Electronic Notes in Theoretical Computer Science 4 (1996)

Input/Output for ELAN

Patrick Viry

Dipartimento di Informatica, Università di Pisa
Corso Italia 40, 56100 Pisa, Italy
Email: viry@di.unipi.it

Abstract

We show how to add Input/Output capabilities to the ELAN rewriting interpreter using a rewrite specification of π-calculus. This I/O system has the advantage of being totally explicit and fit in the same semantic framework than any other "application program". An actual implementation shows the effectiveness of this approach.

1 Introduction

Currently available rewrite interpreters (e.g. OBJ3 [GKK+87], Redux [Bün93], ELAN [KKM95]) offer a nice programming model and are quite efficient, but lack input/output (I/O) capabilities.

For other models of computation like functional or logic programming, the typical approach has been to add extra features implementing I/O. But these features do not fit in the nice and simple underlying model : in order to be able to understand or reason about programs involving I/O, the basic computational model has to be extended in non trivial ways, making it not so nice and simple anymore (see for instance [Gor94] about the monadic approach to functional I/O). The problem is that functional computation and I/O have two different and a priori incompatible interpretations, one in terms of equality and the other in terms of transitions between states.

In the case of rewriting, the underlying semantic model, called rewriting logic [Mes92,MOM93], allows to combine in a simple way equational computations such as function evaluation or abstract data types and non equational computations such as transitions between states.

This is fine on a theoretical point of view, but a straightforward implementation is not realistic, since it would imply matching modulo an arbitrary big equational theory. We have shown in [Vir95] that an effective implementation is possible in many common cases, by orienting equations of the equational theory into rewrite rules, thus ending up with two sets of rules :

[1] Supported by an HCM fellowship, EuroFOCS Network

the reduction rules (denoted with $\longrightarrow$) describe transitions between states and the equational rules (denoted with $\xrightarrow{=}$) implement the equational theory. We introduced three notions of coherence and proved that they are sufficient conditions for three notions of equivalence to hold between the intended semantics of rewriting logic and the actual implementation. Strong coherence is the strongest property, it implies an exact correspondence between steps by reduction rules, but is difficult to assess in the presence of non linear equational rules (rules where more than one occurrence of a variable appears in the left-hand or right-hand side). Equational coherence is the weakest property, and only implies preservation of normal forms. In the middle, weak coherence implies preservation of derivations.

The coherence properties can be verified by checking critical pairs between rewrite rules, and in the weak and strong cases ensuring that a non linear equational rule can never be applied above a reduction rule. The reader is referred to [Vir95] for more details.

In this paper, We take advantage of this result to design and implement an I/O model for rewrite interpreters that will be *totally explicit* in the *same* framework as any other programs.

The model we propose is based on π-calculus [Mil91], a well studied calculus exhibiting processes exchanging messages.

Process calculi are traditionally described by their transition relation $P \xrightarrow{\alpha} Q$ (the process P is able to "perform the action" α and then behave as Q). Implementing this relation by rewriting is not trivial, since arrows of the transition relation are labelled and rewrite steps are not, and attempts to do so require unnatural tricks in order to take labels into account [MOM93].

However, the transition relation is not what we need. It is useful for understanding process equivalence, based on external observation, but we are interested here in the internal steps of process behaviour, described by the so-called reduction relation. In fact, *it is enough to implement internal transitions steps in order to realize an I/O system*,: an external communication is nothing else than an internal communication in a wider context, and the two relations can be defined in terms of each other (see section 2.4).

In a previous paper [Vir96], we gave a rewriting definition of the reduction relation of π-calculus and formally proved its correctness. We first start with recalling this definition, referring to [Vir96] for formal proofs. We then show how it can be used as a basis for adding input/output capabilities to ELAN [Vit94,KKM95], present our implementation and explain design choices. We finally give some examples of programs using this approach.

The source code of the implementation is available from the author.

2 π-calculus and its reduction relation

2.1 Why π-calculus

When choosing a calculus in which to specify explicitly input-output, we first have to decide on a conceptual model. The typical choice is to consider *processes*, able to evolve independently and to synchronize or exchange data by message passing through designated *channels*. This models fits quite well the intuition of sets of boxes connected between themselves and to the outside by some wires.

But then one may argue that π-calculus is *conceptually* too complex for such a "simple" task, and that either an ad-hoc calculus or a *conceptually* simpler calculus such as CCS with value passing or LOTOS [BB89] may do the job.

The first idea is that a special-purpose calculus expressive enough will anyway exhibit all the complexity of π-calculus, hence it is better to rely on a known calculus whose semantic foundation has been well studied. Now why π-calculus ? Is mobility of processes (the ability to dynamically change a configuration) really needed ?

The second idea is that, although introducing mobility makes a calculus more complex on a semantic point of view, *mobility comes for free* on an operational point of view. In order to *explicitly* model exchange of data, a notion of *name* (corresponding e.g. to the bound variables of λ-calculus) together with a notion of explicit substitution are needed. But then there is no difference between, say, substituting an integer value for a name, and substituting a channel name for another name. Or the other way round, we may say that value-passing comes for free when having mobility.

By acknowledging the fundamental role of names and explicit substitutions, we end up with a simpler calculus, in which mobility and exchange of data are modeled by the *same* set of rules.

2.2 π-calculus with explicit substitutions

In this section and the following, we briefly introduce a rewriting implementation of the reduction relation of π-calculus. More details and proofs of the correspondence results can be found in [Vir96].

In the original definition of π-calculus, substitution is a meta-operation, not part of the calculus. But for an actual implementation the substitution operation has to be made explicit, by introducing a substitution operator and the rules defining it (often referred to as the "substitution calculus").

The terms of π-calculus are usually terms with higher-order variables, considered up to α-conversion, that may be bound by binder operators (note that these higher-order variables are different from term variables, and are represented by constant names).

It is possible to design a substitution calculus for terms with higher-order variables (see e.g. [MOM93]), but the resulting calculus is very inefficient,

mainly because one often needs to check if a variable is free or not. Efficient substitution calculi are based on terms with so-called De Bruijn indices, where an higher-order variable is replaced by an integer indicating how many binders to jump over until finding the binder associated with that variable. The change of representation is transparent as there is a one-to-one relationship between well-formed terms with indices and terms with higher-order variables not containing free names. For instance, in the case of λ-calculus, the term $\lambda x.\lambda y.(xy)$ would be represented as $\lambda_x.\lambda_y.(\underline{\mathbf{1}}_x\underline{\mathbf{0}}_y)$ (the subscripts x and y are not actually part of the term, we add them here and in the following for better readability).

Substitution calculi based on terms with indices are more efficient and very close to actual machine implementations. In fact, the different machines designed for implementing functional reduction can be seen as different strategies of applying the substitution rules [HMP95]. In our implementation, we use a substitution calculus inspired by $\lambda\upsilon$-calculus [LRD94], which is one of the simplest given in the literature. Refer to [Les94] for a survey on the various substitution calculi.

The terms of π-calculus with indices are defined as follows (using a syntax more digestible to rewrite interpreters than the usual one) :

- indices are integers, written always underlined, like $\underline{\mathbf{3}}$ or $\underline{\mathbf{n+1}}$. For better readability, we usually add a subscript with a variable name to both indices and binders, as in $(\nu)_x\mathsf{in}(\underline{\mathbf{0}}_x).\mathsf{nil}$.

- processes

$P :=$	nil	the inactive process
	$\mid$ $(\nu)\,P$	restriction (binder)
	$\mid$ $g.P$	guard (see below)
	$\mid$ $P_1 + P_2$	choice
	$\mid$ $P_1 \mid P_2$	parallel composition
	$\mid$ $!\,P$	replication
	$\mid$ $P\sigma$	substitution (see below)

- guards

$g :=$	$\mathsf{in}(c)$	input on channel c (binder)
	$\mid$ $\mathsf{out}(c, x)$	output x on channel c
	$\mid$ $\mathsf{bout}(x)$	bound output (binder)
	$\mid$ τ	internal choice

  Bound output can be defined in terms of other basic operators ($\mathsf{bout}(x).P = (\nu)_c\mathsf{out}(\underline{\mathbf{0}}_c, x).P$), but we choose to introduce it explicitly for technical reasons.

- substitutions

$$\begin{array}{rcl} \sigma & := & [a/] \\ & | & \Uparrow(\sigma) \\ & | & [\uparrow] \end{array}$$

The intuitive meaning of the substitution operators is to map indices as follows :

$[a/]$	$\Uparrow(s)$	$[\uparrow]$
$\underline{\mathbf{0}} \mapsto a$	$\underline{\mathbf{0}} \mapsto \underline{\mathbf{0}}$	$\underline{\mathbf{0}} \mapsto \underline{\mathbf{1}}$
$\underline{\mathbf{1}} \mapsto \underline{\mathbf{0}}$	$\underline{\mathbf{1}} \mapsto s(\underline{\mathbf{1}})[\uparrow]$	$\underline{\mathbf{1}} \mapsto \underline{\mathbf{2}}$
$\ldots$	$\ldots$	$\ldots$
$\underline{\mathbf{n+1}} \mapsto \underline{\mathbf{n}}$	$\underline{\mathbf{n+1}} \mapsto s(\underline{\mathbf{n}})[\uparrow]$	$\underline{\mathbf{n}} \mapsto \underline{\mathbf{n+1}}$

Processes are considered modulo the equations $\mathsf{AC}(+)$ and $\mathsf{AC}(|)$ (associativity and commutativity of the $+$ and $|$ operators) and the following equational rules :

$$\begin{aligned} P + \mathsf{nil} &\xrightarrow{=} P \\ P \mid \mathsf{nil} &\xrightarrow{=} P \\ !\,P &\xrightarrow{=} P \mid !\,P \\ P \mid (\nu)\,Q &\xrightarrow{=} (\nu)\,(P[\uparrow] \mid Q) \end{aligned}$$

Another set of equational rules deals with the application of explicit substitutions. They are inspired by the rules of $\lambda\upsilon$-calculus [LRD94], extended to take into account the three different binding operators present in π-calculus (refer to [Vir96] for details), There are basically two kind of rules, the congruence rules "pushing down" the substitution operators, and the variable substitution rules.

The last two equational rules allow to replace $|$ or $!$ operators at the top of a process with disjunctions :

The expansion rule. Let $P = \alpha_1.P_1 + ... + \alpha_n.P_n$ and $Q = \beta_1.Q_1 + ... + \beta_m.Q_m$, then

$$P \mid Q \quad \xrightarrow{=} \quad \sum_{i=1...n} \alpha_i.(P_i \mid Q) + \sum_{i=1...m} \beta_i.(P \mid Q_i) + \sum_{\substack{i=1...n \\ j=1...m}} Sync(\alpha_i.P_i, \beta_j.Q_j)$$

where

$$Sync(\alpha_i.P_i, \beta_j.Q_j) = \begin{cases} \tau.(P_i[z/] \mid Q_j) & \text{if } \alpha_i = \mathsf{in}(x) \text{ and } \beta_j = \mathsf{out}(x,z) \\ \tau.(\nu)\,(P_i \mid Q_j) & \text{if } \alpha_i = \mathsf{in}(x) \text{ and } \beta_j = \mathsf{bout}(x) \\ \text{(and similarly by swapping the arguments)} & \\ \mathsf{nil} & \text{in all other cases} \end{cases}$$

The replication rule. Let $P = g_1.P_1 + \cdots + g_n.P_n$, then

$$!\,P \quad \xrightarrow{=} \quad g_1.(P_1 \mid !\,P) + \cdots + g_n.(P_n \mid !\,P)$$

A process is in weak disjunctive normal form if is of the form

$(\nu)\ldots(\nu)(g_1.P_1 + \cdots + g_n.P_n)$

with $n \geq 0$. Using the above equational rules, any process can be put in weak disjunctive normal form.

Expansion and replication are not properly rewriting rules because of the variable n, but can be easily simulated using extra hidden operators.

The rewrite relation defined by the above equational rules (modulo AC) does not terminate because of the replication rule that can be applied repeatedly into its own right-hand side. In order to ensure termination, this latter rule has to be applied only when needed in order to compute a weak disjunctive normal form (see [Vir96] for a precise definition). Let us denote $\Longrightarrow$ the derivations using all the above equational rules (modulo AC) according to that strategy, then we have the following correspondence result :

Proposition 2.1 ([Vir96]) *There is a one-to-one correspondence between processes of the original π-calculus (modulo the usual structural axioms) and normal forms with respect to $\Longrightarrow$.*

In the following, $=$ will denote equivalence of processes modulo $\Longrightarrow$.

2.3 The reduction relation

The reduction relation of π-calculus, written $\longrightarrow$, corresponds to internal transitions of processes (the so-called τ-transitions). It is defined as

$$\tau.P + Q \longrightarrow P \qquad \text{(Choice)}$$

$$\frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q} \qquad \frac{P \longrightarrow P'}{(\nu)\,P \longrightarrow (\nu)\,P'}$$

The Choice rule is not a rewrite rule, since it cannot by applied under any context, but it can be considered a rewrite rule if used together with a strategy that permits its application only under the allowable contexts. Choice is the only non equational rewrite rule (denoted with $\longrightarrow$ rather than $\xrightarrow{=}$) : it is interpreted as an irreversible transition between states, whereas all the above equational rules only compute an equivalent form of a given process.

In the following we denote $\longrightarrow$ the rewrite steps applying Choice considered as a rewrite rule with the appropriate strategy. The correspondence result is as follows :

Proposition 2.2 ([Vir96]) *There is a reduction step $P \longrightarrow Q$ in the original π-calculus if and only if there is a rewrite derivation $P \Longrightarrow\!\longrightarrow Q'$, where Q' is structurally equivalent to Q.*

This result is proved by showing strong coherence between equational and reduction rules and applying the results of [Vir95].

The relations $\Longrightarrow$ and $\longrightarrow$ are based on rewriting modulo AC and can be implemented quite efficiently in ELAN.

2.4 Internal vs. External Communications

The reduction relation of π-calculus describes internal moves of processes but is not able to take into account external communications. However, an external communication between a process P and an environment E (another π-calculus process) is nothing else than an internal communication within the combined process $P \mid E$ (at the top level, ruling out communications within P or within E).

Observation of the external communications of P can be defined as predicates as follows :

Definition 2.3 *The observation predicates are defined as :*

$$P \xrightarrow{\mathsf{in}(c)} P' \textit{ iff. } P = (\nu)...(\nu)(\mathsf{in}(c).P' \mid Q)$$

$$P \xrightarrow{\mathsf{out}(c,x)} P' \textit{ iff. } P = (\nu)...(\nu)(\mathsf{out}(c,x).P' \mid Q)$$

The relevance of this definition is stated by the following property :

Proposition 2.4 *There is a step of the reduction relation $P \longrightarrow Q$ if and only if P is of the form $P = (\nu)...(\nu)(P_1 \mid P_2)$, Q is of the form $Q = (\nu)...(\nu)(Q_1[x/] \mid Q_2)$, with $P_1 \xrightarrow{\mathsf{in}(c)} Q_1$ and $P_2 \xrightarrow{\mathsf{out}(c,x)} Q_2$.*

A process P can communicate with an environment E if $P \xrightarrow{\mathsf{in}(c)} P'$ and $E \xrightarrow{\mathsf{out}(c,x)} E'$, or the opposite. Property 2.4 basically says that there P can communicate with E if and only if there is an internal reduction $P \mid E \longrightarrow P' \mid E'$: any transition can be considered equivalently as an internal transition or an external communication, depending what we consider as the external environment.

The whole implementation of I/O is based on this idea : even for implementing external communication, it is enough to implement internal transitions.

Proof of proposition 2.4

- Only if part :

  Consider $P = (\nu)...(\nu)(P_1 \mid P_2)$. If $P_1 \xrightarrow{\mathsf{in}(c)} Q_1$, then P_1 is of the form $P_1 = \mathsf{in}(c).Q_1 + U_1$. Similarly $P_2 = \mathsf{out}(c,x).Q_2 + U_2$. Then using the expansion rule, we have $P = (\nu)...(\nu)(\tau.(Q_1[x/] \mid Q_2) + \mathsf{in}(c).(Q_1 \mid \mathsf{out}(c,x).Q_2 + U_2) + \mathsf{out}(c,x).(\mathsf{in}(c).Q_1 + U_1 \mid Q_2))$, and the Choice rule applies giving $P \longrightarrow (\nu)...(\nu)(Q_1[x/] \mid Q_2)$.

- If part :

  If $P \longrightarrow Q$, then by definition of the reduction relation and the equation $U \mid (\nu)\, V \xrightarrow{=} (\nu)\, (U[\uparrow] \mid V)$, P must be of the form $P = (\nu)...(\nu)(\tau.Q + V)$. The τ symbol can only be introduced by the expansion rule, hence P is of the form $P = (\nu)...(\nu)(\tau.(Q_1[x/] \mid Q_2) + \mathsf{in}(c).(Q_1 \mid \mathsf{out}(c,x).Q_2 + U_2) + \mathsf{out}(c,x).(\mathsf{in}(c).Q_1 + U_1 \mid Q_2)) = \mathsf{in}(c).Q_1 + U_1 \mid \mathsf{out}(c,x).Q_2 + U_2$, thus $P = P_1 \mid P_2$ with $P_1 \xrightarrow{\mathsf{in}(c)} Q_1$ and $P_2 \xrightarrow{\mathsf{out}(c,x)} Q_2$, and $Q = Q_1 \mid Q_2$.

3 Implementation

ELAN [KKM95] is a rewrite interpreter and compiler developed in Nancy, with a strong emphasis on efficiency. Two of its specific features are of particular interest to us :

- It offers a powerful means of defining strategies, which makes it easy to define the Choice rule as a rewrite rule that can be applied only under some contexts, and to encode the lazy application of the replication rule.
- It has a powerful preprocessor that we use for defining rule schemata for the substitution rules, since they must apply to any variable type.

The relations $\Longrightarrow$ and $\longrightarrow$ are thus easily encoded into ELAN.

3.1 Communication scenario

A process in weak disjunctive normal form exhibits the possible external and internal communications :

$$P = \alpha_1.P_1 + \cdots + \alpha_n.P_n + \tau.Q_1 + \cdots + \tau.Q_n$$

The subterms $\alpha_i.P_i$ offer possible external communications, that can be "performed" if the environment accepts them, namely if the corresponding Unix file descriptors are ready for input or output.

The subterms $\tau.Q_j$ correspond to possible internal choices and can possibly be "selected" by applying the Choice rule.

The problem is what to do with a process P containing both $\alpha_i.P_i$ and $\tau.Q_j$ subterms. We can imagine three scenarios :

(i) Perform an internal transition by applying the Choice rule to one of the $\tau.Q_j$

(ii) Perform an external communication, waiting as long as necessary until one is accepted.

(iii) Check if one of the external communications is accepted, if yes perform it, if no perform an internal transition

In the first case, the problem is that applying the Choice rule may not terminate. There exist so-called divergent processes that can perform infinitely many internal moves.

In the second case, the implementation would not be "fair", in the sense that the program may block indefinitely even if a communication would have been possible after performing an internal move.

The third case avoids both these problems, but raises an issue of efficiency since checking if file descriptors are ready for input or output is a costly operation.

We opted for the first scenario in our implementation, leaving to the user the task of ensuring that there are no divergent processes. This choice is motivated by the feeling that nobody would ever want to design divergent processes, and that we may safely consider this case as an error.

3.2 Actual input/output

Reduction to normal form thus computes a term of the form

$$P = \alpha_1.P_1 + \cdots + \alpha_n.P_n$$

where all the α_i's are input or output guards referring to an external channel.

The selection of a particular external communication among all possible ones (selection of one of the α_i) is implemented by adding a new built-in to ELAN, calling the Unix primitive `select`. Given a set of file descriptors (streams) as arguments, `select` returns the ones that are ready for reading and/or writing.

Effective input/output is then performed by implementing the in and out predicates with the corresponding `read` and `write` Unix system calls. This is done by adding another two new built-in operators in the ELAN source code.

Reduction then proceeds again starting from P_i if $\mathsf{out}(c, x).P_i$ had been selected, or from $P_i[x/]$ if $\mathsf{in}(c).P_i$ had been selected and x is the value read. The whole program stops if the normal form nil is reached, indicating no more possible communication.

3.3 Unix interface

Some adaptations of the program have been necessary in order to cope with "real" input/output :

(i) Since external channels correspond to Unix file descriptors, rather than maintaining a table of associations between π-calculus channels and descriptors, we choose to introduce a special constructor for external channels, $\mathsf{extchan}(i)$, where i is the file descriptor.

This also makes possible the addition of a simplification rule : since an input or output guard on an internal channel may never interact with the outside, we may safely remove processes with such guards from the weak disjunctive normal form computed by $\Longrightarrow$.

(ii) A Unix file descriptor must be opened before its use. We introduced in the calculus a new guard for this purpose, $\mathsf{open}(\text{filename}, \text{type}, \mathrm{P_{Succ}}, \mathrm{P_{Fail}})$, with the intuitive meaning of opening the file whose name is given, then behave as $P_{\mathrm{Succ}}[\mathsf{extchan}(\underline{\mathbf{i}})/]$ if the opening succeeded, binding $\underline{\mathbf{0}}$ with the corresponding channel, or behaving as $P_{\mathrm{Fail}}[i/]$ in case of failure, binding $\underline{\mathbf{0}}$ with a system error number. The type argument is used to convert between ELAN and Unix data representation (for instance ELAN integers may be encoded in Unix files as bit fields of various lengths, or even as their printable representation).

The rules for the expansion and replication theorem must be extended in order to take this new guard into account.

We did not address in this stage of prototype the possibility of closing files, but this would be needed as well for practical applications.

(iii) The semantics of communication in π-calculus and of input-output in Unix are quite different. The former is synchronous and atomic, the latter is asynchronous and may fail in the middle of a transfer.

Asynchronicity is actually not a problem, because it is internal to Unix : the semantic of a communication between a process P and the Unix environment is preserved. Atomicity is guaranteed when reading or writing one byte at a time. It is possible to simulate atomicity for bigger transfers as well, but at the expense of efficiency. A real I/O system should certainly provide a choice between these both options.

Possible failure is more problematic because it does not fit in the model of π-calculus. A possible approach would be to extend input and output guards to allow for failure in a way similar to the open guard, at the expense of simplicity in designing processes. Another more "practically" satisfying approach may be to add a notion of exception handling to the calculus, but this seems a non trivial task. For the moment we simply suppose that such events will make the whole program fail (gossipers note : this is not much different from most commercially available software...)

3.4 Value-passing and typing

So far, we have considered the plain monadic π-calculus. As shown in [Mil91], this calculus is powerful enough to encode any kind of structured data, so the game may end here. However, implementing a value passing calculus based on this encoding loses one of the main advantages of rewriting logic, namely the fact of being able to combine various calculi, using for each domain the more adequate calculus without having to do unnatural encodings.

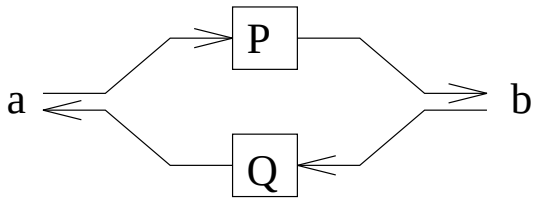
The solution is immediate. Extend the calculus by letting input and output values be not only π-calculus channels, but also any arbitrary data type (indices now range over arbitrary values, including channels). One may for instance write a process $\mathsf{in}(\underline{\mathbf{0}})_x.\mathsf{out}(\__c, 3 + \underline{\mathbf{0}}_x).\mathsf{nil}$, whose behaviour is intuitively "receive an integer value x on a channel c, add it 3 and send the result back on the same channel.

In order to have a conceptual difference between channels and values, some notion of *typing* is called for. This notion of typing should also be able to take into account a possible typing of values.

However, well-typedness cannot be expressed in the many-sorted framework of ELAN, because the type of an index should depend on its value. In our prototype we rely on the user to provide well-typed terms. Classical techniques borrowed from functional programming may be used to guarantee well-typedness.

4 Examples

4.1 Double-way buffer



A double-way buffer consists of two processes in parallel, each of them repeatedly reading a data element from a channel and writing it on the other channel. These processes are most naturally specified using recursive equations :

$$P = \mathsf{in}(a)_x.\mathsf{out}(b, \underline{\mathbf{0}}_x).P$$

$$Q = \mathsf{in}(b)_x.\mathsf{out}(a, \underline{\mathbf{0}}_x).Q$$

We cannot implement directly recursive equations, and need to restate this definition using the replication operator. Note that this is always possible [Mil91] and that the two specifications are weak equivalent (i.e. equivalent up to the internal actions). P and Q are redefined as

$$P \stackrel{def}{=} (\nu)_p(\mathsf{out}(\underline{\mathbf{0}}_p, \mathrm{void}).\mathsf{nil} \mid !\,\mathsf{in}(\underline{\mathbf{0}}_\mathrm{p}).\mathsf{in}(\mathrm{a})_\mathrm{x}.\mathsf{out}(\mathrm{b}, \underline{\mathbf{0}}_\mathrm{x}).\mathsf{out}(\__\mathrm{p}, \mathrm{void}).\mathsf{nil})$$

$$Q \stackrel{def}{=} (\nu)_q(\mathsf{out}(\underline{\mathbf{0}}_q, \mathrm{void}).\mathsf{nil} \mid !\,\mathsf{in}(\underline{\mathbf{0}}_\mathrm{q}).\mathsf{in}(\mathrm{b})_\mathrm{x}.\mathsf{out}(\mathrm{a}, \underline{\mathbf{0}}_\mathrm{x}).\mathsf{out}(\__\mathrm{q}, \mathrm{void}).\mathsf{nil})$$

where void is the only value of the single-valued type of channels that only exchange synchronizations. Intuitively, the process below the replication operator of P (resp. Q) can only be "activated" by an input from channel p (resp. q).

The process $P \mid Q$ reduces to the weak disjunctive normal form

$$(\nu)_p(\nu)_q(\mathsf{in}(a)_x.P' + \mathsf{in}(b)_x.Q')$$

with

$$\begin{aligned} P' = {} & \mathsf{out}(b, \underline{\mathbf{0}}_x).\mathsf{out}(p, void).\mathsf{nil} && (1)\\ \mid {} & \mathsf{in}(b)_x).\mathsf{out}(a, \underline{\mathbf{0}}_x).\mathsf{out}(q, void).\mathsf{nil} && (2)\\ \mid {} & !\,\mathsf{in}(\underline{\mathbf{0}}_p).\mathsf{in}(a)_x.\mathsf{out}(b, \underline{\mathbf{0}}_x).\mathsf{out}(\__p, void).\mathsf{nil} && (3)\\ \mid {} & !\,\mathsf{in}(\underline{\mathbf{0}}_q).\mathsf{in}(a)_x.\mathsf{out}(b, \underline{\mathbf{0}}_x).\mathsf{out}(\__q, void).\mathsf{nil} && (4) \end{aligned}$$

and similarly for Q' by swapping p with q and a with b.

The normal form exhibits the two possible external communications $\mathsf{in}(a)$ and $\mathsf{in}(b)$. As soon as one of them is possible, the communication takes place and the computation proceeds with either P' or Q'. In P', term (1) is the continuation of the buffer process P, term (2) is the buffer process Q, and terms (3) and (4) are the "pools" of processes that are activated by an input on channel p or q.

The two-way buffer example can be trivially extended by adding compu-

tation of the output values, for instance

$$P = \mathsf{in}(a)_x.\mathsf{out}(b, f(\underline{\mathbf{0}}_x)).P$$

$$Q = \mathsf{in}(b)_x.\mathsf{out}(a, g(\underline{\mathbf{0}}_x)).P$$

where f and g are functions defined by rewrite rules. Since these defined operators appear only strictly below process operators, we are guaranteed that strong coherence is preserved [Vir95] and thus that our implementation remains correct.

4.2 Filter

The previous example exhibits the typical programming style of our approach : processes exchanging data, possibly computing output values with defined functions. But process expressions may also appear below defined symbols, as long as no non-linear rewrite rule may ever be applied above a process expression, in order to preserve strong coherence [Vir95].

This is the case for instance with the $if...then...else...$ operator, defined by the following linear rules

$$\text{if true then } x \text{ else } y \longrightarrow x$$

$$\text{if false then } x \text{ else } y \longrightarrow y$$

Then we can write a FILTER process, that repeatedly inputs values on a channel i and copies them on an input channel o only when they verify a given condition :

$$\text{FILTER} = \mathsf{in}(i)_x.\text{if } c(x) \text{ then } \mathsf{out}(o, \underline{\mathbf{0}}_x).\text{FILTER else FILTER}$$

This recursive definition is then restated using the replication operator as in the previous example.

This possibility allows for a more "natural" programming style, for instance closer to CSP/Occam [Hoa78], but the constructs that may appear above processes must be clearly identified in order to ensure the condition about non linear rules.

These constructs may also be the constructors of data types, and we may be able for instance to specify in a unique framework a system of windows each running its own independent process.

5 Conclusion

Starting from a rewriting definition of the reduction relation of π-calculus, we have designed an input/output system for the ELAN rewriting interpreter that is totally explicit in the rewriting framework itself and integrates smoothly with any other "application program".

This system has been implemented in ELAN to show its effectiveness. An important issue yet to be checked is the strategy used for applying the substitution rules. Applying them eagerly is hopelessly inefficient, but particular

strategies correspond to different types of known abstract machines [HMP95] and can achieve the same efficiency once compiled.

We plan to add this system to a future ELAN distribution, and hope that adding I/O capabilities to rewrite interpreters will make these systems very attractive.

References

[BB89] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. In P. H. J. van Eijk, C. A. Vissers, and M. Diaz, editors, *The Formal Description Technique LOTOS*, pages 23–73. Elsevier Science Publishers B. V. (North-Holland), 1989.

[Bün93] R. Bündgen. Reduce the redex $\rightarrow$ ReDuX. In *Proceedings 5th Conference on Rewriting Techniques and Applications, Montreal (Canada)*, number 690 in LNCS, pages 446–450. Springer-Verlag, 1993.

[GKK+87] J. A. Goguen, Claude Kirchner, Hélène Kirchner, A. Mégrelis, J. Meseguer, and T. Winkler. An introduction to OBJ-3. In J.-P. Jouannaud and S. Kaplan, editors, *Proceedings 1st International Workshop on Conditional Term Rewriting Systems, Orsay (France)*, volume 308 of *LNCS*, pages 258–263. Springer-Verlag, July 1987. Also as internal report CRIN: 88-R-001.

[Gor94] A. Gordon. *Functional Programming and Input/Output*. Distinguished Dissertations in Computer Science. Cambridge University Press, 1994. ISBN 0 521 47103.

[HMP95] T. Hardin, L. Maranget, and B. Pagano. Functional back-ends within the weak lambda-sigma-calculus. In *Procs. of Workshop on the Implementation of Functional Languages*, Sept. 1995.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[KKM95] C. Kirchner, H. Kirchner, and M. Vittek. Designing CLP using computational systems. In P. Van Hentenryck and S. Saraswat, editors, *Principles and Practice of Constraint Programming*. The MIT press, 1995.

[Les94] P. Lescanne. From $\lambda\sigma$ to $\lambda\upsilon$, a journey through calculi of explicit substitutions. In Hans Boehm, editor, *Proceedings of the 21st Annual ACM Symposium on Principles Of Programming Languages, Portland (Or., USA)*, pages 60–69. ACM, 1994.

[LRD94] P. Lescanne and J. Rouyer-Degli. The calculus of explicit substitutions $\lambda\upsilon$. Technical Report RR-2222, INRIA-Lorraine, January 1994.

[Mes92] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[Mil91] R. Milner. The polyadic π-calculus: a tutorial. Technical Report ECS-LFCS-91-180, LFCS, University of Edinburgh, 1991.

[MOM93] N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. Technical Report CSL-93-05, SRI International, 1993.

[Vir95] P. Viry. Rewriting modulo a rewrite system. Technical Report TR-95-20, Dipartimento di Informatica, Università di Pisa, 1995.

[Vir96] P. Viry. A rewriting implementation of π-calculus. Technical Report TR-96-29, Dipartimento di Informatica, Università di Pisa, 1996.

[Vit94] Marian Vittek. *ELAN: Un cadre logique pour le prototypage de langages de programmation avec contraintes.* Thèse de Doctorat d'Université, Université Henri Poincaré - Nancy 1, October 1994.