

Certifying confluence in a timed process calculus

Mehdi Dogguy¹ and Stéphane Glondu¹

Laboratoire PPS
Université Paris Diderot

Abstract. We describe a type system for confluent processes using affine usages in a new timed variant of the π -calculus called TAPIS. In any environment, typable programs are guaranteed to have the same behaviour. The operational semantics of the calculus and the proof of correctness of the type system are formalised and proved in Coq.

1 Introduction

Synchronous (or timed) calculi are used to design and implement critical or embedded systems where programmers have to provide a number of formal guarantees on programs' behaviour. In synchronous systems, a global clock organises computations. It defines implicitly a notion of *instant* (or *round* or *phase*...). Each process performs an operation and synchronises with other processes. In that sense, processes evolve at the same speed.

In synchronous systems, computations are driven by the environment. Each process repeatedly interacts with the latter to evolve. Analysing the behaviour of such concurrent processes is not an easy task. The problem becomes even harder when they are not deterministic.

In the present work, we focus on the study of confluence in a synchronous setting; confluence being a technical property that entails determinacy. An early study of confluence in the context of CCS can be found in [Mil89]. More recently, several authors have considered the problem of guaranteeing confluence for models with richer communication mechanisms (values, name passing) by means of typing systems. For instance, [NS97] introduces a type system for “port-uniqueness” that maintains the invariant that all communication channels are point-to-point (one receiver and one sender). In a related direction, [KPT99] introduces a notion of *linear channel* in the context of the π -calculus. In previous work [AD08], the first author has developed a type system with *usages* which guarantees confluence and determinism for a *synchronous* π -calculus whose communication mechanism is based on signals.

As a first contribution, in the present work we introduce a similar type system for a *timed* π -calculus whose communication mechanism is based on ‘asynchronous’ channels. We call the resulting system TAPIS for *timed*, *asynchronous*, π -calculus.

¹ `firstname.surname@pps.jussieu.fr`

The main step towards proving confluence is the so called subject reduction property that states that typing is invariant under reduction. The proof of this property in type systems enriched with usages is surprisingly technical. This state of affairs motivated us to work on a full formalisation of the proof in the Coq proof assistant [CDT10]. This required a one man-year effort and resulted in a 12 Kloc Coq development which is available online [DG11]. This is our second and main contribution. The two main novelties in this development concern the representation in Coq of the TAPIS processes and of the type system with usages. Early work on representing π -calculus in HOL is described in [Moh95] and mimicks a textbook representation. Following work in Coq [Hir97] has adopted a De Bruijn representation of variables. In the present work, we will follow a *locally nameless* approach [ACP⁺08] where bound variables are represented by De Bruijn indexes while free variables are represented by ordinary names. To the best of our knowledge, this is the first time this approach is applied to a language of the π -calculus family.

The proof of the subject reduction property for type systems has become a standard benchmark in the formal treatment of programming languages [PSWZ05]. Several authors have addressed this problem for various calculi, perhaps the examples which are the closest to our work are [HG99, Des00] where following [Hir97] variables are represented in De Bruijn notation and subject reduction is proven for a simple type system for the π -calculus. As far as we know, we provide the first formalisation of the subject reduction property for a type system with usages. Roughly, types with usages are types indexed over an algebra with a partially defined associative, commutative operation. As we will see in section 3.1, this raises non-trivial representation problems.

The rest of the paper is organised as follows. Section 2 introduces TAPIS. Section 3 gives a detailed description of the type system. Finally, section 4 presents the main steps to reach the subjection reduction property.

2 TAPIS

This section gives a detailed description of the syntax and semantics of TAPIS.

We assume the reader is familiar with TCCS [Yi91] which is a timed version of CCS [Mil89] where *time passes exactly when no internal computation is possible*. The same notion of time has been presented in ESTEREL [BG92] and in derived languages such as SL [BS96]. These two languages belong to the family of so called *synchronous* languages and they differ in the way programs can react to the absence of a signal. In ESTEREL, the reaction is immediate while in SL the reaction is at the end of the instant.

All the formalisms mentioned above ((T)CCS, ESTEREL, SL) are based on a communication model where processes synchronise on channels or signals without exchanging values. In [Ama07], the SL model has been extended to include value and name passing in the style of the π -calculus. TAPIS pursues a similar goal starting from TCCS (thus communication is based on channels rather than on signals).

2.1 Syntax

In TAPIS, as in TCCS, we distinguish processes whose behaviour is time-dependent and those which are not. The former rely on a *else-next* operator which is written as $a(x).P \triangleright Q$. This is the process that tries to get an input on channel a during the current instant. If it succeeds, P is launched; otherwise Q is started during the next instant.

The syntax of TAPIS consists of terms and processes and it is defined in Coq as follows:

```

Inductive term :=
| Tindex : index -> term
| Tname  : name -> term
| Tconstr : constr -> list term -> term.
Inductive process :=
| Pnull : process
| Ppar  : process -> process -> process
| Psend : term -> term -> process
| Precv : term -> process -> process
| Precv_elsenext : term -> process -> process -> process
| Precv_dup : term -> process -> process
| Precv_dup_elsenext : term -> process -> process -> process
| Pnu : process -> process
| Pmatch : term -> constr -> nat -> process -> process -> process.

```

A key point in the formalisation of TAPIS is the the representation of the variables. In this work we rely on the *locally nameless* approach, introduced by McKinna, Pollack, Gordon and McBride [MM04, MP99, Gor94]. It consists of using De Bruijn indices for bound variables (**Tindex**) and names for free variables (**Tname**). This technique avoids most problems related to alpha-equivalence since alpha-equivalent terms have syntactically equal representations.

As described in [ACP⁺08], the two basic operations defined over terms and processes are **open** and **close**.

open Takes a De Bruijn index k , two terms u and t and replaces occurrences of k (if any) in t with u . We will note $t@u$ when k is 0, and $t@_k u$ otherwise.
close This does the opposite. It replaces the name z in t with a De Bruijn index k . We will note t^z when k is 0, and $t^k z$ otherwise.

Next, we describe informally the behaviour of programs: 0 (also written **Pnull** in Coq) is the terminated program; $P \mid Q$ (or **Ppar** P Q , $P \parallel Q$) runs P and Q in parallel; $\nu a.P$ (or **Pnu** P) creates a new channel a bound in P and continues as P ; $[t \sqsubseteq p]P_1, P_2$ (or **Pmatch** t k n P_1 P_2 , where $p = k(x_1, \dots, x_n)$, k is a constructor name, and $x_1, \dots, x_n$ are bound in P_1) matches the term t against the pattern p . It continues as θP_1 if there exists a substitution θ matching t against p (written $\text{match}(t, p) = \theta$), and P_2 if no such substitution exists (written $\text{match}(t, p) = \uparrow$); $\bar{a}\langle v \rangle$ (or **Psend** a v) sends a value v over channel a during the instant, and then dies. Finally, the input operator can be categorised according to two criteria: (i) whether it is replicated, and (ii) whether it is time-dependent.

- $a(x).P$ (or `Precv a P`) upon receiving a message on channel a it bounds it to the formal parameter x and proceeds as P . On the other hand, $a(x).P \triangleright Q$ (or `Precv_elsenext a P Q`) is the *else-next* operator described above whose behaviour is time-dependent.
- $!a(x).P$ (or `Precv_dup a P`) is the standard replicated input guarded process. In TAPIS, its behaviour does not depend on time. It offers a service on the channel a , and it is present during all instants an unbounded number of times. On the other hand, $a(x).P ! \triangleright Q$ (or `Precv_dup_elsenext a P Q`) is a time-dependent variant which behaves as the replicated input during the first instant and as Q in the following ones.

We notice that in an untyped language, the last input operator introduced $a(x).P ! \triangleright Q$ can be derived from the replicated input and the *else-next* operator. Indeed, recalling that the replicated input allows to simulate recursive definitions, we could define:

$$a(x).P ! \triangleright Q \stackrel{def}{=} \overline{A} \langle \rangle \mid !A.(\overline{A} \langle \rangle \mid a(x).P \triangleright Q)$$

Unfortunately, the typing of this encoding to guarantee confluence leads to complications. This is the reason we adopt the operator $_ ! \triangleright _$ as a primitive one.

Example 1 (Clock/Alarm). First, we define *pause.P* as an abbreviation for the process $\nu a a(x).0 \triangleright P$. This is the process that suspends in the first instant and then behaves as P . Then, we provide an example of a *clock* process which offers a service to which any other process may subscribe. Each subscriber sending a channel name x on the channel *tac* will receive one message on x , at every instant, and starting from the following instant.

$$!tac(x).pause.(\overline{tac} \langle x \rangle \mid \overline{x} \langle \star \rangle)$$

In Coq, this could be written as follows:

```
let x := Tindex 0 in
let pause P := Pnu (Precv_elsenext x Pnull P) in
let star := Tconstr 0 nil in
let tac := Tname 0 in
Precv_dup tac (pause (Psend tac x // Psend x star))
```

In the Coq development, terms and processes share similar definitions and properties. To cope with these two syntactic categories, we rely on type classes as described in [SO08]. For example, some properties of the `open` and `close` operations mentioned above are gathered in a type class:

```
Class subst_props A B (* ... *) := {
  subst_close_fresh : forall (t:B) (x:name),
    x ∉ FV t -> close x t = t;
  subst_close_open : forall (x:name) (t:B),
    x ∉ FV t -> close x (t @ x) = t;
```

```

subst_fresh : forall (x:name) u (t:B),
  x ∉ FV t -> t[x:=u] = t;
subst_open : forall (x:name) (u:A) (t:B) (w:A),
  locally_closed u ->
    (open t w)[x := u] = open (t[x:=u]) (w[x:=u]);
subst_open_var : forall (x:name) (u:A) (t:B) (y:name),
  locally_closed u -> x <> y ->
    (open t (Tname y))[x := u] = open (t[x := u]) (Tname y);
subst_intro : forall (x:name) (u:A) (t:B),
  locally_closed u -> x ∉ FV t ->
    (open t (Tname x))[x := u] = open t u
}.

```

The attentive reader will notice the overloading of names and notations.

2.2 Equivalence relation

The semantics of the calculus uses an equivalence relation $\equiv$ (named `peq` in Coq) on processes that is defined in table 1.

$P \mid 0 \equiv P$	$P \mid Q \equiv Q \mid P$	$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$
$[\star]$	$(\nu a.P) \mid Q \equiv \nu a.(P \mid Q)$	if $a \notin fn(Q)$
$\nu a. \nu b. P \equiv \nu b. \nu a. P$		

Table 1. Equivalence relation on processes

The scope extrusion rule (marked by $[\star]$ in table 1) is translated in Coq in the following way:

$$\text{Pnu } P \text{ // } Q \equiv \text{Pnu } (P \text{ // lift } Q)$$

This is the only rule that uses the lift operation on De Bruijn indices. It might look unnatural at first sight but it is rather simple and can be explained easily. We assume that processes are well-formed (no orphaned De Bruijn indices). Getting Q under Pnu is simple since Q is supposed to be well-formed. Extracting Q from within Pnu is slightly more complicated since we have to make sure that bound variables are not mixed, which is taken care of by using `lift` Q . In a pure locally nameless way, one can combine `open` and `close` for different De Bruijn indices to get the desired result. However, this is more complicated than the `lift` solution.

2.3 Semantics

The semantics of TAPIS is defined by means of two transition systems which are described in table 2. The first one is $\rightarrow$ (written $\dashrightarrow$ in Coq) and defines the computations during the instant. The second one is $\xrightarrow{tick}$ (written $\dashrightarrow$ in Coq) and defines how the computation moves from one instant to the following one. Informally, the effect of the $\xrightarrow{tick}$ transition is to kill outputs, to re-schedule time-independent processes, and to take the alternative branches of the time-dependent input processes.

$(\text{par}) \frac{P \rightarrow P'}{P \mid Q \rightarrow P' \mid Q}$	$(\nu) \frac{P \rightarrow P'}{\nu a \ P \rightarrow \nu a \ P'}$
$(=^{\text{ind}}_1) \frac{\text{match}(t, p) = \theta}{[t \sqsubseteq p] P_1, P_2 \rightarrow \theta P_1}$	$(=^{\text{ind}}_2) \frac{\text{match}(t, p) = \uparrow}{[t \sqsubseteq p] P_1, P_2 \rightarrow P_2}$
$(\equiv) \frac{P \equiv P' \quad P' \rightarrow Q' \quad Q' \equiv Q}{P \rightarrow Q}$	
$(\text{sync}_1) \frac{}{a(x).P \mid \bar{a}\langle t \rangle \rightarrow P[t/x]}$	$(\text{sync}_2) \frac{}{a(x).P \triangleright Q \mid \bar{a}\langle t \rangle \rightarrow P[t/x]}$
$(!_1) \frac{}{a(x).P ! \triangleright Q \mid \bar{a}\langle t \rangle \rightarrow a(x).P ! \triangleright Q \mid P[t/x]}$	
$(!_2) \frac{}{!a(x).P \mid \bar{a}\langle t \rangle \rightarrow !a(x).P \mid P[t/x]}$	
$(\triangleright_{\text{in}}) \frac{}{a(x).P \xrightarrow{tick} a(x).P}$	$(\triangleright_{\text{else}}) \frac{}{a(x).P \triangleright Q \xrightarrow{tick} Q}$
$(\triangleright_{!}) \frac{}{!a(x).P \xrightarrow{tick} !a(x).P}$	$(\triangleright_{\text{rec}}) \frac{}{a(x).P ! \triangleright Q \xrightarrow{tick} Q}$
$(\triangleright_{\nu}) \frac{P \xrightarrow{tick} P'}{\nu a \ P \xrightarrow{tick} \nu a \ P'}$	$(\triangleright_{\text{out}}) \frac{}{\bar{a}\langle t \rangle \xrightarrow{tick} 0}$
$(\triangleright_{\text{par}}) \frac{P \xrightarrow{tick} P' \quad Q \xrightarrow{tick} Q'}{P \mid Q \not\rightarrow \cdot}$	$\frac{}{P \mid Q \xrightarrow{tick} P' \mid Q'}$

Table 2. TAPIS's transition system

The formalisation in Coq of this transition system uses the locally nameless approach. For example, the transition rule (ν) is written in the following way:

$$(\nu) \frac{\forall x \notin L \quad P@x \rightarrow P'@x}{\text{Pnu } P \rightarrow \text{Pnu } P'}$$

The reader may notice the universal quantification over some finite set of names L . The technique is known as *co-finite quantification* [ACP⁺08]. Intuitively, the set L represents all known names (*e.g.* free names of the considered term/process). In the rule (ν) , the fact that x is *fresh* is formalised by the condition $x \notin L$. Apart from other technical transformations, the same remark applies to any transition rule where the topmost constructor is a binder. Appendix A provides a full description of the transition systems which is closer to the Coq formalisation.

3 Type system

In this section, we introduce a type system for TAPIS which, following the work in [AD08], is intended to guarantee confluence, and therefore the determinism, of typable processes.

3.1 Additions

The type system uses several partial, associative and commutative, binary operations, called *additions*. We are not aware of a canonical way to handle such definitions in Coq. Three approaches were considered (for additions on, *e.g.*, type A):

- binary functions returning an optional value (*i.e.* with type $A \rightarrow A \rightarrow \text{option } A$);
- ternary relations (*i.e.* with type $A \rightarrow A \rightarrow A \rightarrow \text{Prop}$);
- binary functions operating on optional values (*i.e.* with type $\text{option } A \rightarrow \text{option } A \rightarrow \text{option } A$).

The first solution seems the most natural for a partial function, but the asymmetric inputs and output make it difficult to compose. In particular, to express associativity, one needs to explicitly name intermediate results, leading to awkward statements like

```
forall x y z r1 r2,
  f x y = Some r1 /\ f r1 z = Some r2 ->
  exists r3, f y z = Some r3 /\ f x r3 = Some r2
```

which are difficult to manipulate. Besides, the adequacy of such definitions with respect to “informal” definitions given on paper is not immediate, and they usually cannot be used directly in proofs (many auxiliary lemmas are needed).

The second solution, when implemented with inductive predicates, leads to definitions closer to the declarative ones on paper, and gives access to some

frequent reasoning patterns for free thanks to Coq’s support for inductive types (such as the `induction` and `inversion` tactics). However, this solution, like the first one, does not lead to easily composable reasoning and worse, it needs explicit proofs of functionality (*i.e.* that there is at most one z such that $x + y = z$).

The third solution—adopted in our development—is functional and solves the composability issue. This allows the use of regular techniques of reasoning on associative and commutative operations in Coq [BP10]. However, it allows the definition of “exotic” functions that do not correspond to any partial binary operator. These exotic possibilities can be ruled out by proving that `None` is an absorbing element. All the additions described in the following sections are made instances of a type class requiring associativity, commutativity and absorbance that allow the use of generic names and notations ($\oplus$ and $+$). A generic proof of equivalence with functional ternary relations is also provided.

3.2 Usages

Usages may be regarded as the set of granted actions over channels and exchanged data. Formally, a *usage* is an element of the set $L = \{0, 1, \infty\}$ where each element has a special meaning. One cannot perform any actions on resources with usage 0; 1 means that such resource might be used at most once; ∞ means an arbitrary number of uses. Further details about usages can be found in [AD08].

Channel usages during the instant are defined over L^2 . The first component represents available resources for outputs, while the second one is for inputs. Among all the possibilities, there are only two usage families that are interesting in TAPIS. They are represented in figure 1. The rest do not ensure deterministic behaviour or are not very useful (*e.g.* $(1, \infty)$ is not very useful).

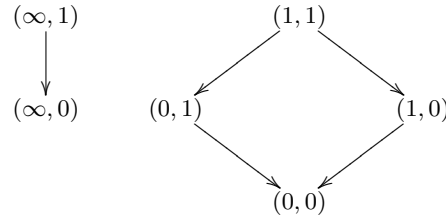


Fig. 1. Usage families

Usages are represented in Coq using three constructs:

- an inductive type for each usage kind :

```
Inductive uk1 := U81 | U80 .
Inductive uk2 := U11 | U01 | U10 | U00 .
```

- an inductive type to represent *usage families*:

```
Inductive usage_family := UK1 | UK2.
```

- a dependent type:

```
Definition usage x := match x with
| UK1 => uk1
| UK2 => uk2
end.
```

`usage` is used in the type of usage-related functions and lemmas. Using dependent types in this situation allows more conciseness and flexibility. Thanks to `usage`, one can write the following to express some definition for both usage kinds:

```
Definition foo {uk} (x y : usage uk) := ...
```

Usages are classified according to various criteria. They are said to be *affine* if they contain a 1, and *neutral* otherwise. *Channel usages* may be classified according to an additional criterion: $u \cdot v^\omega$ is *uniform* when $u = v$, and *non-uniform* otherwise. And for any usage u , `affine`(u) holds if u is affine, `neutral`(u) if it is neutral and `uniform`(u) if it is uniform.

We add the elements of L with a partial operation $\oplus$ such that $0 \oplus u = u \oplus 0 = u$ and $\infty \oplus \infty = \infty$, and which is undefined otherwise (note in particular that $1 \oplus 1$ is undefined). The addition induces an order by $a \leq b$ if there exists c such that $a \oplus c = b$. With respect to this order, 0 is the least element while 1 and ∞ are incomparable. If $a \leq b$ then we define a subtraction operation $b \ominus a$ as the largest c such that $b = a \oplus c$. Therefore, $a \ominus 0 = a$, $1 \ominus 1 = 0$ and $\infty \ominus \infty = \infty$. We also define the *shift* operation, denoted by $\uparrow$, which takes off the first component of a usage when it is not uniform (*i.e.* $\uparrow(u \cdot v^\omega) = v^\omega$ and $\uparrow(u) = u^\omega$).

3.3 Types

Types are organised in levels as described in table 3. Each level is implemented as an inductive type in Coq. T types are user-defined types and are introduced in a *signature* (`envi`), *i.e.* a structure describing all user-defined inductive types and assumed properties on them. The structure is as follows:

- `envi_raw`: a list of user types that stores the following information for each one of them:
 - `data usage UT_usage` $\in \{1, \infty\}$. Types declared with usage 0 cannot be used at all. Therefore, this possibility has not been considered here. The usage indicates how many times a value of this type can be shared, where 1 means *at most once* and ∞ an *arbitrary number of times*.
 - `UT_args` is the list of arguments of the type being defined.
 - `UT_constructors` is the list of constructors with the types of their corresponding arguments. Each constructors is then identified with their position in the list.

- some assumed properties on `envi_raw`:
 - *Affine user-types use only affine or neutral types.* The reverse is not true.

```

envi_wf0 : forall T,
  In T envi_raw ->
  UT_usage T = U1 ->
  (forall t, In t (UT_constructors T) ->
    Forall affine t)
  /\ Forall affine (UT_args T);

```

- *Neutral user-types use only neutral types.*

```

envi_wf1 : forall T,
  In T envi_raw ->
  UT_usage T = U8 ->
  (forall t, In t (UT_constructors T) ->
    Forall neutral t)
  /\ Forall neutral (UT_args T);

```

- *User-types use only uniform types.*

```

envi_wf2 : forall T,
  In T envi_raw ->
  (forall t, In t (UT_constructors T) ->
    Forall uniform t)
  /\ Forall uniform (UT_args T)

```

The partial operation of addition $\oplus$ is extended to types so that:

$$C_u(\sigma) \oplus C_v(\sigma) = C_{u \oplus v}(\sigma)$$

where $C \in \{Ch, T\}$ and provided $u \oplus v$ is defined.

$\kappa ::= T_\infty(\kappa_1, \dots, \kappa_n) \mid Ch_{u^\omega}(\kappa) \mid 1$	Neutral types (u is neutral)
$\lambda ::= T_1(\sigma_1, \dots, \sigma_n) \mid Ch_v^\omega(\sigma)$	Affine types (u is affine)
$\sigma ::= \kappa \mid \lambda$	Uniform types

Table 3. Types

Predicates `affine`, `neutral` and `uniform` are extended to types in a natural way. The shift operation $\uparrow$ is also extended to types: it is the identity for all types except for channel types where it shifts their usage in a non-recursive way.

3.4 Typing rules

Typing contexts (or environments) are finite maps from variable names to types. The development is built on top of the generic finite map interface of Coq's standard library. Additions on types are extended to environments in the following

fashion:

$$(\Gamma_1 \oplus \Gamma_2)(x) = \begin{cases} \Gamma_1(x) & \text{if } \Gamma_2(x) \text{ is undefined,} \\ \Gamma_2(x) & \text{if } \Gamma_1(x) \text{ is undefined,} \\ \Gamma_1(x) \oplus \Gamma_2(x) & \text{otherwise.} \end{cases}$$

Note that $\Gamma_1(x) \oplus \Gamma_2(x)$ may be undefined, and so is $(\Gamma_1 \oplus \Gamma_2)(x)$ in this case.

Predicates **affine**, **neutral** and **uniform** are also extended to typing contexts. The predicate holds if and only if it holds for each type bound in the context. The shift operation ($\uparrow$) is extended to typing contexts as well. It shifts every type bound in the context.

The typing rules are described in table 4. They make sure that processes do not use more resources than what is actually allowed by usages. Each generated channel name comes with its type. We derive judgements of the form:

- $\Gamma \vdash P$, where Γ is a typing context that maps variable names to their types and P is a process.
- $\Gamma \vdash t : s$, where Γ is a typing context, t a term and s a type.

The typing context Γ describes which actions P is allowed to perform. Confluence of typed programs is guaranteed by allowing the following situations only:

- sending at most one value and receiving at most once on the same channel during any instant;
- when a process offers a service, the latter should be unique and accepts an arbitrary number of clients.

Typing rules ($\triangleright_{\text{in}}$), (in), ($!_{\text{in}}$) and ($! \triangleright_{\text{in}}$) describe requirements for input actions.

4 Subject Reduction

The main result proved in Coq and presented here is the *subject reduction* property, *i.e.*, the fact that typing defined in table 4 is preserved by the transition relations described in table 2. In the following we outline the main lemmas needed to reach the result.

The formalisation in Coq of the type system requires that terms and processes are *locally closed*. In particular, we want that any typable object is free of orphaned De Bruijn indices and that free names are bound in the typing context. The following lemmas ensure that these properties hold.

Lemma `typ_term_locally_closed` : forall Γ t s ,
 $\Gamma \vdash t : s \rightarrow \text{locally_closed } t$.

Lemma `typ_process_locally_closed` : forall Γ P ,
 $\Gamma \vdash P \rightarrow \text{locally_closed } P$.

Both lemmas are proved by induction on the typing judgement.

Sometimes, one has to go from one typing derivation for a single variable x to another one for a co-finite set of variables. In those situations, renaming lemmas come to the rescue.

Processes

$(0) \frac{}{\Gamma \vdash 0}$	$(\nu) \frac{\Gamma, a : Ch_{u^\omega}(\sigma) \vdash P \quad \mathbf{uniform}(Ch_{u^\omega}(\sigma))}{\Gamma \vdash \nu a : Ch_{u^\omega}(\sigma) P}$
$(\text{par}) \frac{\Gamma_i \vdash P_i \quad i = 1, 2}{\Gamma_1 \oplus \Gamma_2 \vdash P_1 \mid P_2}$	$(\text{out}) \frac{\Gamma_a \vdash a : Ch_{u \cdot v^\omega}(\sigma) \quad \Gamma_e \vdash e : \sigma \quad u(1) > 0}{\Gamma_a \oplus \Gamma_e \vdash \bar{a}(e)}$
$(\triangleright_{\text{in}}) \frac{\Gamma_a \vdash a : Ch_{u \cdot v^\omega}(\sigma) \quad \Gamma_P, x : \sigma \vdash P \quad u = (0, 1) \quad \uparrow (\Gamma_a \oplus \Gamma_P) \vdash Q}{\Gamma_a \oplus \Gamma_P \vdash a(x).P \triangleright Q}$	$(\text{in}) \frac{\Gamma_a \vdash a : Ch_{u^\omega}(\sigma) \quad \Gamma_P, x : \sigma \vdash P \quad u = (0, 1) \quad \mathbf{uniform}(\Gamma_P, x : \sigma)}{\Gamma_a \oplus \Gamma_P \vdash a(x).P}$
$(! \triangleright_{\text{in}}) \frac{\Gamma_a \vdash a : Ch_{u \cdot v^\omega}(\sigma) \quad \Gamma_P, x : \sigma \vdash P \quad u = (\infty, 1) \quad \uparrow \Gamma_Q \vdash Q \quad \mathbf{neutral}(\Gamma_P)}{\Gamma_a \oplus \Gamma_P \oplus \Gamma_Q \vdash a(x).P ! \triangleright Q}$	$(!_{\text{in}}) \frac{\Gamma_a \vdash a : Ch_{u^\omega}(\sigma) \quad \Gamma_P, x : \sigma \vdash P \quad u = (\infty, 1) \quad \mathbf{uniform}(\sigma) \quad \mathbf{neutral}(\Gamma_P)}{\Gamma_a \oplus \Gamma_P \vdash !a(x).P}$
$(\text{m}_c) \frac{\mathbf{k} : (\sigma_1, \dots, \sigma_n) \rightarrow T_u(\sigma_1, \dots, \sigma_m) \quad \Gamma_1 \vdash e : T_u(\sigma_1, \dots, \sigma_m) \quad \Gamma_2, x_1 : \sigma_1, \dots, x_n : \sigma_n \vdash P_1 \quad \Gamma_1 \oplus \Gamma_2 \vdash P_2}{\Gamma_1 \oplus \Gamma_2 \vdash [e \sqsubseteq \mathbf{k}(x_1, \dots, x_n)]P_1, P_2}$	

Terms

$(\text{var}_c) \frac{}{\Gamma, x : T_u(\sigma) \vdash x : T_u(\sigma)}$	$(c) \frac{\Gamma_i \vdash e_i : \sigma_i \quad i = 1, \dots, n \quad \mathbf{k} : (\sigma_1, \dots, \sigma_n) \rightarrow T_u(\sigma_1, \dots, \sigma_m)}{\Gamma_0 \oplus \Gamma_1 \oplus \dots \oplus \Gamma_n \vdash k(e_1, \dots, e_n) : T_u(\sigma_1, \dots, \sigma_m)}$
$(\text{var}_{\text{ch}}) \frac{\Gamma_1(c) = Ch_{u \cdot v^\omega}(\sigma)}{\Gamma_1 \oplus \Gamma_2 \vdash c : Ch_{u \cdot v^\omega}(\sigma)}$	

Table 4. Typing rules

```

Lemma typ_term_renaming : forall (t:term) n  $\Gamma$  s ty,
  forall x, x  $\notin$  dom  $\Gamma$  -> x  $\notin$  FV t ->  $\Gamma, x:s \vdash (t @ x)$  ty ->
  forall y, y  $\notin$  dom  $\Gamma$  -> y  $\notin$  FV t ->  $\Gamma, y:s \vdash (t @ y)$  ty.
Lemma typ_renaming : forall  $\Gamma$  s (P:process),
  forall x, x  $\notin$  dom  $\Gamma$  -> x  $\notin$  FV P ->  $\Gamma, x:s \vdash P @ x$  ->
  forall y, y  $\notin$  dom  $\Gamma$  -> y  $\notin$  FV P ->  $\Gamma, y:s \vdash P @ y$ .

```

Before reaching the main result, one needs to prove a *substitution lemma*, i.e. the fact that the typing of a term or a process is preserved when replacing a name with a term which has the same type.

```

Lemma open_replace :
  forall (p:nat), forall P, p_size P <= p ->
  forall  $\Gamma_1$   $\Gamma_2$   $\Gamma$  t k n s,
    n  $\notin$  dom  $\Gamma$  ->
    n  $\notin$  FV P ->
     $\Gamma_1, n:s \vdash (\text{process\_open } k \text{ (Tname } n) P)$  ->
    ! $\Gamma_1$  + ! $\Gamma_2$  == ! $\Gamma$  ->
     $\Gamma_2 \vdash t : s$  ->
     $\Gamma \vdash (\text{process\_open } k \text{ t } P)$ .

```

In turn, the substitution lemma requires the following *weakening*, *strengthening*, and *type invariance under equivalence* properties.

- The *weakening* lemma states that enriching the typing context with new names or type of existing names does not break typing of the surrounding object:

```

Lemma weakening : forall  $\Gamma_1$   $\Gamma_2$   $\Gamma_3$  P,
  ! $\Gamma_1$  + ! $\Gamma_2$  == ! $\Gamma_3$  ->  $\Gamma_1 \vdash P$  ->  $\Gamma_3 \vdash P$ .

```

Note that here, + represents the partial, resource-aware, addition on environments, and not a simple juxtaposition, like in resource-unaware type systems.

- The *strengthening* lemma states that removing unused names from the typing context doesn't break typing:

```

Lemma strengthening_p: forall n, forall P, p_size P <= n ->
  forall  $\Gamma$  x s,
    x  $\notin$  FV P -> x  $\notin$  dom  $\Gamma$  -> ( $\Gamma, x:s \vdash P$  ->  $\Gamma \vdash P$ ).

```

- *Type invariance* states that equivalent processes are typable in the same typing context:

```

Instance typ_equiv :
  Proper (eq ==> peq ==> iff) typ_process.

```

In other words: $\forall \Gamma P_1 P_2, P_1 \equiv P_2 \implies \Gamma \vdash P_1 \iff \Gamma \vdash P_2$.

Using these lemmas, one can finally state and prove the subject reduction property:

```

Lemma subject_reduction_1 : forall P P' Γ,
  Γ ⊢ P -> P ---> P' -> Γ ⊢ P'.
Lemma subject_reduction_2 : forall P P' Γ,
  Γ ⊢ P -> P ==> P' -> ↑ Γ ⊢ P'.

```

The size and contents of various Coq files are listed in table 5 and are available in [DG11].

File	Lines	Content
CoreLibrary.v	437	Extensions to the standard library
Syntax.v	744	Definitions of terms, processes and environments
Additions.v	2111	Definitions of (partial) additions
Names.v	2723	Locally nameless related constructs and lemmas
Typing.v	3027	Auxiliary lemmas about typing and names handling
Tpi.v	3073	Main lemmas

Table 5. Coq modules

5 Conclusion and future work

We have introduced a timed π -calculus, named TAPIS, together with a type system to identify confluent processes. The type system relies on usages to control resources. We also formalised TAPIS and its accompanying type system in Coq and showed that the subject reduction property holds. In this development, we have used and extended the locally nameless technique for the representation of names.

Future work could include proof of confluence for typable programs, and extraction of a certified type checker and interpreter.

References

- [ACP⁺08] B. Aydemir, A. Charguéraud, B.C. Pierce, R. Pollack, and S. Weirich. Engineering formal metatheory. *ACM SIGPLAN Notices*, 43(1):3–15, 2008.
- [AD08] R. Amadio and M. Dogguy. On affine usages in signal-based communication. In *Asian Symposium on Programming Languages and Systems*, pages 221–236, 2008.
- [Ama07] R.M. Amadio. The SL synchronous language, revisited. *Journal of Logic and Algebraic Programming*, 70(2):121–150, 2007.
- [BG92] G. Berry and G. Gonthier. The Esterel synchronous programming language. *Science of computer programming*, 19(2):87–152, 1992.
- [BP10] M. Braibant and D. Pous. Rewriting modulo associativity and commutativity in Coq. 2nd Coq Workshop, July 2010. (Available as a HAL report).

- [BS96] F. Boussinot and R. De Simone. The SL synchronous language. *IEEE Trans. on Software Engineering*, 22(4):256–266, 1996.
- [CDT10] The Coq Development Team. The Coq proof assistant reference manual, version 8.3. <http://coq.inria.fr/refman/>, 2010.
- [Des00] J. Despeyroux. A higher-order specification of the π -calculus. *Theoretical Computer Science: Exploring New Frontiers of Theoretical Informatics*, pages 425–439, 2000.
- [DG11] M. Dogguy and S. Glondu. Confluence in TAPIS. <http://www.pps.jussieu.fr/~dogguy/?research/tapis>, 2011.
- [Gor94] A. Gordon. A mechanisation of name-carrying syntax up to alpha-conversion. *Higher Order Logic Theorem Proving and Its Applications*, pages 413–425, 1994.
- [HG99] L. Henry-Gréard. Proof of the subject reduction property for a-calculus in coq. 1999.
- [Hir97] D. Hirschkoﬀ. A full formalisation of π -calculus theory in the calculus of constructions. *Theorem Proving in Higher Order Logics*, pages 153–169, 1997.
- [KPT99] N. Kobayashi, B. C. Pierce, and D. N. Turner. Linearity and the pi-calculus. *ACM Trans. Program. Lang. Syst.*, 21(5):914–947, 1999.
- [Mil89] R. Milner. *Communication and concurrency*. 1989.
- [MM04] C. McBride and J. McKinna. Functional pearl: I am not a number—I am a free variable. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 1–9. ACM, 2004.
- [Moh95] Otmane Aït Mohamed. Mechanizing a pi-calculus equivalence in HOL. In *TPHOLs*, pages 1–16, 1995.
- [MP99] J. McKinna and R. Pollack. Some lambda calculus and type theory formalized. *Journal of Automated Reasoning*, 23(3):373–409, 1999.
- [NS97] U. Nestmann and M. Steffen. Typing confluence. In *Second International ERCIM Workshop on Formal Methods in Industrial Critical Systems*, pages 77–101, 1997.
- [PSWZ05] B. C. Pierce, P. Sewell, S. Weirich, and S. Zdancewic. It is time to mechanize programming language metatheory. In *VSTTE*, pages 26–30, 2005.
- [SO08] M. Sozeau and N. Oury. First-class type classes. In *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics, TPHOLs '08*, pages 278–293, Berlin, Heidelberg, 2008. Springer-Verlag.
- [Yi91] W. Yi. CCS+ time= an interleaving model for real time systems. *Automata, Languages and Programming*, pages 217–228, 1991.

A Tapis's transition system using Coq's syntax

$$\begin{array}{c}
\text{(par)} \frac{P \rightarrow P'}{P \parallel Q \rightarrow P' \parallel Q} \qquad \qquad \qquad (\nu) \frac{\forall x \notin L \quad P@x \rightarrow P'@x}{\text{Pnu } P \rightarrow \text{Pnu } P'} \\
\\
(=^{\text{ind}}_1) \frac{\text{match}(\mathbf{t}, \mathbf{p}) = \theta}{[\mathbf{t} \sqsubseteq \mathbf{p}]P_1, P_2 \rightarrow \theta P_1} \qquad \qquad \qquad (=^{\text{ind}}_2) \frac{\text{match}(\mathbf{t}, \mathbf{p}) = \uparrow}{[\mathbf{t} \sqsubseteq \mathbf{p}]P_1, P_2 \rightarrow P_2} \\
\\
(\equiv) \frac{P \equiv P' \quad P' \rightarrow Q' \quad Q' \equiv Q}{P \rightarrow Q} \\
\\
(\text{sync}_1) \frac{}{\text{Precv } a \, P \parallel \text{Psend } a \, \mathbf{t} \rightarrow P@\mathbf{t}} \\
\\
(\text{sync}_2) \frac{}{\text{Precv_elsenext } a \, P \parallel \text{Psend } a \, \mathbf{t} \rightarrow P@\mathbf{t}} \\
\\
(!_1) \frac{}{\text{Precv_dup } a \, P \parallel \text{Psend } a \, \mathbf{t} \rightarrow \text{Precv_dup } a \, P \parallel P@\mathbf{t}} \\
\\
(!_2) \frac{}{\text{Precv_dup_elsenext } a \, P \parallel \text{Psend } a \, \mathbf{t} \rightarrow \text{Precv_dup_elsenext } a \, P \parallel P@\mathbf{t}} \\
\\
(\triangleright_{\text{in}}) \frac{}{\text{Precv } a \, P \xrightarrow{\text{tick}} \text{Precv } a \, P} \qquad \qquad \qquad (\triangleright_{\text{else}}) \frac{}{\text{Precv_elsenext } a \, P \triangleright Q \xrightarrow{\text{tick}} Q} \\
\\
(\triangleright_{!}) \frac{}{\text{Precv_dup } a \, P \xrightarrow{\text{tick}} \text{Precv_dup } a \, P} \quad (\triangleright_{\text{rec}}) \frac{}{\text{Precv_dup_elsenext } a \, P \, Q \xrightarrow{\text{tick}} Q} \\
\\
(\triangleright_{\nu}) \frac{\forall x \notin L \quad P@x \xrightarrow{\text{tick}} P'@x}{\text{Pnu } P \xrightarrow{\text{tick}} \text{Pnu } P'} \qquad \qquad \qquad (\triangleright_{\text{out}}) \frac{}{\text{Psend } a \, \mathbf{t} \xrightarrow{\text{tick}} \text{Pnull}} \\
\\
(\triangleright_{\text{par}}) \frac{P \xrightarrow{\text{tick}} P' \quad Q \xrightarrow{\text{tick}} Q' \quad P \parallel Q \not\rightarrow \cdot}{P \parallel Q \xrightarrow{\text{tick}} P' \parallel Q'}
\end{array}$$