Translating FSP into LOTOS and Networks of Automata

Gwen Salaun, Jeff Kramer, Frederic Lang, Jeff Magee

Translating FSP into LOTOS and Networks of Automata

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

[1] INRIA, Centre de Recherche Rhône-Alpes / VASY, Montbonnot, France
Frederic.Lang@inria.fr ***
[2] Imperial College, London, UK
{j.kramer,j.magee}@imperial.ac.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 from FSP to LOTOS. Since FSP composite processes are hard to encode into 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 we implemented.

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. At the same time, 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 is no connection between calculi and 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 [16] is a widely-used and expressive process calculus conceived to make the writing of specifications easier and concise. FSP is supported by LTSA, a user-friendly tool which allows to compile FSP specifications into finite state machines known as LTSs (Labelled Transition Systems), to visualise and animate LTSs through graphical interfaces, and to verify LTL properties. On the other hand, LOTOS is an ISO standard [13],

*** G. Salaün currently works at Universidad de Málaga, Spain (salaun@lcc.uma.es).

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 [9], a verification toolbox for asynchronous concurrent systems. CADP allows to deal with very large state spaces, 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 very accessible to everyone, whereas LOTOS requires a better level of expertise. In addition, CADP is a rich and efficient verification toolbox which can complement basic analysis possible with LTSA. We propose to translate FSP specifications into LOTOS to enable FSP users to access the verification means 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 the EXP format which is another input format of CADP. EXP allows the description of networks of automata using parallel composition, 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 extend 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 [7] for instance) because (i) we preferred to keep the expressiveness of the specification and then make the translation of most behavioural operators easier, (ii) high-level models are necessary to use some verification techniques available in CADP, such as compositional verification, (iii) verification of the generated LOTOS code can benefit from the numerous optimisations implemented in the CÆSAR.ADT and CÆSAR [6, 12, 11] compilers of LOTOS available in CADP, which would be too expensive to re-implement at the FSP level.

The translation from FSP to LOTOS/EXP is completely automated in a translator tool we implemented (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 a branching equivalence relation [22] which is stronger than observational equivalence.

The remainder of this paper is organised as follows. Section 2 gives short introductions to FSP, LOTOS, and EXP. Section 3 presents some preliminary definitions that are used in the remainder of the paper. Sections 4 and 5 describe respectively the translation of FSP sequential processes into LOTOS and of FSP composite processes into EXP. In Section 6, we present our tool and its validation. Section 7 illustrates how LTSA and CADP can be used jointly on a simple system. Section 8 ends with some concluding remarks. More details about this work are given in an INRIA technical report [15].

2 FSP, LOTOS, and EXP

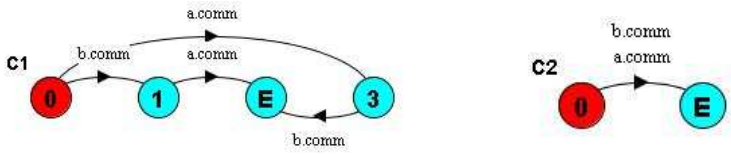
We start with a short description of FSP (Finite State Processes, see [16] for a more complete presentation of the language). FSP can define (i) constants, ranges, and sets, (ii) basic (*i.e.*, sequential) and composite processes, (iii) safety and progress properties (not handled in this work). In the grammar of FSP basic processes below, an upper case identifier P refers to a process identifier, X (or x) is a variable, *act* is a string label, and V is an expression involving arithmetic, comparison, and logical operators with variables. We discard in this paper local process definitions as well as the visibility operateur "`@`", dual of the hiding operator, although the full expressiveness of FSP is taken into account in the translator we implemented (see Section 6).

P_B ::=	$P\,\texttt{(}X_1\texttt{=}V_1, \ldots, X_k\texttt{=}V_k\texttt{)}\ \texttt{=}\ B$	*process definition*
	$+\{A_{e1}, \ldots, A_{en}\}$	*alphabet extension*
	$/\{A'_{r1}/A_{r1}, \ldots, A'_{rn}/A_{rn}\} \setminus \{A_{h1}, \ldots, A_{hn}\}$	*relabel + hide*
B ::=	`stop` $\mid$ `end` $\mid$ `error` $\mid P(V_1, \ldots, V_n)$	*terminations + process call*
	$\mid$ `if` V `then` B_1 `else` B_2	*if-then-else structure*
	$\mid$ `when` V_1 $S_1 \rightarrow B_1$ `|` $\ldots$ `|` `when` V_n $S_n \rightarrow B_n$	*choice*
	$\mid P_1\texttt{(}V_{11}, \ldots, V_{1k}\texttt{)}\texttt{;} \ldots \texttt{;} P_n\texttt{(}V_{n1}, \ldots, V_{nl}\texttt{)}\texttt{;} B$	*sequential composition*
S ::=	$A_1 \rightarrow \ldots \rightarrow A_k$	*sequence of labels*
A ::=	$L_1 \ldots L_n$	*label*
L ::=	*act*	*action*
	$\mid V \mid x : V$	*expression*
	$\mid \{A_1, \ldots, A_n\} \mid x : \{A_1, \ldots, A_n\}$	*set of labels*
	$\mid$ `[`V_1`..`V_2`]` $\mid x :$ `[`V_1`..`V_2`]`	*range*

FSP has an expressive syntax to represent labels A, A_1, etc. Each label is thus the concatenation of lower-case identifiers *act*, expressions V, and integers within a range "`[`V_1`..`V_2`]`" where V_1 and V_2 are integer expressions. A variable x may be associated to some labels, which allows to reuse them later in the behaviour. A basic process definition consists of a process name P, a set of parameters $X_1, \ldots, X_k$ with default values $V_1, \ldots, V_k$, and a sequential behaviour B. This behaviour can be either relabeled, "A'_{ri}/A_{ri}" meaning that each label in A_{ri} renames into labels A'_{ri} (a single label may rename into several labels, thus yielding several transitions), or hidden to the environment, which corresponds to relabeling into the special action τ. FSP uses label prefix matching while applying hiding and relabeling operators. The *alphabet* of a process consists of the set of labels (possibly renamed) occurring in B and the supplementary labels in "$\{A_{e1}, \ldots, A_{en}\}$". The `stop`, `end`, and `error` behaviours correspond to deadlock, successfull termination, and erroneous termination. Process call and *if-then-else* have a standard semantics. The "`|`" operator denotes a choice in which every branch "$S_i \rightarrow B_i$" whose condition V_i evaluates to true may execute nondeterministically; S_i corresponds to a sequence of labels. Finally, a sequential composition is made up of a list of process calls requiring that all these processes terminate. FSP composite processes are defined as follows:

P_C::=$||P(X_1{=}V_1, \ldots, X_k{=}V_k) = C$ *process definition*
$\gg \{A_{p1}, \ldots, A_{pn}\} \backslash \{A_{h1}, \ldots, A_{hn}\}$ *priority + hide*
| $||P(X_1{=}V_1, \ldots, X_k{=}V_k) = C$ *process definition*
$\ll \{A_{p1}, \ldots, A_{pn}\} \backslash \{A_{h1}, \ldots, A_{hn}\}$ *priority + hide*
C::=$SL\ \ P(V_1, \ldots, V_n)\ /\{A'_{r1}/A_{r1}, \ldots, A'_{rn}/A_{rn}\}$ *process call + relabel*
| $SL\ \ (C_1||\ldots||C_n)\ /\{A'_{r1}/A_{r1}, \ldots, A'_{rn}/A_{rn}\}$ *parallel compo. + relabel*
| `if` V `then` C_1 `else` C_2 *if-then-else structure*
SL::=$\{A_1, \ldots, A_m\}$`::`$\{A_1, \ldots, A_n\}$`:` *sharing / labeling*

A composite process definition consists of a name "$||P$" (the symbol "$||$" indicating that P belongs to the class of composite processes), a set of parameters $X_1, \ldots, X_k$ with default values $V_1, \ldots, V_k$, and a composite behaviour C. Priorities can be assigned to labels in C, "$\gg$" (respectively "$\ll$") meaning that labels in "$\{A_{p1}, \ldots, A_{pn}\}$" have lower (respectively higher) priority than all other labels occurring in C. A composite behaviour may be either a call to a basic process, a parallel composition of composite behaviours synchronising on the intersection of their alphabets, or a deterministic choice between composite processes. Relabeling and hiding are also possible using a syntax similar to basic processes. At last, FSP contains two original operators named *process labeling* "`:`" and *process sharing* "`::`" which are always used subsequently (first sharing then labeling, see the grammar above). Process labeling "$\{A_1, \ldots, A_n\}$`:`C" generates an interleaving of as many instances of C 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. Process sharing "$\{A_1, \ldots, A_m\}$`::`C" replaces each label l occurring in C by a choice between labels "$\{A_1 l, \ldots, A_m l\}$". As an illustration, the resulting automata for "`||C1 = {a,b}:P`" and "`||C2 = {a,b}::P`" are given below with "`P = comm.END`".



Example 1. The following specification describes a semaphore. The process `ACCESS` simulates a client which accesses the critical section protected by the process `SEMAPHORE`. The system, called `SEMADEMO`, is made up of the semaphore in charge of three resources `a`, `b`, `c`, and of the process which wants to access them.

```
SEMAPHORE(N=0) = SEMA[N],
SEMA[v:0..1]   = ( 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) ).
```

LOTOS (Language Of Temporal Ordering Specification) is a specification language standardised by ISO [13]. It combines definitions of abstract data types and algebraic processes. The full syntax of a process is the following:

`process` P `[`G_1, ..., G_m`]` (X_1:S_1, ..., X_n:S_n) : $func$:=
B `where` $block_1$, ..., $block_p$

`endproc`

A process defines a list of formal gates $G_{i\in\{1,\ldots,m\}}$ and of parameters $X_{j\in\{1,\ldots,n\}}$ of sort S_j. Each $block_{k\in\{1,\ldots,p\}}$ denotes a data type or process definition. The functionality $func$ of a process is `exit` if it ends by an `exit` behaviour, or `noexit` otherwise. The process behaviour B is formalised in the following grammar. We present only operators that are required for our translation:

$B ::=$	`exit`	*termination*
	$\mid$ $G\ O_1\ \ldots\ O_n$ `[`V`] ;`B $\mid$ `i;`B	*action prefix*
	$\mid$ `[`V`]->`B	*guarded behaviour*
	$\mid$ B_1 `[]` B_2	*choice*
	$\mid$ B_1`>>`B_2	*sequential composition*
	$\mid$ `hide` $G_1, \ldots, G_n$ `in` B	*hiding*
	$\mid$ P`[`$G_1, \ldots, G_n$`]`$(V_1, \ldots, V_m)$	*process call*
	$\mid$ `choice` $X : T$ `[]` B	*choice on values*
$O ::=$	`!`V $\mid$ `?`$X : T$	*emission / reception*
$V ::=$	$X \mid f(V_1, \ldots, V_n)$	*value expression*

Gate identifiers G may be complemented with a set of parameters called offers. An *offer* has either the form "`!`V" which corresponds to the emission of a value V, or the form "`?`$X : T$" which means the reception of a value of sort T in a variable X. A single action can contain several offers, and these offers can be complemented by a guard which constrains the received values. For instance, "`COMM?X:Int[X==1]`" means that `1` is the only value that can be received in `X`. The guarded behaviour "`[`V`]->`B" means that B is executed only if the boolean expression V evaluates to true. The sequential composition "B_1`>>`B_2" executes first B_1 until it reaches an exit, then B_2 is executed. Hiding masks gates within a behaviour, thus producing an hidden event written `i`. Cyclic behaviours may be defined using tail-recursive process calls. The choice on values "`choice` $X : T$ `[]` B" generates a choice between behaviours $B_{V_1}, \ldots, B_{V_n}$ where $V_1, \ldots, V_n$ are all the values of sort T, and variable X is instantiated with value V_1 in B_{V_1}, with value V_2 in B_{V_2}, etc.

The main differences between Fsp and Lotos resides in the treatment of labels. On the one hand, Lotos labels are structured in the form of a static (*i.e.*, determined at compile time) gate, possibly followed by dynamic (*i.e.*, computed at run-time) offers. Label hiding and label synchronisation are determined uniquely by the gate. On the other hand, no such distinction between gate and offers exists in Fsp, where hiding and synchronization depend on full labels, computed at run-time by concatenation of sub-labels and replacement of variables. This difference constitutes the main difficulty of the translation.

Fortunately, Cadp also provides a tool for communicating automata, called Exp.Open [14], whose input language (Exp) features more flexible label handling mechanisms than Lotos. Exp allows to describe parallel compositions of finite state machines using several parallel composition operators, synchronisation vectors, as well as renaming, hiding, cutting, and priority operators. We present the part of the Exp language that is used in this paper. $L_1, L'_1, \ldots$ represent labels, which are merely character strings.

$B ::=$ `total rename` $L_1 \rightarrow L'_1, \ldots, L_n \rightarrow L'_n$ `in` B `end rename` *rename*
| `total hide` $L_1, \ldots, L_n$ `in` B `end hide` *hide*
| `total cut` $L_1, \ldots, L_n$ `in` B `end cut` *cut*
| `total prio all but` $L_1, \ldots, L_n > L'_1, \ldots, L'_k$ `in` B `end prio` *priority (1)*
| `total prio` $L_1, \ldots, L_n >$ `all but` $L'_1, \ldots, L'_k$ `in` B `end prio` *priority (2)*
| $B_1 ||| B_2$ *interleaving*
| `label par` $L_1, \ldots, L_m$ `in` $B_1 || \ldots || B_n$ `end par` *par. compo.*
| `label par` $V_1, \ldots, V_m$ `in` $B_1 || \ldots || B_n$ `end par` *vect. compo.*
$V ::= (L_1|\_) * \ldots * (L_n|\_) \rightarrow L$ *vector*

Rename and hide respectively define a set of labels to be renamed, and a set of labels to be hidden in behaviour B. The `cut` operator is used to cut the transitions that carry some given labels in a transition system. Priority expresses that a set of labels have a higher priority than another set of labels. "`all but` $L_1, \ldots, L_n$" represents all labels except $L_1, \ldots, L_n$. We introduce three forms of parallel composition: "$B_1 ||| B_2$" means that B_1 and B_2 run in parallel without synchronising, "`label par` $L_1, \ldots, L_m$ `in` $B_1 || \ldots || B_n$" means that $B_1, \ldots, B_n$ run in parallel and synchronise all together on labels $L_1, \ldots, L_m$, and "`label par` $V_1, \ldots, V_m$ `in` $B_1 || \ldots || B_n$" means that they synchronise following the constraints expressed by the synchronisation vectors $V_1, \ldots, V_m$. Precisely, a vector "$l_1 * \ldots * l_n \rightarrow l$" produces a transition labeled l if all B_i such that "$l_i \neq \_$" execute all together a transition labeled l_i. Exp.Open provides alternative semantics for the `hide`, `rename`, `cut`, `prio`, and `par` operators, the precise semantics used in this paper being determined by the `total` (which means that a label matches if it matches a regular expression entirely) and `label` (which means that processes synchronise on full labels, and not only on the gate part of the label) keywords. Finally, we mention Svl [8] (Script Verification Language), which allows a high-level and concise description of the calls to the different Cadp tools.

3 Preliminary Definitions

3.1 Environment

When translating an Fsp specification into Lotos/Exp, we need to propagate along the abstract syntax tree of the Fsp specification, information collected during the tree traversal. This information is called an environment and is made of the following objects:

- E is a partial function associating expressions to variables, represented as a set of couples of the form "$x \mapsto v$". Environment E will be used during the translation to store variables defined in Fsp labels but also process parameters and constant definitions. E is initialised with constant definitions which are global to all processes.
- S is a set of labels used with the "`::`" Fsp operator, to be shared between the parallel processes.

– L is a label coming from the "`:`" FSP operator, to be distributed as prefix over the parallel processes.
– R is a relabeling relation represented as a set of elements of the form "$l \mapsto \{l_1, \ldots, l_k\}$", which associates sets of new labels to old labels.
– H is a set of labels to be hidden.
– X is a mapping from variables to sets of labels, represented as a set of elements of the form "$x \mapsto s$", where s is either a range "(v_1, v_2)" or a set of labels "$\{l_1, \ldots, l_k\}$".

An environment is a tuple "$\langle E, M, X \rangle$" where M is a list of tuples "$\langle S, L, R, H \rangle$". We now define some functions used thereafter to formalise the translation. Function *dom* applies to R such that: $dom(\{l_1 \mapsto s_1, \ldots, l_k \mapsto s_k\}) = \{l_1, \ldots, l_k\}$ where $s_{i \in \{1,\ldots,k\}}$ are sets of labels. The dispatching function $\mapsto_d$ associates every element of one source set to all the elements of a target set:

$\{l_1, \ldots, l_k\} \mapsto_d \{l'_1, \ldots, l'_k\} = \{l_1 \mapsto \{l'_1, \ldots, l'_k\}, \ldots, l_k \mapsto \{l'_1, \ldots, l'_k\}\}$

Function $\otimes$ concatenates elements of two sets:

$\{l_1, \ldots, l_k\} \otimes \{m_1, \ldots, m_p\} = \{l_1 m_1, \ldots, l_1 m_p, \ldots, l_k m_1, \ldots, l_k m_p\}$

Function *pm* is a prefix matching test, which takes as input a label l and a set of labels, and returns *true* if one of the labels in the list is a prefix for l:

$pm(l, \{l_1, \ldots, l_n\}) = (\exists i \in \{1, \ldots, n\})(\exists l')\ l = l_i l'$

Function *newlab* takes as input a label l and returns the set of labels after relabeling by R:

$newlab(l, \{l_1 \mapsto s_1, \ldots, l_k \mapsto s_k\}) = \{l'_i l' | (\exists i \in \{1, \ldots, k\})\ l = l_i l' \wedge l'_i \in s_i\}$

3.2 Expressions

Function $f2l_e$ translates FSP expressions (which are of type integer or string) into LOTOS expressions substituting the values of variables *wrt.* environment E:

$f2l_e(x, E) = E(x)$

$f2l_e(f(V_1, \ldots, V_n), E) = f(f2l_e(V_1, E), \ldots, f2l_e(V_n, E))$

Function *vars* extracts the variables appearing in an expression:

$vars(x) = \{x\}$

$vars(f(V_1, \ldots, V_n)) = vars(V_1) \cup \ldots \cup vars(V_n)$

Function *type* computes the type of an expression, that is either `Int` or `String`. This function is standard, and therefore not defined here.

3.3 Translation of a Label

The translation of an FSP label is not easy because FSP labels involve different notions that have no direct counterpart in LOTOS, namely sets, ranges, complex expressions, and variable definitions. Moreover, hiding or renaming of a label can only be made after flattening this high-level notation, as computed by LTSA while generating transition systems from FSP specifications. Consequently, we define two functions, $flatten$ and $flatten_x$, to translate an FSP label into a set of LOTOS labels. Function $flatten$ expands all the variables appearing in the label, whereas $flatten_x$ translates as often as possible into LOTOS variables the

variables defined in the FSP label. In the following, function $flatten$ is used instead of the less space-consuming $flatten_x$ one when hiding or renaming has to be applied on the label.

We start with the definition of the $flatten$ function which is called with an environment E associating one value to every variable appearing in FSP, and returns a set of couples *(label, environment)*. In case a variable is assigned several values (as defined in an FSP range or set), $flatten$ generates as many labels and environments as there are values associated to the variable. At this abstract level, all the label portions are stored in a list using the *cons* and *nil* operator. A variable is defined only once in an FSP specification, so union between variable environments can be used instead of an overloading operation.

$flatten(L_1 \dots L_n, E) =$
$\{(cons(l_1, l_{11}), E_{11} \cup E_1), \dots, (cons(l_1, l_{1m}), E_{1m} \cup E_1), \dots,$
$(cons(l_k, l_{k1}), E_{k1} \cup E_k), \dots, (cons(l_k, l_{kl}), E_{kl} \cup E_k)\}$
where $\{(l_1, E_1), \dots, (l_k, E_k)\} = flatten_l(L_1, E)$,
$\{(l_{11}, E_{11}), \dots, (l_{1m}, E_{1m})\} = flatten(L_2 \dots L_n, E_1)$, and
$\{(l_{k1}, E_{k1}), \dots, (l_{kl}, E_{kl})\} = flatten(L_2 \dots L_n, E_k)$.

The terminal case is defined on an empty list ε as: $flatten(\varepsilon, E) = \{(nil, E)\}$

Function $flatten_l$ flattens a portion of label. In case of an expression indexed by a variable, environment E is extended with the variable x and its value v obtained after evaluation by $f2l_e$.

$flatten_l(act, E) = \{(act, E)\}$
$flatten_l(V, E) = \{(v, E)\}$
$flatten_l(x : V, E) = \{(v, \{x \mapsto v\} \cup E)\}$
where $v = f2l_e(V, E)$.

In case of sets of labels, all the labels are expanded, and environments updated if necessary.

$flatten_l(\{A_1, \dots, A_n\}, E) =$
$\{(l_{11}, E_{11}), \dots, (l_{1k}, E_{1k}), \dots, (l_{n1}, E_{n1}), \dots, (l_{nm}, E_{nm})\}$
$flatten_l(x{:}\{A_1, \dots, A_n\}, E) =$
$\{(l_{11}, \{x \mapsto l_{11}\} \cup E_{11}), \dots, (l_{1k}, \{x \mapsto l_{1k}\} \cup E_{1k}), \dots,$
$(l_{n1}, \{x \mapsto l_{n1}\} \cup E_{n1}), \dots, (l_{nm}, \{x \mapsto l_{nm}\} \cup E_{nm})\}$
where $\{(l_{11}, E_{11}), \dots, (l_{1k}, E_{1k})\} = flatten(A_1, E)$, and
$\{(l_{n1}, E_{n1}), \dots, (l_{nm}, E_{nm})\} = flatten(A_n, E)$.

In case of ranges, all the integer expressions are computed from the range.

$flatten_l(\texttt{[}V_1\texttt{..}V_2\texttt{]}, E) = \{(v_1, E), \dots, (v_r, E)\}$
$flatten_l(x{:}\texttt{[}V_1\texttt{..}V_2\texttt{]}, E) = \{(v_1, \{x \mapsto v_1\} \cup E), \dots, (v_r, \{x \mapsto v_r\} \cup E)\}$
where $v_1 = f2l_e(V_1, E)$, $r = (f2l_e(V_2, E) - f2l_e(V_1, E)) + 1$, and
$(\forall i \in \{1, \dots, r-1\})\ v_{i+1} = v_i + 1$.

Example 2. The FSP label "`lab[x:1..2]`" is translated using $flatten$ as two abstract LOTOS labels (see Section 7 for the concrete notation): "$cons(lab, cons(1, nil))$" and "$cons(lab, cons(2, nil))$".

Function $flatten_x$ generates LOTOS labels keeping variables when the label is not concerned by relabeling or hiding. Function $flatten_x$ returns a set of tuples

"(l, E, X)" where l is a label, E is a variable environment, and X binds variables which are kept while translating to a range of integer values, or a set of labels.

$flatten_x(L_1 \dots L_n, E, X) = \{(cons(l_1, l_{11}), E_{11} \cup E_1, X_{11} \cup X_1), \dots,$
$(cons(l_1, l_{1m}), E_{1m} \cup E_1, X_{1m} \cup X_1), \dots, (cons(l_k, l_{k1}), E_{k1} \cup E_k, X_{k1} \cup X_k),$
$\dots, (cons(l_k, l_{kl}), E_{kl} \cup E_k, X_{kl} \cup X_k)\}$
$flatten_x(\varepsilon, E, X) = \{(nil, E, X)\}$
where $\{(l_1, E_1, X_1), \dots, (l_k, E_k, X_k)\} = flatten_{xl}(L_1, E, X)$,
$\{(l_{11}, E_{11}, X_{11}), \dots, (l_{1m}, E_{1m}, X_{1m})\} = flatten_x(L_2 \dots L_n, E_1, X_1)$,
$\{(l_{k1}, E_{k1}, X_{k1}), \dots, (l_{kl}, E_{kl}, X_{kl})\} = flatten_x(L_2 \dots L_n, E_k, X_k)$,

Function $flatten_{xl}$ translates a portion of an FSP label into a portion of a LOTOS label. All the variables appearing in the FSP label are kept taking advantage of the expressiveness of LOTOS offers. Thus, an FSP set or range is translated as the variable at hand, and the set or range is stored in X that will be used during the translation of the full label to generate a guard constraining the value of the variable. If a variable is part of an expression to be translated, a new variable (y below) is kept in place of this expression.

$flatten_{xl}(act, E, X) = \{(act, E, \emptyset)\}$
$flatten_{xl}(V, E, X) =$
$$\begin{cases} \{(y, \{y \mapsto y\} \cup E, \{y \mapsto (v, v)\})\} & \text{if } \exists z \in vars(V) \ \wedge \ z \in X \\ \{(v, E, \emptyset)\} & \text{otherwise} \end{cases}$$
$flatten_{xl}(x{:}V, E, X) =$
$$\begin{cases} \{(y, \{x \mapsto v, y \mapsto y\} \cup E, \{y \mapsto (v, v)\})\} & \text{if } \exists z \in vars(V) \ \wedge \ z \in X \\ \{(v, \{x \mapsto v\} \cup E, \emptyset)\} & \text{otherwise} \end{cases}$$
where $v = f2l_e(V, E)$.

When translating sets and ranges, several labels with environments E and X are generated. In case a variable x appears as index of the set (resp. range), x is kept as a variable in E and the environment X is extended with all the possible values that can be computed for x from the set (resp. range).

$flatten_{xl}(\{A_1, \dots, A_n\}, E, X) = \{(l_{11}, E_{11}, X_{11}), \dots, (l_{1k}, E_{1k}, X_{1k}), \dots,$
$(l_{n1}, E_{n1}, X_{n1}), \dots, (l_{nm}, E_{nm}, X_{nm})\}$
$flatten_{xl}(x{:}\{A_1, \dots, A_n\}, E, X) =$
$\{(x, \{x \mapsto x\} \cup E, \{x \mapsto \{l_{11}, \dots, l_{1k}, \dots, l_{n1}, \dots, l_{nm}\}\})\}$
where $\{(l_{11}, E_{11}, X_{11}), \dots, (l_{1k}, E_{1k}, X_{1k})\} = flatten_x(A_1, E, X)$, and
$\{(l_{n1}, E_{n1}, X_{n1}), \dots, (l_{nm}, E_{nm}, X_{nm})\} = flatten_x(A_n, E, X)$.
$flatten_{xl}(\texttt{[}V_1..V_2\texttt{]}, E, X) =$
$$\begin{cases} \{(y, \{y \mapsto y\} \cup E, \{y \mapsto (v_1, v_r)\})\} & \text{if } \exists z \in (vars(V_1) \cup vars(V_2)) \wedge z \in X \\ \{(v_1, E, \emptyset), \dots, (v_r, E, \emptyset)\} & \text{otherwise} \end{cases}$$
$flatten_{xl}(x{:}\texttt{[}V_1..V_2\texttt{]}, E, X) = \{(x, \{x \mapsto x\} \cup E, \{x \mapsto (v_1, v_r)\})\}$
where $v_1 = f2l_e(V_1, E)$, $r = (f2l_e(V_2, E) - f2l_e(V_1, E)) + 1$, and
$(\forall i \in \{1, \dots, r-1\})\ v_{i+1} = v_i + 1$.

Example 3. The FSP label "`lab[x:1..2]`" is translated using $flatten_x$ in LOTOS as "$cons(lab, cons(x, nil))$" with a guard "$(x \geq 1) \wedge (x \leq 2)$" which restricts the values of x. Both pieces of specification generate exactly the same labels.

3.4 Hiding or Renaming

Function *horr* tests if a set of flattened labels is concerned by hiding or renaming, and is used while translating sequences of labels to decide if variables may be kept (use of $flatten_x$) or not (use of $flatten$) in the LOTOS code. The list of tuples "$\langle S, L, R, H\rangle$" is applied starting by the first tuple since this list is built adding in head, and tuples have to be applied starting by the most recent one.

$horr(\{l_1, \ldots, l_k\}, [\langle S_1, L_1, R_1, H_1\rangle, \ldots, \langle S_n, L_n, R_n, H_n\rangle]) =$
$\quad \bigvee_{i\in\{1,\ldots,q\}, j\in\{1,\ldots,k\}}$
$\quad (\ pm(s_i ml_j, H_1) \ \vee \ pm(s_i ml_j, dom(R_1))$
$\quad\ \vee horr(\{s_i ml_j\}, [\langle S_2, L_2, R_2, H_2\rangle, \ldots, \langle S_n, L_n, R_n, H_n\rangle]) \ )$
where $S_1 = \{s_1, \ldots, s_q\}$, and $L_1 = m$.

4 Translating FSP Sequential Processes into LOTOS

This section presents function $f2l_p$ which translates a sequence of labels, and function $f2l_b$ which generates LOTOS code for FSP sequential processes.

4.1 Sequence of Labels

The translation of a sequence depends if labels have to be modified (renamed or hidden) during the translation: if it is the case, they are flattened using the $flatten$ function; otherwise variables are kept and function $flatten_x$ is used. We also recall that an FSP label may correspond to several labels in LOTOS. Therefore, the translation of a label may generate a choice in LOTOS with as many branches as labels computed by $flatten$ or $flatten_x$.

Hiding or relabeling required. More formally, it means that $horr(\{l_1, \ldots, l_h\}, M) = true$ where $\{(l_1, E_1), \ldots, (l_h, E_h)\} = flatten(A_1, E)$.

$f2l_p(A_1 \rightarrow \ldots \rightarrow A_k \rightarrow B, \langle E, M, X\rangle) =$
$\quad f2l_l(apply(l_1, M)) \ seq(l_1, M) \ f2l_p(A_2 \rightarrow \ldots \rightarrow A_k \rightarrow B, \langle E_1 \cup E, M, X\rangle)$
$\quad$ `[]` $\ldots$ `[]`
$\quad f2l_l(apply(l_h, M)) \ seq(l_h, M) \ f2l_p(A_2 \rightarrow \ldots \rightarrow A_k \rightarrow B, \langle E_h \cup E, M, X\rangle)$
$f2l_p(B, \langle E, M, X\rangle) = f2l_b(B, \langle E, M, X\rangle)$

The last rule applies when the sequence of labels is empty. Function $f2l_b$ dedicated to the translation of sequential processes is defined in the sequel. Function *apply* computes a set of labels resulting of the application in sequence of the list of tuples on a label. Note that relabeling may replace a single label by several ones, and that prefixing, relabeling and hiding are successively applied.

$apply(l, [\langle S_1, L_1, R_1, H_1\rangle, \ldots, \langle S_n, L_n, R_n, H_n\rangle]) =$
$\quad apply(l''_1, [\langle S_2, L_2, R_2, H_2\rangle, \ldots, \langle S_n, L_n, R_n, H_n\rangle])$
$\quad \cup \ \ldots \ \cup$
$\quad apply(l''_k, [\langle S_2, L_2, R_2, H_2\rangle, \ldots, \langle S_n, L_n, R_n, H_n\rangle])$
where $S_1 = \{s_1, \ldots, s_q\}$, $L_1 = m$, $\{l_1, \ldots, l_p\} = \{s_1 ml, \ldots, s_q ml\}$,
$\quad \{l'_1, \ldots, l'_r\} = apply_R(l_1, R_1) \ \cup \ apply_R(l_p, R_1)$, and
$\quad \{l''_1, \ldots, l''_k\} = apply_H(l'_1, R_1) \ \cup \ apply_H(l'_r, R_1)$.

Functions $apply_R$ and $apply_H$ are resp. in charge of renaming and hiding.

$$apply_R(l,R) = \begin{cases} newlab(l,R) & \text{if } pm(l, dom(R)) \\ l & \text{otherwise} \end{cases}$$

$$apply_H(l,H) = \begin{cases} i & \text{if } pm(l,H) \\ l & \text{otherwise.} \end{cases}$$

Function $f2l_l$ generates a LOTOS choice from a set of labels.

$$f2l_l(\{l_1,\ldots,l_n\}) = \begin{cases} l_1 & \text{if } n = 1 \\ l_1\texttt{;exit [] ... [] }l_n\texttt{;exit} & \text{otherwise} \end{cases}$$

Function *seq* chooses the LOTOS sequential composition "$\gg$" as sequence operator when this operator is preceded by a behaviour (a choice among several labels), and the LOTOS action prefix "`;`" when preceded by a single label.

$$seq(l,M) = \begin{cases} \gg & \text{if } |apply(l,M)| > 1 \\ ; & \text{otherwise} \end{cases}$$

No hiding or relabeling required. In this case, there is $horr(\{l_1,\ldots,l_m\},M) = false$ where $\{(l_1,E_1),\ldots,(l_m,E_m)\} = flatten(A_1,E)$.

$f2l_p(A_1 \to \ldots \to A_k \to B, \langle E,M,X\rangle) =$
  $f2l_l(apply(l_1,M), X_1 \cup X)\ seq(l_1,M,X_1 \cup X)$
    $f2l_p(A_2 \to \ldots \to A_k \to B, \langle E_1 \cup E, M, X_1 \cup X\rangle)$
  `[] ... []`
  $f2l_l(apply(l_h,M), X_h \cup X)\ seq(l_h,M,X_h \cup X)$
    $f2l_p(A_2 \to \ldots \to A_k \to B, \langle E_h \cup E, M, X_h \cup X\rangle)$
$f2l_p(B,\langle E,M,X\rangle) = f2l_b(B,\langle E,M,X\rangle)$
where $\{(l_1,E_1,X_1),\ldots,(l_h,E_h,X_h)\} = flatten_x(A_1,E,X)$.

Extra variables in X generated while flattening labels using $flatten_x$ (see for instance the second rule of $flatten_{xl}$, page 9) are not considered below because they make the notation concise but are not used in the following of the behaviour.

$f2l_l(\{l_1,\ldots,l_n\},X) =$

$$\begin{cases} l_1 & \text{if } n = 1 \\ l_1\texttt{[}G\texttt{];exit(}x_1,\ldots,x_p\texttt{) [] ... [] }l_n\texttt{[}G\texttt{];exit(}x_1,\ldots,x_p\texttt{)} & \text{otherwise} \end{cases}$$

where $X = \{x_1 \mapsto D_1,\ldots,x_p \mapsto D_p\}$, and
  $G = f2l_t(x_1 \mapsto D_1) \wedge \ldots \wedge f2l_t(x_p \mapsto D_p)$.

Function $f2l_t$ generates guards from tuples of X. These guards are used to constrain the values of variables introduced in the translation of labels.

$f2l_t(x \mapsto (v_1,v_2)) = (x \geq v_1) \wedge (x \leq v_2)$
$f2l_t(x \mapsto \{l_1,\ldots,l_k\}) = (x = l_1) \vee \ldots \vee (x = l_k)$

Finally, function *seq* makes the variable passing explicit when the LOTOS sequential composition "$\gg$" is chosen as sequence operator. This is mandatory compared to the expanded translation of labels since variables are preserved here and can be used in the rest of the behaviour.

$$seq(l,M,X) = \begin{cases} \gg\ \texttt{accept}\ x_1 : T_1,\ldots,x_p : T_p\ \texttt{in} & \text{if } |apply(l,M)| > 1 \\ ; & \text{otherwise} \end{cases}$$

where $X = \{x_1 \mapsto D_1,\ldots,x_p \mapsto D_p\}$, and $T_1 = type(x_1),\ldots,T_p = type(x_p)$.

4.2 Sequential Processes

FSP sequential processes are translated into LOTOS processes. FSP allows the definition of local processes which are translated as local processes into LOTOS as well. We only present the translation of a process without local definitions, although this structuring is taken into account into the tool we have implemented (see Section 6). Now, we define the translation from FSP to LOTOS as a function $f2l_b$. Function $func$ computing the process functionality is defined in [15]. Note also that only two concrete LOTOS gates are generated by our translation: `EVENT` which prefixes all the regular FSP labels, and `EVENT_ERROR` which is used to encode the FSP `error` termination.

$f2l_b($`P(`X_1`=`$V_1, \ldots, X_k$`=`V_k`) = `B $+ \{A_{e1}, \ldots, A_{en}\}$
$/\{A'_{r1}/A_{r1}, \ldots, A'_{rn}/A_{rn}\} \setminus \{A_{h1}, \ldots, A_{hn}\}, \langle E, M, X\rangle) =$
`process` P `[EVENT,EVENT_ERROR] (`$X_1 : T_1, \ldots, X_k : T_k$`) :` $func(B)$ `:=`
$f2l_b(B, \langle E_0, [\langle \emptyset, \emptyset, R_0, H_0\rangle, M], X\rangle)$
`endproc`

where $T_1 = type(V_1), \ldots, T_k = type(V_k)$, $E_0 = \{X_1 \mapsto V_1, \ldots, X_k \mapsto V_k\} \cup E$,
$\{(l_{r11}, E_{r11}), \ldots, (l_{r1k}, E_{r1k})\} = flatten(A_{r1}, E)$,
$\{(l'_{r11}, E'_{r11}), \ldots, (l'_{r1k}, E'_{r1k})\} = flatten(A'_{r1}, E)$,
$\{(l_{rn1}, E_{rn1}), \ldots, (l_{rnk}, E_{rnk})\} = flatten(A_{rn}, E)$,
$\{(l'_{rn1}, E'_{rn1}), \ldots, (l'_{rnk}, E'_{rnk})\} = flatten(A'_{rn}, E)$,
$R_0 = \{\{l_{r11}, \ldots, l_{r1k}\} \mapsto_d \{l'_{r11}, \ldots, l'_{r1k}\}, \ldots,$
$\{l_{rn1}, \ldots, l_{rnk}\} \mapsto_d \{l'_{rn1}, \ldots, l'_{rnk}\}\}$,
$\{(l_{h11}, E_{h11}), \ldots, (l_{h1k}, E_{h1k})\} = flatten(A_{h1}, E)$,
$\{(l_{hn1}, E_{hn1}), \ldots, (l_{hnk}, E_{hnk})\} = flatten(A_{hn}, E)$, and
$H_0 = \{l_{h11}, \ldots, l_{h1k}, \ldots, l_{hn1}, \ldots, l_{hnk}\}$.

Usual terminations (`stop` and `end`) have direct equivalent in LOTOS. The `error` termination is translated using a `P_ERROR` process whose behaviour is an endless loop on an `EVENT_ERROR` label.

$f2l_b($`stop`$, \langle E, M, X\rangle) =$ `stop`
$f2l_b($`end`$, \langle E, M, X\rangle) =$ `exit`
$f2l_b($`error`$, \langle E, M, X\rangle) =$ `P_ERROR [EVENT_ERROR]`

A process call is translated as is with as parameter its list of arguments.

$f2l_b(P($`(`$V_1, \ldots, V_n$`)`$, \langle E, M, X\rangle) =$
P `[EVENT,EVENT_ERROR] (`$f2l_e(V_1, E), \ldots, f2l_e(V_n, E)$`)`

The `if` structure is encoded as a LOTOS choice with two branches respectively encoding the `then` and `else` part of the FSP behaviour.

$f2l_b($`if` V `then` B_1 `else` $B_2, \langle E, M, X\rangle) =$
`[`$f2l_e(V, E)$`]->` $f2l_b(B_1, \langle E, M, X\rangle)$
`[]`
`[`$\neg f2l_e(V, E)$`]->` $f2l_b(B_2, \langle E, M, X\rangle)$

The FSP choice is translated into a LOTOS choice with guards.

$f2l_b($`when` V_1 $S_1 \rightarrow B_1$ `|` $\ldots$ `|` `when` V_n $S_n \rightarrow B_n, \langle E, M, X\rangle) =$
`[`$f2l_e(V_1, E)$`]->` $f2l_p(S_1 \rightarrow B_1, \langle E, M, X\rangle)$
`[]` $\ldots$ `[]`
`[`$f2l_e(V_n, E)$`]->` $f2l_p(S_n \rightarrow B_n, \langle E, M, X\rangle)$

Last, a sequential composition is translated similarly in LOTOS.

$f2l_b(P_1\texttt{(}V_{11},\ldots,V_{1k}\texttt{)};\ldots;P_n\texttt{(}V_{n1},\ldots,V_{nl}\texttt{)};B,\langle E,M,X\rangle) =$

$P_1\texttt{[EVENT,EVENT\_ERROR](}f2l_e(V_{11},E),\ldots,f2l_e(V_{1k},E)\texttt{)} \gg \ldots \gg$

$P_n\texttt{[EVENT,...](}f2l_e(V_{n1},E),\ldots,f2l_e(V_{nl},E)\texttt{)} \gg f2l_b(B,\langle E,M,X\rangle)$

No variables are passed along the LOTOS sequential composition because processes involved in the composition are independent of each other: each process has to terminate correctly before starting the next one. In addition, LOTOS local processes are generated for each process called in the sequential composition. This is needed to apply the environment (and possible renaming or hiding) to the definitions of referred processes. As an example, process P_1 is translated as follows, where $\hat{P_1}$ is the process definition of P_1:

`process` P_1 `[EVENT,EVENT_ERROR] (`$X_1 : T_1,\ldots,X_k : T_k$`)` $: func(\hat{P_1})$ `:=`

$f2l_b(\hat{P_1},\langle E_0,M,X\rangle)$

`endproc`

where $T_1 = type(V_1),\ldots,T_k = type(V_k)$, $E_0 = \{X_1 \mapsto V_1,\ldots,X_k \mapsto V_k\} \cup E$.

5 Translating FSP Composite Processes into EXP

Encoding FSP composite processes into LOTOS is tedious for several reasons:

- LOTOS hiding and synchronisation constructs operate on gates, whereas FSP constructs operate on full labels (which are gates + offers in LOTOS).
- LOTOS has no renaming operator. The only way to rename a gate in LOTOS is to instantiate a process with an actual gate different from the formal one. Such a renaming is not always satisfactory, because it only permits injective renaming (different gates cannot be renamed into the same gate), whereas non-injective renaming is allowed in FSP.
- LOTOS does not have a priority operator. The only way to express it is by refactoring the specification to only allow labels with high priority to be executed when necessary.

Consequently, we chose to translate FSP composite processes into the EXP format instead of LOTOS. Translation of an FSP composite process into EXP is made up of three steps. First, all the FSP sequential processes are translated into LOTOS using function $f2l_b$ presented in Section 4. This translation takes into account the possible parameters coming with the process call. Then, SVL [8] scripts are automatically derived to generate a BCG file (which is a computer representation for state/transition models) for each sequential process translated into LOTOS. These BCG descriptions of processes are used in the last step, namely the translation of FSP composite processes into EXP.

Function $f2l_c$ translates composite processes into EXP specifications. The abstract notation "$l.*$" matches labels with l as prefix, and is used to take the prefix matching into account while hiding labels. The environment is only used during the translation of composite processes to store values of process parameters. This is due to the top-down approach our translation is based on, and to the fact that all the FSP operators are directly expressed in EXP.

$f2l_c($ ||P(X$_1$=V$_1$, ..., X$_k$=V$_k$) =
$C \gg \{A_{p1}, \ldots, A_{pn}\} \setminus \{A_{h1}, \ldots, A_{hn}\}, \langle E, M, X\rangle$) =
total hide $l_{h11}, l_{h11}.*, \ldots, l_{h1k}, l_{h1k}.*,$
$\ldots, l_{hn1}, l_{hn1}.*, \ldots, l_{hnk}, l_{hnk}.*$ in
total prio all but $l_{p11}, \ldots, l_{p1k}, \ldots, l_{pn1}, \ldots, l_{pnk}$ >
$l_{p11}, \ldots, l_{p1k}, \ldots, l_{pn1}, \ldots, l_{pnk}$ in
$f2l_c(C, \langle E_0, M, X\rangle)$
end prio
end hide

$f2l_c($ ||P(X$_1$=V$_1$, ..., X$_k$=V$_k$) =
$C \ll \{A_{p1}, \ldots, A_{pn}\} \setminus \{A_{h1}, \ldots, A_{hn}\}, \langle E, M, X\rangle$) =
total hide $l_{h11}, l_{h11}.*, \ldots, l_{h1k}, l_{h1k}.*, \ldots,$
$l_{hn1}, l_{hn1}.*, \ldots, l_{hnk}, l_{hnk}.*$ in
total prio $l_{p11}, \ldots, l_{p1k}, \ldots, l_{pn1}, \ldots, l_{pnk}$ >
all but $l_{p11}, \ldots, l_{p1k}, \ldots, l_{pn1}, \ldots, l_{pnk}$ in
$f2l_c(C, \langle E_0, M, X\rangle)$
end prio
end hide

where $T_1 = type(V_1), \ldots, T_k = type(V_k)$, $E_0 = \{X_1 \mapsto V_1, \ldots, X_k \mapsto V_k\} \cup E$,
$\{(l_{h11}, E_{h11}), \ldots, (l_{h1k}, E_{h1k})\} = flatten(A_{h1}, E)$,
$\{(l_{hn1}, E_{hn1}), \ldots, (l_{hnk}, E_{hnk})\} = flatten(A_{hn}, E)$,
$\{(l_{p11}, E_{p11}), \ldots, (l_{p1k}, E_{p1k})\} = flatten(A_{p1}, E)$, and
$\{(l_{pn1}, E_{pn1}), \ldots, (l_{pnk}, E_{pnk})\} = flatten(A_{pn}, E)$.

As regards the translation of a process call into Exp, process P is duplicated in as many interleaved processes P as there are labels in the labeling set with all the labels of process P prefixed by one label of this set, respectively m_1, m_2, etc. The "label par" statement is used below for renaming purposes, since the rename statement existing in Exp does not allow to rename a single label into several labels (thus producing several transitions from a single one). Therefore, we use synchronisation vectors (usually used for synchronisation purposes) to rename labels ($vectors_r$), and to prefix all the labels of the process by the labels defined in S ($vectors_p$). Function *alpha* computing the alphabet of a process is defined in [15].

$f2l_c(SL\ P(V_1, \ldots, V_n)\ /\{A'_{r1}/A_{r1}, \ldots, A'_{rn}/A_{rn}\}, \langle E, M, X\rangle) =$
label par $vectors_r(R_0, alpha(SL\ P(V_1, \ldots, V_n), \langle E, M, X\rangle))$ in
label par
$vectors_p(\{s_1, \ldots, s_k\}, \{m_1\} \otimes alpha(P(V_1, \ldots, V_n), \langle E, M, X\rangle))$ in
$f2l_{pr}(P(V_1, \ldots, V_n), \langle E, M, X\rangle)$
end par
|||...|||
label par
$vectors_p(\{s_1, \ldots, s_k\}, \{m_p\} \otimes alpha(P(V_1, \ldots, V_n), \langle E, M, X\rangle))$ in
$f2l_{pr}(P(V_1, \ldots, V_n), \langle E, M, X\rangle)$
end par
end par

where $\{s_1, \ldots, s_k\} = flatten_{sh}(SL, E)$, $\{m_1, \ldots, m_p\} = flatten_{lb}(SL, E)$,
$\{(l_{11}, E_{11}), \ldots, (l_{1k}, E_{1k})\} = flatten(A_{r1}, E)$,
$\{(l'_{11}, E'_{11}), \ldots, (l'_{1k}, E'_{1k})\} = flatten(A'_{r1}, E)$,
$\{(l_{n1}, E_{n1}), \ldots, (l_{nk}, E_{nk})\} = flatten(A_{rn}, E)$,
$\{(l'_{n1}, E'_{n1}), \ldots, (l'_{nk}, E'_{nk})\} = flatten(A'_{rn}, E)$, and
$R_0 = \{\{l_{11}, \ldots, l_{1k}\} \mapsto_d \{l'_{11}, \ldots, l'_{1k}\}, \ldots,$
$\{l_{n1}, \ldots, l_{nk}\} \mapsto_d \{l'_{n1}, \ldots, l'_{nk}\}\}$.

Functions $flatten_{sh}$ and $flatten_{lb}$ flatten the sets of labels used as prefixes:
$flatten_{sh}(\{A_1, \ldots, A_m\}::\{A_1, \ldots, A_n\}:, E) = flatten(\{A_1, \ldots, A_m\}, E)$
$flatten_{lb}(\{A_1, \ldots, A_m\}::\{A_1, \ldots, A_n\}:, E) = flatten(\{A_1, \ldots, A_n\}, E)$

Auxiliary functions $f2l_{pr}$, $vectors_r$, and $vectors_p$ are now defined. Function $f2l_{pr}$ refers to the BCG file generated previously from the LOTOS code if it is a sequential process, or calls the $f2l_c$ function if it is a composite process. We present a simplified version of $f2l_{pr}$ since a same process can be referred several times with different parameters. In our translator tool, we indexed such processes with numbers to distinguish their different instances.

$f2l_{pr}(P(V_1, \ldots, V_n), \langle E, M, X\rangle) =$

$$\begin{cases} \texttt{"}P.bcg\texttt{"} & \text{if } is\_sequential(P) \\ f2l_c(P(\hat{X}_1\texttt{=}V_1, \ldots, \hat{X}_n\texttt{=}V_n)\texttt{=}\hat{P}, \langle E, M, X\rangle) & \text{otherwise} \end{cases}$$

where $\hat{X}_{i\in\{1,\ldots,n\}}$ refers to formal parameter identifiers, and $V_{i\in\{1,\ldots,n\}}$ refer to actual values for them.

Function $vector_r$ generates vectors with as left part a single element corresponding to the label to rename, and as right part its new name. Labels which are not concerned by relabeling preserve their original name.

$vectors_r(R, \{l_1, \ldots, l_p\}) =$

$$\begin{cases} l_1 \texttt{ -> } l_1\texttt{,}\ vectors_r(R, \{l_2, \ldots, l_p\}) & \text{if } newlab(l_1, R) = \emptyset \\ l_1 \texttt{ -> } n_1\texttt{,} \ldots \texttt{,} l_1 \texttt{ -> } n_q\texttt{,}\ vectors_r(R, \{l_2, \ldots, l_p\}) & \text{otherwise} \end{cases}$$

where $\{n_1, \ldots, n_q\} = newlab(l_1, R)$.

Function $vector_p$ generates vectors with as left part the label to extend, and derives from it a new label with a prefix taken into a set of prefixes. All the combinations are computed for the set of labels and prefixes in input. Function *cat* corresponds to the concatenation of lists.

$vectors_p(\{s_1, \ldots, s_k\}, \{l_1, \ldots, l_p\}) =$
$l_1 \texttt{ -> } cat(s_1, l_1)\texttt{,} \ldots \texttt{,} l_1 \texttt{ -> } cat(s_k, l_1)\texttt{,}\ vectors_p(\{s_1, \ldots, s_k\}, \{l_2, \ldots, l_p\})$

Similarly to the translation of a process call, for a parallel composition, interleaving of processes is derived, as well as renaming and prefixing using respectively $vectors_r$ and $vectors_p$ functions. Synchronisation sets are made explicit. Below, the composite process C_1 synchronise with the rest of the involved processes on labels $I_1, \ldots, I_q$.

$f2l_c(SL\ (C_1||\ldots||C_n)\ /\{A'_{r1}/A_{r1}, \ldots, A'_{rn}/A_{rn}\}, \langle E, M, X\rangle) =$
`label par` $I_1, \ldots, I_q$ `in`
`(`
`label par` $vectors_r(R_0, alpha(SL\ C_1, \langle E, M, X\rangle))$ `in`
`label par`

$vectors_p(\{s_1,\ldots,s_k\},\{m_1\}\otimes alpha(C_1,\langle E,M,X\rangle))$ `in`
$f2l_c(C_1,\langle E,M,X\rangle)$
`end par`
`end par`
`|||...|||`
`label par` $vectors_r(R_0, alpha(SL\ C_1,\langle E,M,X\rangle))$ `in`
`label par`
$vectors_p(\{s_1,\ldots,s_k\},\{m_p\}\otimes alpha(C_1,\langle E,M,X\rangle))$ `in`
$f2l_c(C_1,\langle E,M,X\rangle)$
`end par`
`end par`
`)`
`||` $f2l_c(SL\ (C_2||\ldots||C_n)\ /\{A'_{r1}/A_{r1},\ldots,A'_{rn}/A_{rn}\},\langle E,M,X\rangle)$
`end par`

where $\{I_1,\ldots,I_q\} = alpha(SL\ C_1\ /\{A'_{r1}/A_{r1},\ldots,A'_{rn}/A_{rn}\},\langle E,M,X\rangle)$,
$\cap\ alpha(SL\ (C_2||\ldots||C_n)\ /\{A'_{r1}/A_{r1},\ldots,A'_{rn}/A_{rn}\},\langle E,M,X\rangle)$,
$\{s_1,\ldots,s_k\} = flatten_{sh}(SL,E)$, $\{m_1,\ldots,m_p\} = flatten_{lb}(SL,E)$,
$\{(l_{r11},E_{r11}),\ldots,(l_{r1k},E_{r1k})\} = flatten(A_{r1},E)$,
$\{(l'_{r11},E'_{r11}),\ldots,(l'_{r1k},E'_{r1k})\} = flatten(A'_{r1},E)$,
$\{(l_{rn1},E_{rn1}),\ldots,(l_{rnk},E_{rnk})\} = flatten(A_{rn},E)$,
$\{(l'_{rn1},E'_{rn1}),\ldots,(l'_{rnk},E'_{rnk})\} = flatten(A'_{rn},E)$, and
$R_0 = \{\{l_{r11},\ldots,l_{r1k}\}\mapsto_d\{l'_{r11},\ldots,l'_{r1k}\},\ldots,$
$\{l_{rn1},\ldots,l_{rnk}\}\mapsto_d\{l'_{rn1},\ldots,l'_{rnk}\}\}$.

The `if` construct is translated as for sequential processes.

$$f2l_c(\texttt{if}\ V\ \texttt{then}\ C_1\ \texttt{else}\ C_2,\langle E,M,X\rangle) = \begin{cases} f2l_c(C_1,\langle E,M,X\rangle) & \text{if } f2l_e(V,E) \\ f2l_c(C_2,\langle E,M,X\rangle) & \text{otherwise} \end{cases}$$

6 Tool and Validation

Translator Tool. We developed an automatic translator from FSP to LOTOS using the SYNTAX and LOTOS NT compiler construction technologies [10]. The tool consists of about 5,000 lines of SYNTAX, 20,000 lines of LOTOS NT, and 500 lines of C. This implementation was split into two main steps: (i) parsing the FSP language and storing the result into an abstract syntax tree, (ii) translating the abstract syntax tree into semantically equivalent LOTOS code. The parsing task was difficult since SYNTAX accepts only LALR(1) grammar as input. Therefore, the abstract FSP grammar as formalised in [16] was refined to a concrete grammar free of ambiguities. We validated the translator on about 10,500 lines of FSP specifications (approx. 2,400 FSP processes) that we reused from [16] or wrote ourselves. In the latter case, we tried to systematically explore all the expressiveness that allows the FSP notation to ensure robustness of our translation. These 10,500 lines of FSP correspond after translation to about 72,000 lines of LOTOS, 2,000 lines of SVL, and 8,000 lines of EXP. This large number of LOTOS lines has two main explanations: (i) LOTOS is more verbose than FSP, for instance

there are more keywords, or gates have to be made explicit; (ii) although we keep variables as often as possible, our translation of FSP sequential processes may flatten labels whereas FSP allows a concise notation for them.

Correctness of the Translation. It is essential to preserve the semantics of the source language after translation into the target one. Indeed, any verification performed with CADP on the specification obtained after translation has to be valid for the initial specification. Our translation preserves semantics of both process calculi *wrt.* a branching equivalence relation [22]. Branching equivalence is the strongest of the weak equivalences found in the literature. Unlike strong equivalence, branching equivalence does not require a strict matching of τ transitions. This is exactly what we need since sequential composition in LOTOS induces τ transitions which do not appear in the semantics of FSP. Semantics preservation modulo branching equivalence is important as it ensures that the properties restricted to visible actions (*e.g.,* safety and fair liveness) verified on the LOTOS specification are indeed properties of the FSP specification. One drawback of branching equivalence might be that it does not preserve τ cycles. For instance, two systems can be branching equivalent even if one system contains a τ cycle which does not appear in the other. However, our translation ensures that all the τ cycles in the FSP specification are preserved in the LOTOS one, and no new cycle is introduced.

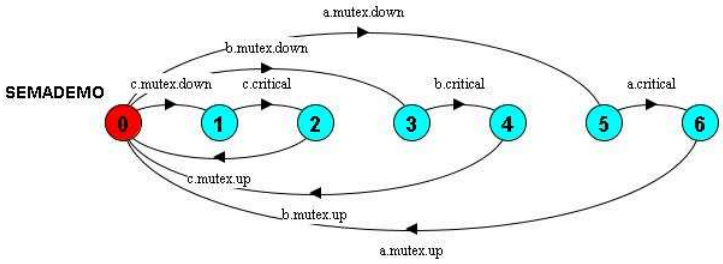
We checked on all the examples that we used for validation that branching equivalence is preserved by our translation. For each specification, this test is performed on LTSs generated by LTSA and CADP *wrt.* the semantics of their respective notations. It was verified automatically using BISIMULATOR [2] for nontrivial examples. BISIMULATOR is a tool of the CADP toolbox which allows to verify the most common notions of behavioural equivalences. The equivalence test cannot be applied directly because FSP labels generated by LTSA and LOTOS labels generated by CADP do not follow the same syntactic conventions. Additionally, the FSP hidden event `tau` corresponds to `i` in LOTOS, and the LOTOS termination event `exit` has no counterpart in FSP. However, CADP allows to systematically transform an LTS with FSP conventions into one with LOTOS conventions, by renaming and cutting labels that match some predefined regular expressions.

7 Application

In this section, we focus on the specification of a semaphore. We present several refinements of this specification, and show how the use of LTSA is complemented by the use of CADP based on the translation from FSP to LOTOS/EXP. The starting point is the specification of the semaphore given in FSP in Example 1. The resulting automaton is made up of 7 states and 9 transitions.

A first refinement is to extend the number of resources ("`{a,b,c}`" but also "`{1,2,3}`") being concerned by the mutual exclusion as well as the number of accesses. This result is obtained by this new specification:

```
||SEMADEMO1 = (  {a,b,c}:ACCESS
```




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

from which LTSA generates an automaton with 13 states and 18 transitions. The next step aims at duplicating both semaphores so that each semaphore is in charge of a single resource.

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

The resulting automaton contains 49 states and 126 transitions and becomes difficult to analyse visually, 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.

```
||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 ).
```

The automaton generated by LTSA has the same number of states and transitions as the former system, but it is impossible to claim that both specifications (`SEMADEMO2` and `SEMADEMO3`) are equivalent. At this level, we use the translation to check this equivalence with the CADP toolbox. We show first some pieces of the code obtained by our translation. The FSP process `SEMAPHORE` is translated as follows in LOTOS:

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

The concrete notation for LOTOS labels is slightly different from the abstract notation we introduced in Section 3. Indeed, a label is systematically represented by the `EVENT` gate followed by an offer consisting of a list of items of types `Int` and `String`, using the following data type:

```
type  Label is IntegerNumber, String
sorts Label
opns  CONS (*! constructor *) : String, Label -> Label
      CONS (*! constructor *) : Int, Label    -> Label
      NIL  (*! constructor *) :               -> Label
endtype
```

If variables appear in one label, the LOTOS `choice` construct is used to distinguish variables and regular string labels. For instance, the FSP label "`lab[x:1..2]`" is translated in LOTOS using the aforementioned concrete syntax as:

```
choice X:Int []
   EVENT !CONS (LAB, CONS (X, NIL)) [(X>=POS(1)) and (X<=POS(2))]
```

Now we show a piece of EXP code generated for the `C_P` process defined in the `SEMADEMO3` system. Processes `ACCESS` and `SEMAPHORE` synchronise on the set of labels appearing at the beginning of the EXP description. This example shows how prefixing labels of `ACCESS` by `B` is done using synchronisation vectors, and how the `exit` label is cut within sequential processes.

```
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)))", ...
in
   label par
      "EVENT !CONS (MUTEX, CONS (DOWN, NIL))"
               -> "EVENT !CONS (B, CONS(MUTEX, CONS (DOWN, NIL)))",
      "EVENT !CONS (CRITICAL, NIL)"
               -> "EVENT !CONS (B, CONS (CRITICAL, NIL))" ,
      "EVENT !CONS (MUTEX, CONS (UP, NIL))"
               -> "EVENT !CONS (B, CONS (MUTEX, CONS (UP, NIL)))"
   in
      total cut exit in "ACCESS.bcg" end cut
   end par
   ||
   ...  total cut exit in "SEMAPHORE.bcg" end cut  ...
end par
```

Processes `SEMADEMO2` and `SEMADEMO3` translated into LOTOS/EXP have been checked strongly equivalent using the BISIMULATOR tool of the CADP toolbox. We illustrated here the use of equivalence checking on FSP designs, but other CADP verification techniques can be used to complement LTSA validation, such as distributed, compositional, or on-the-fly verification to tackle the state explosion problem, or efficient model checking techniques available in EVALUATOR [17]. Last, the debugging stage using CADP does not add any complexity for designers because labels used in counter-examples may be translated in FSP format (see Section 6) using renaming facilities available in CADP.

8 Concluding Remarks

In this paper, our motivation was to reduce the gap between existing tool support for process calculi. We chose here the popular process calculus FSP and the international standard LOTOS. FSP is based on an expressive and concise notation. Therefore, we proposed a translation from FSP to LOTOS (and EXP) to make the joint use of LTSA and CADP possible for FSP users. The translation is completely automated by a tool we implemented. This translator will be integrated in a future release of the CADP toolbox.

As regards related work, to the best of our knowledge, the only proposals focusing on high-level translations between process algebras have been made in the hardware area [20, 23]. Their common goal is to allow verification of asynchronous circuits and architectures. Beyond that, the most related set of works are 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 [18, 4, 21, 1] or into the B method [3]. Motivations of these works are to take advantage of the formal verification means available for the target formalism. Theorem proving allows to fight the state explosion problem and to deal with infinite automata, but is not suitable to prove temporal properties. We preferred model checking because it makes verification steps easier thanks to a full automation and its adequacy to automata-based models.

A first future work is to apply our approach on complex systems, such as web service models described first in BPEL or WS-CDL, and then automatically translated into FSP for analysis purposes [5]. In this case, many interacting services can involve huge underlying state spaces which can be generated and minimised using the optimised means of CADP. Moreover, equivalence checking 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 equivalent [19]. Another perspective is to take FSP safety and progress properties into account, and to translate them into regular alternation-free mu-calculus formulas, input format of the on-the-fly model checker EVALUATOR [17].

References

1. T. Basten and J. Hooman. Process Algebra in PVS. In *Proc. of TACAS'99*, volume 1579 of *LNCS*, pages 270–284. Springer-Verlag, 1999.
2. D. Bergamini, N. Descoubes, C. Joubert, and R. Mateescu. BISIMULATOR: A Modular Tool for On-the-Fly Equivalence Checking. In *Proc. of TACAS'05*, volume 3440 of *LNCS*, pages 581–585. Springer-Verlag, 2005.
3. M. Butler. CSP2B: A Practical Approach to Combining CSP and B. *Formal Aspects of Computing*, 12(3):182–198, 2000.
4. B. Dutertre and S. Schneider. Using a PVS Embedding of CSP to Verify Authentication Protocols. In *Proc. of TPHOLs'97*, volume 1275 of *LNCS*, pages 121–136. Springer-Verlag, 1997.
5. H. Foster, S. Uchitel, J. Magee, and J. Kramer. Tool Support for Model-Based Engineering of Web Service Compositions. In *Proc. of ICWS'05*, pages 95–101. IEEE CSP, 2005.

6. H. Garavel. Compilation of LOTOS Abstract Data Types. In *Proc. of FORTE'89*, pages 147–162. North-Holland, 1989.
7. H. Garavel. OPEN/CÆSAR: An Open Software Architecture for Verification, Simulation, and Testing. In *Proc. of TACAS'98*, volume 1384 of *LNCS*, pages 68–84. Springer-Verlag, 1998.
8. H. Garavel and F. Lang. SVL: A Scripting Language for Compositional Verification. In *Proc. of FORTE'01*, pages 377–394. Kluwer, 2001.
9. H. Garavel, F. Lang, and R. Mateescu. An Overview of CADP 2001. *EASST Newsletter*, 4:13–24, 2001.
10. H. Garavel, F. Lang, and R. Mateescu. Compiler Construction Using LOTOS NT. In *Proc. of TACAS'02*, volume 2304 of *LNCS*, pages 9–13. Springer-Verlag, 2002.
11. H. Garavel and W. Serwe. State Space Reduction for Process Algebra Specifications. *Theoretical Computer Science*, 351(2):131–145, 2006.
12. H. Garavel and J. Sifakis. Compilation and Verification of LOTOS Specifications. In *Proc. of PSTV'90*, pages 379–394. North-Holland, 1990.
13. ISO. LOTOS: a Formal Description Technique based on the Temporal Ordering of Observational Behaviour. Technical Report 8807, International Standards Organisation, 1989.
14. F. Lang. EXP.OPEN 2.0: A Flexible Tool Integrating Partial Order, Compositional, and On-The-Fly Verification Methods. In *Proc. of IFM'2005*, volume 3771 of *LNCS*, pages 70–88. Springer-Verlag, 2005.
15. F. Lang and G. Salaün. Translating FSP into LOTOS and Networks of Automata. Technical report, INRIA, 2007.
16. J. Magee and J. Kramer. *Concurrency: State Models & Java Programs*. Wiley, 1999.
17. R. Mateescu and M. Sighireanu. Efficient On-the-Fly Model-Checking for Regular Alternation-Free Mu-Calculus. *Science of Comp. Progr.*, 46(3):255–281, 2003.
18. M. Nesi. Formalising a Value-Passing Calculus in HOL. *Formal Aspects of Computing*, 11(2):160–199, 1999.
19. G. Salaün, L. Bordeaux, and M. Schaerf. Describing and Reasoning on Web Services using Process Algebra. In *Proc. of ICWS'04*, pages 43–51. IEEE CSP, 2004.
20. G. Salaün and W. Serwe. Translating Hardware Process Algebras into Standard Process Algebras: Illustration with CHP and LOTOS. In *Proc. of IFM'05*, volume 3771 of *LNCS*, pages 287–306. Springer-Verlag, 2005.
21. H. Tej and B. Wolff. A Corrected Failure-Divergence Model for CSP in ISABELLE/HOL. In *Proc. of FME'97*, volume 1313 of *LNCS*, pages 318–337. Springer-Verlag, 1997.
22. R. J. van Glabbeek and W. P. Weijland. Branching Time and Abstraction in Bisimulation Semantics. *Journal of the ACM*, 43(3):555–600, 1996.
23. X. Wang, M. Z. Kwiatkowska, G. K. Theodoropoulos, and Q. Zhang. Towards a Unifying CSP approach to Hierarchical Verification of Asynchronous Hardware. In *Proc. of AVoCS'04*, volume 128 of *ENTCS*, pages 231–246, 2005.