)$", consisting of a set Q of *states*, a set A of symbols called *labels* or *actions*, a *labeled transition relation* "$\rightarrow \subseteq Q \times A \times Q$", and an *initial state* "$q_0 \in Q$". As usual, we write "$q_1 \xrightarrow{a} q_2$" instead of "$(q_1, a, q_2) \in \rightarrow$".

Following CCS, Lts is the semantic model underlying FSP, Lotos, and Exp.Open: to each process can be associated an Lts that defines the *behaviour* of the process exhaustively. In addition, an Lts usually has a special symbol that denotes an internal action of the process. This symbol is generally written τ in theoretical work, and more concretely written "**i**" in Lotos/Cadp and "**tau**" in FSP/Ltsa.

The Lts model used in Ltsa also has a special sink state (i.e., a state without outgoing transitions) modeling an error of the system, called *error state*. Such an error state can be encoded in the above Lts model as a normal state that contains a single self-looping transition labeled by a special error symbol.

To decide whether two processes are equivalent, one has to compare the Ltss associated to each process. To this aim, we follow the approach based on *graph bisimulations*. Of interest in this work are *strong bisimulation* [Par81], which captures the fact that two processes have exactly the same behaviour, including internal actions (τ-transitions), and *branching bisimulation* [vGW89], which captures the fact that two processes have similar behaviours, except differences on internal actions provided they do not affect the choices of non-internal actions available from branching bisimilar states. Two processes are *strongly equivalent* (respectively *branching equivalent*) if their corresponding LTSs are strongly bisimilar (respectively branching bisimilar). More formally, let "$(Q, A, \rightarrow, q_0)$" be an LTS, and q_1 and q_2 be states of that LTS:

- q_1 and q_2 are *strongly bisimilar* if there exists a relation "$R \subseteq Q \times Q$" such that "$R(q_1, q_2)$" and (1) for each transition "$q_1 \xrightarrow{a} q_1'$", there is a transition "$q_2 \xrightarrow{a} q_2'$" such that "$R(q_1', q_2')$", and (2) for each transition "$q_2 \xrightarrow{a} q_2'$", there is a transition "$q_1 \xrightarrow{a} q_1'$" such that "$R(q_2', q_1')$".
- q_1 and q_2 are *branching bisimilar* if there exists a relation "$R \subseteq Q \times Q$" such that "$R(q_1, q_2)$" and (1) for each transition "$q_1 \xrightarrow{a} q_1'$", either "$a = \tau$" and "$R(q_1', q_2)$", or there is a path "$q_2 \xrightarrow{\tau *} q_2' \xrightarrow{a} q_2''$" such that "$R(q_1, q_2')$" and "$R(q_1', q_2'')$", and (2) for each transition "$q_2 \xrightarrow{a} q_2'$", either "$a = \tau$" and "$R(q_2', q_1)$", or there is a path "$q_1 \xrightarrow{\tau *} q_1' \xrightarrow{a} q_1''$" such that "$R(q_2, q_1')$" and "$R(q_2', q_1'')$".

Two LTSs "$(Q_i, A_i, \rightarrow_i, q_{0i})$ $(i \in \{0, 1\})$" are strongly bisimilar (respectively branching bisimilar) if the states q_{00} and q_{01} are strongly bisimilar (respectively branching bisimilar) in the LTS "$(Q_0 \uplus Q_1, A_0 \cup A_1, \rightarrow_0 \cup \rightarrow_1, q_{00})$", where "$Q_0 \uplus Q_1$" denotes the disjoint union of Q_0 and Q_1.

In every class of strongly bisimilar (respectively branching bisimilar) LTSs, there exists a unique representative (modulo a renaming of states) that is minimal in number of states and transitions. We call LTS minimization the computation of this representative, for which there exists efficient algorithms [PT87, GV90, KS90] and tools [BO05, GLMS07].

The LTS model also allows temporal logic formulas to be verified by evaluation on the initial state of the LTS. For instance, LTSA allows the specification and verification of safety and progress properties themselves written in FSP, and CADP allows the specification and verification of temporal logic formulas expressed in the regular alternation-free μ-calculus [MS03].

2.2 FSP

FSP (*Finite State Processes*) is a recent process calculus [MK99, MK06] originally proposed to design software architectures [MDEK95, Mag99]. FSP allows Booleans, integers, constant character strings, and sets to represent data, as well as *processes* to represent behaviours. An FSP process may be either *basic* (i.e., sequential) or *composite* (i.e., built from parallel compositions of processes).

We give here a short presentation of FSP processes in the form of an abstract grammar, which allows us to get rid of details of FSP's concrete syntax. We omit the "`@`" visibility operator, whose treatment is close to its dual hiding operator, although our tool presented in Section 4 supports this operator. Also, we do not handle safety and progress properties which, in further work, could be translated into regular alternation free modal μ-calculus formulas in order to be verified using the EVALUATOR [MS03] tool of CADP. A comprehensive concrete syntax of FSP is described in Magee and Kramer's book [MK06].

Figures 1 and 2 present the grammar of FSP basic and composite processes, respectively. In the grammar, and also in the sequel, we use "..." and indexed terms to represent sequences of arbitrary length. For instance, "$V_1 \texttt{,} \ldots \texttt{,} V_n$" represents a possibly empty sequence of terms separated by commas. Note that "..." should not be confused with the "`..`" terminal symbol of FSP. We use the symbol P (or "$P_1, P_2, \ldots$") to represent process identifiers, x (or "$x_1, x_2, \ldots$") to represent variables, *act* to represent character strings, and V (or "$V_1, V_2, \ldots$") to represent data expressions. To avoid details about the concrete syntax of data expressions, we consider that an expression is either a variable x, or the application of a function $\mathbf{f}$ to expressions "$V_1, \ldots, V_n$", written "$\mathbf{f}(V_1, \ldots, V_n)$". Without loss of generality, literal constants can be considered as functions without parameters.

FSP has an expressive syntax to represent labels. FSP labels, written A (or "$A', A_1, A_2, \ldots$"), are concatenations of sublabels written L (or "$L', L_1, L_2, \ldots$"), each of which is either a character string *act*, an expression V, a nonempty set of labels "$\{A_1, \ldots, A_n\}$", or a nonempty integer range "$V_1..V_2$", where V_1 and V_2 are integer expressions. An FSP label thus denotes a set of label strings obtained by (combinatorial) concatenation of sublabel strings. When a variable x is associated to a sublabel, such as in "$x\texttt{:}V_1 \texttt{..} V_2$", x is assigned any value in the label set corresponding to the sublabel.

A basic process definition D_b consists of:

- a process name P;

D_b	$::=$	$P\texttt{(}x_1\texttt{=}V_1,\ldots,x_k\texttt{=}V_k\texttt{)}\ \texttt{=}\ B_0$	process definition
		$D_{l1},\ldots,D_{lm}$	local processes
		$+\{A_{e_1},\ldots,A_{e_n}\}$	alphabet extension
		$/\{A'_{r_1}/A_{r_1},\ldots,A'_{r_p}/A_{r_p}\}\ \backslash\{A_{h_1},\ldots,A_{h_q}\}$	relabeling & hiding
D_l	$::=$	$P\texttt{[}x_1^1\texttt{:}L_1^1\texttt{]}\ldots\texttt{[}x_1^n\texttt{:}L_1^n\texttt{]}\ \texttt{=}\ B_1,$	local process definition
		$\ldots$	
		$P\texttt{[}x_m^1\texttt{:}L_m^1\texttt{]}\ldots\texttt{[}x_m^n\texttt{:}L_m^n\texttt{]}\ \texttt{=}\ B_m$	
B	$::=$	**stop**	deadlock termination
	$\mid$	**end**	normal termination
	$\mid$	**error**	erroneous termination
	$\mid$	$A \to B_0$	prefixing
	$\mid$	$P\texttt{(}V_1,\ldots,V_n\texttt{)}\texttt{;}B_0$	global process call
	$\mid$	$P\texttt{[}V_1\texttt{]}\ldots\texttt{[}V_n\texttt{]}$	local process call
	$\mid$	**if** V **then** B_1 **else** B_2	conditional branching
	$\mid$	**when** $V_1\ B_1\ \texttt{\mid}\ \ldots\ \texttt{\mid}$ **when** $V_n\ B_n$	choice
A	$::=$	$L_1 \ldots L_n$	label
L	$::=$	act	action
	$\mid$	$V \mid x\texttt{:}V$	expression
	$\mid$	$\{A_1,\ldots,A_n\} \mid x\texttt{:}\{A_1,\ldots,A_n\}$	label set
	$\mid$	$V_1\texttt{..}V_2 \mid x\texttt{:}V_1\texttt{..}V_2$	range

Figure 1: Abstract grammar of the FSP language: basic processes

- a (possibly empty) list of data parameters "x_i ($i \in 1..k$)", each data parameter being assigned a default value V_i;

- a basic behaviour B_0, described below;

- a (possibly empty) list of local process definitions "$D_{l1}, \ldots, D_{lm}$";

- a (possibly empty) set of relabeling rules "$\{A'_{r_1}/A_{r_1}, \ldots, A'_{r_p}/A_{r_p}\}$", which apply to the labels of B_0, where "$A_{r_i}, A_{r'_i}$ ($i \in 1..p$)" are label expressions: each label in the label set corresponding to A_{r_i} renames into labels corresponding to A'_{r_i} (i.e., a single label may rename into several labels);

- a (possibly empty) set of FSP labels "$\{A_{h_1}, \ldots, A_{h_q}\}$" to be hidden in B_0, i.e., renamed into the internal action **tau**;

- a (possibly empty) set of labels "$\{A_{e_1}, \ldots, A_{e_n}\}$" which, together with the set of non-hidden labels occurring in B_0 constitutes the *alphabet* of the process.

Each local process is defined by an ordered set of equations, each of the form "$P\,\texttt{[}x_i^1\texttt{:}L_i^1\texttt{]} \ldots \texttt{[}x_i^n\texttt{:}L_i^n\texttt{]} = B_i$", where "$x_i^1, \ldots, x_i^n$" are variables and each label L_i^j does not contain expressions. In the concrete syntax, each "x_i^j ($j \in 1..n$)" is optional, but we make them mandatory in the abstract syntax so as to simplify the presentation of translation rules. Parsing into the abstract syntax may thus require adding some dummy variables x_i^j for those L_i^j not preceded by a variable in the concrete syntax. Also, FSP's concrete syntax allows the definition of several local processes with same name but different arities. Instead, we assume that parsing has associated a unique name to each local process, which corresponds to a particular ordered set of equations D_l.

A local process call of the form "$P\ \texttt{[}V_1\texttt{]} \ldots \texttt{[}V_n\texttt{]}$" is substituted by the first B_i such that "$L_i^1, \ldots, L_i^n$" contain respectively the values "$V_1, \ldots, V_n$", in which each "x_i^j ($j \in 1..n$)" is replaced by V_j. If no such B_i exists, then the process call is equivalent to "**error**".

As regards hiding and relabeling, FSP uses *label prefix matching*, which means that the rules apply to label prefixes. For instance, as regards hiding, a label is hidden if some of its prefixes belongs to the set of labels to be hidden.

The operational semantics of FSP can be expressed in terms of an Lts. Informally, the semantics of sequential behaviours is the following:

- The "**stop**" behaviour corresponds to deadlock termination. No transition can be derived from "**stop**".

- The "**end**" behaviour corresponds to successful termination. It does not produce a transition but, if it occurs in the left part of the sequential composition operator "**;**", then the execution of the right part immediately starts.

- The "**error**" behaviour corresponds to erroneous termination. It is modeled by the error state.

- "$A \rightarrow B_0$" corresponds to the prefixing of behaviour B_0 by any action a belonging to A. It produces a transition labeled by a and then behaves as B_0, in which every variable x possibly defined in A is replaced by its value.

- "$P\texttt{(}V_1, \ldots, V_n\texttt{)}\,\texttt{;}\,B_0$" corresponds to the execution of the basic global process P with actual parameters "$V_1, \ldots, V_n$", followed by B_0 once P has terminated succesfully. FSP's concrete syntax also allows calls of the form "$P\texttt{(}V_1, \ldots, V_n\texttt{)}$" (not followed by a behaviour B_0), which is parsed into "$P\texttt{(}V_1, \ldots, V_n\texttt{)}\,$**;end**" in the abstract syntax.

- "$P\,\texttt{[}V_1\texttt{]} \ldots \texttt{[}V_n\texttt{]}$" corresponds to the execution of the local process P, indexed by "$V_1, \ldots, V_n$". A local process call cannot be followed by another behaviour.

- "**if** V **then** B_1 **else** B_2" behaves as B_1 if V evaluates to true, and as B_2 otherwise.

- "**when** V_1 B_1 | $\ldots$ | **when** V_n B_n" behaves nondeterministically as any branch B_i whose condition V_i evaluates to true.

D_c	$::=$	$\texttt{\|\|}P\texttt{(}x_1\texttt{=}V_1, \ldots, x_k\texttt{=}V_k\texttt{)} \texttt{ = } C_0$	process definition
		$op_p\ \{A_{p_1}, \ldots, A_{p_n}\}\ \backslash\{A_{h_1}, \ldots, A_{h_q}\}$	priority & hiding
op_p	$::=$	$\gg$	high priority operator
	$\|$	$\ll$	low priority operator
C	$::=$	$P\texttt{(}V_1, \ldots, V_n\texttt{)}$	process call
	$\|$	$C_1 \texttt{\|\|} \ldots \texttt{\|\|} C_n$	parallel composition
	$\|$	$C_0/\{A'_1/A_1, \ldots, A'_n/A_n\}$	relabeling
	$\|$	$\{A_1, \ldots, A_n\}\texttt{:}C_0$	labeling
	$\|$	$\{A_1, \ldots, A_n\}\texttt{::}C_0$	sharing
	$\|$	**if** V **then** C_1 **else** C_2	conditional branching
	$\|$	**forall** $[x_1\!:\!L_1] \ldots [x_n\!:\!L_n]\ C_0$	replication

Figure 2: Abstract grammar of the FSP language: composite processes

A composite process definition D_c consists of:

- a process name P, the symbol "||" which precedes P indicating that P belongs to the class of composite processes;
- a (possibly empty) list of data parameters "x_i ($i \in 1..k$)", each data parameter being assigned a default value V_i;
- a composite behaviour C_0, described below;
- a (possibly empty) list of labels "$\{A_{p_1}, \ldots, A_{p_n}\}$" that are assigned either higher (symbol "$\ll$") or lower (symbol "$\gg$") priority than all other labels occurring in C_0;
- a (possibly empty) set of FSP labels "$\{A_{h_1}, \ldots, A_{h_q}\}$" to be hidden, i.e., renamed into **tau**.

The semantics of composite behaviours C is the following:

- "$P\texttt{(}V_1, \ldots, V_n\texttt{)}$" corresponds to a (basic or composite) process call.
- "$C_1 \texttt{||} \ldots \texttt{||} C_n$" corresponds to the parallel composition of the composite behaviours "$C_1, \ldots, C_n$". All behaviours among "$C_1, \ldots, C_n$" that contain a common label in their alphabets must synchronise all together on that label.
- "$C_0/\{A'_1/A_1, \ldots, A'_n/A_n\}$" corresponds to the relabeling of C_0, which has the same semantics as for basic processes.
- "$\{A_1, \ldots, A_n\}\texttt{:}C_0$", called *process labeling*, generates an interleaving of as many instances of C_0 as there are labels in "$\{A_1, \ldots, A_n\}$". All the labels of each instance are prefixed by the label of "$\{A_1, \ldots, A_n\}$" associated to this instance. It is assumed that "$n \neq 0$".
- "$\{A_1, \ldots, A_n\}\texttt{::}C_0$", called *process sharing*, replaces each label l occurring in C_0 by a choice between labels "$A_1 l, \ldots, A_n l$". It is assumed that "$n \neq 0$".
- "**if** V **then** C_1 **else** C_2" behaves as C_1 if V evaluates to true, and as C_2 otherwise.
- "**forall** $[x_1\!:\!L_1] \ldots [x_n\!:\!L_n]\ C_0$" corresponds to the parallel composition of as many instances of C_0 as there are valuations of "$x_1, \ldots, x_n$" such that the value of each x_i belongs to the set of labels "L_i ($i \in 1..n$)". In each instance of C_0, each "x_i ($i \in 1..n$)" is replaced by its value in the corresponding valuation.

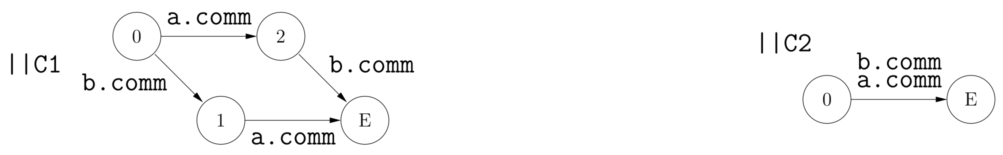


Figure 3: Process labeling and process sharing in FSP

Example 1 *An illustration of process labeling and process sharing is given in Figure 3. The figure shows the automata corresponding to the following processes:*

```
P = comm -> end
||C1 = {a, b}: P
||C2 = {a, b}:: P
```

□

An FSP specification consists of a set of basic (D_b) and composite (D_c) process definitions. We note "$\hat{P}$" the process definition corresponding to the basic or composite process P, and we note "$\hat{P}[V_1', \ldots, V_k']$" the process definition corresponding to P, in which the default parameter values "$V_1, \ldots, V_k$" are replaced by "$V_1', \ldots, V_k'$". For instance, if "$\hat{P}$" corresponds to:

$$\texttt{||}P\texttt{(}x_1\texttt{=}V_1, \ldots, x_k\texttt{=}V_k\texttt{) = } C \gg \{A_{p_1}, \ldots, A_{p_n}\} \setminus \{A_{h_1}, \ldots, A_{h_q}\}$$

then "$\hat{P}[V_1', \ldots, V_k']$" corresponds to:

$$\texttt{||}P\texttt{(}x_1\texttt{=}V_1', \ldots, x_k\texttt{=}V_k'\texttt{) = } C \gg \{A_{p_1}, \ldots, A_{p_n}\} \setminus \{A_{h_1}, \ldots, A_{h_q}\}.$$

The behaviour of the whole FSP specification is that of a particular process, which may be either selected by the user, or chosen by default. We call this particular process the *main process* of the FSP specification.

Example 2 *The following specification describes in FSP's concrete notation a semaphore inspired from an example in [MK06]. The indexed process notation "*`SEMA[v:Int]`*" represents two processes named "*`SEMA[0]`*" and "*`SEMA[1]`*", which are mutually recursive. The "*`ACCESS`*" process simulates a client which accesses the critical section protected by the "*`SEMAPHORE`*" process. The main process, called "*`SEMADEMO`*", is composed of an instance of the "*`SEMAPHORE`*" process that models a semaphore in charge of three resources "*`a`, `b`, `c`*", and an instance of the "*`ACCESS`*" process that wants to access these resources.*

```
range Int = 0..1

SEMAPHORE (N = 0) = SEMA[N],
SEMA[v:Int]       = (up -> SEMA[v+1] | when (v > 0) down -> SEMA[v-1]).
ACCESS            = (mutex.down -> critical -> mutex.up -> ACCESS).
||SEMADEMO        = ({a,b,c}:ACCESS || {a,b,c}::mutex:SEMAPHORE(1)).
```

The Lts *corresponding to the exhaustive behaviour of "*`||SEMADEMO`*" process is depicted in Figure 4.* □

2.3 LOTOS

Lotos (*Language Of Temporal Ordering Specification*) is a specification language for distributed open systems, standardised by Iso [ISO89]. Lotos combines a *data part* based on algebraic abstract data types to define data and their operations, and a *control part* to define (sequential and parallel) processes, inspired from the CCS [Mil89] and CSP [Hoa85] process algebras. In this section, we do not present Lotos data part, which is not intensively used in the translation since FSP does not handle complex data types. Their translation into Lotos makes no particular difficulty. We do not present Lotos parallel composition operators

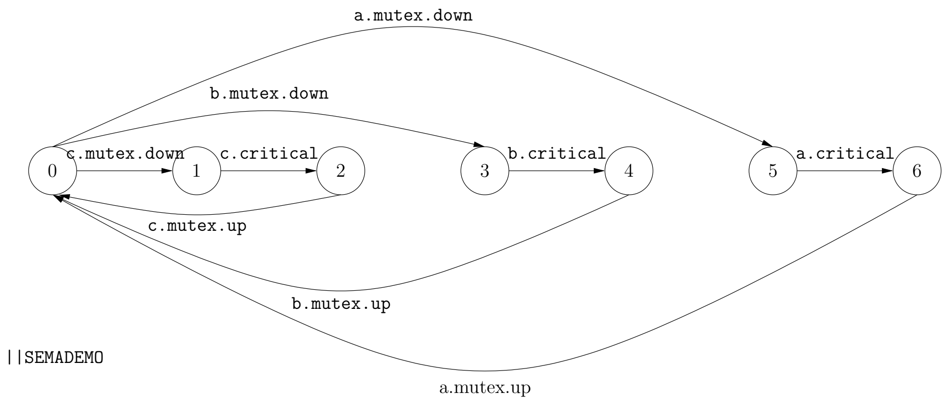


Figure 4: Transition system computed by LTSA for the "SEMADEMO" specification

$$
\begin{array}{ll}
D ::= & \textbf{process}\ P\,\texttt{[}G_1, \ldots, G_n\texttt{]}\,\texttt{(}X_1\texttt{:}\ T_1\texttt{,}\ \ldots\texttt{,}X_m\texttt{:}\ T_m\texttt{)}\texttt{:}\ (\textbf{exit} \mid \textbf{noexit})\ \texttt{:=} \\
 & \quad B_0\ [\textbf{where}\ D_0 \ldots D_p] \\
 & \textbf{endproc}
\end{array}
$$

Figure 5: Syntax of a LOTOS process definition

either, since LOTOS is used only as target language for translating FSP sequential processes, the EXP.OPEN language (see Section 2.4) being used as target language for FSP composite processes.

A LOTOS process has the syntax given in Figure 5. It consists of:

- a process name P;
- a list of *gate parameters* "$G_1, \ldots, G_n$";
- a list of *data parameters* "$X_1, \ldots, X_m$" of respective types (in LOTOS terminology: *sorts*) "$T_1, \ldots, T_m$";
- a *functionality* among "**exit**" if P may end by the "**exit**" behaviour, and "**noexit**" otherwise;
- a behaviour B_0;
- and a possible set of local process definitions "$D_0, \ldots, D_p$".

Figure 6 (page 9) gives the grammar of the subset of LOTOS behaviours used in this article, which consists of sequential behaviours only. The operational semantics of sequential behaviours can be expressed in terms of an LTS. Intuitively, the semantics is the following:

- The "**stop**" behaviour corresponds to deadlock termination. No transition can be derived from "**stop**".
- The "**exit**" behaviour corresponds to normal termination. It produces a transition labeled by "**exit**" and then behaves like "**stop**".
- "$\texttt{[}V\texttt{]} \rightarrow B_0$" behaves either as B_0 if the expression V evaluates to true, or as "**stop**" otherwise.
- "$B_1\,\texttt{[]}\,B_2$" behaves nondeterministically, either as B_1 or as B_2.
- "$B_1\texttt{>>}B_2$" behaves as B_1 until B_1 terminates normally, i.e., produces a transition labeled by "**exit**". This transition is then consumed by the "**>>**" operator and turned into an internal action"**i**", followed by the behaviour of B_2.

B	$::=$	**stop**	deadlock termination
	$\mid$	**exit**	normal termination
	$\mid$	$\texttt{[}V\texttt{]} \rightarrow B_0$	guarded behaviour
	$\mid$	$B_1 \,\texttt{[]}\, B_2$	choice
	$\mid$	$B_1 \texttt{>>} B_2$	sequential composition
	$\mid$	$P\,\texttt{[}G'_1, \ldots, G'_n\texttt{]}\,(V_1, \ldots, V_m)$	process call
	$\mid$	**choice** $X : T$ `[]` B_0	value choice
	$\mid$	$A\texttt{;}\ B_0$	action prefix
	$\mid$	**hide** $G_1, \ldots, G_n$ **in** B_0	hiding
A	$::=$	$G\ O_1\ \ldots\ O_n\ \texttt{[}V\texttt{]}$	(guarded) visible action
	$\mid$	**i**	internal action
O	$::=$	$\texttt{!}V \mid \texttt{?}X : T$	data emission / reception
V	$::=$	$X \mid \mathbf{f}(V_1, \ldots, V_n)$	value expression

Figure 6: Syntax of a subset of (sequential) Lotos behaviours

- "$P\,\texttt{[}G'_1, \ldots, G'_n\texttt{]}\,(V_1, \ldots, V_m)$" corresponds to a call to process P. If P is defined as in Figure 5, this call behaves as B_0 in which the formal gate parameters "$G_1, \ldots, G_n$" are replaced respectively by the actual gate parameters "$G'_1, \ldots, G'_n$", and the formal data parameters "$X_1, \ldots, X_m$" are replaced respectively by the actual values (expressions) "$V_1, \ldots, V_m$". Cyclic behaviours may be defined using tail-recursive process calls.

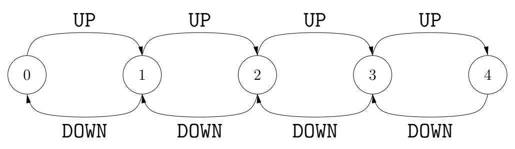
- "**choice** $X : T$ `[]` B_0" behaves as a nondeterministic choice between all instances of B_0 in which X is replaced by some value in T.

- "$G\ O_1\ \ldots\ O_n\ \texttt{[}V\texttt{]}\texttt{;}\ B_0$" corresponds to the prefixing of behaviour B_0 by action "$G\ O_1\ \ldots\ O_n\ \texttt{[}V\texttt{]}$", where G is a gate, "$O_1, \ldots, O_n$" are data expressions called *offers*, and V is a Boolean data expression called *guard*. Each offer O_i has either the form "$\texttt{!}V_i$", which corresponds to the emission of a value V_i, or "$\texttt{?}X_i : T_i$", which corresponds to the reception of any value V_i of sort T_i, stored in a variable X_i. If the guard V in which every variable X_i is replaced by V_i evaluates to true, then "$G\ O_1\ \ldots\ O_n\ \texttt{[}V\texttt{]}\texttt{;}\ B_0$" produces a transition labeled by "$G\ !V_1\ \ldots\ !V_n$" and then behaves as B_0 in which every variable X_i is replaced by V_i. Otherwise, it behaves as "**stop**". For instance, "$\mathit{RECV}\ \texttt{?}X : \mathit{Nat}\ \texttt{[}X \geq 1\texttt{]}\texttt{;}\ \mathit{SEND}\ \texttt{!}X\texttt{;}$ **stop**" produces either a transition labeled by "$\mathit{RECV}\ !0$" followed by a transition labeled by "$\mathit{SEND}\ !0$", and then stops, or a transition labeled by "$\mathit{RECV}\ !1$" followed by a transition labeled by "$\mathit{SEND}\ !1$", and then stops. The special action "**i**" corresponds to an internal action and can neither have offers nor guards.

- "**hide** $G_1, \ldots, G_n$ **in** B_0" behaves as B_0, except that for every transition produced by B_0, the gate of which belongs to "$G_1, \ldots, G_n$", the transition label is replaced by the internal action "**i**".

Example 3 *The following example describes in* Lotos*'s concrete syntax two sequential processes named "*`SEMAPHORE_LOTOS`*" and "*`ACCESS_LOTOS`*".*

```
process SEMAPHORE_LOTOS [UP, DOWN] (N : Nat) : noexit :=
   [ N < 4 ] ->
      UP; SEMAPHORE_LOTOS [UP, DOWN] (N + 1)
   []
   [ N > 0 ] ->
      DOWN; SEMAPHORE_LOTOS [UP, DOWN] (N - 1)
endproc
```



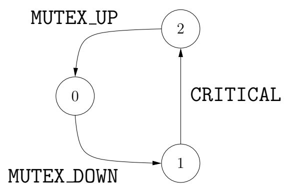


Figure 7: Ltss generated by the Cæsar tool of Cadp for the instances "`SEMAPHORE_LOTOS [UP, DOWN] (0)`" (left) and "`ACCESS_LOTOS [MUTEX_UP, MUTEX_DOWN, CRITICAL]`" (right) of the sequential processes defined in Example 3.

```
process ACCESS_LOTOS [MUTEX_UP, MUTEX_DOWN, CRITICAL] : noexit :=
   MUTEX_DOWN;
      CRITICAL;
         MUTEX_UP;
            ACCESS_LOTOS [MUTEX_UP, MUTEX_DOWN, CRITICAL]
endproc
```

The Lts*s corresponding to the instances "*`SEMAPHORE_LOTOS [UP, DOWN] (0)`*" and "*`ACCESS_LOTOS [MUTEX_UP, MUTEX_DOWN, CRITICAL]`*" are given in Figure 7.* □

2.4 EXP.OPEN

Exp.Open 2.0 [Lan05] is a tool of the Cadp toolbox that allows all applications written using the Open/Cæsar [Gar98] application programming interface to be executed directly on networks of communicating automata. Cadp contains Open/Cæsar applications for step-by-step and random simulation, temporal logic verification, equivalence checking, test generation, etc. For instance, the evaluation of a temporal logic formula described in the file "`prop.mcl`" on the network of automata described in the file "`spec.exp`" using the Open/Cæsar application of Cadp named Evaluator [MS03] can be done using the single command "`exp.open spec.exp evaluator prop.mcl`".

The input language of Exp.Open, which we also call Exp.Open, allows the description of such networks using synchronisation vectors, and generalisations of several parallel composition, renaming, hiding, cutting, and priority operators taken from the process algebras CCS, CSP, Lotos, E-Lotos, and μCrl.

While Lotos synchronisation rules depend on the gate name and only allow synchronisations of transitions that have the same label, Exp.Open allows more flexible label handling mechanisms, such as synchronisations determined by regular expressions, and renaming, hiding, cutting, and synchronisation rules that may depend either upon the gate part of labels as in Lotos, or upon labels as a whole. This additional flexibility of Exp.Open with respect to Lotos will be appropriate when translating FSP concurrent constructs, whose semantics is not easily expressible in Lotos. For this reason, we use Exp.Open instead of Lotos as the target language for FSP concurrent behaviours.

Despite this generality, Exp.Open satisfies nice congruence properties inherited from process algebras, namely: strong bisimulation is a congruence for all Exp.Open operators, and branching bisimulation [vGW89], observational equivalence [Mil89], trace equivalence (also known as language equivalence), weak trace equivalence [BHR84], and safety equivalence [BFG+91] (among others) are congruences for all Exp.Open operators except priority.

We present in Figure 8 (page 11) the part of the Exp.Open language that is used in this article. "$L_1, L'_1, L_2, \ldots$" represent labels, which are merely character strings. In the case of "**rename**", "**hide**", "**cut**", and "**prio**", they may also be regular expressions aimed to match labels. As FSP and Lotos, Exp.Open expressions have an operational semantics defined in terms of an Lts:

- "`"`F`.bcg"`" is the name of a file describing an Lts. Its format called Bcg (*Binary Coded Graph*) [GLMS07] allows a compact representation of very large Ltss.

$$
\begin{array}{llll}
B & ::= & \texttt{"}F\texttt{.bcg"} & \text{graph} \\
 & \mid & \textbf{total rename } L_1 \rightarrow L'_1, \ldots, L_n \rightarrow L'_n \textbf{ in } B_0 \textbf{ end rename} & \text{rename} \\
 & \mid & \textbf{total hide } L_1, \ldots, L_n \textbf{ in } B_0 \textbf{ end hide} & \text{hide} \\
 & \mid & \textbf{total cut } L_1, \ldots, L_n \textbf{ in } B_0 \textbf{ end cut} & \text{cut} \\
 & \mid & \textbf{total prio } L_1, \ldots, L_n > \textbf{all but } L_1, \ldots, L_n \textbf{ in } B_0 \textbf{ end prio} & \text{priority (1)} \\
 & \mid & \textbf{total prio all but } L_1, \ldots, L_n > L_1, \ldots, L_n \textbf{ in } B_0 \textbf{ end prio} & \text{priority (2)} \\
 & \mid & \textbf{label par } L_1, \ldots, L_m \textbf{ in } B_1 \texttt{ || } \ldots \texttt{ || } B_n \textbf{ end par} & \text{parallel (1)} \\
 & \mid & \textbf{label par } V_1, \ldots, V_m \textbf{ in } B_1 \texttt{ || } \ldots \texttt{ || } B_n \textbf{ end par} & \text{parallel (2)} \\
 & \mid & B_1 \texttt{ ||| } B_2 & \text{parallel (3)} \\
V & ::= & A_1 * \ldots * A_n \rightarrow L & \text{sync. vector} \\
A & ::= & L & \text{action} \\
 & \mid & \_ & \text{inaction}
\end{array}
$$

Figure 8: Syntax of a subset of the Exp.Open language

- "**total rename** $L_1 \rightarrow L'_1, \ldots, L_n \rightarrow L'_n$ **in** B_0 **end rename**" behaves as B_0 except that every label matching one of the "L_i $(i \in 1..n)$" is replaced by the corresponding L'_i.
- "**total hide** $L_1, \ldots, L_n$ **in** B_0 **end hide**" behaves as B_0 except that every label matching one of the "L_i $(i \in 1..n)$" is replaced by the internal action "**i**".
- "**total cut** $L_1, \ldots, L_n$ **in** B_0 **end cut**" behaves as B_0 except that every transition whose label matches one of the "L_i $(i \in 1..n)$" is cut, thus potentially making unreachable some states of B_0.
- "**total prio** $L_1, \ldots, L_n >$ **all but** $L_1, \ldots, L_n$ **in** B_0 **end prio**" behaves as B_0 except that every transition whose label matches one of the "L_i $(i \in 1..n)$" takes priority over all other transitions. For "**total prio all but** $L_1, \ldots, L_n > L_1, \ldots, L_n$ **in** B_0 **end prio**", the priority relation is inversed.
- "**label par** $L_1, \ldots, L_m$ **in** B_1 `||` ... `||` B_n **end par**" behaves as the concurrent execution of "$B_1, \ldots, B_n$" with mandatory (n-ary) synchronisation on the labels "$L_1, \ldots, L_m$".
- "**label par** $V_1, \ldots, V_m$ **in** B_1 `||` ... `||` B_n **end par**" behaves as the concurrent execution of "$B_1, \ldots, B_n$" with synchronisation following the constraints expressed by the synchronisation vectors "$V_1, \ldots, V_m$". Precisely, a synchronisation vector (between n expressions "$B_1, \ldots, B_n$", with "$n \geq 1$") is a term of the form "$A_1 * \ldots * A_n \rightarrow L$", where each A_i is either a label, which corresponds to an action of B_i, or the special symbol "_", which corresponds to inaction of B_i. In a given state, the vector "$A_1 * \ldots * A_n \rightarrow L$" produces a transition labeled by L if all B_i such that "$A_i \neq \_$" execute all together a transition labeled by A_i. We call n the length of the synchronisation vector.
- "B_1 `|||` B_2" behaves as the concurrent execution of B_1 and B_2 without synchronisation.

Exp.Open provides other variants of the "**hide**", "**rename**", "**cut**", "**prio**", and "**par**" operators (see [Lan05] for more details). The semantics of each variant is determined by the keyword ("**total**" and "**label**" for the operators described above) that precedes the operator name.

3 Translating FSP Processes into LOTOS and EXP.OPEN

In this section, we describe how a process P of an FSP specification is translated into Lotos and Exp.Open.

3.1 Preliminary Definitions

We will present the translation from FSP to LOTOS using first-order logic and its usual notions of variables, (open and closed) terms, and formulas. Sets may be defined either in extension in the form "$\{e_1, \ldots, e_n\}$", or in intension in the form "$\{t \mid F(x_1, \ldots, x_n)\}$", where t is a term and F a formula whose free variables "$x_1, \ldots, x_n$" are variables of t. The latter denotes the set of closed instances of the term t, such that the valuation of "$x_1, \ldots, x_n$" satisfies the formula "$F(x_1, \ldots, x_n)$". All sets mentioned in this article will be finite.

We represent a partial function from a set S_1 to a set S_2 as a set of couples of the form "$e_1 \mapsto e_2$", where e_1 and e_2 are elements of S_1 and S_2, respectively. We assume that, for a given e_1, at most one e_2 exists such that "$e_1 \mapsto e_2$" belongs to the set. The domain of a function f, denoted by "$\mathsf{dom}(f)$", is defined as the set of elements "$e_1 \in S_1$" such that there exists a couple of the form "$e_1 \mapsto e_2$" in f. In this case, we write "$f(e_1) = e_2$". If "$e_1 \notin \mathsf{dom}(f)$" then "$f(e_1)$" is not legal (undefined value). We represent the empty list by "$()$" and the list of head e and tail T by "$e :: T$".

During the translation from FSP to LOTOS, we will use the following functions and predicates:

- We write "$l{\cdot}m$" the concatenation of labels l and m. We write "ϵ" the neutral element of concatenation, i.e. such that "$(\forall l)\ \epsilon \cdot l = l \cdot \epsilon = l$".
- The *dispatching function* "$\mapsto_d$" takes two sets of labels. It returns a partial function from labels to sets of labels, which associates every element of the first set to the second:

$$\{l_i \mid i \in 1..n\} \mapsto_d \{m_j \mid j \in 1..p\} = \{l_i \mapsto \{m_j \mid j \in 1..p\} \mid i \in 1..n\}$$

- Function "$\otimes$" takes two sets of labels and returns the set of labels obtained by (combinatorial) concatenation of labels taken in each set:

$$\{l_i \mid i \in 1..n\} \otimes \{m_j \mid j \in 1..p\} = \{l_i \cdot m_j \mid i \in 1..n\ \wedge\ j \in 1..p\}$$

- The *prefix matching test* "$\mathsf{pm?}$" takes as inputs a label l and a set of labels, and evaluates to `true` if one of the labels in the list is a prefix for l:

$$\mathsf{pm?}(l, \{l_i \mid i \in 1..n\}) = ((\exists i \in 1..n)\ (\exists m)\ l = l_i \cdot m)$$

- A *relabeling* is a partial function from labels to sets of labels, such that a single label may be replaced by several ones, yielding several transitions. Function "$\mathsf{relabel}$" takes as inputs a label l and a relabeling, and returns the set of labels obtained after relabeling every prefix of l that belongs to the domain of the relabeling:

$$\mathsf{relabel}(l, R) = \begin{cases} \{m' \cdot l' \mid l = m \cdot l' \wedge m \in \mathsf{dom}(R) \wedge m' \in R(m)\} & \text{if } \mathsf{pm?}(l, \mathsf{dom}(R)) \\ \{l\} & \text{otherwise} \end{cases}$$

- In the sequel, FSP sequential composition will have to be translated into the LOTOS sequential composition operator "B_1>>B_2", whose semantics introduces an internal action "**i**" between the end of B_1 and the beginning of B_2, as explained page 8. This internal action does not exist in the semantics of FSP sequential composition. We will see that, to ensure a strong equivalence between the source FSP specification and the target LOTOS specification of a sequential process[1], those internal actions can be removed by using LTS minimisations modulo branching bisimulation. Therefore, we must distinguish such "**i**" actions from the internal actions obtained by hiding of FSP labels, which must appear in the LTS corresponding to the specification. Therefore, we consider a different LOTOS label written "`TAU`", as well as the following function "hide", which takes as inputs a label l and a set of labels H, and returns "`TAU`" if l has to be hidden, or l otherwise:

$$\mathsf{hide}(l, H) = \begin{cases} \texttt{TAU} & \text{if } \mathsf{pm?}(l, H) \\ l & \text{otherwise} \end{cases}$$

[1]Note that weaker equivalences are not congruences in concurrent languages which contain priority operators, such as FSP. Therefore, strong equivalence is an important requirement as regards the semantic correctness of the translator.

As regards FSP data expressions, we will also use the standard function "type", which computes the type of an FSP expression.

The translation of an FSP specification into LOTOS/EXP.OPEN requires to collect and propagate along the abstract syntax tree of the FSP specification, information about the context of the process under translation. Such context information, called an *environment*, consists of the following elements:

- E, called *variable environment*, is a partial function from variables to LOTOS expressions. E is initialised with the constant definitions, which are global to all processes, and will be extended to store the value of parameters and variables.
- X, called *constraint environment*, is a partial function from variables to integer ranges of the form "(v_1, v_2)" corresponding to the set of natural numbers ranging from v_1 to v_2. For a variable x, "$X(x)$" denotes the set of numbers in which x may take its value.
- M, called *relabeling environment*, is a list of tuples "(R, H)" where R is a relabeling and H is a set of labels to be hidden.

3.2 Translating Data and Label Expressions

Given a variable environment E, an FSP data expression is translated into a LOTOS data expression using the "$\mathsf{f2l_e}$" function defined below. We assume that every FSP data operator written "$\mathbf{f}$" can be translated into a LOTOS data operator "$\overline{\mathbf{f}}$". Indeed, FSP contains a fixed set of data operators, which can be easily translated into LOTOS data operators, defined using first order conditional algebraic equations. The precise translation of FSP data operators into LOTOS is standard and out of the scope of this paper.

$\mathsf{f2l_e}$: FSP expression × variable environment → LOTOS expression

$$\begin{array}{rcl} \mathsf{f2l_e}(x, E) & = & E(x) \\ \mathsf{f2l_e}(\mathbf{f}(V_1, \ldots, V_n), E) & = & \overline{\mathbf{f}}(\mathsf{f2l_e}(V_1, E), \ldots, \mathsf{f2l_e}(V_n, E)) \end{array}$$

Both FSP and LOTOS have a rich syntax of expressions to represent labels, so that each label expression evaluates into a set of labels. However, label expressions are structured much differently in each of these languages.

On the one hand, FSP label expressions are concatenations of smaller label expressions. It is not always possible to say at compile-time whether a label expression will be renamed or hidden, because the hiding or renaming operator will act differently on the different values of the label expression. The label expression has to be *expanded*, i.e., replaced by its values (a set of labels) to determine which labels of this set are to be renamed or hidden. This is how LTSA operates while generating a transition system corresponding to an FSP specification.

On the other hand, LOTOS labels are more structured, since they consist of a static part (the gate) and an evaluable part (the offers). The fact that a label expression will be renamed (through gate instantiation) or hidden can be determined statically because it only depends on the gate part. This is how the CÆSAR compiler of LOTOS operates.

Therefore, the translation from FSP labels into LOTOS cannot be straightforward: in some cases, we can translate an FSP label expression into a single LOTOS label expression, but in many cases, we must expand the FSP label expression into several LOTOS labels, depending on the operations performed on the labels.

To translate labels, we thus define two functions, named "expand" and "$\mathsf{expand_x}$", defined below, which translate an FSP label expression in a given environment into a set of tuples consisting of a LOTOS label and an updated environment. Function "expand" expands each FSP label expression into a set of couples consisting of a LOTOS label without variables and a variable environment that associates to each variable occurring in the FSP label the value given by the expansion. For instance, the FSP label "x : 0..2" is translated by "expand" into the set "$\{(0, \{x \mapsto 0\}), (1, \{x \mapsto 1\}), (2, \{x \mapsto 2\})\}$" corresponding to all possible values for the FSP label and subsequent bindings for x. By contrast, function "$\mathsf{expand_x}$" keeps the range variables occurring in the FSP label expression as this allows the translation of FSP labels into more compact sets of LOTOS labels. It thus expands each FSP label expression into a set of triples consisting of a LOTOS label which may contain variables, a variable environment, and a constraint environment that associates the appropriate range to each variable occurring in the FSP label. For instance, the FSP label "$x : 0..2$" is translated by "$\mathsf{expand_x}$" into the set "$\{(x, \{x \mapsto x\}, \{x \mapsto (0, 2)\})\}$". During the translation, function "expand" will be used instead of "$\mathsf{expand_x}$" only when required for a correct translation of FSP hiding or renaming be possible.

Functions "expand" and "$\mathsf{expand_x}$" use respectively the auxiliary functions "$\mathsf{expand_l}$" and "$\mathsf{expand_{lx}}$" defined thereafter, which expands a sublabel.

expand : FSP label × variable environment → (expanded label × variable environment) set

$$\mathsf{expand}(L_1 \ldots L_n, E) = \{(l_1 \cdot \ldots \cdot l_n, E_{n+1}) \mid E_1 = E \wedge (\forall i \in 1..n)\ (l_i, E_{i+1}) \in \mathsf{expand_l}(L_i, E_i)\}$$

$\mathsf{expand_l}$: FSP sublabel × variable environment → (expanded label × variable environment) set

$$\begin{array}{rcl}
\mathsf{expand_l}(act, E) & = & \{(act, E)\} \\
\mathsf{expand_l}(V, E) & = & \{(\mathsf{f2l_e}(V, E), E)\} \\
\mathsf{expand_l}(x\!:\!V, E) & = & \{(v, E \cup \{x \mapsto v\})\} \text{ where } v = \mathsf{f2l_e}(V, E) \\
\mathsf{expand_l}(\{A_1, \ldots, A_n\}, E) & = & \bigcup_{i \in 1..n} \mathsf{expand}(A_i, E) \\
\mathsf{expand_l}(x\!:\!\{A_1, \ldots, A_n\}, E) & = & \bigcup_{i \in 1..n} \{(l_i, E_i \cup \{x \mapsto l_i\}) \mid (l_i, E_i) \in \mathsf{expand}(A_i, E)\} \\
\mathsf{expand_l}(V_1..V_2, E) & = & \{(i, E) \mid i \in \mathsf{f2l_e}(V_1, E)..\,\mathsf{f2l_e}(V_2, E)\} \\
\mathsf{expand_l}(x\!:\!V_1..V_2, E) & = & \{(i, E \cup \{x \mapsto i\}) \mid i \in \mathsf{f2l_e}(V_1, E)..\,\mathsf{f2l_e}(V_2, E)\}
\end{array}$$

$\mathsf{expand_x}$: FSP label × variable environment × constraint environment
→ (expanded label × variable environment × constraint environment) set

$$\begin{array}{l}
\mathsf{expand_x}(L_1 \ldots L_n, E, X) = \\
\quad \{(l_1 \cdot \ldots \cdot l_n, E_{n+1}, X_{n+1}) \mid E_1 = E \wedge X_1 = X \wedge (\forall i \in 1..n)\ (l_i, E_{i+1}, X_{i+1}) \in \mathsf{expand_{lx}}(L_i, E_i, X_i)\}
\end{array}$$

$$\mathsf{expand_{lx}} : \text{FSP sublabel} \times \text{variable environment} \times \text{constraint environment} \\ \rightarrow (\text{expanded label} \times \text{variable environment} \times \text{constraint environment}) \text{ set}$$

$$\begin{array}{rcl}
\mathsf{expand_{lx}}(act, E, X) & = & \{(act, E, X)\} \\
\mathsf{expand_{lx}}(V, E, X) & = & \{(\mathsf{f2l_e}(V, E), E, X)\} \\
\mathsf{expand_{lx}}(x\!:\!V, E, X) & = & \{(v, E \cup \{x \mapsto v\}, X)\} \\
 & & \text{where } v = \mathsf{f2l_e}(V, E) \\
\mathsf{expand_{lx}}(\{A_1, \ldots, A_n\}, E, X) & = & \bigcup_{i \in 1..n} \mathsf{expand_x}(A_i, E, X)\} \\
\mathsf{expand_{lx}}(x\!:\!\{A_1, \ldots, A_n\}, E, X) & = & \bigcup_{i \in 1..n} \{(l_i, E_i \cup \{x \mapsto l_i\}, X_i) \mid (l_i, E_i, X_i) \in \mathsf{expand_x}(A_i, E, X)\} \\
\mathsf{expand_{lx}}(V_1..V_2, E, X) & = & \{(x, E \cup \{x \mapsto x\}, X \cup \{x \mapsto (\mathsf{f2l_e}(V_1, E), \mathsf{f2l_e}(V_2, E))\})\} \\
 & & \text{where } x \text{ is an unused variable} \\
\mathsf{expand_{lx}}(x\!:\!V_1..V_2, E, X) & = & \{(x, E \cup \{x \mapsto x\}, X \cup \{x \mapsto (\mathsf{f2l_e}(V_1, E), \mathsf{f2l_e}(V_2, E))\})\}
\end{array}$$

Example 4
$\mathsf{expand}(\texttt{lab[x:1..2]}, \emptyset) = \{(lab \cdot 1, \{x \mapsto 1\}), (lab \cdot 2, \{x \mapsto 2\})\}$
$\mathsf{expand_x}(\texttt{lab[x:1..2]}, \emptyset, \emptyset) = \{(lab \cdot x, \{x \mapsto x\}, \{x \mapsto (1, 2)\})\}$ $\square$

3.3 Relabel Test

We now define the "relabel?" function, which tests whether a set of labels is affected by hiding or renaming contained in a relabeling environment. This function is used when translating sequences of labels, to decide which of the "expand" or "$\mathsf{expand_x}$" functions has to be used. Indeed, if hiding or renaming has an effect on the list of labels, then variables must be totally expanded, i.e., the "expand" function must be used.

$$\mathsf{relabel?} : \text{expanded labels} \times \text{relabeling environment} \rightarrow \text{Boolean}$$

$$\begin{array}{rcl}
\mathsf{relabel?}(\{l_i \mid i \in 1..n\}, ()) & = & \mathsf{false} \\
\mathsf{relabel?}(\{l_i \mid i \in 1..n\}, (R, H) :: M) & = & (\exists i \in 1..n)\ \mathsf{pm?}(l_i, H \cup \mathsf{dom}(R)) \ \vee\ \mathsf{relabel?}(\{l_i \mid i \in 1..n\}, M)
\end{array}$$

3.4 Translating Sequential Processes into LOTOS

FSP sequential processes are translated into Lotos processes. If E_0 is the initial environment containing the definitions of constants, and the main process P of the FSP specification is sequential, then P is translated into "$\mathsf{f2l_{sd}}(\hat{P}, E_0, ())$", where "$\mathsf{f2l_{sd}}$" is defined below. It uses the auxiliary functions "$\mathsf{f2l_{lp}}$", which translates a local FSP process (defined as a set of equations) into a Lotos process, "func", which computes the Lotos functionality resulting from the translation of an FSP process, and "$\mathsf{f2l_b}$", defined thereafter.

$\mathsf{f2l_{sd}}$: FSP sequential process definition $\times$ variable environment $\times$ relabeling environment $\rightarrow$ LOTOS process

$$\mathsf{f2l_{sd}}\left(\begin{array}{l} P\texttt{(}x_1\texttt{=}V_1,\ldots,x_k\texttt{=}V_k\texttt{) =} \\ B_0, D_{l_1},\ldots,D_{l_m} \\ +\{A_{e_1},\ldots,A_{e_n}\} \\ /\{A'_{r_1}/A_{r_1},\ldots,A'_{r_p}/A_{r_p}\} \\ \backslash\{A_{h_1},\ldots,A_{h_q}\} \end{array}\right), E, M) \quad = \quad \left(\begin{array}{l} \textbf{process}\ P'\ \texttt{[EVENT,TAU,ERROR] :}\ \mathsf{func}(B_0)\ \texttt{:=} \\ \quad \mathsf{f2l_b}(B_0, E_0, (R,H) :: M) \\ \quad \textbf{where} \\ \qquad \mathsf{f2l_{lp}}(D_{l_1}, E_0, (R,H) :: M) \\ \qquad \ldots \\ \qquad \mathsf{f2l_{lp}}(D_{l_m}, E_0, (R,H) :: M) \\ \textbf{endproc} \end{array}\right)$$

where P' is an unused name
$E_0 = \{x_1 \mapsto V_1,\ldots,x_k \mapsto V_k\} \cup E$
$(\forall i \in 1..p)\ S_i = \{l \mid (l,E) \in \mathsf{expand}(A_{r_i}, E_0)\}$
$(\forall i \in 1..p)\ S'_i = \{l \mid (l,E) \in \mathsf{expand}(A'_{r_i}, E_0)\}$
$R = \bigcup_{i\in 1..p} S_i \mapsto_d S'_i$
$H = \bigcup_{i\in 1..q} \{l \mid (l,E) \in \mathsf{expand}(A_{h_i}, E_0)\}$

$\mathsf{f2l_{lp}}$: FSP local process definition $\times$ variable environment $\times$ relabeling environment $\rightarrow$ LOTOS process

$$\mathsf{f2l_{lp}}\left(\begin{array}{l} P\texttt{[}x_1^1\texttt{:}L_1^1\texttt{]}\ldots\texttt{[}x_1^n\texttt{:}L_1^n\texttt{] =}\ B_1, \\ \ldots, \\ P\texttt{[}x_m^1\texttt{:}L_m^1\texttt{]}\ldots\texttt{[}x_m^1\texttt{:}L_m^n\texttt{] =}\ B_m \end{array}\right), E, M) \quad =$$

$$\left(\begin{array}{l} \textbf{process}\ P\ \texttt{[EVENT,TAU,ERROR]}\ \texttt{(}x_1\texttt{:}T_1,\ldots,x_n\texttt{:}T_n\texttt{)}\ \texttt{:}\ F\ \texttt{:=} \\ \quad (\\ \qquad [C_1]\ \rightarrow \mathsf{f2l_b}(B_1, \{x_1^1 \mapsto x_1,\ldots,x_1^n \mapsto x_n\} \cup E, M) \\ \qquad [] \\ \qquad [\neg C_1]\ \rightarrow \\ \qquad\quad (\\ \qquad\qquad [C_2]\ \rightarrow \mathsf{f2l_b}(B_2, \{x_2^1 \mapsto x_1,\ldots,x_2^n \mapsto x_n\} \cup E, M) \\ \qquad\qquad [] \\ \qquad\qquad [\neg C_2]\ \rightarrow \\ \qquad\qquad\quad \ldots \\ \qquad\qquad\quad (\\ \qquad\qquad\qquad [C_m]\ \rightarrow \mathsf{f2l_b}(B_m, \{x_m^1 \mapsto x_1,\ldots,x_m^n \mapsto x_n\} \cup E, M) \\ \qquad\qquad\qquad [] \\ \qquad\qquad\qquad [\neg C_m]\ \rightarrow \mathsf{f2l_b}(\textbf{error}, E, M) \\ \qquad\qquad\quad) \\ \qquad\qquad\quad \ldots \\ \qquad\quad) \\ \quad) \\ \textbf{endproc} \end{array}\right)$$

where $x_1,\ldots,x_n$ are unused variables
$(\forall i \in 1..n)\ T_i = \mathsf{type}(x_i)$
$F = \mathsf{func}(B_1\ \texttt{|}\ \ldots\ \texttt{|}\ B_m)$
$(\forall i \in 1..m, j \in 1..n)\ S_i^j = \{l \mid (l,E') \in \mathsf{expand_l}(L_i^j, E)\}$
$(\forall i \in 1..m)\ C_i = (x_1 \in S_i^1) \wedge \ldots \wedge (x_n \in S_i^n)$

$$\mathsf{func} : \text{FSP sequential behaviour} \rightarrow \{\mathbf{exit}, \mathbf{noexit}\}$$

$$\mathsf{func}(B) \;=\; \begin{cases} \mathbf{exit} & \text{if } \mathsf{exit}(B) \\ \mathbf{noexit} & \text{otherwise} \end{cases}$$

$$\begin{aligned}
\text{where}\quad & \mathsf{exit}(\mathbf{stop}) = \mathsf{false} \\
& \mathsf{exit}(\mathbf{end}) = \mathsf{true} \\
& \mathsf{exit}(\mathbf{error}) = \mathsf{false} \\
& \mathsf{exit}(P\,\texttt{(}V_1, \ldots, V_n\texttt{)}\,\texttt{;}\ B) = \mathsf{exit}(B) \\
& \mathsf{exit}(P\,\texttt{[}V_1\texttt{]} \ldots \texttt{[}V_n\texttt{]}) = \mathsf{false} \\
& \mathsf{exit}(\mathbf{if}\ V\ \mathbf{then}\ B_1\ \mathbf{else}\ B_2) = \mathsf{exit}(B_1) \ \vee\ \mathsf{exit}(B_2) \\
& \mathsf{exit}(\mathbf{when}\ V_1 \rightarrow B_1\ \texttt{|}\ \ldots\ \texttt{|}\ \mathbf{when}\ V_n\ \rightarrow B_n) = \textstyle\bigvee_{i \in 1..n} \mathsf{exit}(B_i) \\
& \mathsf{exit}(A \rightarrow B) = \mathsf{exit}(B)
\end{aligned}$$

The translation from FSP sequential behaviours into Lotos is done by the "$\mathsf{f2l_b}$" function defined in Figure 9. "$\mathsf{f2l_b}$" also uses auxiliary functions "$\mathsf{apply_{RH}}$", "$\mathsf{f2l_s}$", and "$\mathsf{f2l_{sx}}$", defined below.

Lotos behaviours generated by the translation contain three gates, named "`EVENT`", "`TAU`", and "`ERROR`". Every Lotos visible label is made of the "`EVENT`" gate with an offer corresponding to a visible label obtained by translation of an FSP label using function "expand" or "$\mathsf{expand_x}$". The choice between "expand" and "$\mathsf{expand_x}$" depends whether A has to be relabeled: if so, A is expanded using the "expand" function; if not, the "$\mathsf{expand_x}$" function is used instead. The "`ERROR`" gate is used to encode FSP error termination. At last, the "`TAU`" gate is used to encode the FSP internal action as already explained in Section 3.1.

Function "$\mathsf{apply_{RH}}$" computes a set of labels resulting from a list of operations (renaming and hiding) on labels. It uses the auxiliary functions "$\mathsf{apply_R}$" and "$\mathsf{apply_H}$", defined below, which compute a set of labels resulting from renaming and hiding, respectively.

$$\mathsf{apply_{RH}} : \text{expanded label set} \times \text{relabeling environment} \rightarrow \text{expanded label set}$$

$$\begin{aligned}
\mathsf{apply_{RH}}(\{l_i \mid i \in 1..n\}, ()) \;&=\; \{l_i \mid i \in 1..n\} \\
\mathsf{apply_{RH}}(\{l_i \mid i \in 1..n\}, (R, H) :: M) \;&=\; \mathsf{apply_{RH}}(\mathsf{apply_H}(\mathsf{apply_R}(\{l_i \mid i \in 1..n\}, R), H), M)
\end{aligned}$$

$$\mathsf{apply_R} : \text{expanded label set} \times \text{relabeling} \rightarrow \text{expanded label set}$$

$$\mathsf{apply_R}(\{l_i \mid i \in 1..n\}, R) = \textstyle\bigcup_{i \in 1..n} \mathsf{relabel}(l_i, R)$$

$$\mathsf{apply_H} : \text{expanded label set} \times \text{expanded label set} \rightarrow \text{expanded label set}$$

$$\mathsf{apply_H}(\{l_i \mid i \in 1..n\}, H) = \{\mathsf{hide}(l_i, H) \mid i \in 1..n\}$$

$\mathsf{f2l_b}$: FSP sequential behaviour $\times$ variable environment $\times$ relabeling environment $\rightarrow$ LOTOS behaviour

$$\mathsf{f2l_b}(\mathbf{stop}, E, M) = \mathbf{stop}$$

$$\mathsf{f2l_b}(\mathbf{end}, E, M) = \mathbf{exit}$$

$$\mathsf{f2l_b}(\mathbf{error}, E, M) = \texttt{P\_ERROR [ERROR]}$$

where the LOTOS process `P_ERROR` is defined as:

```
process P_ERROR [ERROR] : noexit :=
    ERROR; P_ERROR [ERROR]
endproc
```

$$\mathsf{f2l_b}(P\texttt{(}V_1, \ldots, V_n\texttt{);}\ B_0, E, M) = P'\ \texttt{[EVENT, TAU, ERROR] >>}\ \mathsf{f2l_b}(B_0, E, M)$$

where P' is the LOTOS process defined by $\mathsf{f2l_{sd}}(\hat{P}[V_1, \ldots, V_n], E, M)$

$$\mathsf{f2l_b}(P\texttt{[}V_1\texttt{]} \ldots \texttt{[}V_n\texttt{]}, E, M) = P\ \texttt{[EVENT, TAU, ERROR] (}\mathsf{f2l_e}(V_1, E, M), \ldots, \mathsf{f2l_e}(V_n, E, M)\texttt{)}$$

$$\mathsf{f2l_b}(A \rightarrow B_0, E, M) = \begin{cases} \begin{pmatrix} \mathsf{f2l_s}(\mathsf{apply_{RH}}(\{l_1\}, M), \mathsf{f2l_b}(B_0, E_1 \cup E, M)) \\ \texttt{[]}\ \ldots\ \texttt{[]} \\ \mathsf{f2l_s}(\mathsf{apply_{RH}}(\{l_h\}, M), \mathsf{f2l_b}(B_0, E_h \cup E, M)) \end{pmatrix} & \text{if } \mathsf{relabel?}(S, M) \\ \begin{pmatrix} \mathsf{f2l_{sx}}(l'_1, X_1, \mathsf{f2l_b}(B_0, E'_1 \cup E, M)) \\ \texttt{[]}\ \ldots\ \texttt{[]} \\ \mathsf{f2l_{sx}}(l'_m, X_m, \mathsf{f2l_b}(B_0, E'_m \cup E, M)) \end{pmatrix} & \text{otherwise} \end{cases}$$

where $\{(l_i, E_i) \mid i \in 1..h\} = \mathsf{expand}(A, E)$
$S = \{l_1, \ldots, l_h\}$
$\{(l'_i, E'_i, X_i) \mid i \in 1..m\} = \mathsf{expand_x}(A, E, X)$

$$\mathsf{f2l_b}(\mathbf{if}\ V\ \mathbf{then}\ B_1\ \mathbf{else}\ B_2, E, M) = \begin{pmatrix} \texttt{[}\,\mathsf{f2l_e}(V, E)\texttt{]} \rightarrow \mathsf{f2l_b}(B_1, E, M) \\ \texttt{[]} \\ \texttt{[}\neg\,\mathsf{f2l_e}(V, E)\texttt{]} \rightarrow \mathsf{f2l_b}(B_2, E, M) \end{pmatrix}$$

$$\mathsf{f2l_b}(\mathbf{when}\ V_1\ B_1\ \texttt{|}\ \ldots\ \texttt{|}\ \mathbf{when}\ V_n\ B_n, E, M) = \begin{pmatrix} \texttt{[}\,\mathsf{f2l_e}(V_1, E)\texttt{]} \rightarrow \mathsf{f2l_b}(B_1, E, M) \\ \texttt{[]}\ \ldots\ \texttt{[]} \\ \texttt{[}\,\mathsf{f2l_e}(V_n, E)\texttt{]} \rightarrow \mathsf{f2l_b}(B_n, E, M) \end{pmatrix}$$

Figure 9: Definition of function “$\mathsf{f2l_b}$”

Functions "$\mathsf{f2l_s}$" and "$\mathsf{f2l_{sx}}$", defined below, generate either a LOTOS choice from a set of labels, or a single label if the set is a singleton. They also choose the appropriate LOTOS sequential composition operator between "`>>`" and "`;`", depending whether the set of labels is a singleton or not.

$\mathsf{f2l_s}$: expanded label set × LOTOS behaviour → LOTOS behaviour

$$\mathsf{f2l_s}(\{l_i \mid i \in 1..n \ \wedge \ n > 0\}, B) \ = \begin{cases} l_1\texttt{;}\ B & \text{if } n = 1 \\ \texttt{(}l_1\texttt{; }\textbf{exit}\texttt{ [] } \ldots \texttt{ [] } l_n\texttt{; }\textbf{exit}\texttt{) >> } B & \text{otherwise} \end{cases}$$

$\mathsf{f2l_{sx}}$: expanded label set × constraint environment × LOTOS behaviour → LOTOS behaviour

$$\mathsf{f2l_{sx}}(l, \{x_j \mapsto (v_j, v'_j) \mid j \in 1..m\}, B) \ = \ \textbf{choice } x_1\texttt{:}T_1\texttt{,} \ldots \texttt{,}x_m\texttt{:}T_m \texttt{ [] ([}V\texttt{]} \rightarrow l\texttt{;}B\texttt{)}$$
$$\text{where} \quad V = \textstyle\bigwedge_{j \in 1..m}((x_j \geq v_j) \ \wedge \ (x_j \leq v'_j))$$
$$(\forall i \in 1..m)\ T_i = \mathsf{type}(x_i)$$

3.5 Process alphabets

Due to the semantics of the parallel composition operator of FSP, the translation of composite processes requires to compute the alphabet of a process, i.e., its set of reachable labels. Function "alph" computes such alphabets. We first define below "alph" for sequential processes, then for composite processes. The auxiliary function "$\mathsf{alph_m}$" computes the alphabet of a process definition.

alph (sequential processes) : FSP sequential behaviour × variable environment → expanded label set

$$\begin{aligned}
\mathsf{alph}(\textbf{stop}, E) &= \emptyset \\
\mathsf{alph}(\textbf{end}, E) &= \emptyset \\
\mathsf{alph}(\textbf{error}, E) &= \emptyset \\
\mathsf{alph}(A \rightarrow B_0, E) &= \textstyle\bigcup_{(l,E') \in \mathsf{expand}(A,E)} (\{l\} \cup \mathsf{alph}(B_0, E \cup E')) \\
\mathsf{alph}(P\texttt{(}V_1\texttt{,} \ldots \texttt{,}V_k\texttt{);}\ B_0, E) &= \mathsf{alph_m}(\hat{P}[V_1, \ldots, V_k], E, M) \cup \mathsf{alph}(B_0, E) \\
\mathsf{alph}(P\texttt{[}V_1\texttt{]} \ldots \texttt{[}V_n\texttt{]}, E) &= \emptyset \\
\mathsf{alph}(\textbf{if } V \textbf{ then } B_1 \textbf{ else } B_2, E) &= \begin{cases} \mathsf{alph}(B_1, E) & \text{if } \mathsf{f2l_e}(V, E) = \mathsf{true} \\ \mathsf{alph}(B_2, E) & \text{otherwise} \end{cases} \\
\mathsf{alph}(\textbf{when } V_1 \rightarrow B_1 \ \texttt{|} \ldots \texttt{|} \textbf{when } V_n \ \rightarrow B_n, E) &= \textstyle\bigcup_{i \in 1..n \ \wedge \ \mathsf{f2l_e}(V_i, e) = \mathsf{true}} \mathsf{alph}(B_i, E)
\end{aligned}$$

alph (composite processes) : FSP composite behaviour $\times$ variable environment $\rightarrow$ expanded label set

$$\mathsf{alph}(P\texttt{(}V_1,\ldots,V_k\texttt{)},E) = \mathsf{alph_m}(\hat{P}[V_1,\ldots,V_k],E)$$

$$\begin{aligned}\mathsf{alph}(C_0/\{A'_1/A_1,\ldots,A'_n/A_n\},E) &= \mathsf{apply_R}(\mathsf{alph}(C_0,E),\textstyle\bigcup_{i\in 1..n} S_i \mapsto_d S'_i)\\ &\quad \text{where } (\forall i\in 1..n)\ S_i=\{l \mid (l,E')\in \mathsf{expand}(A_i,E)\}\\ &\quad\qquad\ (\forall i\in 1..n)\ S'_i=\{l \mid (l,E')\in \mathsf{expand}(A'_i,E)\}\end{aligned}$$

$$\mathsf{alph}(\{A_1,\ldots,A_n\}\texttt{::}C_0,E) = \textstyle\bigcup_{i\in 1..n}\{l \mid (l,E')\in \mathsf{expand}(A_i,E)\}\otimes \mathsf{alph}(C_0,E)$$

$$\mathsf{alph}(\{A_1,\ldots,A_n\}\texttt{:}C_0,E) = \textstyle\bigcup_{i\in 1..n}\{l \mid (l,E')\in \mathsf{expand}(A_i,E)\}\otimes \mathsf{alph}(C_0,E)$$

$$\mathsf{alph}(\textbf{if } V \textbf{ then } C_1 \textbf{ else } C_2,E) = \begin{cases}\mathsf{alph}(C_1,E) & \text{if } \mathsf{f2l_e}(V,E)\\ \mathsf{alph}(C_2,E) & \text{otherwise}\end{cases}$$

$$\mathsf{alph}(C_1\,\texttt{||}\ldots\texttt{||}\,C_k,E) = \textstyle\bigcup_{i\in 1..k}\mathsf{alph}(C_i,E)$$

$$\begin{aligned}\mathsf{alph}(\textbf{forall } [x_1\texttt{:}L_1]\ldots[x_n\texttt{:}L_n]\ C_0,E) &= \textstyle\bigcup_{l_1\in S_1,\ldots,l_n\in S_n}\mathsf{alph}(C_0,E\cup\{x_1\mapsto l_1,\ldots,x_n\mapsto l_n\})\\ &\quad \text{where } (\forall i\in 1..n)\ S_i=\{l_i \mid (l_i,E')\in \mathsf{expand}(L_i,E)\}\end{aligned}$$

$\mathsf{alph_m}$ (sequential processes) : FSP sequential process definition $\times$ variable environment $\rightarrow$ expanded label set

$$\mathsf{alph_m}\left(\begin{pmatrix}P\texttt{(}x_1\texttt{=}V_1\texttt{,}\ldots\texttt{,}x_k\texttt{=}V_k\texttt{)}\ \texttt{=}\\ B,D_{l_1},\ldots,D_{l_m}\\ +\{A_{e_1},\ldots,A_{e_n}\}\\ /\{A'_{r_1}/A_{r_1},\ldots,A'_{r_p}/A_{r_p}\}\\ \backslash\{A_{h_1},\ldots,A_{h_q}\}\end{pmatrix},E\right) = \begin{pmatrix}\mathsf{apply_R}(\mathsf{apply_H}(\mathsf{alph}(B,E_0)\cup\\ \bigcup_{i\in 1..m}\mathsf{alph_m}(D_{l_i},E_0),H_0),R_0)\cup\\ \bigcup_{i\in 1..n}\{l \mid (l,E')\in \mathsf{expand}(A_{e_i},E)\}\end{pmatrix}$$

$$\begin{aligned}\text{where } E_0 &= \{x_1\mapsto V_1,\ldots,x_k\mapsto V_k\}\cup E\\ (\forall i\in 1..p)\ S_i &= \{l \mid (l,E')\in \mathsf{expand}(A_{r_i},E)\}\\ (\forall i\in 1..p)\ S'_i &= \{l \mid (l,E')\in \mathsf{expand}(A'_{r_i},E)\}\\ R_0 &= \textstyle\bigcup_{i\in 1..p} S_i\mapsto_d S'_i\\ H_0 &= \textstyle\bigcup_{i\in 1..q}\{l \mid (l,E')\in \mathsf{expand}(A_{h_i},E)\}\end{aligned}$$

$$\mathsf{alph_m}\left(\begin{pmatrix}P\,\texttt{[}x_1^1\texttt{:}L_1^1\texttt{]}\ldots\texttt{[}x_1^n\texttt{:}L_1^n\texttt{]}\ \texttt{=}\,B_1,\\ \ldots,\\ P\,\texttt{[}x_m^1\texttt{:}L_m^1\texttt{]}\ldots\texttt{[}x_m^n\texttt{:}L_m^n\texttt{]}\ \texttt{=}\,B_m\end{pmatrix},E\right) = \begin{pmatrix}\bigcup_{i\in 1..m}\bigcup_{l_1\in S_i^1\wedge\ldots\wedge l_n\in S_i^n}\\ \mathsf{alph}(B_i,E\cup\{x_i^1\mapsto l_1,\ldots,x_i^n\mapsto l_n\})\end{pmatrix}$$

$$\begin{aligned}&\text{where } (\forall i\in 1..m)(\forall j\in 1..n)\\ &\qquad S_i^j=\{l \mid (l,E')\in \mathsf{expand_l}(L_i^j,E)\}\end{aligned}$$

$\mathsf{alph_m}$ (composite processes) : FSP composite process definition $\times$ variable environment $\rightarrow$ expanded label set

$$\mathsf{alph_m}\left(\begin{pmatrix}\texttt{||}P\texttt{(}x_1\texttt{=}V_1,\ldots,x_k\texttt{=}V_k\texttt{)}\ \texttt{=}\ C\\ \gg\{A_{p_1},\ldots,A_{p_n}\}\\ \backslash\{A_{h_1},\ldots,A_{h_q}\}\end{pmatrix},E\right) = \mathsf{apply_H}(\mathsf{alph}(C,E_0),H_0)$$

$$\begin{aligned}\text{where } E_0 &= \{x_1\mapsto V_1,\ldots,x_k\mapsto V_k\}\cup E\\ H_0 &= \textstyle\bigcup_{i\in 1..q}\{l \mid (l,E')\in \mathsf{expand}(A_{h_i},E)\}\end{aligned}$$

3.6 Translating Composite Processes into EXP.OPEN

If E_0 is the initial environment containing the definitions of constants, the FSP composite process P of interest is translated into the Exp.Open expression "$\mathsf{f2l_{cd}}(\hat{P}, E_0)$" where "$\mathsf{f2l_{cd}}$" is defined below.

$\mathsf{f2l_{cd}}$: FSP composite process definition $\times$ variable environment $\rightarrow$ Exp.Open code

$$\mathsf{f2l_{cd}}(\left(\begin{array}{l} \texttt{||}P\texttt{(}x_1\texttt{=}V_1\texttt{,}\ldots\texttt{,}x_k\texttt{=}V_k\texttt{) = }C \\ \gg\{A_{p_1},\ldots,A_{p_n}\} \\ \backslash\{A_{h_1},\ldots,A_{h_q}\} \end{array}\right), E) = \left(\begin{array}{l} \textbf{total prio } \texttt{"ERROR"} > \textbf{all but } \texttt{"ERROR"} \textbf{ in} \\ \quad \textbf{total cut } \texttt{"exit"} \textbf{ in} \\ \quad\quad \textbf{total hide} \\ \quad\quad\quad \texttt{"EVENT !}l_1 \cdot \texttt{.*", ..., "EVENT !}l_m \cdot \texttt{.*"} \\ \quad\quad \textbf{in} \\ \quad\quad\quad \textbf{total prio} \\ \quad\quad\quad\quad \textbf{all but } \texttt{"EVENT !}l'_1\texttt{"},\ldots,\texttt{"EVENT !}l'_p\texttt{" >} \\ \quad\quad\quad\quad\quad \texttt{"EVENT !}l'_1\texttt{"},\ldots,\texttt{"EVENT !}l'_p\texttt{"} \\ \quad\quad\quad \textbf{in} \\ \quad\quad\quad\quad \mathsf{f2l_c}(C, \{x_1 \mapsto V_1, \ldots, x_k \mapsto V_k\} \cup E) \\ \quad\quad\quad \textbf{end prio} \\ \quad\quad \textbf{end hide} \\ \quad \textbf{end cut} \\ \textbf{end prio} \end{array}\right)$$

$$\text{where} \quad \{(l_i, E_i) \mid i \in 1..m\} = \bigcup_{i\in 1..q} \mathsf{expand}(A_{h_i}, E)$$
$$\{(l'_i, E'_i) \mid i \in 1..p\} = \bigcup_{i\in 1..n} \mathsf{expand}(A_{p_i}, E)$$

Note that the generated Exp.Open code contains regular expressions of the form "`EVENT !`$l_i \cdot$`.*`", where "`.*`" is the regular expression that matches any (possibly empty) sequence of labels. This implements the label prefix matching.

The definition of "$\mathsf{f2l_{cd}}$" details only the case of the "$\gg$" priority operator. The "$\ll$" operator is handled similarly, except that

$$\textbf{all but } \texttt{"EVENT !}l'_1\texttt{"},\ldots,\texttt{"EVENT !}l'_p\texttt{" > "EVENT !}l'_1\texttt{"},\ldots,\texttt{"EVENT !}l'_p\texttt{"}$$

is replaced by

$$\texttt{"EVENT !}l'_1\texttt{"},\ldots,\texttt{"EVENT !}l'_p\texttt{" > } \textbf{all but } \texttt{"EVENT !}l'_1\texttt{"},\ldots,\texttt{"EVENT !}l'_p\texttt{"}.$$

Function "$\mathsf{f2l_c}$", defined in Figure 10 (page 22), translates composite processes into Exp.Open code. It uses the auxiliary functions "alph" already defined, as well as "$\mathsf{vec_r}$", "$\mathsf{f2l_{pr}}$", and "$\mathsf{f2l_{rc}}$", which will be detailed below. The renaming rules of the form "`"EVENT !\(.*\)"` $\rightarrow$ `"EVENT !`$m \cdot$`\1"`" correspond to the prefixing of every label by m. The regular expression "`\1`" in the right-hand side stands for the sequence of characters matched by the regular expression located in the (first) occurrence of "`\(...\)`" in the left-hand side.

As already seen earlier, the error state is modeled as a sink state that contains a single (self-looping) transition labeled "`ERROR`". The priority given to action "`ERROR`" over all other actions in the generated Exp.Open code ensures that if a component is in the error state, then the whole specification is also in the error state since "`ERROR`" is the only action that can be executed.

The "**rename**" operator of Exp.Open allows many-to-one renaming, i.e., several labels may be renamed into the same label, but it does not allow one-to-many renaming, i.e., a single label may not be renamed into several labels, such as in FSP. However, one-to-many renaming can be implemented in Exp.Open using synchronisation vectors of length 1, i.e., of the form "$L_1 \rightarrow L_2$". Function "$\mathsf{vec_r}$", defined below, is used to generate such synchronisation vectors.

$\mathsf{f2l_c}$: FSP composite behaviour $\times$ variable environment $\rightarrow$ EXP.OPEN code

$$\mathsf{f2l_c}(P(V_1,\ldots,V_n),E) = \mathsf{f2l_{pr}}(P(V_1,\ldots,V_n),E)$$

$$\begin{aligned}\mathsf{f2l_c}(C_0/\{A'_1/A_1,\ldots,A'_n/A_n\},E) = {} & \textbf{label par } \mathsf{vec_r}(R,\mathsf{alph}(C_0,E)) \textbf{ in } \mathsf{f2l_c}(C_0,E) \textbf{ end par}\\ & \text{where } (\forall i\in 1..n)\ S_i=\{l \mid (l,E')\in \mathsf{expand}(A_i,E)\}\\ & \phantom{\text{where }} (\forall i\in 1..n)\ S'_i=\{l \mid (l,E')\in \mathsf{expand}(A'_i,E)\}\\ & \phantom{\text{where }} R=\textstyle\bigcup_{i\in 1..n} S_i \mapsto_d S'_i\end{aligned}$$

$$\begin{aligned}\mathsf{f2l_c}(\{A_1,\ldots,A_n\}::C_0,E) = {} & \textbf{label par } \mathsf{vec_r}(R,\mathsf{alph}(C_0,E)) \textbf{ in } \mathsf{f2l_c}(C_0,E) \textbf{ end par}\\ & \text{where } S=\textstyle\bigcup_{i\in 1..n}\{l \mid (l,E')\in \mathsf{expand}(A_i,E)\}\\ & \phantom{\text{where }} R=\{(l,l'l) \mid l\in \mathsf{alph}(C_0,E)\wedge l'\in S\}\end{aligned}$$

$$\mathsf{f2l_c}(\{A_1,\ldots,A_n\}:C_0,E) = \begin{cases}\left(\begin{array}{l}\textbf{total rename}\\ \quad \texttt{"EVENT !\textbackslash(.*\textbackslash)"} \rightarrow \texttt{"EVENT !}m_1\cdot\texttt{\textbackslash 1"}\\ \textbf{in}\\ \quad \mathsf{f2l_c}(C_0,E)\\ \textbf{end rename}\end{array}\right) & \text{if } p=1\\ \mathsf{f2l_c}(\{m_1\}:C_0 \;||\; \ldots \;||\; \{m_p\}:C_0,E) & \text{otherwise}\end{cases}$$

$$\text{where } \{m_i \mid i\in 1..p\} = \textstyle\bigcup_{i\in 1..n}\{l \mid (l,E')\in \mathsf{expand}(A_i,E)\}$$

$$\mathsf{f2l_c}(\textbf{if } V \textbf{ then } C_1 \textbf{ else } C_2,E) = \begin{cases}\mathsf{f2l_c}(C_1,E) & \text{if } \mathsf{f2l_e}(V,E)\\ \mathsf{f2l_c}(C_2,E) & \text{otherwise}\end{cases}$$

$$\mathsf{f2l_c}(C_1 \;||\; \ldots \;||\; C_n,E) = \left(\begin{array}{l}\textbf{label par } l_1,\ldots,l_q \textbf{ in}\\ \quad \mathsf{f2l_c}(C_1,E)\\ \quad ||\\ \quad \mathsf{f2l_c}(C_2 \;||\; \ldots \;||\; C_n,E)\\ \textbf{end par}\end{array}\right)$$

$$\text{where } \{l_i \mid i\in 1..q\} = \mathsf{alph}(C_1,E)\cap \mathsf{alph}(C_2 \;||\; \ldots \;||\; C_n,E)$$

$$\begin{aligned}\mathsf{f2l_c}(\textbf{forall } [x_1:L_1]\ldots[x_n:L_n]\ C_0,E) = {} & \mathsf{f2l_{rc}}(C_0,\{E\cup\{x_1\mapsto l_1,\ldots,x_n\mapsto l_n\} \mid (\forall i\in 1..n)\ l_i\in S_i\})\\ & \text{where } (\forall i\in 1..n)\ S_i=\{l_i \mid (l_i,E')\in \mathsf{expand}(L_i,E)\}\end{aligned}$$

Figure 10: Definition of function "$\mathsf{f2l_c}$"

$\mathsf{vec_r}$: relabeling × expanded label set → EXP.OPEN synchronisation vectors

$$\mathsf{vec_r}(R, S) \quad = \quad \texttt{"}l_1\texttt{"} \rightarrow \texttt{"}l_1'\texttt{"}\texttt{,} \ \ldots\ \texttt{,}\ \texttt{"}l_n\texttt{"} \rightarrow \texttt{"}l_n'\texttt{"}$$
$$\text{where } \{(l_i, l_i') \mid i \in 1..n\} = \{(l, l') \mid l \in S \ \wedge \ l' \in \mathsf{relabel}(l, R)\}$$

Function "$\mathsf{f2l_{pr}}$", defined below, translates a sequential or composite process call into EXP.OPEN code. In the case of a sequential process, the process call is replaced by a BCG graph corresponding to the LOTOS process obtained by translation of the sequential process with appropriate parameters, minimized modulo branching bisimulation to eliminate "**i**" transitions that are generated from the LOTOS sequential composition operator "`>>`", as explained in Section 3.1. This graph is obtained automatically by using the CÆSAR.ADT and CÆSAR compilers for LOTOS, and the BCG_MIN minimization tool, all available in CADP. In the case of a composite process, the process call is replaced by an EXP.OPEN expression that inlines the call to the composite process. Note that the translation terminates, because FSP composite processes are not recursive.

$\mathsf{f2l_{pr}}$: FSP process call × variable environment → EXP.OPEN code

$$\mathsf{f2l_{pr}}(P\texttt{(}V_1, \ldots, V_n\texttt{)}, E) \quad = \quad \begin{cases} \texttt{"}P'\texttt{.bcg"} & \text{if } P \text{ is a sequential process} \\ \mathsf{f2l_{cd}}(\hat{P}[V_1, \ldots, V_n], E) & \text{otherwise} \end{cases}$$

where P' is the LOTOS process defined by $\mathsf{f2l_{sd}}(\hat{P}[V_1, \ldots, V_n], E, ())$
and $\texttt{"}P'\texttt{.bcg"}$ is the BCG graph of P' minimized for branching bisimulation

Function "$\mathsf{f2l_{rc}}$", defined recursively below, is an auxiliary function used to translate **forall** processes.

$\mathsf{f2l_{rc}}$: FSP composite behaviour × variable environment set → EXP.OPEN code

$$\mathsf{f2l_{rc}}(C, \{E_0\}) = \mathsf{f2l_c}(C, E_0)$$

$$\mathsf{f2l_{rc}}(C, \{E_0, \ldots, E_{n+1}\}) = \left(\begin{array}{l} \textbf{label par } l_1, \ldots, l_q \textbf{ in} \\ \quad \mathsf{f2l_c}(C, E_0) \\ \quad \texttt{||} \\ \quad \mathsf{f2l_{rc}}(C, \{E_1, \ldots, E_{n+1}\}) \\ \textbf{end par} \end{array} \right)$$
$$\text{where } \{l_i \mid i \in 1..q\} = \mathsf{alph}(C, E_0) \cap \bigcup_{i \in 1..n+1} \mathsf{alph}(C, E_i)$$

4 Tool and Validation

We developed an automatic translator tool from FSP to LOTOS and EXP.OPEN, which is named FSP2LOTOS, implemented using the SYNTAX+TRAIAN compiler construction technology [GLM02]. It consists of about 5,000 lines of SYNTAX code, 20,000 lines of LOTOS NT code, and 600 lines of C code. The FSP2LOTOS tool consists of two parts:

- The *front-end* parses the input FSP program and builds an abstract syntax tree. The front-end was quite hard to implement, because the abstract grammar given in [MK06] is not directly implementable in SYNTAX, which needs LALR(1) grammars. Therefore, the grammar given in [MK06] was refined to a concrete LALR(1) grammar.

- The *back-end* translates the abstract syntax tree into code. It generates a LOTOS file containing the definition of sequential processes and an EXP.OPEN file semantically equivalent to the main process.

  In addition, FSP2LOTOS generates a verification script in the SVL language [GL01], which automates the generation of LTSs. In particular, this script generates (using the CÆSAR.ADT and CÆSAR compilers for LOTOS) and minimizes (using the BCG_MIN tool) the BCG graphs corresponding to FSP sequential processes composed in the main process.

We also developed a new shell-script named FSP.OPEN, which provides an interface between FSP specifications and the OPEN/CÆSAR application programming interface. FSP.OPEN encapsulates the full translation from FSP to EXP.OPEN using FSP2LOTOS and SVL, and a call to the EXP.OPEN tool on the generated network of automata. This allows OPEN/CÆSAR applications to be executed directly on FSP specifications. For instance, the evaluation of a temporal logic formula described in the file "`prop.mcl`" on the FSP specification described in the file "`spec.lts`" using the OPEN/CÆSAR application of CADP named EVALUATOR [MS03] can be done using the single command "`fsp.open spec.lts evaluator prop.mcl`".

We applied FSP2LOTOS and FSP.OPEN on a benchmark of FSP examples[2], which includes all examples provided with the LTSA distribution [MK06] (except features unsupported by FSP2LOTOS such as fluents and properties), as well as unitary tests that we wrote ourselves. It consists of 714 FSP files containing $2,781$ translatable processes. This represents $198,964$ FSP lexical tokens[3] in total. For the whole benchmark, FSP2LOTOS produced $1,097,497$ LOTOS lexical tokens, $169,247$ EXP.OPEN lexical tokens, and $137,682$ SVL lexical tokens. The explanations for these apparently large amount of code are the following:

- LOTOS generally allows a less concise style than FSP. For instance, it requires more keywords and gates have to be declared explicitly and passed as parameters to each process call.
- Although FSP variables are translated as often as possible into LOTOS variables, the translation may expand concise FSP labels into many LOTOS labels.

It is essential for the validity of verifications performed with CADP that the LOTOS/EXP.OPEN code obtained after translation has the same semantics as the source FSP specification. Therefore, we developed an automatic checker, which verifies that strong equivalence is preserved by the translation. The checker, illustrated in Figure 11 (page 25), works as follows:

- In a first step, the checker generates using LTSA (which is accessed in non-interactive mode) the LTS in ALDEBARAN format (file extension "`.aut`") corresponding to the main process of the source FSP specification. This LTS is then slightly transformed by the program AUT2CADP that we developed (255 lines of C code): the FSP error state is replaced by a sink state labeled by the "`ERROR`" symbol, and labels in FSP notations are converted into labels in CADP notations. The resulting LTS is then translated into the compact BCG (*Binary Coded Graph*) format of CADP (file extension "`.bcg`"), using the BCG_IO tool of CADP.
- In a second step, the checker generates the LOTOS, EXP.OPEN, and SVL files corresponding to the translation of the source FSP specification using FSP2LOTOS. The checker then calls the SVL tool of CADP to generate from these files a network of automata in the EXP.OPEN format, which corresponds semantically to the main process of the source FSP specification. Note that this network of automata includes renaming of the LOTOS labels (which are not written using the same conventions as LTSA) into the LTSA notation. This is an important feature that allows the FSP user to easily understand the behaviour of the translation into LOTOS.
- In a third step, the checker compares the BCG graph generated in the first step with the EXP.OPEN network of automata generated in the second step, modulo strong bisimulation. The comparison is performed automatically using the BISIMULATOR [BDJM05] on-the-fly equivalence checking tool of CADP, which responds by true or false and is even capable of producing a counter-example in case the graphs are not strongly bisimilar.

[2]We are looking forward to enriching our benchmark with additional examples. Examples may be sent to `cadp@inrialpes.fr`.

[3]A lexical token is either a keyword, a symbol, or an identifier of the considered language. Comments are excluded. Measuring code size in lexical tokens is more fair than in number of characters or number of lines, which depend on non-significant factors such as indenting style or identifier conventions.

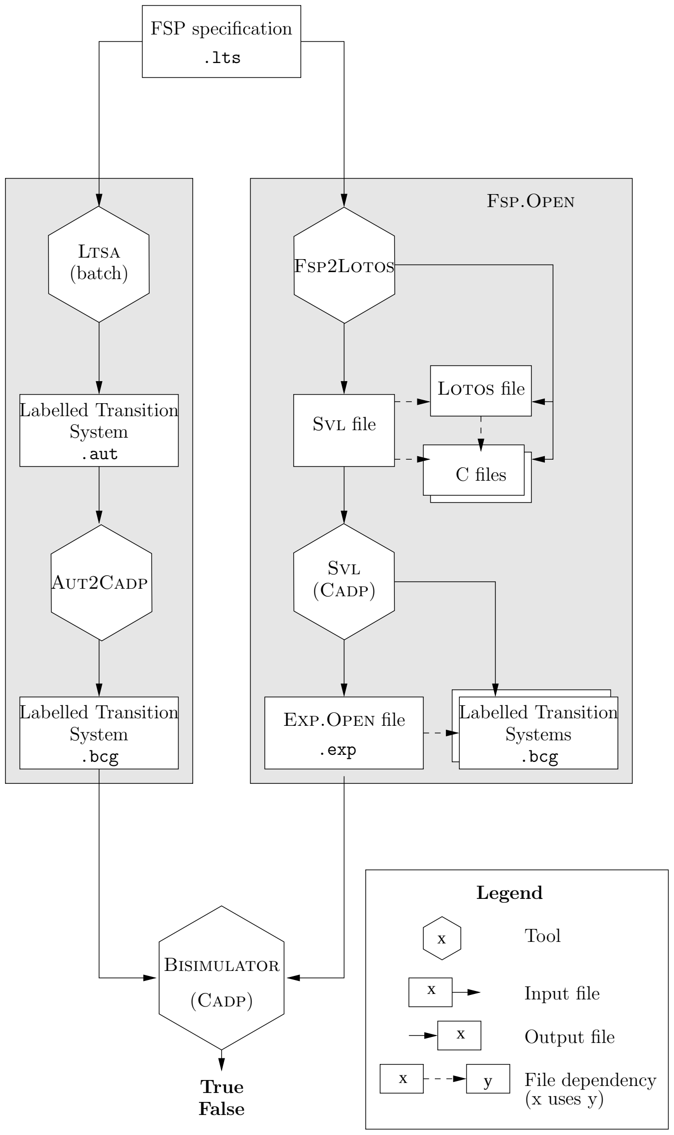


Figure 11: Principles of the automated checker used to validate FSP2LOTOS.

We validated the Fsp2Lotos translator using the aforementioned automated checker on all the examples of our database, and for each example the checker returned the answer *true*, thus showing that both specifications (before and after translation) are strongly equivalent.

So far, the largest FSP specification processed using Fsp2Lotos had 1,658 FSP tokens (213 lines), which is already quite large (although not huge) given the conciseness of the FSP language. The code generated from this specification consists of 389 Svl tokens, 984 Exp.Open tokens, and 7,729 Lotos tokens. These numbers are far below the code sizes already processed by the Cadp tools, which has been used to verify specifications consisting of thousands of lines of code[4].

Although the translation rules implemented in our tool have been formally defined, we cannot claim that they have been formally proven. Given that FSP, Lotos, and Exp.Open are based upon the same Lts semantic model, defined using structural operational semantic rules, and that strong bisimulation is a congruence for all operators, a formal proof would consist in showing, using a structural induction hypothesis, that for each rule, the FSP process in the left-hand side of the rule yields the same Lts transitions as the Lotos or Exp.Open process in the right-hand side, modulo the different encodings of labels and error state. Doing this using a theorem prover such as, e.g., Pvs would probably not raise much technical difficulty. However, due to the size and number of (source and target) languages involved, encoding the translation scheme and underlying theory (languages, semantics, and bisimulations) in such a theorem prover is itself a quite manpower consuming task that we have not considered as a priority so far.

5 Application

In this section, we present several refinements of an FSP specification of a semaphore. We show how Cadp can be used in complement to Ltsa, using the translation from FSP to Lotos and Exp.Open.

The starting point is the FSP specification of the semaphore given in Example 2 (page 7), whose corresponding graph (Figure 4, page 8) has 7 states and 9 transitions.

A first refinement is to extend the number of resources ("`{1,2,3}`" in addition to "`{a,b,c}`") accessed in mutual exclusion, as well as the number of accesses, leading to the following specification "`SEMADEMO1`":

```
||SEMADEMO1 = (
                  {a,b,c}:ACCESS
               || {a,b,c,[1..3]}::mutex:SEMAPHORE(1)
               || [1..3]:ACCESS
               ).
```

For "`SEMADEMO1`", Ltsa generates a graph with 13 states and 18 transitions.

The next refinement aims at duplicating both semaphores so that each semaphore is in charge of a single resource. This leads to the following specification "`SEMADEMO2`":

```
||SEMADEMO2 = (
                  {a,b,c}:ACCESS
               || {a,b,c}::mutex:SEMAPHORE(1)
               || [1..3]::mutex:SEMAPHORE(1)
               || [1..3]:ACCESS
               ).
```

For "`SEMADEMO2`", Ltsa generates a graph with 49 states and 126 transitions, which is difficult to analyse visually, in particular because all the transitions between resources "`{a,b,c}`" and "`{1,2,3}`" are interleaved.

The last refinement defines the specification as a composition of two composite processes being dedicated to one resource. This leads to the following specification "`SEMADEMO3`":

```
||C_P = ({a,b,c}:ACCESS || {a,b,c}::mutex:SEMAPHORE(1)).
||C_Q = ([1..3]:ACCESS || [1..3]::mutex:SEMAPHORE(1)).
||SEMADEMO3 = (C_P || C_Q).
```

[4]See for instance the list of case studies done using Cadp at `http://www.inrialpes.fr/vasy/cadp/case-studies`.

For “SEMADEMO3”, LTSA generates a graph which has the same size as “SEMADEMO2”. However, it is impossible to check using LTSA that “SEMADEMO2” and “SEMADEMO3” are equivalent. Instead, the translation to LOTOS/EXP.OPEN allows the BISIMULATOR [BDJM05] tool of CADP to be used to verify that, indeed, “SEMADEMO2” and “SEMADEMO3” are strongly equivalent.

The following code is an excerpt of the LOTOS code generated by FSP2LOTOS. We only show here the code generated for the “SEMAPHORE” and “SEMA” sequential processes. Additional code (of similar size) is generated for the “ACCESS” sequential process. Note that label concatenation “·” used in Section 3.2 is implemented using the “CONS” and “NIL” list constructor operations, which are defined using LOTOS abstract data types.

```
process SEMAPHORE [EVENT, TAU, ERROR] (N:Int): noexit :=
   SEMA [EVENT, TAU, ERROR] (N)
where
   process SEMA [EVENT, TAU, ERROR] (N:Int): noexit :=
      EVENT !CONS (UP, NIL);
         SEMA [EVENT, TAU, ERROR] (N + POS(1))
      []
      [V > POS(0)] -> EVENT !CONS (DOWN, NIL);
         SEMA [EVENT, TAU, ERROR] (N - POS(1))
   endproc
endproc
```

The following is an excerpt of the EXP.OPEN code generated by FSP2LOTOS (comments were added by hand). We only show here the code generated for the “C_P” composite process. Similar code (same size) is generated for “C_Q”. To save space, we have replaced this code by dots at the end of the following excerpt.

```
total prio "ERROR" > all but "ERROR" in
  (*
   * this part of the EXP.OPEN code corresponds to
   * C_P = ({a, b, c}:ACCESS || {a, b, c}::mutex:SEMAPHORE)
   *)
  label par
    "EVENT !CONS (A, CONS (MUTEX, CONS (DOWN, NIL)))",
    "EVENT !CONS (A, CONS (MUTEX, CONS (UP, NIL)))",
    "EVENT !CONS (B, CONS (MUTEX, CONS (DOWN, NIL)))",
    "EVENT !CONS (B, CONS (MUTEX, CONS (UP, NIL)))",
    "EVENT !CONS (C, CONS (MUTEX, CONS (DOWN, NIL)))",
    "EVENT !CONS (C, CONS (MUTEX, CONS (UP, NIL)))"
  in
    (* {a, b, c}:ACCESS *)
    (
      (* a:ACCESS *)
      total rename "EVENT !\(.*\)" -> "EVENT !CONS (A, \1)" in
        "ACCESS.bcg"
      end rename
      |||
      (* b:ACCESS *)
      total rename "EVENT !\(.*\)" -> "EVENT !CONS (B, \1)" in
        "ACCESS.bcg"
      end rename
      |||
      (* c:ACCESS *)
      total rename "EVENT !\(.*\)" -> "EVENT !CONS (C, \1)" in
        "ACCESS.bcg"
      end rename
    )
    ||
    (* {a, b, c}::mutex:SEMAPHORE *)
    label par
      "EVENT !CONS (MUTEX, CONS (UP, NIL))" ->
              "EVENT !CONS (A, CONS (MUTEX, CONS (UP, NIL)))",
      "EVENT !CONS (MUTEX, CONS (UP, NIL))" ->
              "EVENT !CONS (B, CONS (MUTEX, CONS (UP, NIL)))",
      "EVENT !CONS (MUTEX, CONS (UP, NIL))" ->
              "EVENT !CONS (C, CONS (MUTEX, CONS (UP, NIL)))",
      "EVENT !CONS (MUTEX, CONS (DOWN, NIL))" ->
              "EVENT !CONS (A, CONS (MUTEX, CONS (DOWN, NIL)))",
      "EVENT !CONS (MUTEX, CONS (DOWN, NIL))" ->
              "EVENT !CONS (B, CONS (MUTEX, CONS (DOWN, NIL)))",
      "EVENT !CONS (MUTEX, CONS (DOWN, NIL))" ->
              "EVENT !CONS (C, CONS (MUTEX, CONS (DOWN, NIL)))",
      "ERROR" -> "ERROR" in
        (* mutex:SEMAPHORE *)
        total rename "EVENT !\(.*\)" -> "EVENT !CONS (MUTEX, \1)" in
          "SEMAPHORE.bcg"
        end rename
    end par
  end par
  |||
  (*
   * the part corresponding to C_Q is similar as above,
   * with A, B, C replaced by POS (1), POS (2), and POS (3)
   *)
  ...
end prio
```

This example illustrates the use of equivalence checking of FSP specifications, but other verification techniques available in Cadp to tackle the state explosion problem, such as distributed, compositional, or on-the-fly verification, can be used to complement Ltsa validation. For instance, one can use the Evaluator [MS03] model checker of Cadp to verify μ-calculus formulas on-the-fly. The counterexamples provided by Cadp are easy to translate back automatically into FSP format, using the label renaming facilities available in Cadp.

6 Related Work

Several works aimed at combining the theories and notations of CSP, ACP, and CCS [HLP81, AZ81, Bro83, Mil87, HH06]. The long-term goal of these papers is to unifying theories of concurrent programming, and accordingly they focus on theoretical aspects of the aforementioned process calculi. As an example, in [HH06], the authors consider CCS and CSP and formalise a set of links between common parts of CCS and CSP theories. Codifying the similarities between their respective theories enables them to be used in combination while preserving their benificial differences. Our objective is different since we consider calculi equipped with verification tools, and propose a solution to allow the joint use of existing tools.

As regards high-level translations between process algebras, several proposals have been made in the hardware area [SS05, WKTZ05]. In [SS05], the authors propose a translation from the hardware process algebra Chp to Lotos. Thus, it makes possible to verify Chp designs of asynchronous circuits and architectures using the Cadp toolbox. This approach was applied in practice for the verification of an Asynchronous Network-on-Chip architecture [SSTV07]. [WKTZ05] starts with a Balsa description of circuits, and sketches a translation from Balsa to Csp in order to verify Balsa programs using Fdr.

Two other initiatives consider Lotos as target language of high-level language encodings. In the framework of the French national project TopCased, which gathers numerous industrial (Airbus, Thales, CS-SI, ...) and academic partners (Inria, Cnrs, Toulouse Universities, ...), a new language named Fiacre has been designed as an intermediate model between high-level models and verification toolboxes such as Tina and Cadp. The connection to Cadp has led to a translator from Fiacre into Lotos named Flac [BBF$^+$08, BGLV08]. Also a translator to Lotos from a variant of E-Lotos [ISO01] named Lotos-NT [Sig04] is under construction at Inria/Vasy.

Another group of related works concerns those advocating the encoding of process calculi (mainly Acp, Ccs, Csp and their dialects) into higher-order logics, inputs of theorem provers such as Hol, Pvs, Isabelle [Nes99, DS97, TW97, BH99] or into the B method [But00], motivated by the availability of formal verification support for the target formalism. Theorem proving is a means to fight against the state explosion problem and to deal with infinite automata, but is not suitable to prove temporal properties. Instead, we focused on model checking because it makes verification steps easier (especially for non-expert users) thanks to a full automation and its adequacy to automata-based models.

In [CMS95], the authors present an alternative solution to translation approaches to verify process algebras. The Pac (Process Algebra Compiler) is a front-end generator for process-algebra-based verification tools. It produces routines for parsing and unparsing programs being given a description of the syntax and semantics of a language. Thus, the Pac provides a useful tool for expanding the repertoire of languages that tools can support. The current prototype only includes a back-end for the Concurrency Workbench (Cwb).

Another way to use Cadp to verify Fsp specifications would have been to use a lower-level translation from Fsp to an intermediate language such as the one advocated in the If toolset [BGM02]. If is built upon a specification language based on communicating extended timed automata. So far, the If toolset is mainly connected to high-level modelling languages such as Sdl and Uml. Several validation tools have been developed and connected (mainly Cadp) to analyse If descriptions. We preferred to connect Fsp with Lotos and Exp.Open because it avoids the state explosion that a lower-level encoding might induce.

Other proposals and initiatives have emerged to favour a joint use of verification tools: Rushby and his colleagues [Rus06] propose a joint use of an SMT (Satisfiability Modulo Theories) solver, model checking techniques, and theorem proving. A similar work [FMM$^+$06] focuses as well on a combination of SMT Solvers and Interactive Proof Assistants. Last, let us emphasize the Fmics-Jeti initiative [MNS05] (*Electronic Tool Integration Platform*) which aims at facilitating access to a managed collection of analysis tools.

At last, a preliminary version of this work has been published in [SKLM07]. The current article contains the following updates and additions:

- A related work section has been written and integrated.
- The conference paper contained only excerpts of the translation rules, whereas the current article presents all translation rules in more details.
- Some translation rules have been simplified when possible, so that smaller Lotos/Exp.Open code is generated. In addition, the translation process now preserves a strong equivalence relation (instead of branching equivalence) between the source FSP specification and the target Lotos/Exp.Open code.
- We have enhanced our validation procedure, which now checks automatically that the graphs generated using Fsp2Lotos and Cadp are equivalent to those generated using Ltsa.

7 Concluding Remarks

The motivation of this work was to reduce the gap between existing tool support for process calculi. We chose here the process calculus FSP and the Lotos international standard. We proposed a translation from FSP to Lotos and Exp.Open to make the joint use of Ltsa and Cadp possible for FSP users. The translation is completely automated, and implemented within the Fsp2Lotos and Fsp.Open tools, which we validated on many examples using formal verification tools of Cadp, such as the Bisimulator [BDJM05] Lts equivalence checker. Fsp2Lotos has been distributed within Cadp since beta-version 2007-p (January 2009) and Fsp.Open since beta-version 2008-d (July 2009).

As regards the lessons learnt from our experience in making gateways between formalisms and tools, we think that supporting a high-level encoding between process algebra is a good solution as these languages are based on the same kernel of operators, which makes the translation rather straightforward for most of them. Lotos is an appropriate target calculus because, beyond the numerous validation and verification tools available, it contains various behavioural operators that can be freely combined, but also has an expressive notation to describe abstract data types, the presence of which is sometimes essential to ensure a correct encoding. In [SS05], the authors managed to encode all the operators of the hardware process algebra Chp into Lotos.

However, each process algebra comes with its own specificities and subtleties that may make the high-level translation of all the operators difficult. For instance, in the case of FSP, priorities and the label prefix matching semantics of hiding and relabeling cannot be easily translated into Lotos, which prevented us to benefit from Lotos composition operators. To ameliorate this, an automata-based language such as Exp.Open can be used in order to complement the process algebra translation by providing a large number of parallel composition, hiding, relabeling, and priority operators, among others. When a pure process algebra translation is not possible, a mixed translation targeting both a process algebra and an automata-based language may therefore be an adequate solution to encode the whole expressiveness of a calculus.

A perspective of this work is to apply our approach on complex systems, for instance on Web service models described first in Bpel [A+05] or Ws-Cdl [KBR], and then automatically translated into FSP for analysis purposes [FUMK05]. In this case, the interaction of services can involve huge underlying state spaces, which require efficient generation and minimisation tools such as those available in Cadp. Moreover, the equivalence checking tool available in Cadp can help in Web services to ensure that an abstract specification of a problem and its solution described as a composition of services are formally equivalent [SBS06]. Another perspective is to take FSP safety and progress properties into account, and to translate them into regular alternation-free μ-calculus formulas, which can be checked using the Evaluator [MS03] on-the-fly model checker of Cadp.

Acknowledgements

The authors warmly thank Hubert Garavel (head of the Inria/Vasy project-team) for suggesting this work, and for his constant support and encouragements. They are also grateful to Wendelin Serwe (Inria/Vasy) for his valuable help on technical aspects during the implementation of the translator.

References

[A+05] T. Andrews et al. *Business Process Execution Language for Web Services (WSBPEL)*. BEA Systems, IBM, Microsoft, SAP AG, and Siebel Systems, February 2005.

[AZ81] Egidio Astesiano and Elena Zucca. Semantics of CSP via Translation into CCS. In *Proc. of the 10th International Symposium on Mathematical Foundations of Computer Science (MFCS'81)*, volume 118 of *Lecture Notes in Computer Science*, pages 172–182. Springer, 1981.

[BBF+08] Bernard Berthomieu, Jean-Paul Bodeveix, Patrick Farail, Mamoun Filali, Hubert Garavel, Pierre Gaufillet, Frédéric Lang, and François Vernadat. FIACRE: an Intermediate Language for Model Verification in the TOPCASED Environment. In Jean-Claude Laprie, editor, *Proceedings of the 4th European Congress on Embedded Real-Time Software ERTS'08 (Toulouse, France)*. SIA (the French Society of Automobile Engineers), AAAF (the French Society of Aeronautic and Aerospace), and SEE (the French Society for Electricity, Electronics, and Information & Communication Technologies), January 2008.

[BDJM05] Damien Bergamini, Nicolas Descoubes, Christophe Joubert, and Radu Mateescu. BISIMULATOR: A Modular Tool for On-the-Fly Equivalence Checking. In Nicolas Halbwachs and Lenore Zuck, editors, *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2005 (Edinburgh, Scotland, UK)*, volume 3440 of *Lecture Notes in Computer Science*, pages 581–585. Springer Verlag, April 2005.

[BFG$^+$91] Ahmed Bouajjani, Jean-Claude Fernandez, Susanne Graf, Carlos Rodríguez, and Joseph Sifakis. Safety for Branching Time Semantics. In *Proceedings of 18th ICALP*. Springer Verlag, July 1991.

[BGLV08] Bernard Berthomieu, Hubert Garavel, Frédéric Lang, and François Vernadat. Verifying Dynamic Properties of Industrial Critical Systems Using TOPCASED/FIACRE. *ERCIM News*, 75:32–33, October 2008.

[BGM02] Marius Bozga, Susanne Graf, and Laurent Mounier. IF-2.0: A Validation Environment for Component-Based Real-Time Systems. In Kim G. Larsen and Ed Brinksma, editors, *Proceedings of the Conference on Computer-Aided Verification CAV'2002 (Copenhagen, Denmark)*, volume 2404 of *Lecture Notes in Computer Science*. Springer Verlag, July 2002.

[BH99] T. Basten and J. Hooman. Process Algebra in Pvs. In *Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'99 (Amsterdam, The Netherlands)*, volume 1579 of *Lecture Notes in Computer Science*, pages 270–284. Springer Verlag, 1999.

[BHR84] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A Theory of Communicating Sequential Processes. *Journal of the ACM*, 31(3):560–599, July 1984.

[BO05] Stefan Blom and Simona Orzan. Distributed state space minimization. *International Journal on Software Tools for Technology Transfer*, 7(3):80–291, 2005.

[Bro83] Stephen D. Brookes. On the Relationship of CCS and CSP. In *Proc. of the 10th Colloquium Automata, Languages and Programming (ICALP'83)*, volume 154 of *Lecture Notes in Computer Science*, pages 83–96. Springer, 1983.

[But00] M. Butler. Csp2B: A Practical Approach to Combining Csp and B. *Formal Aspects of Computing*, 12(3):182–198, 2000.

[CMS95] Rance Cleaveland, Eric Madelaine, and Steve Sims. A Front-End Generator for Verification Tools. In Uffe H. Engberg, Kim G. Larsen, and Arne Skou, editors, *Proceedings of TACAS'95 Tools and Algorithms for the Construction and Analysis of Systems (Aarhus, Denmark)*, May 1995. Also available as INRIA Research Report RR-2612.

[DS97] B. Dutertre and S. Schneider. Using a PVS Embedding of CSP to Verify Authentication Protocols. In *Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics TPHOLs'97 (Murray Hill, NJ, USA)*, volume 1275 of *Lecture Notes in Computer Science*, pages 121–136. Springer Verlag, 1997.

[FMM$^+$06] Pascal Fontaine, Jean-Yves Marion, Stephan Merz, Leonor Prensa Nieto, and Alwen Fernanto Tiu. Expressiveness + Automation + Soundness: Towards Combining SMT Solvers and Interactive Proof Assistants. In *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'06 (Vienna, Austria)*, volume 3920 of *Lecture Notes in Computer Science*, pages 167–181. Springer Verlag, 2006.

[FUMK05] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Tool Support for Model-Based Engineering of Web Service Compositions. In *Proceedings of the IEEE International Conference on Web Services ICWS'05*, pages 95–101. IEEE Computer Society, 2005.

[Gar89a] Hubert Garavel. *Compilation et vérification de programmes LOTOS*. Thèse de Doctorat, Université Joseph Fourier (Grenoble), November 1989.

[Gar89b] Hubert Garavel. Compilation of LOTOS Abstract Data Types. In Son T. Vuong, editor, *Proceedings of the 2nd International Conference on Formal Description Techniques FORTE'89 (Vancouver B.C., Canada)*, pages 147–162. North-Holland, December 1989.

[Gar90] Hubert Garavel. CÆSAR Reference Manual. Rapport SPECTRE C18, Laboratoire de Génie Informatique — Institut IMAG, Grenoble, November 1990.

[Gar98] Hubert Garavel. OPEN/CÆSAR: An Open Software Architecture for Verification, Simulation, and Testing. In Bernhard Steffen, editor, *Proceedings of the First International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'98 (Lisbon, Portugal)*, volume 1384 of *Lecture Notes in Computer Science*, pages 68–84, Berlin, March 1998. Springer Verlag. Full version available as INRIA Research Report RR-3352.

[GL01] Hubert Garavel and Frédéric Lang. SVL: a Scripting Language for Compositional Verification. In Myungchul Kim, Byoungmoon Chin, Sungwon Kang, and Danhyung Lee, editors, *Proceedings of the 21st IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems FORTE'2001 (Cheju Island, Korea)*, pages 377–392. IFIP, Kluwer Academic Publishers, August 2001. Full version available as INRIA Research Report RR-4223.

[GLM02] Hubert Garavel, Frédéric Lang, and Radu Mateescu. Compiler Construction using LOTOS NT. In Nigel Horspool, editor, *Proceedings of the 11th International Conference on Compiler Construction CC 2002 (Grenoble, France)*, volume 2304 of *Lecture Notes in Computer Science*, pages 9–13. Springer Verlag, April 2002.

[GLMS07] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes. In Werner Damm and Holger Hermanns, editors, *Proceedings of the 19th International Conference on Computer Aided Verification CAV'2007 (Berlin, Germany)*, volume 4590 of *Lecture Notes in Computer Science*, pages 158–163. Springer Verlag, July 2007.

[GS06] Hubert Garavel and Wendelin Serwe. State Space Reduction for Process Algebra Specifications. *Theoretical Computer Science*, 351(2):131–145, February 2006.

[GV90] Jan Friso Groote and Frits Vaandrager. An Efficient Algorithm for Branching Bisimulation and Stuttering Equivalence. In M. S. Patterson, editor, *Proceedings of the 17th ICALP (Warwick)*, volume 443 of *Lecture Notes in Computer Science*, pages 626–638. Springer Verlag, 1990.

[HH06] Jifeng He and C. A. R. Hoare. CSP Is a Retract of CCS. In *Proc. of the First International Symposium on Unifying Theories of Programming (UTP'06)*, volume 4010 of *Lecture Notes in Computer Science*, pages 38–62. Springer, 2006.

[HLP81] Matthew Hennessy, W. Li, and Gordon D. Plotkin. A First Attempt at Translating CSP into CCS. In *Proc. of the 2nd International Conference on Distributed Computing Systems (ICDCS'81)*, pages 105–115. IEEE Computer Society, 1981.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[ISO89] ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève, September 1989.

[ISO01] ISO/IEC. Enhancements to LOTOS (E-LOTOS). International Standard 15437:2001, International Organization for Standardization — Information Technology, Genève, September 2001.

[KBR] N. Kavantzas, D. Burdett, and G. Ritzinger. *Web Services Choreography Description Language 1.0.* W3C. W3C Working Draft 27 April 2004.

[KS90] P. C. Kanellakis and S. A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Information and Computation*, 86(1):43–68, May 1990.

[Lan02] Frédéric Lang. Compositional Verification using SVL Scripts. In Joost-Pieter Katoen and Perdita Stevens, editors, *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2002 (Grenoble, France)*, volume 2280 of *Lecture Notes in Computer Science*, pages 465–469. Springer Verlag, April 2002.

[Lan05] Frédéric Lang. EXP.OPEN 2.0: A Flexible Tool Integrating Partial Order, Compositional, and On-the-fly Verification Methods. In Jaco van de Pol, Judi Romijn, and Graeme Smith, editors, *Proceedings of the 5th International Conference on Integrated Formal Methods IFM'2005 (Eindhoven, The Netherlands)*, volume 3771 of *Lecture Notes in Computer Science*, pages 70–88. Springer Verlag, November 2005. Full version available as INRIA Research Report RR-5673.

[Lan06] Frédéric Lang. Refined Interfaces for Compositional Verification. In Elie Najm, Jean-François Pradat-Peyre, and Véronique Viguié Donzeau-Gouge, editors, *Proceedings of the 26th IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems FORTE'2006 (Paris, France)*, volume 4229 of *Lecture Notes in Computer Science*, pages 159–174. Springer Verlag, September 2006. Full version available as INRIA Research Report RR-5996.

[Mag99] J. Magee. Behavioral Analysis of Software Architectures Using LTSA. In *Proceedings of the 21st International Conference on Software Engineering ICSE'99*, pages 634–637. ACM Press, 1999.

[MDEK95] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In *Proceedings of the 5th European Software Engineering Conference ESEC'95 (Sitges, Spain)*, volume 989 of *Lecture Notes in Computer Science*, pages 137–153. Springer Verlag, 1995.

[Mil87] M. Millington. *Theories of Translation Corrections for Concurrent Programming Languages*. PhD thesis, LFCS, School of Informatics, University of Edinburgh, 1987.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[MK99] Jeff Magee and Jeff Kramer. *Concurrency: State Models and Java Programs*. Wiley, 1999.

[MK06] Jeff Magee and Jeff Kramer. *Concurrency: State Models and Java Programs*. Wiley, 2006 edition, April 2006.

[MNS05] Tiziana Margaria, Ralf Nagel, and Bernhard Steffen. Remote Integration and Coordination of Verification Tools in JETI. In *Proceedings of the 12th IEEE International Conference on the Engineering of Computer-Based Systems ECBS'05 (Greenbelt, MD, USA)*, pages 431–436. IEEE Computer Society, 2005.

[MS03] Radu Mateescu and Mihaela Sighireanu. Efficient On-the-Fly Model-Checking for Regular Alternation-Free Mu-Calculus. *Science of Computer Programming*, 46(3):255–281, March 2003.

[Nes99] M. Nesi. Formalising a Value-Passing Calculus in HOL. *Formal Aspects of Computing*, 11(2):160–199, 1999.

[Par81] David Park. Concurrency and Automata on Infinite Sequences. In Peter Deussen, editor, *Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer Verlag, March 1981.

[PT87] Robert Paige and Robert E. Tarjan. Three Partition Refinement Algorithms. *SIAM Journal of Computing*, 16(6):973–989, December 1987.

[Rus06] John M. Rushby. Tutorial: Automated Formal Methods with PVS, SAL, and Yices. In *Proceedings of the 4th IEEE International Conference on Software Engineering and Formal Methods SEFM'06 (Pune, India)*, page 262. IEEE Computer Society, 2006.

[SBS06] Gwen Salaün, Lucas Bordeaux, and Marco Schaerf. Describing and Reasoning on Web Services using Process Algebra. *International Journal of Business Process Integration and Management*, 1(2):116–128, 2006.

[Sig04] Mihaela Sighireanu. LOTOS NT User's Manual (Version 2.4). INRIA projet VASY. `ftp://ftp.inrialpes.fr/pub/vasy/traian/manual.ps.Z`, June 2004.

[SKLM07] Gwen Salaün, Jeff Kramer, Frédéric Lang, and Jeff Magee. Translating FSP into LOTOS and Networks of Automata. In Jim Davies, Wolfram Schulte, and Jin Song Dong, editors, *Proceedings of the 6th International Conference on Integrated Formal Methods IFM'2007 (Oxford, United Kingdom)*, volume 4591 of *Lecture Notes in Computer Science*, pages 558–578. Springer Verlag, July 2007.

[SS05] Gwen Salaün and Wendelin Serwe. Translating Hardware Process Algebras into Standard Process Algebras — Illustration with CHP and LOTOS. In Jaco van de Pol, Judi Romijn, and Graeme Smith, editors, *Proceedings of the 5th International Conference on Integrated Formal Methods IFM'2005 (Eindhoven, The Netherlands)*, volume 3771 of *Lecture Notes in Computer Science*. Springer Verlag, November 2005. Full version available as INRIA Research Report RR-5666.

[SSTV07] Gwen Salaün, Wendelin Serwe, Yvain Thonnart, and Pascal Vivet. Formal Verification of CHP Specifications with CADP — Illustration on an Asynchronous Network-on-Chip. In Peter Beerel, Marly Roncken, Mark Greenstreet, and Montek Singh, editors, *Proceedings of the 13th IEEE International Symposium on Asynchronous Circuits and Systems ASYNC 2007 (Berkeley, California, USA)*, pages 73–82. IEEE Computer Society Press, March 2007.

[TW97] H. Tej and B. Wolff. A Corrected Failure-Divergence Model for Csp in Isabelle/Hol. In *Proceedings of the 4th International Symposium of Formal Methods Europe FME'97 (Graz, Austria)*, volume 1313 of *Lecture Notes in Computer Science*, pages 318–337. Springer Verlag, 1997.

[vGW89] R. J. van Glabbeek and W. P. Weijland. Branching-Time and Abstraction in Bisimulation Semantics (extended abstract). CS R8911, Centrum voor Wiskunde en Informatica, Amsterdam, 1989. Also in proc. IFIP 11th World Computer Congress, San Francisco, 1989.

[WKTZ05] X. Wang, M. Z. Kwiatkowska, G. K. Theodoropoulos, and Q. Zhang. Towards a Unifying Csp approach to Hierarchical Verification of Asynchronous Hardware. In *Procedings of the 4th International Workshop on Automated Verification of Critical Systems AVoCS'04 (London, UK)*, volume 128 of *Electronic Notes in Theoretical Computer Science (ENTCS) series*, pages 231–246, 2005.