Timed Process Algebras: A Tutorial *

M. Hennessy
University of Sussex

Abstract

In this set of notes we give an overview of a particular approach to describing timed concurrent systems. The starting point is a well–developed semantic theory of process algebras based on testing. This consists of an operational semantics for a "time–free" process description language, a behavioural equivalence based on *testing* and an algebraic characterisation of this equivalence. We add to this language one timing construct, a *time–out* operator, and show how the theory can be extended to this time enriched language.

This extended language is certainly restricted in its ability to describe timing phenomena but in the last section we show how it may be used as the basis for more expressive timed process description languages.

*Lecture notes for the International Summer School on *Process Design Calculi*, Martoberdorf, 1992

1 Labelled Transition Systems

In this series of lectures we will be interested in the extending an existing semantic theory of standard process algebras to encompass at least some aspects of timed systems. Accordingly in this first section we will review the situation for standard "untimed process algebras".

In general a process algebra may be viewed as a specification or description language for communicating concurrent systems which emphasises their conceptual structure. Essentially a process algebra consists of a collection of combinators for constructing new descriptions from existing ones together with a set of laws or equations for manipulating these descriptions. A large number of process algebras have by now been proposed in the literature; we will try to avoid getting embroiled in the details of these individual languages, each with their own advantages and disadvantages, by simply choosing to work with our favourite, *CCS*, [Mil89]; however most of what we say is equally applicable to other process calculi such as *CSP* and *ACP*. We will also work as much as possible at the more abstract level of intensional operational semantics.

We describe concurrent systems in terms of their ability to perform *actions*. For the most part the nature of these actions will be unspecified and we will simply assume a set of possible actions Act; typically these are some form of synchronisations with other concurrent systems. But we will have need for a special action symbol τ, which we assume is not in Act, to represent internal synchronisations or activity of a system. For convenience we use Act_τ to represent $Act \cup \{\tau\}$, ranged over by α, while a ranges over the external actions Act. The ability of a concurrent system to perform these uninterpreted actions can be conveniently represented in terms of *labelled transition systems*:

Definition 1.1 A labelled transition system is a triple $< P, Act_\tau, \longrightarrow >$ where

P is a set of process states

Act_τ is, as already explained, a set of external actions Act and special action τ

$\longrightarrow$ is a subset of $P \times Act_\tau \times P$; we write $p \stackrel{\alpha}{\longrightarrow} q$ instead of $(p, \alpha, q) \in \longrightarrow$

□

Intuitively $p \stackrel{\alpha}{\longrightarrow} q$ means that in the state p the process may perform the action α and thereby be transformed into the the state q.

An arbitrary process algebra may be given an intensional semantics by interpreting it as a labelled transition system. A standard method for doing so is by structural operational semantics, [Plo81]. The simplest way to explain this is to consider an example, the language *CCS*. In *CCS* communication or synchronisation is a binary operation, i.e. it involves only two participants, one sending the synchronisation signal and the other receiving it. Accordingly we assume that Act is equipped with a complementation operation $\overline{a}$ and informally we view $\overline{a}$ as the action of sending a synchronisation signal on a virtual communication channel a while the action a represents its receipt.

The terms of the language are defined by

$$\begin{array}{rcl} p & ::= & \Omega \mid nil \mid P \mid \alpha.p \\ & & \mid p + p \mid p \mid p \\ & & \mid p[S] \mid p\backslash A,\ A \subseteq Act \end{array}$$

We have the constant processes *nil* which can do no actions and Ω which can only diverge and we are allowed the use of process identifiers such as P which will have associated with them a definition of the form

$$P \Leftarrow t.$$

The combinators of the language are

action prefixing: $\alpha.p$ is a process which can perform the action α and then become the process p

choice: $p + q$ is the process which can act like p or q

parallel: $p \mid q$ is the process which consists of two subprocesses p and q running in parallel

restriction: $p\backslash A$ is a process which acts like p except that all actions in A and their complements are local to p

renaming: $p[S]$ is a process which acts like p except that the actions performed are relabelled using the relabelling function S; it is assumed that S is a function over Act which preserves complementation and which is almost everywhere the identity. We will also assume that $S(\tau) = \tau$.

When writing terms in this language we will observe the usual rules of precedence between the operators,

$$\backslash A \;=\; [S] \;>\; \text{prefixing} \;>\; \mid \;>\; +$$

and we will usually omit trailing occurrences of *nil*.

This informal description of the intended meaning of the combinators is made precise by providing the language with an operational semantics. This involves defining a relation $\stackrel{\alpha}{\longrightarrow}$ between process terms for each action α. To do so we assume the existence of a *declaration*, i.e. a set of definitions of the form

$$P \Leftarrow p,$$

one for each identifier; the term associated with an identifier in a declaration D will be referred to as its *body*, and denoted by $D(P)$; for convenience we will assume that each occurrence of any identifier in the bodies of a declaration are *guarded*, i.e. are contained within terms of the form $\alpha.q$. This is not necessary but is will make various definitions more straightforward as we will not have to deal with suspect definitions such as

$$P \Leftarrow P$$

or

$$P \Leftarrow P \mid a.$$

Assuming some declaration we can define the relations $\stackrel{\alpha}{\longrightarrow}$ as the least relations which satisfy the rules given in Figure 1. The obvious symmetric counterparts to $(Op2)$ and $(Op3)$ are omitted and the predicate A admits α used in $(Op4)$ has the obvious definition: A admits α unless α or $\overline{\alpha} \in A$.

With this definition we can now view *CCS* as a labelled transition system where

$(Op1)$ $\alpha.p \xrightarrow{\alpha} p$

$(Op2)$ $p \xrightarrow{\alpha} p'$ implies $p + q \xrightarrow{\alpha} p'$

$(Op3)$ $p \xrightarrow{\alpha} p'$ implies $p \mid q \xrightarrow{\alpha} p' \mid q$

$(Op4)$ $p \xrightarrow{\alpha} p'$ implies $p\backslash A \xrightarrow{\alpha} p'\backslash A$ provided A admits α

$(Op4)$ $p \xrightarrow{\alpha} p'$ implies $p[S] \xrightarrow{S(\alpha)} p'[S]$

$(Op5)$ $D(P) \xrightarrow{\alpha} p'$ implies $P \xrightarrow{\alpha} p'$

$(Op6)$ $p \xrightarrow{a} p',\ q \xrightarrow{\overline{a}} q'$ implies $p \mid q \xrightarrow{\tau} p' \mid q'$

$(Op7)$ $\Omega \xrightarrow{\tau} \Omega$

Figure 1: Operational semantics of *CCS*

the process states consists of all the terms in the language

the next-state relations $\xrightarrow{\alpha}$ are defined in Figure 1.

Alternatively we can view every process term p as a labelled transition system obtained by restricting the set of process states to those accessible from p; note that this labelled transition system is rooted. Not much distinction will be made between these two slightly different views of the intensional semantics of *CCS*. We will also generally consider the process states of a transition system as simply processes.

As an example of a process description consider the definition

$$\begin{aligned} VM &\Longleftarrow coin.(VM_t + VM_c) \\ VM_t &\Longleftarrow \overline{tea}.VM \\ VM_c &\Longleftarrow \overline{coffee}.VM \end{aligned}$$

This is a simple vending machine which when given a coin will perform exactly one of the actions $\overline{tea}$ or $\overline{coffee}$. It has slightly different behaviour than the vending machine defined by

$$\begin{aligned} VM^i &\Longleftarrow coin.(\tau.VM_t + \tau.VM_c) \\ VM_t &\Longleftarrow \overline{tea}.VM \\ VM_c &\Longleftarrow \overline{coffee}.VM \end{aligned}$$

When given a coin this machine will also provide either tea or coffee but, in contrast to VM there is no knowing which will be provided. The difference between these two machines can be seen by putting them in parallel with a user such as

$$User_t \iff \overline{coin}.tea.\overline{happy}$$

One can check that the process $(User_t \mid VM)\backslash A$, where $A = \{coin, tea, coffee\}$ will always reach the happy state, i.e. will always perform the action $\overline{happy}$. On the other hand the process $(User_t \mid VM^i)\backslash A$ may or may not reach this state depending on what the vending machine chooses to provide.

The intensional semantics of a process algebra as expressed in a labelled transition system gives a relatively abstract view of how a process behaves in terms of the actions it performs. However two processes, or process descriptions, could have very different representations as labelled transition systems and still be considered to be *extensionally* equivalent in the sense of providing more or less the same behaviour to any potential user. For example the process descriptions $(User_t \mid VM)\backslash A$ and $\overline{happy}.nil$ yield different labelled transition systems but one can argue that they should be considered to be extensionally equivalent. Much research has been carried out into what exactly extensional equivalence should mean and a number of viable alternatives have emerged. In these lectures we will concentrate on one possibility, called *testing equivalence* [Hen88], where informally two processes are deemed to be extensionally equivalent if there is no test which can possibly distinguish between them.

This can be formalised at the level of labelled transition systems. We can view a given labelled transition system as providing processes to be experimented upon while the experimenters are also furnished by a labelled transition system. To keep things simple we just assume that these two labelled transition systems are actually the same so that we have processes experimenting upon themselves. A process proceeds by a series of interactions between the process and the experimenter and is considered a success if the experimenter eventually reaches a *success* state. Success can be formalised in a number of ways but let us just decide that $Success = \{\, e \mid \text{for all } \alpha \in Act\ e \stackrel{\alpha}{\not\longrightarrow} \}$. The interactions between the experimenter and the process, or more generally the progress of an experiment, can be formalised as a binary relation over pairs of processes. But we have already seen an operator which captures this form of interaction, namely $\mid$. Admittedly it was defined as part of the syntax of *CCS* but we can easily imagine $\mid$ as being defined over an arbitrary labelled transition system, or even pairs of labelled transition systems provided their set of actions match properly. For our purpose it is sufficient to consider it as an operator which takes an arbitrary labelled transition system and constructs a new labelled transition system whose process states consists of pairs of process states from the original labelled transition system and whose next state relation is determined by the rules $(Op3)$, with α equal to τ, and $(Op6)$. However we will not be interested in how the interaction takes place and to emphasise that these moves are part of an experiment we use the slightly different notation

$$e \mid p \ \mapsto\ e' \mid p'$$

to mean that $e \mid p \stackrel{\tau}{\longrightarrow} e' \mid p'$. An *application* of e to p is then given as a maximal computation of the form:

$$e \mid p \equiv e_0 \mid p_0 \ \mapsto\ e_1 \mid p_1 \ldots \mapsto\ e_n \mid p_n \ \mapsto\ \ldots$$

that is, it is either infinite or has a maximal element $e_m \mid p_m$ from which no further derivation can be made. Such an application is considered to be successful if the experimenter reaches a successful state, i.e. there is a k such that $e_k \in \mathit{Success}$. We then say that p *guarantees* e if every application of e to p is successful. Finally we have

> For all processes p, q we write $p \sqsubseteq q$ if p *guarantees* e implies q *guarantees* e for every experimenter e.

The associated equivalence relation, the kernel of $\sqsubseteq$, is denoted by $\simeq$.

This extensional equivalence has been studied in depth in [Hen88] and it has been applied to the process algebra *CCS* in particular in [DH84]. It is this equivalence and its associated theory which we wish to develop for timed process algebras.

2 Timed labelled Transition Systems

There are a large number of proposals in the literature for adding time to process algebras; to such an extent that it is very difficult to compare and contrast the often competing schemes. Some of the abundance is due to the profusion of different process algebras but if we abstract to the level of intensional operational semantics in terms of labelled transition systems then the situation is somewhat better. At this level of abstraction the diversity is caused primarily by the decision as to how to model the passage of time. The approach which we take is essentially the same as many researchers, [NS90, Wan91, MT90, Han91, BB92], although there are differences in the details of the individual languages used and some of the assumptions about time. In this exposition we will gloss over many of the technical details which may be found in [Reg92, HR91].

Part of the success of process algebras is due to the fact that they describe processes at a relatively abstract level of detail. We do not know if processes will ever actually perform the actions they are capable of nor indeed is any knowledge presupposed about the nature of these actions; for example whether they are instantaneous, all take the same length of time or are variable. It is precisely because no commitments have been made on these questions that process algebras can be used in many different applications and have a reasonable mathematical theory. However now that we wish to add some notion of time to these process descriptions it is necessary to be at least a little more specific. Nevertheless we wish to remain as abstract as possible and to deviate as little as necessary from standard process algebras. In this way we can at least hope to continue to reap their benefits.

Our proposal is to simply add to labelled transitions systems a new special action called σ to model the passage of time. More generally one could envisage adding an arbitrary number of such special actions, $\sigma_1, \sigma_2, \ldots$, each one representing the passage of time relative to a particular clock as in [BG88]. However at least for the moment let us keep things simple by just assuming that there is only one clock in existence.

Definition 2.1 A t-labelled transition system is a triple $< P, Act_{\tau\sigma}, \longrightarrow >$ where

P is a set of process states

$Act_{\tau\sigma}$ is a set of actions of the form $Act \cup \{\tau, \sigma\}$ where τ, σ are two special action symbols not in Act

$\longrightarrow$ is a subset of $P \times Act_{\tau\sigma} \times P$

□

Of course σ can not be considered simply as any old action; it represents the passage of time and presumably has at least some characteristics different from standard synchronisation actions. Indeed much of the proliferation of timed process algebras is due to the properties which researchers deem this action should have. Rather than discussing these properties in the abstract let us see an example of a t-labelled transition system obtained by extending *CCS* with a timing construct. The syntax of the new language, which we call *TPL*, is defined by adding one new clause:

$$p ::= \lfloor p \rfloor(p).$$

Intuitively this is a time-out operator; the process $\lfloor p \rfloor(q)$ acts like the process p before a clock signal arrives. But if it can do nothing before this event then when the signal arrives it subsequently behaves like q.

This is formalised by giving an operational semantics to the language *TPL*. Underlying this semantics is a particular informal view of the nature of *CCS* descriptions which may not be to everybodys liking. This view is captured by the informal assumptions:

a process is determined by its ability to perform actions or synchronisations

all actions, apart from σ are assumed to be instantaneous

a process will wait indefinitely until it can engage in a synchronisation

once a process can engage in a synchronisation it will do so without delay

The definition of operational semantics is given in two phases. The first defines the relations $\stackrel{\alpha}{\longrightarrow}$ for $\alpha \in Act_\tau$. This is straightforward as one simply adds to the rules of Figure 1 the new rule

$$(Op8) \quad p \stackrel{\alpha}{\longrightarrow} p' \quad \text{implies} \quad \lfloor p \rfloor(q) \stackrel{\alpha}{\longrightarrow} p'.$$

The second phase defines the relation $\stackrel{\sigma}{\longrightarrow}$ between arbitrary processes in *TPL*. The rules determining $\stackrel{\sigma}{\longrightarrow}$ are given in Figure 2. Most of these rules are influenced by the properties we wish to associate with the passage of time; other viewpoints would lead to different rules. For example $(W1)$ which says that for any external action a the process $a.p$ can wait:

$$a.p \stackrel{\sigma}{\longrightarrow} a.p.$$

This is because we think of a process as idling until such time as its environment provides an opportunity to synchronise. However if one wishes to envisage processes which are *insistent* in the sense that they require immediate synchronisations then this rule would not form part of the operational semantics. We will return to this point later, in the final section. Note that $\tau.p$ can not wait as the τ represents some unknown opportunity to synchronise and we wish to enforce our intuitive assumption that synchronisations should happen as soon as they are possible. For the same reason we insist in $(W3)$ that $p \mid q$ can only delay if p and q can not synchronise. Again if one had a different view of processes then this rule would need to be changed.

$(W1)$	$a.p \xrightarrow{\sigma} a.p$	
	$nil \xrightarrow{\sigma} nil$	
$(W2)$	$p \xrightarrow{\sigma} p',\ q \xrightarrow{\sigma} q'$	implies $p + q \xrightarrow{\sigma} p' + q'$
$(W3)$	$p \xrightarrow{\sigma} p',\ q \xrightarrow{\sigma} q',$ $p \mid q \not\xrightarrow{\tau}$	implies $p \mid q \xrightarrow{\sigma} p' \mid q'$
$(W4)$	$p \xrightarrow{\sigma} p'$	implies $p\backslash A \xrightarrow{\sigma} p'\backslash A$
$(W5)$	$p \xrightarrow{\sigma} p'$	implies $p[S] \xrightarrow{\sigma} p'[S]$
$(W6)$	$p \not\xrightarrow{\tau}$	implies $\lfloor p \rfloor(q) \xrightarrow{\sigma} q$
$(W7)$	$D(P) \xrightarrow{\sigma} p'$	implies $P \xrightarrow{\sigma} p$

Figure 2: The passage of time in *TEPL*

It is worth examining in detail the behaviour of the timeout construct. In $\lfloor p \rfloor(q)$ if p can perform any action α then it will do so and the "exception" process q will be discarded. If on the otherhand no opportunity for synchronisation is forthcoming, and this includes the supposition that $p \not\xrightarrow{\tau}$, then when the clock tick arrives the new residual will be q, i.e. under those assumptions $\lfloor p \rfloor(q) \xrightarrow{\sigma} q$. Thus $(a.p \mid \lfloor \overline{a}.q \rfloor(r))\backslash\{a\}$ can only perform a τ move to $(p \mid q)\backslash\{a\}$ while $(b.p \mid \lfloor a.q \rfloor(r))\backslash\{a, b\}$, under the assumption that $a \neq \overline{b}$, can only wait by performing a σ move and become $(b.p \mid r)\backslash\{a, b\}$. Notice that the language does not have a simple delay construct. But it can be implemented as $\lfloor nil \rfloor(p)$; this process can do nothing until the first time cycle when it becomes p. Since this delay construct will be frequently used we introduce an abbreviation for it.

Definition 2.2 We use $\sigma.p$ to denote the term $\lfloor nil \rfloor(p)$. This notation should be intuitive because the only action it can perform is σ to become p. □

With these rules we now have an interpretation of the language *TPL* as a t-labelled transition system where the process states consists of the terms from *TPL* and the next-state relations $\xrightarrow{\mu}$ are defined by the rules just outlined. In order to illustrate these rules we now consider a simple example, again a vending machine.

$$\begin{aligned} VM_d &\Longleftarrow coin.\sigma.VM_t \\ VM_t &\Longleftarrow \lfloor \overline{tea}.VM_d \rfloor(VM_c) \\ VM_c &\Longleftarrow \lfloor \overline{coffee}.VM_d \rfloor(VM_d) \end{aligned}$$

The vending machine will accept a coin as before and then after a time cycle will be in the state VM_t. Here it will produce tea if requested or after another time cycle will produce coffee. The next time cycle will bring the machine back to the original state VM_d where it will produce nothing until it receives another coin. Although its

behaviour is somewhat more complicated than the original vending machine it can still be used successfully by users who want a particular drink. For example $(User_t \mid VM)\backslash A$, where as before $A = \{coin, tea, coffee\}$, will always reach the happy state and the same is true of $User_c$ defined by

$$User_c \iff \overline{coin}.coffee.\overline{happy}.nil$$

However users have to be a little careful of how long they wait before accepting their drink. For example the user $User_t^2$ defined by

$$User_t^2 \iff \overline{coin}.\sigma^2.tea.\overline{happy}.nil,$$

where $\sigma^2.p$ is an abbreviation for $\sigma.\sigma.p$, will not reach the happy state although the corresponding user who wants coffee will have no problem.

We now reconsider various properties that one might want to associate with the special action σ. Each of the following lemmas refer to the particular t-labelled transition system determined by TPL.

Lemma 2.3 *(Time-determinism) If $p \xrightarrow{\sigma} q$ and $p \xrightarrow{\sigma} r$ then q and r are syntactically identical.*

This is a natural property to associate with the passage of time although there are process algebras which do not have this property, [Gro90].

Lemma 2.4 *(Maximal progress) If $p \xrightarrow{\sigma}$ then $p \not\xrightarrow{\tau}$*

This is the formal counterpart to our fourth informal assumption about the nature of processes. Again there are timed process algebras which do not satisfy this property, [NS90, MT90]. One advantage of assuming maximal progress is that one can easily describe processes in which particular actions are forced to happen; if p can perform the action a then when placed in the context $(\overline{a}.nil \mid [\ ])\backslash A$ it will be forced to synchronise.

Lemma 2.5 *(Patience) If $p \not\xrightarrow{\tau}$ then $p \xrightarrow{\sigma}$*

This is the formal counterpart to our second informal assumption about the nature of processes which says that a process will idle until such time as it can synchronise. It is not an especially popular assumption. For example it is not satisfied in [NS90, MT90, BB92] as they have insistent actions but it does have its proponents, [Wan91]. Typically if a process algebra does not take on board the maximal progress assumption then it also does not satisfy patience as some way of enforcing actions to happen is required.

Lemma 2.6 *(Persistence) If $p \in CCS$ then $p \xrightarrow{a} q$ and $p \xrightarrow{\sigma} r$ implies $r \xrightarrow{a} q$.*

This property is not enjoyed by all processes in the language; a simple example is $\lfloor a.p \rfloor(nil)$. It is not present in most of the proposed timed process algebras; as we will see it leads to a certain lack in descriptive power. However a form of it occurs in [Wan91].

The extensional equivalence of the previous section can be generalised to this timed setting by working with t-labelled transition systems instead of the standard transition systems. Here the process being tested, p, and the experimenter e are assumed to be

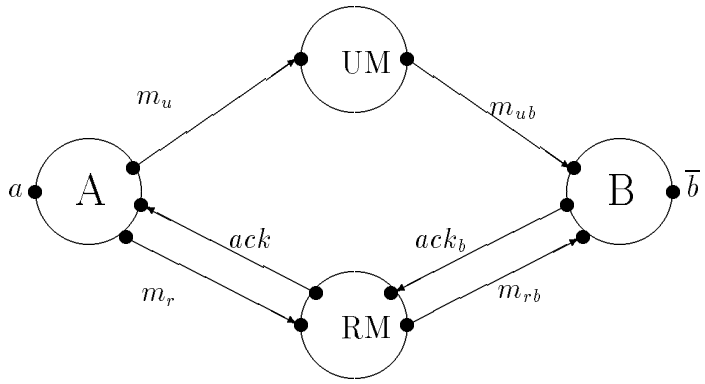

Figure 3: The security costs protocol

states in a t-labelled transition systems and the interaction relation $\mapsto$ is defined in exactly the same way; a parallel operator is defined between arbitrary t-labelled transition systems using the rules $(Op3)$ with α equal to τ, $(Op6)$ and $(W3)$ and $e \mid p \mapsto e' \mid p'$ if $e \mid p \stackrel{\tau}{\longrightarrow} e' \mid p'$ or $e \mid p \stackrel{\sigma}{\longrightarrow} e' \mid p'$. This leads as before to a semantic preorder defined by:

> in a t-labelled transition system $p \sqsubseteq_t q$ if p *guarantees* e implies q *guarantees* e for every experimenter e,

and the associated equivalence relation is denoted by $\simeq_t$.

Question: Design two tests which show that the processes VM and VM_d are incomparable with respect to $\sqsubseteq_t$. □

We aim to show that this new semantic equivalence between timed systems enjoys many of the desirable properties of the original testing equivalence $\simeq$. In particular it has a reasonable mathematical theory and an associated proof system based on equational rewriting.

We end this section with a slightly more extended example of a timed process description using *TPL*. We view it as a pro-typical example of the potential use of *TPL*; only one process involved uses a timing construct and the remainder are described using the syntax of standard *CCS*. Thus the timing considerations are restricted to a very small part of the system.

The "security costs protocol" , shown diagramatically in Figure 3, describes a simple method for sending a message between two distributed ports which we call A and B where transmission of a message across a reliable medium is considered expensive. The sender, A, initially sends the message across an unreliable medium only resending across the reliable medium if an acknowledgement has not arrived before a "time-out" occurs. To keep the example simple we assume that acknowledgements always use the reliable medium.

This reliable medium is defined by

$$RM \Leftarrow m_r.\overline{m}_{rb}.RM + ack_b.\overline{ack}.RM$$

It accepts a message destined for it, called m_r and sends it on to B in the form of the action $\overline{m}_{rb}$. Alternatively it acts as conduit for acknowledgements from B to A: it accepts an acknowledgement from B in the form of the action ack_b and passes it on as $\overline{ack}$ to A. The unreliable medium is defined by

$$UM \Leftarrow m_u.(\tau.UM + \tau.\overline{m}_{ub}.UM)$$

It accepts a message and then nondeterministically decides to either lose it and return to the original state UM or to send it on to B in the form of the action $\overline{m}_{ub}$. The receiver B is defined by

$$B \Leftarrow m_{rb}.\overline{b}.\overline{ack}_b.B \ + \ m_{ub}.\overline{ack}_b.\overline{b}.\overline{ack}_b.B.$$

It can accept a message from the reliable medium, m_{rb}, in which case it immediately transmits the message, modelled here by the the action $\overline{b}$ and sends an acknowledgement to A to say that the message has been transmitted and so that A can accept another input. Alternatively it can accept a message from the unreliable medium in which case it immediately acknowledges that the message has successfully come across the unreliable medium and then proceeds as before. Finally the sender A is defined by

$$A \Leftarrow a.\overline{m}_u.\lfloor ack.ack.A \rfloor (\overline{m}_r.ack.A).$$

This is the only place where timing enters the description. A accepts a message for transmission, modelled by the action a, and initially sends it to the unreliable medium. It awaits an acknowledgement that the message has successfully made it across the medium and then awaits another acknowledgement, in this case acknowledging that the message has been transmitted, before accepting new input. However if no acknowledgement is forthcoming when the clock cycle comes the message is resent over the reliable medium and A waits for an acknowlegdement of transmission before accepting new input.

The entire system is modelled by

$$SYS \Leftarrow (A \mid RM \mid UM \mid B) \backslash I$$

where I is the set of all the actions used by the processes except the input and output, a, b.

Later on we hope to show that SYS is equivalent to the specification

$$SPEC \Leftarrow a.(\tau.\sigma.\overline{b}.SPEC + \tau.\overline{b}.SPEC)$$

which is exactly the behaviour one would expect of the system.

Question: If the sender A is replaced by

$$a.\overline{m}_u.(ack.ack.A + \sigma.\overline{m}_r.ack.A)$$

does the extensional behaviour of the system remain the same ? □

Question: Show that for the protocol to work it is essential for A to wait an acknowlegdement that B has transmitted the message before accepting a new message for transmission. □

3 Acceptances and Barbs

In this section we investigate the properties of processes which determine their ability to guarantee tests. This is essential if we are going to develop a theory of $\simeq_t$; it is easy to demonstrate that two processes are not extensionally equivalent as one only has to provide a test which distinguishes them but to show that they are equivalent we must establish that they guarantee exactly the same set of tests. Rather than consider the reaction of processes with respect to *all* tests we extract the relevant behaviour of their operational semantics which in effect determines which equivalence class of $\simeq_t$ a process belongs to.

Before we start we need some definitions which are given relative to an arbitrary t-labelled transition system although they apply equally well to labelled transition systems since the latter may be considered as t-labelled transition systems where the relation $\stackrel{\sigma}{\longrightarrow}$ is empty. For any $s \in (Act_\sigma)^*$ let the relations $\stackrel{s}{\Longrightarrow}$ be defined in the obvious way:

1. $p \stackrel{\varepsilon}{\Longrightarrow} p$
2. $p \stackrel{\mu}{\longrightarrow} p',\ p' \stackrel{s}{\Longrightarrow} q$ implies $p \stackrel{\mu.s}{\Longrightarrow} q$
3. $p \stackrel{\tau}{\longrightarrow} p',\ p' \stackrel{s}{\Longrightarrow} q$ implies $p \stackrel{s}{\Longrightarrow} q$
4. $p \stackrel{s}{\Longrightarrow} p',\ p' \stackrel{\tau}{\longrightarrow} q$ implies $p \stackrel{s}{\Longrightarrow} q$.

Let $S(p)$ denote the set $\{\, a \in Act \mid p \stackrel{a}{\Longrightarrow} \}$ and we say that p is *stable* if $p \not\stackrel{\tau}{\longrightarrow}$. Finally we say that p *diverges* if there exists an infinite sequence $p \stackrel{\tau}{\longrightarrow} p_1 \stackrel{\tau}{\longrightarrow} \ldots \stackrel{\tau}{\longrightarrow} p_k \stackrel{\tau}{\longrightarrow} \ldots$; we use $p \Uparrow$ to denote divergence and $p \Downarrow$ to denote the converse, *convergence*.

We first recapitulate on the situation for untimed calculi, testing in an arbitrary labelled transition system. Here the crucial property of a process is the set of its *acceptances*; these consists of a sequence of actions s together with a finite set of actions which it is ready to accept after having performed the sequence s. Formally

1. Ω is an acceptance
2. A is an acceptance if A is a finite subset of Act
3. $a\underline{a}$ is an acceptance if $\underline{a}$ is an acceptance and $a \in Act_\sigma$.

So an acceptance has the form sX where s is a sequence from Act_σ^* and X is either Ω or a finite subset of Act. Acceptances are compared in the following way: let $\ll_a$ be the least relation between acceptances which satisfies

$\Omega \ll_a \underline{a}$ for any acceptance $\underline{a}$

$A \ll_a A'$ if $A \subseteq A'$

$a\underline{a} \ll_a a\underline{a}'$ if $\underline{a} \ll_a \underline{a}'$

One can check that $\ll_a$ is actually a partial order between acceptances and it may be characterised by saying

$$sX \ll_a tY$$

if either X is Ω and s is a prefix of t or $s = t$ and X and Y are both finite subsets of Act with $X \subseteq Y$. This partial order is generalised to a preorder between sets of acceptances by letting $S \ll_a T$ if for each $\underline{a} \in T$ there exists an $\underline{a}' \in S$ such that $\underline{a}' \ll_a \underline{a}$. The acceptance sX is *generated* by the process p if $p \stackrel{s}{\Longrightarrow} q$ for some q such that

if X is Ω then $q \Uparrow$

if X is a finite subset of Act then q is stable and $X = S(q)$.

We use $Acc(p)$ to denote the set of acceptances generated by p.

These definitions enable us to state a condition which is sufficient to ensure that processes in an arbitrary labelled transition system are related extensionally. Of course in a labelled transition system acceptances have no occurrence of the timed action σ.

Theorem 3.1 *In any labelled transition system* $Acc(p) \ll_a Acc(q)$ *implies* $p \sqsubseteq q$.

Proof: See [Hen88]. □

The converse depends essentially on the expressive power of the labelled transition system in question. For example if in the labelled transition system every process which can do an a action can also do a b action with the same effect then the two processes $a.nil + b.nil$ and $\tau.a.nil + b.nil$ will not be distinguishable; but they will in a labelled transition system which has a process which can not do a b action but can do an a action to a terminated state. We will not go detail about the exact expressive power necessary. Instead let us just say that a labelled transition system is *sufficiently expressive* if it contains a denotation for every finite term in $fCCSseq$, i.e. terms in CCS which only use the combinators nil, $+$ and prefixing by actions in Act_τ.

Theorem 3.2 *In a sufficiently expressive finitely branching labelled transition system* $p \sqsubseteq q$ *implies* $Acc(p) \ll_a Acc(q)$.

Proof: See [Hen88]. □

Question: Is this true if the condition on finite branching is dropped ? □

This is the situation, which is well-known, for testing in labelled transition systems. Let us now consider t-labelled transition systems. In fact we will restrict our attention to particular kinds of t-labelled transition systems, essentially those having the properties of TPL discussed in the previous section.

Definition 3.3 A t-labelled transition system is called *regular* if it satisfies

1. (Time-determinism) If $p \xrightarrow{\sigma} q$ and $p \xrightarrow{\sigma} r$ then $q = r$
2. (Maximal progress) If $p \xrightarrow{\sigma}$ then $p \not\xrightarrow{\tau}$
3. (Patience) If $p \not\xrightarrow{\tau}$ then $p \xrightarrow{\sigma}$

□

One can easily show that the characterisation of testing in terms of acceptances for labelled transition systems is no longer true when we consider regular t-labelled transition systems.

Example 3.4 Let p, q denote the terms $a + \tau.b$, $\tau.a + \tau.(a + b)$ respectively. Then $Acc(p) \ll_a Acc(q)$ but $p \not\sqsubseteq_t q$; they can be differentiated by the test $\sigma.(\overline{a}.fail + \overline{b})$. This is guaranteed by p because when the clock tick happens all possible synchronisations will have occurred, in particular all τ actions, and so only b is possible. However in q when the clock tick occurs a is also possible and using it to synchronise with the tester leads to the terminal unsuccessful state $fail \mid nil$. □

So in order to characterise testing in regular t-labelled transition systems we need to take into account more information about processes than that contained in their acceptances. But before tackling this problem another aspect of the example deserves comment. Both p and q are terms in *CCS* and are extensionally equivalent for untimed testing, i.e. $p \simeq q$ in the labelled transition system determined by *CCS*, but when considered as timed processes, i.e. as terms of *TPL*, they are not equivalent, $p \not\simeq_t q$. This is because although they are untimed processes which can not be distinguished using untimed tests there is a timed test which can tell them apart!. A more striking example of this phenomenon, taken from [Lan89], is given by the two processes

$$coin.(tea + hit.tea) + coin.(coffee + hit.coffee)$$

and

$$coin.(tea + hit.coffee) + coin.(coffee + hit.tea)$$

whose behaviour as labelled transition systems are given in Figure 4. They have exactly the same acceptances, namely

$$\begin{array}{rl} \epsilon\{\{coin\}\} & \\ coin\{\{tea, hit\}\} & coin\{\{coffee, hit\}\} \\ coin.tea\{\emptyset\} & \\ coin.coffee\{\emptyset\} & \\ coin.hit.\{\{tea\}\} & coin.hit.\{\{coffee\}\} \\ coin.hit.tea.\{\emptyset\} & \\ coin.coffee.tea.\{\emptyset\} & \end{array}$$

But they can be distinguished by the timed test $\overline{coin}.(\overline{tea}. + \sigma.\overline{hit}.\overline{tea})$. This test says that if you can not do a *tea* action immediately after doing a *coin* action then you will be able to do so after performing a *hit* action.

This phenomenon may strike the reader as odd but on reflection it is not unnatural. These processes are really timed systems which when viewed as labelled transitions systems have their timing features abstracted away to such an extent that they can no longer be distinguished. Moreover this abstraction from time is consistent in the sense that so long as we only test using similarly abstracted processes then this level of abstraction can be maintained and we obtain coherent theory of "time-free" process descriptions. But once tests with timing information are allowed this level of abstraction is fractured and we must consider all the timing features of the process descriptions.

In fact it will be instructive as a first step to characterise timed testing as it applies to these apparently "time-free" processes. Again we do wish to work directly with the particular languages *CCS* and *TPL* but instead work at the level of transition systems.

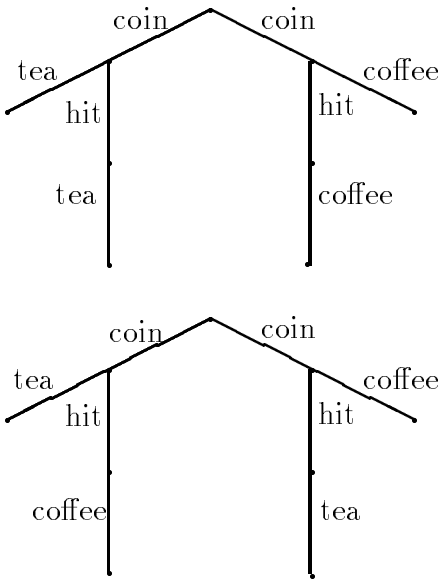

Figure 4: Langerak's Vending Machines

The appropriate notion of a "time-free" process in an arbitrary t-labelled transition system is given by the following definition.

Definition 3.5 In any t-labelled transition system a process p is called

1. σ-constant whenever $p \stackrel{\sigma}{\longrightarrow} p$ if and only if $p \stackrel{\tau}{\not\longrightarrow}$
2. and h-σ-constant (for hereditarily σ-constant) whenever it is σ-constant and if $p \stackrel{\mu}{\longrightarrow} q$ for any μ in $Act_{\tau\sigma}$ then q is also h-σ-constant.[1]

□

One can check that in the t-labelled transition system of *TPL* all the processes from the sublanguage *CCS* are h-σ-constant and informally one can argue that it is precisely this property which enables them to be viewed abstractly as labelled transition systems as for h-σ-constant processes the timing information is completely schematic.

As a first step we characterise testing in a regular t-labelled transition system for h-σ-constant processes. The required generalisation of acceptances is the notion of barbs. They can be defined by adding one clause to the definition of acceptances:

1. Ω is a barb
2. A is a barb if A is a finite subset of Act
3. $\mu\underline{b}$ is a barb if $\underline{b}$ is a barb and $\mu \in Act_\sigma$
4. $A\underline{b}$ is a barb if $\underline{b}$ is a barb and A is a finite subset of Act

[1]Note that strictly speaking we should define the set of h-σ-constant processes as the largest set of processes all of whose elements satisfy these two conditions.

A barb therefore has the form

$$s_1 A_1 s_2 A_2 ... s_k X$$

where each $s_i \in (Act \cup \sigma)^*$, each A_i is a finite subset of Act and the final X is either Ω or again a finite subset of Act. This barb is generated by the process p if there is a derivation of the form

$$p \stackrel{s_1}{\Longrightarrow} p_1 \stackrel{s_2}{\Longrightarrow} p_2 \ldots \stackrel{s_k}{\Longrightarrow} p_k$$

where each p_i is stable, for $1 \leq i < k$ $S(p_i) = A_i$ and if the final X is Ω then $p_k \Uparrow$ and otherwise p_k is also stable with $S(p_k)$ equal to X. We use $Barb(p)$ to denote the set of barbs generated by p. Barbs are ordered in much the same way as acceptances: $\ll_b$ is the least relation over barbs which satisfies

$\Omega \ll_b \underline{b}$ for any barb $\underline{b}$

$A \ll_b A'$ if $A \subseteq A'$

$a\underline{b} \ll_b a\underline{b'}$ if $\underline{b} \ll_b \underline{b'}$

$A\underline{b} \ll_b A'\underline{b'}$ if $\underline{b} \ll_b \underline{b'}$ and $A \subseteq A'$

and is lifted to sets of barbs in exactly the same way as acceptances.

Theorem 3.6 *In any regular t-labelled transition system if p and q are h-σ-constant then $Barb(p) \ll_b Barb(q)$ implies $p \sqsubseteq_t q$.* □.

Question: Which of the clauses in the definition of regular t-labelled transition system can be dropped while maintaining this result ? □

Once again the converse depends on the expressive power of the t-labelled transition system in question and let us say that it is sufficiently expressive if it contains a denotation for every term in the language defined using *nil*, + and prefixing by every action in $Act_{\tau\sigma}$. We then obtain

Theorem 3.7 *In any finite branching regular t-labelled transition system if p and q are h-σ-constant then $p \sqsubseteq_t q$ implies $Barb(p) \ll_b Barb(q)$.* □

This means that for a large class of t-labelled transition systems barbs capture exactly the ability of "time-free" processes to guarantee tests. Is the same true for arbitrary processes ? Unfortunately the answer is no.

Example 3.8 Let p, q be the processes $\tau.(b + \sigma.a) + \tau.(a + c)$ and $a + b$. We will soon be able to check that $p \sqsubseteq_t q$ but their barbs are not properly related. The barb $\{a, b\}a$ is generated by q but p can not generate any comparable barb, i.e. any barb $\underline{b}$ such that $\underline{b} \ll_b \{a, b\}a$. □

So although barbs are satisfactory for processes whose timing behaviour is very simple they are too discriminating for arbitrary timed processes. We need to restrict attention to *standard* barbs, i.e. barbs of the form

$$s_1 A_1 \sigma s_2 A_2 \sigma s_3 ... s_k X$$

where each s_i is in Act^* i.e. they do not contain occurrences of σ. Let $SBarb(p)$ be the standard barbs associated with the process p.

Theorem 3.9 *In any sufficiently expressive finite-branching regular t-labelled transition system* $p \sqsubseteq_t q$ *if and only if* $SBarb(p) \ll_b SBarb(q)$. □

Question: Define a closure operator, C, on sets of barbs with the property that $B \ll_b B'$ if and only if $\mathcal{C}(B') \subseteq \mathcal{C}(B)$. □

Question: A t-labelled transition system is *divergence free* if it contains no infinite τ derivations. Define a closure operator $\mathcal{C}'$ on barbs with the property $\mathcal{C}'(B)$ is finite if B is finite and which satisfies $Barb(p) \ll_b Barb(q)$ if and only if $\mathcal{C}'(Barb(p)) \subseteq \mathcal{C}'(Barb(q))$, for p, q in a finite-branching divergence free t-labelled transition system. □

4 Equations

Here we show that the extensional equivalences can be captured by sets of equations or more generally inequations. The motivation for this work is that these equations form the basis for a proof system or syntactic transformation system for proving different descriptions extensionally equivalent. The results in this section are necessarily dependent on the particular process algebra we have chosen to work with and indeed the ability to obtain a simple equational characterisation of the extensional relations is a useful criterion when comparing different formalisms.

For the moment let us confine our attention to finite terms i.e. terms which do not use any process identifiers; we call these sublanguages *fCCS* and $fTPL$. Consider the proof system in Figure 5. It essentially allows rewriting of sub-terms using a set of equations E. Let us write $t \leq_E u$ to denote that $t \leq u$ can be derived in this proof system. The idea is to find a set of inequations E with the property that two finite processes, p, q are extensionally ordered if and only if $p \leq_E q$. The inequations E consist of pairs of terms which may be written as

$$t \leq t'$$

where both t and t' are allowed to use the operators of the language in question together with variables. So for example

$$x + y \leq y + x$$

is certainly a reasonable inequation for *fCCS*. We will also use the equation

$$t = t'$$

as a shorthand notation for the two inequations $t \leq t'$ and $t' \leq t$. Note that the proof system in general manipulates *open* terms, i.e. terms containing variables, while the langauges in which we are interested, *fCCS* and *fTPL* are variable-free. But the proof system allows instantiation of variables using Rule 4 and so inequalities involving variable-free terms or process terms can be derived. Note also that we only require the proof system to capture the extensional preorder between process terms and not open terms. The latter would be a much more difficult requirement.

We first review the situation for *fCCS* and untimed testing which is well-known from [DH84]. The first problem is that by definition the relation $\leq_E$ is preserved by all contexts whereas this is not the case for $\sqsubseteq$. The standard example is $a \sqsubseteq \tau.a$ but $b + a \not\sqsubseteq b + \tau.a$ as $b + a$ guarantees the test $\overline{b}$. The plus operator of *CCS* creates this problem for many of the usual extensional preorders and the standard response is to replace $\sqsubseteq$ with the relation $\sqsubseteq^+$ defined by

1.	*Reflexivity*	$\dfrac{}{t \leq t}$	
2.	*Transitivity*	$\dfrac{t \leq t',\ t' \leq t''}{t \leq t''}$	
3.	*Substitution*	$\dfrac{\underline{t} \leq \underline{t}'}{f(\underline{t}) \leq f(\underline{t}')}$	for every operator f
4.	*Instantiation*	$\dfrac{t \leq t'}{t\rho \leq t'\rho}$	for every substitution ρ
5.	*Inequations*	$\dfrac{}{t \leq t'}$	for every inequation $t \leq t'$ in E
6.	$\Omega - Rule$	$\dfrac{}{\Omega \leq t}$	

Figure 5: The Proof System

$p \sqsubseteq^+ q$ if for some action a not occurring in $p,\ q,\ \ a + p \sqsubseteq a + q$.

One can show that $\sqsubseteq$ is preserved by all the operators of *CCS* and moreover is the largest such relation contained in $\sqsubseteq$. Accordingly we transfer our attention to giving an inequational characterisation for $\sqsubseteq^+$ and the related congruence $\simeq^+$.

Question: Show that for any p, q, r the following is true:

$$\begin{array}{rcl} ((\tau.p + \tau.q) \mid r)\backslash A & \simeq^+ & \tau.(p \mid r)\backslash A + \tau.(q \mid r)\backslash A \\ (a.p \mid \overline{a}.q \mid r)\backslash\{a\} & \simeq^+ & \tau.(p \mid q \mid r)\backslash\{a\} \end{array}$$

□

Many of the required equations are not of any particular interest and are relegated to the appendix; these simply state obvious properties of restriction, renaming, Ω and $+$, most of which are discussed at length in [Mil89]. Here we will concentrate on two aspects of the equational system, the interleaving law which relates parallel composition to nondeterminism and the laws governing τ. The first is given in Figure 6 and applies only to terms of the form $\sum_I \alpha_i.x_i \mid \sum_J \beta_j.y_j$. This parallel term can be rewritten to a nondeterministic term which essentially has a prefix for each of the possible actions which x and y can perform either individually or together. The τ laws, of which there are five, are given Figure 7.

Let E_1 denote the collection of all these inequations.

Theorem 4.1 *For all p, q in fCCS $p \sqsubseteq^+ q$ if and only if $p \leq_{E_1} q$.*

For $x = \sum_I \alpha_i.x_i$ and $y = \sum_J \beta_j.y_j$

$$x \mid y \quad = \quad \sum_I \alpha_i.(x_i \mid y) + \sum_J \beta_j.(x \mid y_j) + \sum_{\alpha_i = \overline{\beta}_j} \tau.(x_i \mid y_j) \quad (I1)$$

Figure 6: Interleaving Law for *CCS*

Proof: See [DH84]. □

Instead of finding directly an equational characterisation for $\sqsubseteq_t$ over *fTPL* let us first consider the sublanguage *CCS*. Again we have to consider $\sqsubseteq_t^+$ in place of $\sqsubseteq_t$ and so we want to find a set of equations E_2 with the property that for all finite *CCS* terms $p \sqsubseteq_t q$ if and only if $p \leq_{E_2} q$. The obvious place to start is with the set E_1. It turns out that all the equations in the appendix are still valid as is the interleaving law; we only have to modify the τ-laws. The first two remain true for $\sqsubseteq_t^+$ but the remaining three are in general false. The equation $(\tau 3)$ is still true if α is equal to τ but for external actions it is false in general. For example

$$a.b + \tau.a.c \not\sqsubseteq_t \tau.(a.b + a.c).$$

The test $\sigma.\overline{a}.\overline{c}$ distinguishes them or alternatively the barb $\{a\}ab$ of $\tau.(a.b + a.c)$ can not be properly matched by any barb of $a.b + \tau.a.c$. We have already seen in the previous section the essential reason why $(\tau 4)$ is not true:

$$a + \tau.b \not\sqsubseteq_t \tau.(a + b)$$

because of the test $\sigma.(a.\mathit{fail} + b)$. Finally the equation $(\tau 5)$ is true of $\sqsubseteq_t$ but not of $\sqsubseteq_t^+$ because the same reasoning as in the previous case will show that

$$a + \tau.b \not\sqsubseteq_t (a + b).$$

$$\begin{aligned}
\tau.x + \tau.y &\leq \tau.x && (\tau 1)\\
\alpha.x + \alpha.y &= \alpha.(\tau.x + \tau.y) && (\tau 2)\\
\alpha.x + \tau.(\alpha.y + z) &= \tau.(\alpha.x + \alpha.y + z) && (\tau 3)\\
x + \tau.y &\leq \tau.(x + y) && (\tau 4)\\
\tau.x &\leq x && (\tau 5)
\end{aligned}$$

Figure 7: τ Laws for *CCS*

$$
\begin{array}{rcll}
\tau.x + \tau.y & \leq & \tau.x & (\tau N1) \\
\alpha.x + \alpha.y & = & \alpha.(\tau.x + \tau.y) & (\tau N2) \\
\tau.x + y & = & \tau.(\tau.x + y) + \tau.x & (\tau N3) \\
\tau.x + \tau.y & \leq & \tau.(x + y) & (\tau N4) \\
\tau.x & \leq & \tau.x + x & (\tau N5)
\end{array}
$$

Figure 8: The new τ laws

The new τ laws are given in Figure 8. The least two inequations, $(\tau 4)$ and $(\tau 5)$ survive in the slightly weaker form of $(\tau N4)$ and $(\tau N5)$ but $(\tau 3)$ disappears entirely to be replaced by a new law $(\tau N3)$ which is only concerned with τs. If we let E_2 denote this revised set of equations we now have

Theorem 4.2 *For every pair of finite CCS processes p $\sqsubseteq_t^+ q$ if and only if $p \leq_{E_2} q$.* $\Box$

When we move to the full language, *fCCS*, then we have to add equations for the extra operator $\lfloor\ \rfloor(\ )$. The interaction with the operators Ω, $\backslash A$ and $[S]$ are straightforward and given in the appendix. There is an interesting interplay between delay and the two operators *nil* and prefixing captured by the equations

$$
\begin{array}{rcll}
nil & = & \lfloor nil \rfloor(nil) & (\sigma 1) \\
a.x & = & \lfloor a.x \rfloor(a.x) & (\sigma 2)
\end{array}
$$

These follow because all processes definable in *TPL* are patient. There are two further laws governing the behaviour of $\lfloor\ \rfloor(\ )$:

$$
\begin{array}{rcll}
\lfloor\lfloor x \rfloor(y)\rfloor(z) & = & \lfloor x \rfloor(z) & (\sigma 3) \\
\lfloor x \rfloor(y) + \lfloor u \rfloor(v) & = & \lfloor x + u \rfloor(y + v) & (\sigma 4)
\end{array}
$$

The first reflect the fact that in $\lfloor p \rfloor(q)$ the arrival of the time cycle preempts all the activity of p including possible σ moves it is capable of while the second is true because the passage of time is deterministic. There are also three laws governing it's interaction with τ:

$$
\begin{array}{rcll}
\lfloor \tau.x \rfloor(y) & = & \tau.x & (\sigma\tau 1) \\
\tau.\lfloor x \rfloor(y) & = & \tau.\lfloor x \rfloor(\tau.y) & (\sigma\tau 2) \\
x + \tau.\lfloor y \rfloor(z) & \leq & \tau.\lfloor x + y \rfloor(z) & (\sigma\tau 3)
\end{array}
$$

Finally we must add a new interleaving law. The existing one $(I1)$ can only be used for terms in *sum forms*, i.e. terms of the form $\alpha_1.t_1 + \ldots + \alpha_k.t_k$ but with the new time-out operator it is obvious that not all terms can be reduced to these forms. The new interleaving law is given in Figure 9; it is very similar in style to the previous one except that it now applies to terms of the form $\lfloor t \rfloor(u)$ where t is a sum form.

Let E_3 denote this new revised set of equations which must include $(I1)$ as it can not be derived from the more general $(I2)$.

For $x = \lfloor \sum_I \alpha_i.x_i \rfloor (x_\sigma)$ and $y = \lfloor \sum_J \beta_j.y_j \rfloor (y_\sigma)$

$$x \mid y \quad = \quad \lfloor \sum_I \alpha_i.(x_i \mid y) + \sum_J \beta_j.(x \mid y_j) + \sum_{\alpha_i = \overline{\beta}_j} \tau.(x_i \mid y_j) \rfloor (x_\sigma \mid y_\sigma) \quad (I2)$$

Figure 9: The new interleaving law

Theorem 4.3 *For any pair of terms in fTPL, $p \sqsubseteq_t^+ q$ if and only if $p \leq_{E_3} q$.* □

This completes the story for finite terms and the moral is that there is very little difference between the untimed and timed languages; in essence one gets a slightly more complicated interleaving law and the τ laws need to be changed. It should however be pointed out that the choice of timing construct, $\lfloor\ \rfloor(\ )$ originally defined in [NS90], was not arbitrary. For example one would have problems if the less general construct $\sigma.-$ were used instead as it would be difficult to find a useful interleaving law. The problem is that a process such as $a \mid \sigma.b$ is not persistent. It can initially perform an a action to $nil \mid \sigma.b$ but after the time click it becomes the process $a \mid b$ which can no longer do an a action to a similar state. All processes definable in the finite language without parallel and with the time-out replaced by $\sigma.-$ are persistent. It therefore follows that there can be no interleaving law which will eliminate the parallel construct $\mid$ from the process $a \mid \sigma.b$. The time-out construct enables us to define non-persistent processes which occur naturally in the presence of parallelism.

We end this section with a brief survey of how recursive definitions are treated in the proof system. The first obvious rule, which we call (Rec), is that recursive definitions may be unwound:

$$P \quad = \quad D(P)$$

Another more complicated rule is called *Recursion Induction*:

> If $\{ P_i \Longleftarrow D_i \mid i \in I \}$ is a declaration and for each $i \in I$ there is a process q_i such that $D_i\{\underline{q}/\underline{P}\} \leq q_i$ then, for each $i \in I$, $P_i \leq q_i$.

To prove the soundness of this rule is not straightforward. It relies essentially on the fact that if a process p guarantees a test e then there is a finite approximation to p, a finite term obtained by unwinding the definitions used in p to some arbitrary depth, which also guarantees e.

Recursion Induction together with the equations will not however give a complete proof system; neither will stronger forms of induction such as Scott Induction. The essential reason is that $\simeq$ is not even partially recursive but the set of theorems in a proof system is r.e.. However one way to say that "recursion + equations" is complete for the entire language is to introduce an infinitary form of induction which in [Hen88] is called ω-induction. One can then show for arbitrary terms in *CCS* that $p \sqsubseteq_t^+ q$ if and only if $p \leq q$ can be derived using the inequations E_1 together with (Rec) and ω-induction.

It is interesting to note that the corresponding result is *not* true for *TPL*; it is because all processes are patient. For example

$$a.nil \sqsubseteq_t A$$

where A is defined by $A \Longleftarrow \lfloor a \rfloor(A)$ and this can not be proved by the equations even with the help of ω-induction!. We need to add one more rule, called the *Stability* rule, to the proof system:

$$\frac{}{\sum_{i \in I} \alpha_i.t_i \leq S}$$

where S is defined by $S \Longleftarrow \lfloor \sum_{i \in I} \alpha_i.t_i \rfloor(S)$.

A quite useful form of induction is called *Unique Fixpoint Induction*. The form is identical that of Recursion Induction but the inequality is replaced by equality.

> If $\{ P_i \Longleftarrow D_i \mid i \in I \}$ is a declaration and for each $i \in I$ there is a process q_i such that $D_i\{\underline{q}/\underline{P}\} = q_i$ then for each $i \in I$, $P_i = q_i$

Unfortunately it is unsound for the standard extensional equivalences including $\simeq$ and $\simeq_t$. However for a large class of definitions it is sound and it is the rule which tends to be most used in the literature. If we ensure it is used only for definitions where in the bodies all occurrences of process names are all guarded by external actions and do not occur within occurrences of $|$ then we will not run into difficulty.

Question: Show that if we allow process names to be guarded by σ then Unique Fixpoint Induction becomes unsound.
Give an example which shows that if we allow occurrences of process names within $|$, even if they are guarded by external actions, then the rule is also unsound. □

5 An Example Proof

In this section we discuss how one might use the proof system to prove properties of of timed systems. We use as an example the security costs protocol of section two.

As in the untimed case the proof methodology consists in using the proof system as the basis for a transformation system for manipulating process descriptions. The most common use of the resulting transformation system is to transform a description of a high-level specification of a system into a more detailed description of a proposed implementation. Refering to the example in section two this means transforming $SPEC$ into SYS. Since the proof system is complete in some sense in theory one should be able to transform any two behaviourally related processes into each other using the inequations. But is practice it is virtually essential to augment the set of equations with more useful transformations. For example

$$\begin{aligned} x \mid y &= y \mid x \\ x \mid nil &= nil \mid x \\ (x \mid y) \mid z &= x \mid (y \mid z) \end{aligned}$$

are all sound and are not derivable in the complete proof system. Two other interesting transformations involving parallel are

$$\begin{aligned} ((\tau.x_1 + \tau.x_2) \mid y)\backslash A &= \tau.(x_1 \mid y)\backslash A + \tau.(x_2 \mid y)\backslash A \\ (a.x \mid \overline{a}.y \mid z)\backslash A &= \tau.(x \mid y \mid z)\backslash A \quad \text{if } a \in A \end{aligned}$$

However the most useful transformation rule is a generalisation of the interleaving law to the case where an arbitrary number of processes are running in parallel and a restriction is in force. The timed expansion theorem is a straightforward generalisation of the corresponding theorem for CCS, [Mil89]:

Let P denote a process of the form $(P_1 \mid \ldots \mid P_n)\backslash A$ where $n \geq 1$. Then

$$P = \lfloor imm \rfloor (delay) \qquad (EXP)$$

where

$$\begin{aligned} imm \;=\; & \sum\{\, \alpha_i.(P_1 \mid \ldots \mid P_i' \mid \ldots \mid P_n)\backslash A \;\mid\; P_i \xrightarrow{\alpha_i} P_i',\ A \text{ admits } \alpha_i \,\} \\ & + \sum\{\, \tau.(P_1 \mid \ldots \mid P_i' \mid \ldots \mid P_j' \mid \ldots \mid P_n)\backslash A \;\mid\; P_i \xrightarrow{a} P_i',\ P_j \xrightarrow{\overline{a}} P_j' \,\} \end{aligned}$$

and

$$delay \;=\; \sum\{\, (P_1^\sigma \mid \ldots \mid P_n^\sigma)\backslash A \mid P_i \xrightarrow{\sigma} P_i^\sigma \,\}$$

This is quite a powerful rule although somewhat complicated. However there are simpler versions when P has particular properties. For example, using the same notation, we have

If $P \xrightarrow{\tau}$ then $P = imm$ $\qquad (EXP\tau)$

If P is σ-constant, i.e. $P \xrightarrow{\sigma} P$, then $P = imm$ $\qquad (EXP\sigma)$

Note that in each of these cases the transformation is identical to that of the untimed case, in [Mil89].

Syntactic versions of the expansion theorem can be obtained by assuming that each P_i has a particular form. For example if each P_i is a σ-form, i.e. has the form $\sum_{j\in I_i} \lfloor Q_{ij} \rfloor (P_i^\sigma)$ then the calculation of imm leads to

$$\begin{aligned} & \{\, \alpha_{ij}.(P_1 \mid \ldots \mid Q_{ij}' \mid \ldots \mid P_n)\backslash A \;\mid\; Q_{ij} \xrightarrow{\alpha_i} Q_{ij}',\ A \text{ admits } \alpha_i \,\} \\ & + \sum\{\, \tau.(P_1 \mid \ldots \mid Q_{ij} \mid \ldots \mid Q_{kl} \mid \ldots \mid P_n)\backslash A \;\mid\; \alpha_{ij} = \overline{\alpha_{kl}} \,\} \end{aligned}$$

while $delay$ remains the same. If each P_i is a sum-form, i.e. has the simpler form $\sum_{j\in I_i}\{Q_{ij}\}$ then we obtain

$$P = imm$$

where imm is calculated syntactically as above.

Let us now see how to use the expansion theorem together with the other equations to prove $SPEC = SYS$ where these processes are defined in section 2. This will follow immediately by Unique Fixpoint Induction if we can prove

$$SYS = a.(\tau.\sigma.\overline{b}.\ SYS + \tau.\overline{b}.\ SYS).$$

Note that this represents a sound application of Unique Fixpoint Induction since in the body all occurrences of SYS are guarded.

Let us start by manipulating SYS. By unwinding each of the recursive definitions, applying $EXP\sigma$ and rewinding we obtain

$$SYS = a.(A_1 \mid RM \mid UM \mid B)\backslash I$$

where A_1 is $\overline{m}_u.\lfloor ack.\ ack.A \rfloor(\overline{m}_r.\ ack.A)$. This procedure of unwinding definitions, applying some form of the expansion theorem and rewinding some of the resulting processes is a very frequently used proof tactic. In fact the unwinding and rewinding of definitions is so persuasive we will not mention it in future; instead we only indicate the variety of expansion theorem used. The next application is ($EXP\tau$) from which we obtain

$$SYS = a.(A_2 \mid RM \mid UM_1 \mid B)\backslash I$$

where A_2 is $\lfloor ack.\ ack.A \rfloor(\overline{m}_r.\ ack.A)$ and UM_1 is $\tau.\ UM + \tau.\overline{m}_{ub}.\ UM$. By applying the rule

$$((\tau.x_1 + \tau.x_2) \mid y)\backslash A = \tau.(x_1 \mid y)\backslash A + \tau.(x_2 \mid y)\backslash A$$

we obtain

$$SYS = a.(\tau.S_1 + \tau.S_2)$$

where S_1, S_2 denote $(A_2 \mid RM \mid UM \mid B)\backslash I$, $(A_2 \mid RM \mid \overline{m}_{ub}.\ UM \mid B)\backslash I$ respectively.

1. We show $S_1 = \sigma.\tau.\overline{b}.\ SYS$
   Applying (EXP) we obtain

   $$S_1 = \lfloor nil \rfloor((\overline{m}_r.\ ack.A \mid RM \mid UM \mid B)\backslash I).$$

   The proof proceeds by five more applications of various forms of the expansion theorem:

   $$\begin{array}{rcll} S_1 & = & \sigma.(\overline{m}_r.\ ack.A \mid RM \mid UM \mid B)\backslash I & (EXP\sigma) \\ & = & \sigma.\tau.(\ ack.A \mid \overline{m}_{rb}.\ RM \mid UM \mid B)\backslash I & (EXP\tau) \\ & = & \sigma.\tau.\tau.(\ ack.A \mid RM \mid UM \mid \overline{b}.\overline{ack}_b.B)\backslash I & (EXP\tau) \\ & = & \sigma.\tau.\tau.\overline{b}.(\ ack.A \mid RM \mid UM \mid \overline{ack}_b.B)\backslash I & (EXP\sigma) \\ & = & \sigma.\tau.\tau.\overline{b}.\tau.(\ ack.A \mid \overline{ack}.\ RM \mid UM \mid B)\backslash I & (EXP\tau) \\ & = & \sigma.\tau.\tau.\overline{b}.\tau.\tau.(A \mid RM \mid UM \mid B)\backslash I & (EXP\tau) \end{array}$$

   Using the equation ($\tau N2$) one can prove

   $$\alpha.x = \alpha.\tau.x$$

   and by its repeated application we obtain $S_1 = \sigma.\tau.\tau.\overline{b}.\ SYS$ and one further application gives the required $S_1 = \sigma.\tau.\overline{b}.\ SYS$.

2. We show $S_2 = \tau.\overline{b}.\ SYS$.
   The approach is very similar. The expansion theorem is applied repeatedly to obtain a skeleton of the behaviour and then simplification laws, usually based on the elimination of τs, are applied. With four applications of the expansion theorem we obtain

   $$S_2 = \tau.\tau.(\overline{b}.\tau.S_3 + \tau.S_4)$$

where S_3 represents ($ack.A \mid RM \mid UM \mid \overline{ack_b}.B)\backslash I$ and S_4 the process ($ack.A \mid RM \mid UM \mid \overline{b}.\,\overline{ack_b}.B)\backslash I$. One application of ($EXP\sigma$) gives $S_4 = \overline{b}.S_3$ so we now have

$$S_2 = \tau.\tau.(\overline{b}.\tau.S_3 + \tau.\overline{b}.S_3).$$

On the other hand three applications of ($EXP\tau$) gives

$$S_3 = \tau.\tau.\tau.SYS.$$

which leads to

$$S_2 = \tau.\tau.(\overline{b}.\tau.\tau.\tau.\tau.\ SYS + \tau.\overline{b}.\tau.\tau.\tau.\ SYS).$$

The same τ-reduction rule, $\alpha.x = \alpha.\tau.x$, reduces this to

$$S_2 = \tau.(\overline{b}.\ SYS + \tau.\overline{b}.\ SYS).$$

Another derived equation is $x + \tau.x = \tau.x$ which when applied gives the required

$$S_2 = \tau.\overline{b}.\ SYS.$$

Combining these two sub–proofs we now have

$$SYS = a.(\tau.\sigma.\tau.\overline{b}.\ SYS + \tau.\overline{b}.\ SYS)$$

Applying the equation ($\sigma\tau 2$) we obtain the required

$$SYS = a.(\tau.\sigma.\overline{b}.\ SYS + \tau.\overline{b}.\ SYS)$$

This completes the proof that $SYS = SPEC$ and as we have seen it consists of a large number of applications of the expansion theorem with periodic interventions using τ-reduction rules. The proof is no different in style than corresponding proofs for time-free processes. In other words we can apply the techniques originally developed for standard process algebras to prove properties of at least some types of time dependent systems. Performing such proofs is undoubtedly tedious but they are eminently suited to mechanical assistance. Software systems have already been developed which help in the development of these proofs, [Lin91, MV89], and they can easily be extended to handle *TPL*. For example the above proof has been carried out by the system *PAM*, [Lin91].

6 Extensions

The language we have investigated is somewhat simple and is best viewed as the core of more extensive and more useful timed languages. This core can be extended in many ways and the choice is probably best made in the light of intended applications. Here we briefly sketch two possibilities; one concentrates on the passage of time and introduces more constructs for the manipulation of the implicit time variable underlying the operational semantics while the other adds *urgent* or *insistent* actions.

Manipulating Time

To make descriptions in the language more compact one can easily extend the syntax with a variety of notational conventions; a large number may be found in [NS90]and here we will examine a small selection. For any $k \geq 0$ $\sigma^k.p$ can be viewed as a shorthand for $\sigma.\ldots.\sigma.p$ which means delay for $k+1$ time-cycles before continuing like p. More generally we can define a *delay start* operator, $\lfloor p \rfloor^k(q)$. Intuitively $\lfloor p \rfloor^k(q)$ behaves like p provided p can perform an action within k clock cycles and otherwise, after the k^{th} occurrence of the clock tick, it behaves like q. As an example of its use consider

$$\begin{array}{rcl} VM & \Longleftarrow & coin.VM' \\ VM' & \Longleftarrow & \lfloor \sigma^2.\overline{tea}.VM + \sigma^3.\overline{coffee}.VM \rfloor^{30}(VM) \end{array}$$

After receiving a coin tea is ready in two clock cycles while coffee takes three and after thirty seconds the machine reverts to its original state and the coin is lost.

There are two ways of viewing the extension of the language with the operator $\lfloor\ \rfloor^k(\ )$. In the first the syntax of the language is actually extended by adding an infinite set of new operators, $\lfloor\ \rfloor^k(\ )$, one for each $k \geq 0$. The operational semantics of the language must now be also extended to cover processes which use the new operators; this amounts to adding extra clauses to the structural operational rules in Figure 2. The appropriate rules, which reflect our intuition, are

$$p \stackrel{\alpha}{\longrightarrow} p' \quad \text{implies} \quad \lfloor p \rfloor^k(q) \stackrel{\alpha}{\longrightarrow} p'$$

$$p \stackrel{\tau}{\not\longrightarrow} \quad \text{implies} \quad \lfloor p \rfloor^0(q) \stackrel{\sigma}{\longrightarrow} q$$

$$p \stackrel{\sigma}{\longrightarrow} p' \quad \text{implies} \quad \lfloor p \rfloor^{k+1}(q) \stackrel{\sigma}{\longrightarrow} \lfloor p' \rfloor^k(q)$$

One can show that with this extension one still obtains a regular t-labelled transition system and therefore the characterisation of testing in terms of barbs also applies to this language. However it is necessary to check that the behaviour preorder is preserved by the new operator, i.e. $p \sqsubseteq_t p'$, $q \sqsubseteq_t q'$ implies $\lfloor p \rfloor^k(q) \sqsubseteq_t \lfloor p' \rfloor^k(q')$. Finally one must find equations which capture entirely the behaviour of the new operator, i.e. equations which when added to the set given in section four provide a complete axiomatisation for the extended language. In this case the required equations are

$$\begin{array}{rcl} \lfloor t \rfloor^0(u) & = & \lfloor t \rfloor(u) \\ \lfloor \lfloor t \rfloor(u) \rfloor^{k+1}(r) & = & \lfloor t \rfloor(\lfloor u \rfloor^k(r)) \end{array}$$

The other approach is simply to view $\lfloor\ \rfloor^k(\ )$ as a notational convention or a convenient shorthand for writing more complicated terms. In this case one need to supply the translation from $\lfloor t \rfloor^k(u)$ into the core language for each possible k, t and u. The essence of this translation is provided by the above equations because as it happens every process can be transformed into the form $\lfloor p \rfloor(q)$ for some p and q.

Question: Given an operational semantics for a *watchdog* operator $\lceil\ \rceil^k(\ )$ where intuitively $\lceil p \rceil^k(q)$ acts like p until the k^{th} clock cycle; from then on it acts like q.

Show that $\sqsubseteq_t$ is preserved by this operator and give a set of characterising equations for it. □

In the definition of these type of operators *constants* over the time domain are used. In [Wan91] a further step is taken, namely the introduction of *time-variables*. In order to take advantage of their presence syntactic constructs need to be generalised also. So parameterised definitions of the form

$$P(\mathtt{t}) \Longleftarrow p$$

are now allowed where $\mathtt{t}$ is a time-variable which may occur in p. The behaviour of the process p should in general depend on the value assigned to the occurrences of its time-variables and one way of ensuring this is to introduce a construct such as $\lfloor\ \rfloor^{\mathtt{t}}(\ )$ or more generally $\lfloor\ \rfloor^{\mathtt{e}}(\ )$ where $\mathtt{e}$ is some *time-expression*, i.e. an expression which evaluates to a time when all its variables are instantiated. We will continue to use $\sigma^{\mathtt{e}}.p$ as a shorthand for $\lfloor nil \rfloor^{\mathtt{e}}(p)$. Moreover we can allow special kinds of *input* actions which may read in the values to be associated with these variables. These take the form $c?\mathtt{t}$ which intuitively means "read in the value of the time-variable $\mathtt{t}$ from the channel c". For the sake of symmetry we should also allow *output* actions of the form $c!\mathtt{e}$. As an example consider the definition

$$\begin{array}{rcl} T_0 & \Longleftarrow & settime?\mathtt{t}.T_1(\mathtt{t}) \\ T_1 & \Longleftarrow & \sigma^{\mathtt{t}}.T_2 \\ T_2 & \Longleftarrow & \overline{timeout}.T_0 \end{array}$$

Initially this process is ready to input a time which is used to instantiate the variable $\mathtt{t}$. So for example the time can be set to 30 by the action

$$T_0 \stackrel{settime?30}{\longrightarrow} T_1(30).$$

The process $T_1(30)$ can only perform thirty consecutive σ actions and then perform a timeout to arrive at its original state.

In [Wan91] a further form of action prefixing is allowed, $a@\mathtt{t}.p$, which enables the behaviour of a process to depend on the length of delay before the action is performed. This is still a patient process and so can delay indefinitely, i.e. perform σ actions. But whenever one is performed the value associated with the variable $\mathtt{t}$ is increased. Since we are assuming discrete time this means that we have the the rule

$$a@\mathtt{t}.p \stackrel{\sigma}{\longrightarrow} a@\mathtt{t}.(p[\mathtt{t}+1/\mathtt{t}]).$$

When it does get around to performing the action a the variable $\mathtt{t}$ is instantiated to zero,

$$a@\mathtt{t}.p \stackrel{a}{\longrightarrow} p[0/\mathtt{t}],$$

so that in effect in a move of the form

$$a@\mathtt{t}.p \stackrel{\sigma}{\longrightarrow} \ldots \stackrel{\sigma}{\longrightarrow}\stackrel{a}{\longrightarrow} p'$$

the term p' is obtained by instantiating the variable $\mathtt{t}$ in p to the number of clock cycles in the sequence $\stackrel{\sigma}{\longrightarrow} \ldots \stackrel{\sigma}{\longrightarrow}$. A simple example of its use is in the definition of a stop-watch.

$$\begin{array}{rcl} S & \Longleftarrow & start.G \\ G & \Longleftarrow & stop@\mathtt{t}.D(t) \\ D(t) & \Longleftarrow & display!t.S \end{array}$$

The possible computations from S are of the form

$$S \xrightarrow{start} G \xrightarrow{\sigma} \ldots \xrightarrow{\sigma} \xrightarrow{stop} \xrightarrow{display!k} S$$

where k is the amount of lapsed time, i.e. the length of the sequence $\xrightarrow{\sigma} \ldots \xrightarrow{\sigma}$.

A slightly more complicated form of timer may be defined by

$$\begin{array}{rcl} W & \Longleftarrow & settime?\mathtt{t}.S(\mathtt{t}) \\ S(\mathtt{t}) & \Longleftarrow & start.G(\mathtt{t}) \\ G(\mathtt{t}) & \Longleftarrow & \lfloor\, stop@\mathtt{u}.S(\mathtt{t}-\mathtt{u})\rfloor^{\mathtt{t}}(\overline{timeout}.W) \end{array}$$

Here W can receive a time, say 10, and then be started to become $G(10)$. In this state it awaits 10 clock cycles and then performs the timeout action. But while waiting it can also be stopped. For example after 6 clock cycles it is in the state $\lfloor\, stop@\mathtt{u}.S(\mathtt{10}-(\mathtt{u}+\mathtt{6}))\rfloor^{\mathtt{t}}(\overline{timeout}.W)$ where it can perform the action $stop$ to the state $S(4)$. In this state it can be restarted at will and the remaining 4 clock cycles will be counted down - unless it is stopped once more.

Recapitulating the language in question now looks like

$$\begin{array}{rcl} p & ::= & \Omega \mid nil \mid P \mid \alpha.p \mid \quad \mid p+p \mid p \mid p \\ & & \mid\ p[S] \mid p\backslash A,\ A \subseteq Act \\ & & \lfloor p\rfloor^{\mathtt{e}}(p) \mid a@\mathtt{t}.p \mid a?\mathtt{t}.p \mid a!\mathtt{e}.p \end{array}$$

where $\mathtt{t}$ ranges over a set of time-variables and $\mathtt{e}$ over some language for time expressions and of course further timing constructs of interest may be added. We will not formally define the operational semantics as it can easily be constructed from the outline given above. It does once more lead to a regular t-labelled transition system and therefore the basic results about the behavioural preorder carry over.

However when extending the equational theory of TPL to this new language considerable care needs to be taken. This is because both forms of prefixing, $a?\mathtt{t}$ and $a@\mathtt{t}$ act like binders for the time-variable $\mathtt{t}$; for example for the term $a?\mathtt{t}.p$ all occurrences of $\mathtt{t}$ in the subterm p are bound. Terms with free occurrences of variables, i.e. occurrences which are not bound, can not directly be given an operational semantics as their behaviour will in general depend on how these variables are instantiated. So let us call a term with no free variables a *process*. Then the operational semantics and the resulting testing preorder applies to directly to processes. However the preorder can be extended to arbitrary terms in a standard way by defining

> $p \sqsubseteq_t q$ if for every instantiation of time-variables ρ the processes $p\rho$, $q\rho$ are related, i.e. $p\rho \sqsubseteq_t q\rho$.

Thus the presence of variables brings us outside the standard theory of equational algebras as presented in, say, [Gue81]. As a simple example the validity of a statement such as

$$\lfloor p\rfloor^{\mathtt{e}}(q) \simeq^{+} \lfloor p'\rfloor^{\mathtt{e}'}(q')$$

depends on the relationship between the two time-expressions $\mathtt{e}$ and $\mathtt{e}'$. However instead of getting involved in this issue we finish the subsection with an example of the use of the language. The reader interested in the treatment of values is refered to [HI91].

We describe a simple distributed implementation of the vending machine using a timer and a separate unit for brewing the drinks. The timer is defined as follows:

$$\begin{array}{rcl} T & \Longleftarrow & settime?\mathtt{t}.W(\mathtt{t}) \\ W(\mathtt{t}) & \Longleftarrow & \lfloor\ reset.T\rfloor^{\mathtt{t}-1}(T') \\ T' & \Longleftarrow & \overline{timeout}.T + reset.T \end{array}$$

while the independent unit for brewing the tea and coffee is given by:

$$\begin{array}{rcl} C & \Longleftarrow & coin.settime!4.B \\ B & \Longleftarrow & \sigma^2.\overline{tea}.F + \sigma^3.\overline{coffee}.F + timeout.C \\ F & \Longleftarrow & \overline{reset}.C \end{array}$$

Let the implementation be defined by

$$I \Longleftarrow (C \mid T)\backslash A$$

where A is the set $\{ settime, timeout, reset\}$. The behaviour of this system is slightly different than the other vending machines we have seen. In particular it has some genuinely nondeterministic behaviour. After the fourth clock tick one may still obtain a drink but this is not guaranteed. The complete behaviour is defined by

$$\begin{array}{rcl} S & \Longleftarrow & coin.S' \\ S' & \Longleftarrow & \lfloor\sigma^2.\overline{tea}.S + \sigma^3.\overline{coffee}.S\rfloor^3(\overline{tea}.S + \overline{coffee}.S + \tau.S) \end{array}$$

We now outline a proof that I is an implementation of S, i.e. $I \simeq^+ S$, by showing how to transform one into the other using a proof system based on that in section four. This involves a use of Unique Fixpoint Induction; we will show that

$$\begin{array}{rcl} I & = & coin.I' \\ I' & = & \lfloor\sigma^2.\overline{tea}.I + \sigma^3.\overline{coffee}.I\rfloor^3(\overline{tea}.I + \overline{coffee}.I + \tau.I) \end{array}$$

for some term I', from which the result will then follow. It is quite straightforward to discover the required term I'. By two applications of an interleaving law we obtain

$$I = coin.\tau.(B \mid W(4))\backslash A.$$

So let I' denote the term $(B \mid W(4))\backslash A$. Using τ-absorption we obtain

$$I = coin.I'$$

and therefore it remains to show

$$I' = \lfloor\sigma^2.\overline{tea}.I + \sigma^3.\overline{coffee}.I\rfloor(\overline{tea}.I + \overline{coffee}.I + \tau.I)$$

By expanding out recursive definitions and doing some rearrangements I' may be rewritten to a form susceptible to an expansion theorem:

$$((\lfloor nil\rfloor(\sigma.\overline{tea}.F + \sigma^2.\overline{coffee}.F + timeout.C)) \mid (\lfloor\ reset.T\rfloor^3(T')))\backslash A.$$

On applying the theorem we obtain

$$I' = \sigma.I_1$$

where

$$I_1 \text{ is } ((\sigma.\overline{tea}.F + \sigma^2.\overline{coffee}.F + \ timeout.C) \mid (\lfloor reset.T \rfloor^2(T')))\backslash A.$$

Repeating this procedure we obtain

$$I_1 = \sigma.I_2$$

where

$$I_2 \text{ is } ((\overline{tea}.F + \sigma^1.\overline{coffee}.F + \ timeout.C) \mid (\lfloor reset.T \rfloor^1(T')))\backslash A.$$

This time the expansion theorem gives

$$I_2 = \lfloor \overline{tea}.Y \rfloor(I_3)$$

where

$$Y \text{ is } (F \mid \lfloor reset.T \rfloor^1(T')$$

and

$$I_3 \text{ is } ((\overline{tea}.F + \overline{coffee}.F + \ timeout.C) \mid (\lfloor reset.T \rfloor^0(T')))\backslash A.$$

One application of the theorem gives $Y = \tau.I$ which together with another application gives

$$I_3 = \lfloor \overline{tea}.\tau.I + \overline{coffee}.\tau.I \rfloor(I_4)$$

where I_4 stands for $((\overline{tea}.F + \overline{coffee}.F + \ timeout.C) \mid T')\backslash A$. With a final application we obtain

$$I_4 = \overline{tea}.\tau.I + \overline{coffee}.\tau.I + \tau.I.$$

Recapitulating we have shown that

$$I' = \sigma^2.\lfloor \overline{tea}.\tau.I \rfloor(\lfloor \overline{tea}.\tau.I + \overline{coffee}.\tau.I \rfloor(\overline{tea}.\tau.I + \overline{coffee}.\tau.I + \tau.I))$$

and by using τ-absorption this reduces to

$$I' = \sigma^2.\lfloor \overline{tea}.I \rfloor(\lfloor \overline{tea}.I + \overline{coffee}.I \rfloor(\overline{tea}.I + \overline{coffee}.I + \tau.I).)$$

However by using the axioms $(\sigma 1 - \sigma 4)$ this can easily be rearranged to the required

$$I' = \lfloor \sigma^2.\overline{tea}.I + \sigma^3.\overline{coffee}.I \rfloor^3(\overline{tea}.I + \overline{coffee}.I + \tau.I).$$

Although the description of the system is in the language extended with time-variables they did not appear very much in this proof. Indeed the only appearance is hidden in the first application of an expansion theorem to I'; the rest of the proof is entirely within the sub-language free of time-variables. However the example was carefully chosen. If the time was set at a large number such as 100 then we would need a hundred applications of the expansion theorem and an inordinate amount of syntactic manipulation. Clearly it would be preferable to have rules which are uniformly applicable regardless of actual values associated with time-variables, i.e. are schematic in some sense. If the use of time-variables was more sophisticated, such rules would be essential. It should be possible to adapt those in [Hen91] for general value–passing processes.

Insistent Actions

Throughout these notes we have assumed that processes in our language are *patient* in that they satisfy the condition

$$\text{if } p \not\xrightarrow{\tau} \text{ then } p \xrightarrow{\sigma}$$

which means that processes will wait indefinitely until they can perform a synchronisation. This gives a particular flavour to the language and its theory. It is certainly necessary in order to obtain the simple alternative characterisation of testing in terms of barbs but it also necessitates the introduction of the Stability rule into the proof system. Here we discuss briefly the effect of relaxing this condition.

In the operational semantics patience is enforced by the rule $(W1)$ for prefixing

$$a.p \xrightarrow{\sigma} a.p$$

which is in addition to the standard rule $(Op1)$

$$\alpha.p \xrightarrow{\alpha} p.$$

This means that we have built into the language a patient form of prefixing. Let us now extend the language with *insistent prefixing* denoted by $\alpha : p$. The operational semantics for this construct is given by the one standard rule

$$\alpha : p \xrightarrow{\alpha} p.$$

Because of the absence of a rule corresponding to $(W1)$ this means that in $\alpha : p$ the action α can not be delayed until the next clock cycle. For example if a is not the complement of b then the process $a : p \mid b.q$ can not delay, i.e. can not perform a σ action; it must perform either the action a or the action b. In the process $(a : p \mid b.q)\backslash\{b\}$ the action a is forced to happen while the process $(a : p \mid b.q)\backslash\{a, b\}$ is both deadlocked, i.e. can not perform a synchronisation action, and also can not delay, i.e. can not perform σ. Note that none of these processes satisfy the condition of *patience*.

Although the use of patient prefixing often makes process descriptions simpler sometimes insistent prefixing is very useful. A typical instance of its use is when describing a one place buffer which intuitively waits indefinitely for input but when it contains a datum insists on outputing it immediately. Using both kinds of prefixing such a buffer could be described by

$$\begin{aligned} B &\Longleftarrow in.B_1 \\ B_1 &\Longleftarrow \overline{out} : B \end{aligned}$$

This buffer has two "states", B which is a patient state and B_1 an insistent state. We can connect k of these buffers together, as in [Hoa85], to obtain a buffer of size k but as is turns out most of the states of this buffer are insistent. For example let $B2$ be defined by

$$B2 \Longleftarrow B \ll B$$

where $\ll$ is the chaining operator from [Hoa85]. In general $X \ll Y$ is defined to be the process $(X[S_l] \mid Y[S_r])\backslash\{ mid\}$ where mid is a special channel, S_l is the renaming which is the identity except that out is mapped to mid while S_r maps in to mid; in effect $\ll$ connects the out port of X to the in port of Y and internalises the resulting connection. One can show that the process $B \ll B$ obtained by joining two of these kinds of buffers together is testing equivalent to the specification S defined by

$$\begin{array}{rcl} S & \Longleftarrow & in.S_1 \\ S_1 & \Longleftarrow & in : S_2 + \overline{out} : S \\ S_2 & \Longleftarrow & \overline{out} : S \end{array}$$

Note that here all of the states except the initial one are insistent. However the time behaviour of these buffers can be designed at will by introducing appropriate occurrences of σ.

The addition of insistent prefixing adds considerable descriptive power to the but it also alters considerably the resulting theory of testing. For a start standard barbs no longer characterise the testing preorder as they are not sufficiently expressive. As a simple example $a : nil$ and $a.nil$ have exactly the same barbs but they can be distinguished by the test $\sigma.\overline{a}$. Nevertheless it is possible to extend the notion of barb so as to capture testing for the extended language.

Question: Redefine *SBarb* and the partial order between them, $\ll_b$, so that $p \sqsubseteq_t q$ if and only if $SBarb(p) \ll_b SBarb(q)$ for every pair of processes p, q in the extended language. □

An equational characterisation can also be found and in some ways the resulting algebraic theory is simpler. The Stability rule is no longer necessary as patient prefixing can be expressed in terms of insistent prefixing and the time–out operator. We have the equation

$$a.t \;=\; \lfloor a : t \rfloor (a : t).$$

In some sense insistent prefixing always takes higher priority over the patient variety. For example we have the laws

$$\begin{array}{rcl} a.t \mid b : u & = & a : t \mid b : u \\ a.t + b : u & = & a : t + b : u \end{array}$$

However the development of the algebraic theory of the language with insistent prefixing is outside the scope of these lecture notes.

Question: Formulate a sound interleaving law for insistent sum terms. □

A The standard laws

The first set of equations deal with nondeterminism:

$$\begin{aligned} x + x &= x && (+1)\\ x + y &= y + x && (+2)\\ x + (y + z) &= (x + y) + z && (+3)\\ x + nil &= x && (+4) \end{aligned}$$

The next set deal with restriction and renaming:

$$\begin{aligned} nil\backslash A &= nil && (res1) & \qquad nil[S] &= nil && (ren1)\\ a.x\backslash A &= nil \text{ if } a \text{ or } \overline{a} \in A && (res2) & \qquad (\alpha.x)[S] &= S(\alpha).x[S] && (ren2)\\ \alpha.x\backslash A &= \alpha.(x\backslash A) \text{ if } \alpha \text{ and } \overline{\alpha} \notin A && (res3) & \qquad (x + y)[S] &= x[S] + y[S] && (ren3)\\ (x + y) \setminus a &= x \setminus a + y \setminus a && (res4) \end{aligned}$$

The final set essentially says that all the operators apart from prefixing by an external action are strict.

$$\begin{aligned} \tau.\Omega &= \Omega && (\Omega 1) & \qquad \Omega \setminus a &= \Omega && (\Omega 4)\\ x + \Omega &= \Omega && (\Omega 2) & \qquad \Omega[S] &= \Omega && (\Omega 5)\\ x \mid \Omega &= \Omega && (\Omega 3) \end{aligned}$$

When the language is extended with the time-out construct $\lfloor\ \rfloor(\ )$ we also need the obvious laws:

$$\begin{aligned} \lfloor\Omega\rfloor(x) &= \Omega && (\Omega 5)\\ (\lfloor x\rfloor(y))\backslash A &= \lfloor x\backslash A\rfloor(y\backslash A) && (res5)\\ (\lfloor x\rfloor(y))[S] &= \lfloor x[S]\rfloor(y[S]) && (ren4) \end{aligned}$$

References

[BB92] J. C. M. Baeten and J. A. Bergstra. Discrete time process algebra. Technical Report P9208, University of Amsterdam, 1992.

[BG88] G. Berry and G. Gonthier. The synchronous programming language ESTEREL: design, semantics, implementation. Report 842, INRIA, Centre Sophia-Antipolis, Valbonne Cedex, 1988. To appear in *Science of Computer Programming*.

[DH84] R. DeNicola and M. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 24:83–113, 1984.

[Gro90] J.F. Groote. Specification and verification of real time systems in ACP. Report CS-R9015, CWI, Amsterdam, 1990. An extended abstract appeared in L. Logrippo, R.L. Probert and H. Ural, editors, *Proceedings* 10^{th} *International Symposium on Protocol Specification, Testing and Verification*, Ottawa, pages 261–274, 1990.

[Gue81] I. Guessarian. *Algebraic Semantics*. Lecture Notes in Computer Science vol 99, 1981.

[Han91] Hans A. Hansson. *Time and Probability in Formal Design of Distributed Systems*. Ph.d. thesis, Uppsala University, 1991.

[Hen88] M. Hennessy. *An Algebraic Theory of Processes*. MIT Press, 1988.

[Hen91] M. Hennessy. A proof system for communicating processes with value–passing. *Formal Aspects of Computer Science*, 3:346–366, 1991.

[HI91] M. Hennessy and A. Ingolfsdottir. A theory of communicating processes with value-passing. *Information and Computation*, to appear, 1991.

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

[HR91] M. Hennessy and T. Regan. A process algebra for timed systems. Technical Report 5/91, CSAI, University of Sussex, 1991.

[Lan89] R. Langerak. A testing theory for LOTOS using deadlock detection. In E. Brinksma, G. Scollo, and C.A. Vissers, editors, *Proceedings of the Ninth International Conference on Protocol Specification, Testing and Verification*, 1989.

[Lin91] H. Lin. Process Algebra Manipulator: User Guide. In K. Larsen and A. Skou, editors, *Proceedings of CAV91*, Lecture Notes in Computer Science, 1991. also available as Susex Computer Science Techniucal Report 9/91.

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

[MT90] F. Moller and C. Tofts. A temporal calculus of communicating systems. In *Proc. Concur 90*, pages 401–415. Springer-Verlag, 1990. LNCS 458.

[MV89] S. Mauw and G.J. Veltink. An introduction to PSF_d. In J. Díaz and F. Orejas, editors, *TAPSOFT89, vol 2*, volume 352 of *Lecture Notes in Computer Science*, pages 272–285, 1989.

[NS90] X. Nicollin and J. Sifakis. The algebra of timed processes ATP: Theory and application. Technical Report RT-C26, Laboratoire de Génie Informatique de Grenoble, 1990. to appear in Information and Computation.

[Plo81] G.D. Plotkin. A structural approach to operational semantics. Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[Reg92] T. Regan. *A Process Algebra for Real-Time Systems.* Ph.d. thesis, University of Sussex, 1992.

[Wan91] Wang Yi. *A Calculus of Real Time Systems.* Ph.D. thesis, Chalmers University, 1991.