R12516 Copy 1

AN INTRODUCTION TO TIMED CSP

by

Jim Davies
Steve Schneider

Technical Monograph PRG-75
ISBN 0-902928-57-0

August 1989

Oxford University Computing Laboratory
Programming Research Group
11 Keble Road
Oxford OX1 3QD
England

Oxford University Computing Laboratory
Programming Research Group
11 Keble Road
Oxford OX1 3QD
England

Electronic mail: jdavies@uk.ac.ox.prg (JANet)
sas@uk.ac.ox.prg (JANet)

Prologue

Timed CSP is an extension of Communicating Sequential Processes which includes timing information. It can be used to model time-dependent properties of concurrent systems. An algebraic notation is employed in the definition of processes, capturing the behaviour of a system in a clear and intuitive manner. A uniform hierarchy of semantic models for this notation is presented in [Re88]. Each semantic model identifies a process with a set of possible behaviours: by reasoning about these sets, we may establish properties of the corresponding processes.

This monograph contains two papers on Timed CSP. The first of these introduces the language of Timed CSP, aimed at those familiar with Hoare's book on CSP, [H85]. The second presents a complete proof system for reasoning about the most useful class of Timed CSP specifications: behavioural specifications on timed failures. Together, these two papers provide a foundation for the specification and design of real-time concurrent systems using Timed CSP.

Contents

An Introduction To Timed CSP

Jim Davies and Steve Schneider

Oxford University Computing Laboratory
Programming Research Group
11 Keble Road
Oxford OX1 3QD

Abstract. This paper is an introduction to the language of Timed CSP. The syntax is presented and explained through several examples of timed interaction. The subsequent chapters show how the syntax may be used to represent aspects of time-critical behaviour. The use of the semantic models in specification is demonstrated. The paper ends with a brief discussion of future research directions.

1 Introduction

This paper is intended as an introduction to Timed CSP for those already familiar with Tony Hoare's book *Communicating Sequential Processes*, [H85]. Because of this, we may assume that our readers are aware of the advantages of using such a notation to reason about the properties of concurrent systems. There is a need for a similar treatment of real-time communicating processes, where correctness may depend upon subtle timing considerations; applications include communication protocols and operating systems.

A number of timed models have been postulated for Hoare's Communicating Sequential Processes, notably in [J82], [Z86], and [BG87]. However, the hierarchy of compatible models for Timed CSP, presented by Reed and Roscoe in [RR86], [RR87], and [Re88] has the following advantages:

- the models are compatible with the existing untimed models of CSP.
- infinite hiding and infinite alphabet transformations are possible.
- deadlock and divergence may be distinguished.
- divergence may be distinguished from the possibility of divergence.
- the models are arranged in a hierarchy, permitting timewise refinement of specifications.

In this paper, we introduce the models of the hierarchy, and show how they can be used in the specification and design of concurrent systems. We begin by introducing the syntax of Timed CSP, together with some examples of its application. We show how to model aspects of timed interaction: timeouts and interrupts. Timed CSP descriptions of two protocols are given as examples. We present the semantic models, and illustrate their use in the specification and verification of Timed CSP processes. Finally, we discuss the enhancements that are being made to the theory of Timed CSP.

2 Timed CSP: Syntax

2.1 Assumptions

We make a small number of assumptions about timing in a distributed system:

- there is a non-zero lower bound, δ, on the length of the time interval between any two events in the history of a sequential process.
- there is no lower bound on the time interval between two independent actions, such as those performed by two processes running asynchronously in parallel.
- the times at which events occur in the system relate to a conceptual global clock: time passes at the same rate in each process.
- hidden events occur as soon as they become available.

For further details, and a more mathematical treatment, see [Re88].

2.2 Processes

The process algebra *TCSP* is essentially that of untimed CSP, with the addition of a delay operator *WAIT*:

$$\begin{aligned} P \;::=\;& \perp \mid STOP \mid SKIP \mid WAIT\ t \mid a \rightarrow P \mid \\ & P \mathbin{\Box} P \mid P \sqcap P \mid P \parallel P \mid P \mathbin{{}_A\|_B} P \mid P \mathbin{|||} P \mid \\ & P\,;P \mid P \setminus A \mid f^{-1}(P) \mid f(P) \mid \mu X \bullet F(X) \end{aligned}$$

Although the semantic treatment of the operators is quite different, the intuition behind their use remains the same. In many cases, the new semantics reflect this intuition more closely: the introduction of time information allows us to draw new distinctions. The notion of process alphabet introduced in [H85] has been discarded; synchronisation of events is achieved by means of an alphabeticised parallel operator.

Basic Operators

$\perp$	The divergent process $\perp$ can perform no observable actions. However, internal activity may continue indefinitely; it is live-locked, like an infinite loop.
STOP	The deadlock process *STOP* cannot perform any external actions; neither can it make any internal progress.
SKIP	This construct models the successful termination of a process, signalled by the occurrence of the special event ✓; this is the only event that this process may engage in.
WAIT t	The delay operator, *WAIT* t, models the delayed successful termination of a process, introducing a delay of time t before the special event ✓ becomes available. The processes *WAIT* 0 and *SKIP* are equivalent.

Prefixing

$a \rightarrow P$	The process $a \rightarrow P$ represents a process which is initially prepared to engage in event a. A short delay (δ) follows the occurrence of event a, and the process then behaves as P. The delay introduced by the prefix operator models the minimum time required for a sequential process to recover from participation in an event. Longer recovery times can be modelled by explicit use of the *WAIT* operator.

Choice

$P \sqcap Q$ The notion of nondeterministic choice remains unchanged with the introduction of time. The process $P \sqcap Q$ represents an internal choice between P and Q: the environment cannot interfere. The indexed version of the operator may be used to model infinite nondeterministic choice.

$P \Box Q$ The notion of deterministic choice is also unchanged from the untimed models. The process $P \Box Q$ represents an external choice between the two processes. Control is passed to the first process to perform an external action; the choice is made with the co-operation of the environment.

Parallelism

$P\ {}_X\|_Y\ Q$ The alphabeticised parallel operator provides for synchronisation in Timed CSP. In the parallel combination $P\ {}_X\|_Y\ Q$, process P may perform only those events drawn from the set X. Similarly, process Q is restricted to events from the set Y. The two processes must synchronise on events from the intersection $X \cap Y$.

$P \parallel Q$ The simple parallel operator is a special case of the alphabeticised operator: when the set arguments are omitted, they are taken to be the universal set of events Σ. In the parallel combination $P \parallel Q$, the two processes must synchronise on all events.

$P \mid\mid\mid Q$ The interleaved parallel combination of two TCSP processes performs as the two components acting independently: there is no synchronisation between them.

Sequential Composition

$P\ ;\ Q$ The sequential composition operator is used to transfer control from one process to another. In the process $P;Q$, the transfer is effected once P signals successful termination, modelled by the occurrence of the special event $\checkmark$. The sequential composition operator hides this event from the environment, with the result that the event occurs as soon as possible.

Abstraction

$P \setminus A$ — The hiding operator allows ns to abstract from internal detail, hiding certain events from the environment of a process. This has the effect of forcing these events to occur as soon as they become available: the events are no longer observable, but the delays are retained. In ordinary CSP, the set of events to be hidden must be finite; in Timed CSP, this restriction no longer applies. The new models support infinite hiding.

Renaming

$f(P)$, $f^{-1}(P)$ — The notion of process relabelling is unchanged in the timed version of CSP. The process $f(P)$ is obtained by applying the alphabet transformation f to the events in the description of process P. The second form of process relabelling, $f^{-1}(P)$, allows ns to consider all of the possible processes which behave in a fashion similar to P.

Recursion

$\mu X \bullet F(X)$ — Recursion in Timed CSP introduces a delay of time δ, similar to that of the prefix operator, as illustrated by this equivalence:

$$\mu X \bullet F(X) \equiv F(\mathit{WAIT}\,\delta \,;\, \mu X \bullet F(X))$$

2.3 Alphabets

In Hoare's treatment of CSP, [H85], each process P is associated with a unique set of events αP, the process alphabet. If P appears in a parallel combination of processes, events from αP require the co-operation of P. In the approach to CSP used by Roscoe and Reed in [Ro82] and [Re88] the need for process alphabets is removed by the introduction of an alphabeticised parallel operator.

This operator is parametrised by two sets of events. In the parallel combination

$$P\ {}_A\|_B\ Q$$

process P may perform only those events in set A, process Q may perform only those events in set B, and the two processes mnst co-operate on events drawn from the intersection of A and B.

Using Hoare's approach, we may restrict the behaviour of a process P by placing it in parallel with the process $STOP$, parametrised by a suitable alphabet. For example, the process $STOP_{\{a\}} \parallel P$ behaves as P, with all occurrences of event a blocked. Without alphabets, we use the set of all events, Σ, to construct an equivalent process using the alphabeticised parallel operator:

$$STOP\ {}_{\{a\}}\|_{\Sigma}\ P$$

It is still necessary to identify the set of possible actions of a process — we use $\sigma(P)$ to denote the set of all events in which process P may participate.

2.4 Algebra

A number of properties of untimed CSP processes are no longer true in Timed CSP. It is perhaps instructive to examine these, and the reasons for their disappearance:

- $P \parallel STOP \not\equiv STOP \qquad \text{if } P \neq \bot$

P may still make internal progress: the right-hand side is a stable process, but the left-hand side need not be. Internal activity is not prevented by the need to synchronise with the stable process $STOP$.

- $(a \rightarrow P) \setminus a \not\equiv P \setminus a$

Even though the a event occurs instantaneously, it is followed by a delay of δ, which is still recorded. Thus we now have the equivalence $(a \rightarrow P) \setminus a \equiv WAIT\ \delta ; (P \setminus a)$

- $\mu X \bullet F(X) \not\equiv F(\mu X \bullet F(X))$

Any recursion takes time δ to unfold; this delay is present before each recursive call. The correct equivalence is $\mu X \bullet F(X) \equiv F(WAIT\ \delta ; \mu X \bullet F(X))$.

- $(a \rightarrow P) \mid\mid\mid (b \rightarrow Q) \not\equiv \begin{array}{l} a \rightarrow (P \mid\mid\mid (b \rightarrow Q)) \\ \Box \\ b \rightarrow ((a \rightarrow P) \mid\mid\mid Q) \end{array}$

We have *true* parallelism, not *time-slice parallelism*, so the left-hand process may engage in the two events a and b within an interval of time which is arbitrarily small. The right-hand side, however, describes a sequential process: after performing the first event, time δ must elapse before it can perform another.

- $P \sqcap (Q \,\square\, R) \not\equiv (P \sqcap Q) \,\square\, (P \sqcap R)$

This law fails because we now have more information about the history of a process. We can do *refusal testing*: we have a record of the events refused by a process throughout its history. If event a is offered only by process Q, the knowledge that event a is refused by the left-hand side resolves the nondeterministic choice. On the right-hand side, only one of the choices is resolved: R is still a possibility.

Many of the identities established in [H85] are retained...

$$\begin{aligned}
P \,\square\, STOP &\equiv P \\
P ; (Q ; R) &\equiv (P ; Q) ; R \\
STOP \,|||\, Q &\equiv Q \\
SKIP ; Q &\equiv Q \\
(a \to P) ; Q &\equiv a \to (P ; Q) \qquad \text{if } a \neq \checkmark
\end{aligned}$$

...and the introduction of time brings more:

$$\begin{aligned}
WAIT\, t_1 ; WAIT\, t_2 &\equiv WAIT\,(t_1 + t_2) \\
WAIT\, t_1 \parallel WAIT\, t_2 &\equiv WAIT\, max\{t_1, t_2\} \\
(WAIT\, t_1 \,|||\, WAIT\, t_2) ; P &\equiv WAIT\, min\{t_1, t_2\} ; P
\end{aligned}$$

2.5 Example

Consider the process VM defined below:

$$VM \;\widehat{=}\; coin \to rattle \to WAIT(d - \delta) ; drink \to VM$$

This is intended to represent a simple vending machine. Following the insertion of a coin, a rattling sound may be heard as the coin drops. A drink is then offered by the machine; this offer is made no earlier than time d after the rattle. If the drink is accepted, the machine returns to its initial state.

We are considering timed behaviour, and there is a minimum delay of time δ between participation in events. To ensure that the drink becomes available after the specified interval, the delay is represented by a $WAIT$ of time $d - \delta$.

However, the observation of the rattling sound should not be necessary for the correct behaviour of the machine, so we abstract from the event, using the hiding operator to conceal it from the environment.

$$VM \setminus rattle \;\equiv\; coin \to WAIT\, d ; drink \to (VM \setminus rattle)$$

Notice that the hidden event *rattle* occurs as soon as possible.

3 Timed Interaction

3.1 A Simple Timeout

One important aspect of time-critical behaviour is the *timeout*: a change of state in which control is passed from one process to another, if the first performs no external actions in a given period of time. We can represent this behaviour in Timed CSP using the deterministic choice operator, the sequential composition operator, and a suitable delay. Consider the *TCSP* process given by

$$(P \,\square\, \mathit{WAIT}\ t)\,;\,Q$$

where P and Q are also *TCSP* processes. If P performs no external actions by time t, then the special event $\checkmark$ is made available by the delay construct. The presence of the sequential composition operator forces this event to occur as soon as it becomes available, removing the external choice and passing control to process Q: events from P are no longer available. If P performs an external action before time t, then the choice is resolved in favour of P, and the *WAIT t* process is no longer present.

3.2 Example

As an illustration of the use of Timed CSP, we will consider a simple representation of a sensitive vending machine and its user. The sensitive nature of the machine lies in its response to a kick: if it is kicked while the coin is still dropping, it may refuse to dispense a drink. In untimed CSP, the machine may be described as follows:

$$SVM \;\widehat{=}\; coin \rightarrow \begin{pmatrix} drink \rightarrow SVM \\ \square \\ kick \rightarrow (STOP \sqcap drink \rightarrow SVM) \end{pmatrix}$$

As we are unable to observe the coin's progress inside the machine, we cannot determine the effect of a *kick* event between the insertion of a coin and the attempted removal of a drink. If the machine is confronted with a user such as *USE*, where

$$USE \;\widehat{=}\; coin \rightarrow kick \rightarrow drink \rightarrow USE$$

then this sequence of events may lead to deadlock:

$$USE \parallel SVM \;\equiv\; coin \rightarrow kick \rightarrow \begin{pmatrix} STOP \\ \sqcap \\ drink \rightarrow (USE \parallel SVM) \end{pmatrix}$$

Timed CSP allows us to include more information in our description. In the case of the vending machine, the time at which it is kicked affects the outcome of the *kick* event; if we allow time for the coin to drop, a kick will do no harm. Consider the description of a time-sensitive vending machine given by:

$$TSVM \;\widehat{=}\; coin \rightarrow \begin{pmatrix} kick \rightarrow STOP \\ \Box \\ WAIT\,1 \end{pmatrix} ; \begin{pmatrix} drink \rightarrow TSVM \\ \Box \\ kick \rightarrow drink \rightarrow TSVM \end{pmatrix}$$

Providing that the user does not kick the machine within a second of inserting the coin, a drink will become available. If we have a user whose patience extends to three seconds,

$$TUSE \;\widehat{=}\; coin \rightarrow WAIT\,3 ; kick \rightarrow drink \rightarrow TUSE$$

then we can guarantee a satisfactory outcome.

$$TUSE \parallel TSVM \;\widehat{=}\; coin \rightarrow WAIT\,3 ; kick \rightarrow drink \rightarrow (TUSE \parallel TSVM)$$

Including timing information in our description has the effect of resolving the nondeterminism. The outcome of a *kick* is dependent upon when the *kick* occurs; any untimed description must include an element of nondeterminism. The timed process *TSVM* is a *timewise refinement* of the untimed process *SVM*.

3.3 A Timeout Operator

The timeout construct presented above is adequate for most purposes, but it has one undesirable property. In the *TCSP* process

$$(P \Box WAIT\;t) ; Q$$

control is passed to process Q if P performs no external events before time t. However, the same thing will happen if P terminates successfully; the sequential composition operator prevents P from signalling successful termination to the environment.

We now define a timeout operator for Timed CSP, using the syntactic equivalence:

$$P \overset{t}{\triangleright} Q \;\widehat{=}\; (P \Box (WAIT\;t ; b \rightarrow Q)) \setminus b$$

where $b \notin \sigma(P) \cup \sigma(Q)$. If P performs no external events before time t, the event b removes control; as a hidden event, it occurs as soon as possible (at time t)

and resolves the choice against P. The process then behaves as $Q \setminus b$, which is equivalent to Q by our choice of b.

If P engages in an external action before the timeont occurs, then the process continues to behave as $P \setminus b$, which is equivalent to P. If this process should terminate successfully, then the special event ✓ will be observed by the environment.

This operator will be a useful *TCSP* process constructor. As a guide to its application, consider the following identities:

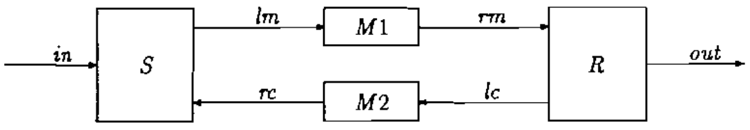
$$\begin{aligned}
(P \overset{t_1}{\triangleright} Q) \overset{t_1+t_2}{\triangleright} R &\equiv P \overset{t_1}{\triangleright} (Q \overset{t_2}{\triangleright} R) \\
(P \sqcap Q) \overset{t}{\triangleright} R &\equiv (P \overset{t}{\triangleright} R) \sqcap (Q \overset{t}{\triangleright} R) \\
(P \Box Q) \overset{t}{\triangleright} R &\equiv (P \overset{t}{\triangleright} R) \Box (Q \overset{t}{\triangleright} R) \\
((a \to P) \overset{t}{\triangleright} Q) \setminus X &\equiv \mathit{WAIT}\,\delta\,; (P \setminus X) \qquad (a \in X, t > 0)
\end{aligned}$$

4 Protocols

One area of study involving the analysis of timed concurrent behaviour is the design and verification of communication protocols: distributed algorithms for facilitating the transfer of information. We can use the notation of Timed CSP to produce clear, concise descriptions of these protocols.

4.1 The Alternating Bit Protocol

We present a timewise refinement of the alternating bit protocol presented in [PS88]. This protocol consists of a sending process S and a receiving process R, and operates over a medium represented by wires $M1$ and $M2$:



The sending process operates in the following fashion: a messagc is inpnt along channel *in*, tagged with a bit value, and output along channel *lm*. Consecutive messages are tagged with alternating bits. The process awaits a confirmation bit on channel *rc*; this should match the bit value of the last transmission. If no such bit arrives within a specified time, the process times out and retransmits the message with the same bit value. This behaviour is captured by the following *TCSP* process:

$$\begin{aligned} S &\triangleq S_0 \\ S_b &\triangleq in?x \to lm!x.b \to S_{x.b} \\ S_{x.b} &\triangleq (rc?a \to (\textbf{if } a = b \textbf{ then } S_{1-b} \text{ else } S_{x.b})) \overset{t}{\rhd} lm!x.b \to S_{x.b} \end{aligned}$$

The receiving process complements this behaviour: a message received along the channel *rm* is stripped of its bit value, which is transmitted on channel *lc* as an acknowledgement bit. If the bit value matches the bit value of the previous message, the message is discarded. Otherwise, the message is output on channel *out*. A *TCSP* representation might be:

$$\begin{aligned} R &\triangleq R_1 \\ R_b &\triangleq rm?x.c \to (\textbf{if } c = b \textbf{ then } lc!c \to R_c \text{ else } out!x \to lc!c \to R_c) \end{aligned}$$

We then have a formal description of the protocol in Timed CSP. This permits a rigorous analysis of its behaviour and a clear description of the interface with the environment. If we obtain a similar description of the communication medium, then we may use Timed CSP to verify that the protocol will function correctly. For example, we may describe the wires $M1$ and $M2$ using *TCSP*.

A suitable representation for $M1$ and $M2$ is given by the *TCSP* process below:

$$\begin{aligned} RW_n &\triangleq \prod_{k=0}^{n} RW_{k,n} \\ RW_{0,n} &\triangleq in?x \to out!x \to RW_n \\ RW_{k+1,n} &\triangleq in?x \to RW_{k,n} \end{aligned}$$

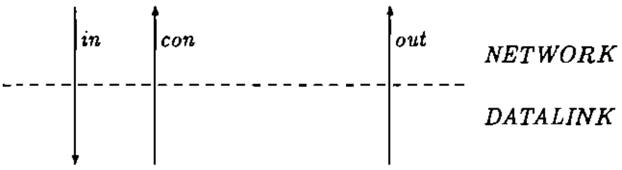
This process successfully transmits at least $\frac{1}{n}$ of the inputs with which it is presented; it can never discard n consecutive inputs. We can use this to model a communication medium in which the probability of losing n consecutive messages is negligible. If we relabel the channels, we can obtain *TCSP* representations of wires $M1$ and $M2$.

4.2 Local Area Network Protocol

Local Area Networks can be used to connect systems that need to communicate over fairly short distances: up to a few kilometres apart. In most cases, data is *broadcast* across a transmission medium, perceived by all stations or nodes. One protocol designed for use in this situation is the *Ethernet* protocol introduced in [MB76].

This protocol operates by accepting a packet of data and attempting to transmit it across the broadcast medium. The medium is monitored throughout the transmission, which is halted in the case of interference. This procedure is called *collision detection*. If a collision occurs, the transmitter waits for a random amount of time before trying again. If the same message is interrupted too many times, then an error is reported.

In this section, we are not concerned with the mechanism of transmission. We wish to describe the service provided by an ethernet-like protocol to each node of a network. In the ISO seven-layer model, see [T81], this corresponds to the interface between the datalink layer and the network layer. The datalink layer is the parallel combination of the datalink components of all of the network nodes; at each of these the interface consists of three channels: *in*, *con* and *out*.



At node i, messages are transmitted to the datalink along channel $i.in$. The datalink then processes the message, and attempts to transmit the contents to the other nodes on the network. In our representation, the outcome of such an attempt is nondeterministic. If the message m is successfully transmitted then the network layer is notified using channel $i.con$. If not, then a failure is reported using the same channel.

The factors influencing the success or failure of an attempt cannot be determined from the network layer, but knowledge of the datalink structure allows us to include timing information in our description. For example, if the time taken for the datalink to pass a message m to the transmission medium is t_m, a successful transmission cannot be reported within time t_m of an input.

In ethernet-like protocols, if the transmission is interrupted, the datalink backs off for a random period, then begins again without informing the network layer, abandoning the current message only after 15 consecutive attempts have failed; only then is the failure reported. If the backoff time is restricted to the interval $[t_{min}, t_{max}]$, then the failure report for a message m may become available at any time between $15t_{min}$ and $15(t_m + t_{max})$. Because of this, if the transmission of a message m is to succeed, it must succeed by time $15(t_m + t_{max})$.

We can capture this timing information in Timed CSP. If we choose N to represent the set of node identifiers, and let $t_{i,j}$ be the time taken for a signal to travel from node i to node j, we may represent the service provided by the datalink layer as follows:

$$\begin{aligned}
DATALINK &\mathrel{\widehat{=}} \mathop{|||}_{i \in N} NODE_i \\
NODE_i &\mathrel{\widehat{=}} i.in?m \rightarrow (SUCCESS_{i,m} \sqcap FAILURE_i) \\
SUCCESS_{i,m} &\mathrel{\widehat{=}} (WAIT\ I_S)\ ;\ \mathop{|||}_{j \in N} WAIT\ t_{i,j}\ ;\ j.out!m \rightarrow STOP \\
&\qquad\qquad\qquad |||< \\
&\qquad\qquad\quad (\ i.con!success \rightarrow NODE_i \\
FAILURE_i &\mathrel{\widehat{=}} (WAIT\ I_F)\ ;\ i.con!failure \rightarrow NODE_i
\end{aligned}$$

where

$$\begin{aligned}
WAIT\ I &\mathrel{\widehat{=}} \mathop{\sqcap}_{t \in I} WAIT\ t \\
I_S &\mathrel{\widehat{=}} [t_m, 15(t_m + t_{max})] \\
I_F &\mathrel{\widehat{=}} [15t_{min}, 15(t_m + t_{max})]
\end{aligned}$$

Note that the arrival of a report on the channel *con* is preceded by a nondeterministic delay. This is modelled by a nondeterministic choice over an interval of time.

5 Simple Interrupts

An *interrupt* is a signal that interrupts the execution of a process. Subsequent behaviour may be determined by an interrupt handler: a process that identifies the nature of the interrupt signal and acts accordingly. In some cases, an interrupted process may resume execution at the point of interruption; the process state is stored for the duration of the interrupt. In others, the existing process is terminated, to be replaced by another or simply restarted.

It is possible to model the first class of interrupts in Timed CSP; we simply suspend all internal activity present in the interrupted process. This requires a new semantic definition, outside the scope of an introduction to the notation of Timed CSP.

In this section we will address the second class of interrupts, the *simple* interrupts. These can be used to model any situation in which the *internal* state of the interrupted process P need not be preserved; any external actions of P before the interrupt can be recorded and acted upon during and after the interrupt has occurred.

5.1 A First Approach

Consider the *TCSP* process

$$(P \parallel\!| \ a \rightarrow SKIP) \ ; Q$$

where $a \notin \sigma(P)$. This behaves as process P until the first occurrence of event a. This signals the successful termination of the first construct, passing control to process Q. Event a acts as an interrupt event to process P. We can use this construction to model simple interrupts.

The fact that the behaviour following an interrupt is the same as that following successful termination of P need not concern us. In many cases, process P will never terminate successfully. If we require that P should be able to signal successful termination, or that there should be a choice of interrupt events, we can use the *TCSP* interrupt operator presented later in this section.

Further, we can dispense with the interrupt event altogether, automatically removing control from process P at a particular time. In the process

$$(P \parallel\!| \ WAIT \ t) \ ; Q$$

the first construct will terminate at time t, transferring control to process Q.

5.2 Example

We consider a simple pinball machine. To play, the user must insert a coin, and press the start button. At the end of the game, the machine returns to its initial state. To add interest, the user may cause an interrupt during play by tilting the machine; this results in the immediate termination of play:

$$PINBALL \ \widehat{=} \ coin \rightarrow start \rightarrow (PLAY \parallel\!| \ tilt \rightarrow SKIP) \ ; PINBALL$$

The user's interaction with the machine is not limited to the above three events. During the game, two flippers are provided; the action of these is described by the processes below:

$$\begin{aligned} LEFTFLIP \ &\widehat{=} \ left \rightarrow WAIT \ (d - 2\delta) \ ; LEFTFLIP \\ RIGHTFLIP \ &\widehat{=} \ right \rightarrow WAIT \ (d - 2\delta) \ ; RIGHTFLIP \end{aligned}$$

Observe that occurrences of the event *left*, representing the triggering of the left flipper are restricted; there is a minimum delay d between any two occurrences. A similar restriction is placed upon the event *right*. However, each flipper should be independent of the other; there are no bounds on the interval between consecutive *left* and *right* events.

We make no other observations during the game, which may end at any time after the short interval required for the ball to enter play. This is modelled using an infinite nondeterministic choice:

$$PLAY \;\widehat{=}\; (LEFTFLIP \mid\mid\mid RIGHTFLIP) \mid\mid\mid \bigsqcap_{t \in [3,\infty)} WAIT\ t$$

Finally, we may cause an interrupt at a higher level. For example, the user may inadvertently cut off the power to the machine; if this happens, no further interaction is possible. This may be represented with another simple interrupt construct:

$$MACHINE \;\widehat{=}\; (PINBALL \mid\mid\mid unplug \rightarrow SKIP)\ ;\ STOP$$

5.3 A Simple Interrupt Operator

If the same *TCSP* process may be interrupted by more than one event, or if we wish to allow successful termination, the simple interrupt construct is not suitable. Instead, we may use the *TCSP* operator defined by the syntactic equivalence below. In this expression, I denotes the set of possible interrupt events, and b_I denotes the set $\{b_i \mid i \in I\}$, the set of hidden synchronisation events.

$$P \underset{i \in I}{\nabla} Q(i) \;\widehat{=}\; \left(\begin{array}{l} \left(\begin{array}{l} P\ ;\ a \rightarrow SKIP \\ \mid\mid\mid \\ i : I \rightarrow SKIP \end{array} \right) ; \left(\begin{array}{l} a \rightarrow SKIP \\ \Box \\ b_i : b_I \rightarrow Q(i) \end{array} \right) \\ \\ {}_{\Sigma}\|_{I \cup b_I \cup \{a\}} \\ \\ \left(\begin{array}{l} a \rightarrow a \rightarrow SKIP \\ \Box \\ i : I \rightarrow b_i \rightarrow SKIP \end{array} \right) \end{array} \right) \setminus b_I \cup \{a\}$$

where the following alphabet conditions hold:

$$I \cap \sigma(P) = \emptyset$$
$$(b_I \cup \{a\}) \cap (\sigma(P) \cup \textstyle\bigcup_{i \in I} \sigma(Q(i))) = \emptyset$$

The synchronisation events should be chosen to make them independent of the processes, and P should not be able to interfere with its own interrupt signals. To see that this is a satisfactory definition of the interrupt operator, consider the possible behaviours of the process:

- if process P has not terminated successfully, and no interrupts have occurred, then any of the interrupt events i are available to the environment. The synchronisation event a is hidden from the environment, but may not occur; although it is available in the lower half of the parallel combination, it remains blocked (by P) in the upper half.
- if an interrupt i occurs, then the corresponding synchronisation event b_i is enabled. This event is hidden from the environment, and will occur as soon as it becomes available in both halves of the parallel combination. At the same time, P has been interrupted, and control in the upper half has been passed to a deterministic choice between a and a set of synchronisation events. However, i has occurred, so the lower half is willing to participate only in the event b_i. As all of these events have been hidden, b_i occurs immediately, and control passes to $Q(i)$, the correct interrupt handler.
- if P terminates successfully, the hidden event a is enabled, and occurs at once. In the upper half, control again passes to the deterministic choice, but this time only the event a is being offered by the lower half, which must have synchronised upon the first a. Thus a second hidden a occurs, and the entire construct terminates successfully.

5.4 Example

To illustrate the use of this operator, we return to our pinball machine. The latest model $PINBALL_2$, has a unpleasant feature: if a coin is inserted while a game is being played, the machine abandons the current game and is prepared to start a new one. We now have two interrupt events during play: *tilt* and *coin*. The behaviour following an interrupt depends upon the nature of the interrupt event:

$$\begin{aligned} PINBALL_2 &\triangleq coin \to GAME \\ GAME &\triangleq start \to (PLAY \mathop{\nabla}_{i \in I} HANDLE(i))\ ;\ PINBALL_2 \end{aligned}$$

where the set I contains only the events *tilt* and *coin*, and the interrupt handler is defined as follows:

$$\begin{aligned} HANDLE(tilt) &\triangleq SKIP \\ HANDLE(coin) &\triangleq GAME \end{aligned}$$

5.5 A Timed Interrupt Operator

We can define a timed interrupt operator for Timed CSP. This has the effect of transferring control from one process to another, after a predetermined period of time. Unlike the construct presented in section 5.1, this operator permits the first process to signal successful termination.

$$P \underset{t}{\lightning} Q \;\; \widehat{=} \;\; \left(\begin{array}{l} \left(\begin{array}{l} P \, ; \, a \to SKIP \\ ||| \\ WAIT \; t \, ; \, b \to SKIP \end{array} \right) ; \left(\begin{array}{l} a \to SKIP \\ ||| \\ b \to Q \end{array} \right) \\ \\ {}_{\Sigma}\|_{\{a,b\}} \\ \\ \left(\begin{array}{l} a \to a \to SKIP \\ \Box \\ b \to b \to SKIP \end{array} \right) \end{array} \right) \setminus \{a, b\}$$

where $\{a, b\} \cap (\sigma(P) \cup \sigma(Q)) = \emptyset$. Note that a delay of 2δ is introduced when control is passed by this operator: the second process starts execution at time $t + 2\delta$. This is reflected in the following identity:

$$(P \underset{t_1}{\lightning} Q) \underset{t_1+t_2+2\delta}{\lightning} R \;\; \equiv \;\; P \underset{t_1}{\lightning} (Q \underset{t_2}{\lightning} R)$$

We can extend the definition of the interrupt operator to allow interrupts over an interval, occurring nondeterministically:

$$P \underset{T}{\lightning} Q \;\; \widehat{=} \;\; \left(\begin{array}{l} \left(\begin{array}{l} P \, ; \, a \to SKIP \\ ||| \\ WAIT \; T \, ; \, b \to SKIP \end{array} \right) ; \left(\begin{array}{l} a \to SKIP \\ ||| \\ b \to Q \end{array} \right) \\ \\ {}_{\Sigma}\|_{\{a,b\}} \\ \\ \left(\begin{array}{l} a \to a \to SKIP \\ \Box \\ b \to b \to SKIP \end{array} \right) \end{array} \right) \setminus \{a, b\}$$

where $\{a, b\} \cap (\sigma(P) \cup \sigma(Q)) = \emptyset$ and T represents a finite interval of time. The construct $WAIT \; T$ is as defined at the end of section 4.2:

$$WAIT \; T \;\; \widehat{=} \;\; \prod_{t \in T} WAIT \; t$$

6 Timed CSP: Semantics

Timed CSP has been given a variety of semantic models. These can be used to produce formal descriptions of process behaviour. In each model, a timed CSP process is identified with a set of possible behaviours. A typical element of a semantic set is a tuple, whose elements represent different aspects of a possible behaviour.

The untimed models consider the traces, refusals and stabilities of a process; these are closely related to the traces, refusals and divergences of untimed CSP. The timed models address the timed equivalents of these components: timed traces, timed refusals, and timed stabilities. As the untimed models are not the subject of this paper, we will omit the prefix *timed* where no ambiguity will arise.

6.1 Traces

A timed trace of a Timed CSP process is a finite sequence of observable events in the history of that process, each labelled with the time at which it occurs. The events are presented in chronological order. In the simpler models of timed CSP — those without timed refusal information — we must identify the times at which events first become available, in order to reach a satisfactory definition of hiding. We do this by placing a hat upon an event whenever it occurs at the first moment of availability.

As an example, consider the trace set of the process

$$P \;\hat{=}\; WAIT\,1\,;(a \rightarrow b \rightarrow STOP)$$

$\langle(0.5, a), (1, b)\rangle$	is not a trace of P: the first event cannot take place before time 1.
$\langle(1, a), (2, b)\rangle$	is a trace of P
$\langle(1, \hat{a}), (2, b)\rangle$	is a trace of P
$\langle(1.5, \hat{a}), (2, b)\rangle$	is not a trace of P: the hat on a is incorrect, as the event first becomes available at time 1.
$\langle(2, a), (2 + \delta, \hat{b})\rangle$	is a trace of P.
$\langle(2, b)\rangle$	is not a trace of P: the first event must be an a.
$\langle(1.5, b), (1, a)\rangle$	is not a trace of P: the sequence of times in any trace must be non-decreasing.

6.2 Refusals

Timed refusals represent the times at which events may be refused during the observation of a given trace; they are not simple extensions of untimed refusal sets. A timed refusal is finite union of refusal tokens; a refusal token is the product of a finite half-open interval of time with a set of events. A typical element of a refusal set is thus a (*time,event*) pair. The restrictions on the composition of a refusal set allow us to consider only those observations made in a *finite* period of time. As in the case of untimed CSP, (*trace,refusal*) pairs are termed *failures*. We present two different explanations of refusals:

Timed refusals are refusals

We may interpret a timed refusal as the set of (*time,event*) pairs *refused* in a possible history of the process. In the failure $(s, \aleph)$, we consider the refusal set $\aleph$ to be the set of (*time,event*) pairs that the process may refuse to engage in given that it performs the trace s. In the case of our example process P:

$(\langle(1,a)\rangle, ([0, 1+\delta) \times \{b\}))$	is a failure of P: event a may take place at time 1, and in this case, event b may be refused up until time $1+\delta$
$(\langle(1,a)\rangle, ([0,2) \times \{a\}))$	is a failure of P: if event a occurs as soon as it becomes available, then it is refused up until that time *and* from that time onwards.
$(\langle(2,a)\rangle, ([0,2) \times \{a\}))$	is not a failnre of P, as event a must be available from time 1 nntil such time as it occurs.
$(\langle(1,a)\rangle, ([0,\infty) \times \{a\}))$	is not a failure of any process: the alleged refusal describes infinite behaviour.

Note that there is no reason why a pair (t, a) should not be present in both the trace s and the refusal set $\aleph$. In the second failure above, the process refuses a up until time 1. Having engaged in the event at time 1, it refuses any further offers of the same event. This apparent contradiction is not present in the alternative exposition given below, which also permits an intuitive explanation of the hiding operator.

Timed refusals are forcing sets

Alternatively, we may view timed CSP processes as entities upon which the global environment may experiment. In addition to simply observing events, this environment may *force* events over finite intervals of time: if the process is prepared to perform an event when the environment forces it, it must occur instantaneously. Events that are not forced may still occur, if available.

A timed refusal may be viewed as the history of such an experiment; a timed trace is a possible result. The presence of a pair (t, a) in a refusal set corresponds to the act of forcing event a at time t. In the light of this interpretation, we reconsider the failure set of process P:

$(\langle(1,a)\rangle,([0,1+\delta)\times\{b\})$	is a failure of P: if event b is forced over the interval $[0, 1+\delta)$, an event a may be observed at time 1.
$(\langle(1,a)\rangle,([0,2)\times\{a\}))$	is a failure of P: if event a is forced over the interval $[0,2)$, it must occur as soon as it becomes available (at time 1).
$(\langle(2,a)\rangle,([0,2)\times\{a\}))$	is not a failure of P: a should have occurred at time 1.
$(\langle(2,a)\rangle,\emptyset)$	is a failure of P: if a is not forced, it may occur at any time at which it is available.

The advantages of this intuition are

- the interpretation of the empty refusal set, always possible for any trace, is more easily understood.
- hiding a set of events A corresponds to forcing them upon a process.
- the sequential composition operator works by forcing any available occurrence of the special event $\checkmark$.

It should be observed that the *global environment* is merely an intuitive device; we cannot model it as a TCSP process — such a process may permit or prevent the occurrence of events, but cannot *force* their occurrence.

6.3 Stability

In addition to the traces and refusals of a process, which record the response and reaction to external stimuli, we are also interested in the internal activity of a process. We wish to know whether or not the process is making internal progress, whether the process has stabilised. Once a process has stabilised, there can be no further changes of state without an external action occurring. The concept of stability is dual to that of divergence, discussed in earlier models of CSP.

Intuitively, we may imagine a red light on the back of a process that is extinguished once all internal activity has ceased. The stability component of a behaviour is the earliest time by which the light is *guaranteed* to be off. In the failures-stability model for Timed CSP, a typical behaviour of a process is represented by a triple $(s, \alpha, \aleph)$: the stability value α is the earliest time by which the process must have stabilised, given that it exhibits the external behaviour described by trace s and refusal $\aleph$.

Choosing α to represent the earliest time of guaranteed stability means that every (*trace,refusal*) pair of a given process is associated with a unique stability value. Similarly, in the timed stability model, where behaviours are (*trace,stability*) pairs, each trace is associated with a unique stability value.

6.4 Models

The semantic models for CSP and Timed CSP presented in [Re88] form a hierarchy, ordered according to the information contained in a typical element of a semantic set. All of the models are based upon metric spaces, and projection mappings have been defined, allowing the various aspects of a process's behaviour to be considered separately. In the diagram below, these mappings are represented by arrows connecting the various models.

$$\begin{tikzcd}
 & & TM_{FS} \arrow[d] \arrow[ddll] \arrow[ddrr] & & \\
 & & M_{FS} \arrow[dl] \arrow[dr] & & \\
TM_F \arrow[r] \arrow[ddrr] & M_F \arrow[dr] & & M_S \arrow[dl] & TM_S \arrow[l] \arrow[ddll] \\
 & & M_T & & \\
 & & TM_T \arrow[u] & &
\end{tikzcd}$$

As we move up the hierarchy, further aspects of process behaviour are revealed. The lower models in the hierarchy, M_T, M_F, M_S and M_{FS} have no timed information at all; they are the traces, failures and stability models of untimed CSP. The lowest timed model, TM_T, identifies only the timed traces of a process. Higher models identify failures and stabilities, allowing us address both external response and internal activity.

In [Re88], a semantic function is defined for each model. The complexity of the semantics reflects the amount of information required for a successful treatment of timed concurrency. As an example, we present the semantic clause for the prefix operator in the timed failures model:

$$\begin{aligned}
\mathcal{F}_T[\![a \rightarrow P]\!] \;\widehat{=}\;& \{(\langle\rangle, \aleph) \mid a \notin \sigma(\aleph)\} \\
& \cup \\
& \{(\langle(t,a)\rangle^\frown(s + (t+\delta)), \aleph) \mid t \geqslant 0 \wedge a \notin \sigma(\aleph \restriction t) \wedge \\
& \qquad\qquad\qquad\qquad (s, \aleph \dot{-} (t+\delta)) \in \mathcal{F}_T[\![P]\!]\}
\end{aligned}$$

where $\mathcal{F}_T$ is the semantic function from $TCSP$ to TM_F.

A behaviour of $a \rightarrow P$ may arise in one of two ways. If event a has not yet occurred, then it must be available at all times, hence a must not be in $\sigma(\aleph)$, the set of all events in refusal set $\aleph$. If a has occurred, at time t, and is recorded as the first event of the trace, then it must have been available before time t. The process may not participate in any event during an interval of length δ, hence any event may be refused. The subsequent behaviour of the process must be a behaviour of P, translated through time $t + \delta$.

The notation employed above will be defined formally in section 7.2.

7 Specification

In this section, we show how the requirements placed upon a system may be translated into Timed CSP specifications. As the semantic sets represent sets of possible behaviours, we can write each specification as a predicate upon the semantics of a process.

We have seen how the syntax of Timed CSP can be used to produce formal descriptions of time-critical systems. These descriptions can be used to suggest an implementation, or as an algorithmic representation of an existing system at a suitable level of abstraction. Further, we can determine whether or not such a description meets our formalised requirements.

7.1 Behavioural Specifications

We consider a behavioural specification to be a predicate on a typical behaviour of a process. If this predicate holds of all possible behaviours of a process, we say that the process satisfies the specification. This permits us to define a relation between *TCSP* processes and behavioural specifications: the satisfaction operator **sat** for a process P and a specification $S(s, \aleph)$:

$$P \textbf{ sat } S(s,\aleph) \;\; \widehat{=} \;\; \forall (s,\aleph) \in \mathcal{F}_T[\![P]\!] \bullet S(s,\aleph)$$

The form of a behavioural specification identifies the model employed. In the above example, the parameters s and $\aleph$ indicate that the timed failures model, TM_F, is being used. This extends the definition made in [H85]. In [Re88], Reed defines specifications as predicates on the entire semantic set of a process. We define predicates on a typical element of that set. Reed's specifications permit a more detailed analysis of the process semantics; ours are more suited to the capture of general requirements upon a process.

For every behavioural specification S, we can define a corresponding Reed specification R_S as follows:

$$R_S(P) \;\; \widehat{=} \;\; P \textbf{ sat } S(s,\aleph)$$

However, not every Reed specification has a corresponding behavioural specification. For example, the predicate

$$(\langle(1,a),(2,b)\rangle,\emptyset) \;\; \in \;\; \mathcal{F}_T[\![P]\!]$$

cannot be written as a behavioural specification. It states that $(\langle(1,a),(2,b)\rangle,\emptyset)$ is a possible behaviour of P, and to decide upon its truth we need to examine the whole of the semantic set.

We are interested in the correctness of processes. Behavioural specifications reflect this: they insist that every possible behaviour is acceptable. To state that a process *may* participate in a certain event at a certain time, or refuse a certain event at another, without further information, is of little use. We are interested in what can be *guaranteed* about a process behaviour.

7.2 Notation

To capture our requirements as predicates, we define a number of operators on timed traces and timed refusals. A more extensive list of semantic operators is given as an appendix to this paper.

Timed traces are sequences of (*time,event*) pairs. We write $s_1 {}^\frown s_2$ to represent the concatenation of traces s_1 and s_2, and $\#s$ to represent the length of s. As in [H85], we define the relation **in** as follows:

$$s_1 \;\mathbf{in}\; s_2 \;\hat{=}\; \exists\, u, v \bullet u {}^\frown s_1 {}^\frown v = s_2$$

This relation holds whenever the first trace is a contiguous subsequence of the second.

The *last* operator is defined on non-empty timed traces, returning the last event in a trace:

$$last(s {}^\frown \langle (t, a) \rangle) \;\hat{=}\; a$$

while the *end* operator is defined for all traces:

$$\begin{aligned} end(\langle\rangle) &\;\hat{=}\; 0 \\ end(s {}^\frown \langle (t, a) \rangle &\;\hat{=}\; t \end{aligned}$$

The definition of the restriction operator includes the clauses for hatted events in a trace. These are present only in the lower models of the hierarchy; timed refusal information makes them unnecessary.

$$\begin{array}{rcll} \langle\rangle \upharpoonright A & \hat{=} & \langle\rangle & \\ (\langle (t, a) \rangle {}^\frown s) \upharpoonright A & \hat{=} & \langle (t, a) \rangle {}^\frown (s \upharpoonright A) & \text{if } a \in A \\ (\langle (t, a) \rangle {}^\frown s) \upharpoonright A & \hat{=} & s \upharpoonright A & \text{otherwise} \\ (\langle (t, \hat{a}) \rangle {}^\frown s) \upharpoonright A & \hat{=} & \langle (t, \hat{a}) \rangle {}^\frown (s \upharpoonright A) & \text{if } a \in A \\ (\langle (t, \hat{a}) \rangle {}^\frown s) \upharpoonright A & \hat{=} & s \upharpoonright A & \text{otherwise} \end{array}$$

This operator restricts the trace to events drawn from a given set A.

We define two operators on timed refusals. The first restricts the refusal set to events that may be refused after a specified time:

$$\aleph \upharpoonright t \;\hat{=}\; \aleph \cap ([t, \infty) \times \Sigma)$$

where Σ denotes the set of all events. The second operator yields the set of events mentioned in the refusal set $\aleph$.

$$\sigma(\aleph) \;\hat{=}\; \{a \in \Sigma \mid \exists\, t \bullet (t, a) \in \aleph\}$$

If an event is not in $\sigma(\aleph)$, then it is not refused during the behaviour $(s, \aleph)$.

To illustrate the use of these operators, we define:

$$(s, \aleph) \;\hat{=}\; (\langle(1,a),(3,c)\rangle, [0,3) \times \{a,b\})$$

and observe that:

$$\begin{aligned} last(s) &= c \\ end(s) &= 3 \\ s \upharpoonright \{a,b\} &= \langle(1,a)\rangle \\ \sigma(\aleph) &= \{a,b\} \\ \aleph \upharpoonleft 2 &= [2,3) \times \{a,b\} \end{aligned}$$

7.3 Example

Recall the definition made in section 2, of a simple vending machine:

$$VM \;\hat{=}\; coin \rightarrow rattle \rightarrow WAIT(d - \delta)\, ;\, drink \rightarrow VM$$

This process satisfies the following behavioural specifications:

$$\begin{aligned} S_1(s) &\;\hat{=}\; \langle(t_1, rattle),(t_2, drink)\rangle \text{ in } s \Rightarrow t_2 \geqslant t_1 + d \\ S_2(s,\aleph) &\;\hat{=}\; last(s) = rattle \Rightarrow drink \notin \sigma(\aleph \upharpoonleft (end(s) + d)) \\ S_3(s,\sigma) &\;\hat{=}\; last(s) = coin \Rightarrow \sigma = end(s) + \delta \end{aligned}$$

These capture the following requirements:

- a drink is not available until time d has elapsed, following the occurrence of *rattle*.
- when a *rattle* is heard, then a drink *will* be available after a further time d.
- at a time δ after the insertion of a coin, the machine has stabilised: all internal activity has ceased.

That the process meets these specifications could be verified with reference to the semantics. With more complex process descriptions, this is impractical. Fortunately, there is an alternative. We can derive a number of inference rules transforming a behavioural specification on a *TCSP* process to behavioural specifications on its components. Indeed, a complete proof system for timed failures specifications is given in [DS89].

7.4 Proof

A proof of correctness of a Timed CSP process is a verification that it meets a given specification. For example, we might wish to prove that the simple vending machine *VM* meets the first of the above specifications. We can write this proof requirement using the **sat** relation:

$$VM \quad \mathbf{sat} \quad S_1(s, \aleph)$$

and establish the truth of this assertion by demonstrating that the predicate $S_1(s, \aleph)$ holds for a typical element of the semantic set $\mathcal{F}_T[\![VM]\!]$.

A more complex requirement can be placed upon the parallel combination of the time-sensitive vending machine and its user:

$$TUSE \parallel TSVM \quad \mathbf{sat} \quad last(s) = kick \Rightarrow drink \notin \sigma(\aleph \upharpoonright (end(s) + \delta))$$

This captures the requirement that the user cannot break the machine by kicking it; a drink will always become available. Recalliug the definitions of the component processes

$$TSVM \quad \widehat{=} \quad coin \rightarrow \begin{pmatrix} kick \rightarrow STOP \\ \Box \\ WAIT\ 1 \end{pmatrix} ; \begin{pmatrix} drink \rightarrow TSVM \\ \Box \\ kick \rightarrow drink \rightarrow TSVM \end{pmatrix}$$

$$TUSE \quad \widehat{=} \quad coin \rightarrow WAIT\ 3 ; kick \rightarrow drink \rightarrow TUSE$$

we can see that this is guaranteed by the combination of the user's restraint and the invulnerability of the machine after a certain time interval. More formally, we can show that:

$$\begin{aligned} TSVM \quad \mathbf{sat} \quad & last(s) = kick \wedge end(s \upharpoonright coin) + 2 \leqslant end(s) \\ & \qquad \Rightarrow drink \notin \sigma(\aleph \upharpoonright (end(s) + \delta)) \\ TUSE \quad \mathbf{sat} \quad & last(s) = kick \Rightarrow (end(s \upharpoonright coin) + 2 \leqslant end(s) \\ & \qquad\qquad \wedge\ drink \notin \sigma(\aleph \upharpoonright (end(s) + \delta))) \end{aligned}$$

and use the following inference rule (from [DS89]) to establish that the required result holds:

$$\frac{\begin{array}{l} P_1 \ \mathbf{sat}\ T_1(s, \aleph) \\ P_2 \ \mathbf{sat}\ T_2(s, \aleph) \\ T_1(s, \aleph_1) \wedge T_2(s, \aleph_2) \Rightarrow S(s, \aleph_1 \cup \aleph_2) \end{array}}{P_1 \parallel P_2 \ \mathbf{sat}\ S(s, \aleph)}$$

To establish that a parallel combination meets a given specification S, it is sufficient to find two specifications, one for each component, that yield S for a combination of behaviours. More precisely, a typical failure of $P_1 \parallel P_2$ must satisfy:

- any trace of $P_1 \parallel P_2$ is a trace of each component,
- any refusal set of $P_1 \parallel P_2$ will be the union of two refusal sets: one from each of the component processes.

The parallel combination refuses to participate in an event e whenever either or both of its components refuses e.

Similar rules exist to verify that the simpler, sequential processes *TUSE* and *TSVM* satisfy the two specifications given above. Using these rules, we can establish any property of a process that can be captured as a behavioural specification.

8 Discussion

This paper was intended as a brief introduction to Timed CSP. The ideas and notation presented in the previous sections provide the foundation for a uniform theory of timed concurrency. As it stands, Timed CSP is a powerful tool for capturing requirements in a clear and concise fashion, and communicating these requirements to others. Additions being made to the theory, including a complete proof system for behavioural specifications, will simplify the analysis and verification of processes; software tools can be developed to assist in this.

The additions being made to the theory include:

Instability sets: In [Bl89], Blamey develops an alternative treatment of process stability, associating each failure with a set of *instabilities*, rather than a single stability value. Using this approach, we may obtain a basis for a complete proof system for all models of Timed CSP.

Event times: The δ delay, present at each prefix, can be replaced by a function, associating a different delay with each event. This yields a more intuitive treatment of sequential processes, and permits event refinement.

Time-slice parallelism: The parallel operators given in this paper can be used to describe true parallelism. However, we may wish to model multiprocessing behaviour, in which the execution of a process may be suspended. A new semantic definition is required, and has already been formulated in [D89].

Timewise refinement: We can use the structure of the hierarchy to refine processes and specifications, adding the timing information of a higher model.

A number of other directions are also being pursued; these include the addition of probabilistic models to the heirarchy, the development of software tools, and a methodology for process design and implementation.

Acknowledgments

The authors would like to thank Bill Roscoe, Mike Reed, and Stephen Blamey for laying the foundations; Jim Woodcock, Carroll Morgan, Richard Miller, Greg Abowd, Dave Jackson and Michael Goldsmith for advice; Alice King-Farlow and Elizabeth Schneider for inspiration; Phil, Alan, Jeremy, and Colin for coffee and biscuits. This work was supported by grants from SERC and BP.

References

[B89] S.R. Blamey, *TCSP Processes as Predicates,* (to appear) Oxford 1989.

[BG87] A. Boucher and R. Gerth, *A Timed Failures Model for Extended Communicating Scquential Processes*, ICALP 1987 Springer LNCS.

[D89] J.W. Davies, *A Time-Slice Parallel Operator for Timed CSP*, (to appear) Oxford 1989.

[DS89] J.W. Davies and S.A. Schneider, *Factorising Proofs in Timed CSP*, in this volume.

[H85] C.A.R. Hoare, *Communicating Sequential Processes,* Prentice-Hall International 1985.

[J82] G. Jones, *A Timed Model for Communicating Processes,* Oxford University D.Phil Thesis 1982.

[MB76] R.M. Metcalfe and D.R. Boggs, *Ethernet: Distributed Packet Switching for Local Computer Networks*, Communications of the ACM, July 1976, **395–404**.

[OH83] E.R. Olderog and C.A.R. Hoare, *Specification-oriented Semantics for Communicating Processes*, Springer LNCS **154** 1983, **561–572**. (Also, Acta Informatica 23 1986, 9-66.).

[PS88] K. Paliwoda and J.W. Sanders, *The Sliding-Window Protocol in CSP,* Oxford University Programming Research Group Technical Monograph **66**, 1988.

[Re88] G.M. Reed, *A Uniform Mathematical Theory for Real-time Distributed Computing,* Oxford University D.Phil thesis 1988.

[RR86] G.M. Reed and A.W. Roscoe, *A Timed Model for Communicating Sequential Processes*, Proceedings of ICALP'86, Springer LNCS **226** (1986), **314-323**; Theoretical Computer Science **58** 1988, **249–261**.

[RR87] G.M. Reed and A.W. Roscoe, *Metric Spaces as Models for Real-time Concurrency*, Proceedings of the Third Workshop on the Mathematical Foundations of Programming Language Semantics, LNCS **298** 1987, **331–343**.

[Ro82] A.W. Roscoe, *A Mathematical Theory of Communicating Processes,* Oxford University D.Phil thesis 1982.

[T81] A.S. Tanenbaum, *Computer Networks,* Prentice Hall International 1981.

[Z86] A.E. Zwarico, *A Formal Model of Real-Time Computing*, University of Pennsylvania Technical Report 1986.

A Notation for Specification

We define a number of operators on timed traces, timed failures and timed refusals. These will be useful in formulating definitions and specifications of Timed CSP processes.

A.1 Timed Traces

Timed traces are sequences of $(\mathit{time}, \mathit{event})$ pairs. We write $s_1 \frown s_2$ to represent the concatenation of traces s_1 and s_2, and $\#s$ to represent the length of s. As in [H85], we define the relation **in** as follows:

$$s_1 \ \mathbf{in}\ s_2 \ \ \widehat{=} \ \ \exists\, u, v \bullet u \frown s_1 \frown v = s_2$$

This relation holds whenever the first trace is a contiguous subsequence of the second.

First and Last

The *first* and *last* operators are defined upon non-empty traces, returning the first and last events in a trace, respectively:

$$\begin{aligned} \mathit{first}(\langle (t,a)\rangle \frown s) \ &\widehat{=}\ a \\ \mathit{last}(s \frown \langle (t,a)\rangle) \ &\widehat{=}\ a \end{aligned}$$

and the *begin* and *end* operators are defined for all traces:

$$\begin{aligned} \mathit{begin}(\langle\rangle) \ &\widehat{=}\ \infty \\ \mathit{begin}(\langle (t,a)\rangle \frown s) \ &\widehat{=}\ t \\ \mathit{end}(\langle\rangle) \ &\widehat{=}\ 0 \\ \mathit{end}(s \frown \langle (t,a)\rangle) \ &\widehat{=}\ t \end{aligned}$$

The values chosen for the empty trace are the most convenient for the subsequent mathematics.

During, Before and After

We define the *during*, *before*, and *after* operators on timed traces. The first returns the subsequence of the trace with times drawn from set I. The others return the parts of the trace before and after the specified time.

$$\begin{array}{rcll}
\langle\rangle \uparrow I & \widehat{=} & \langle\rangle & \\
(\langle(t,a)\rangle^\frown s) \uparrow I & \widehat{=} & \langle(t,a)\rangle^\frown(s \uparrow I) & \text{if } t \in I \\
 & & (s \uparrow I) & \text{otherwise} \\
s \upharpoonright t & \widehat{=} & s \uparrow [0,t] & \\
s \upharpoonleft t & \widehat{=} & s \uparrow (t,\infty) &
\end{array}$$

where I is a set of time values. In the case that $I = \{t\}$ for some time t, we may omit the set brackets. The *before* operator, $\upharpoonright$, is also used to denote the restriction of a trace to events drawn from a given set. If the second argument of the operator is a set, then:

$$\begin{array}{rcll}
\langle\rangle \upharpoonright A & \widehat{=} & \langle\rangle & \\
(\langle(t,a)\rangle^\frown s) \upharpoonright A & \widehat{=} & \langle(t,a)\rangle^\frown(s \upharpoonright A) & \text{if } a \in A \\
 & \widehat{=} & s \upharpoonright A & \text{otherwise} \\
(\langle(t,\hat{a})\rangle^\frown s) \upharpoonright A & \widehat{=} & \langle(t,\hat{a})\rangle^\frown(s \upharpoonright A) & \text{if } a \in A \\
 & \widehat{=} & s \upharpoonright A & \text{otherwise}
\end{array}$$

Stripping

If timed refusals are not being considered, the events in a timed trace may be labelled with hats; the operator *hstrip* strips the hats from a timed trace:

$$\begin{array}{rcl}
hstrip(\langle\rangle) & \widehat{=} & \langle\rangle \\
hstrip(\langle(t,a)\rangle^\frown s) & \widehat{=} & \langle(t,a)\rangle^\frown hstrip(s) \\
hstrip(\langle(t,\hat{a})\rangle^\frown s) & \widehat{=} & \langle(t,a)\rangle^\frown hstrip(s)
\end{array}$$

whereas the operator *tstrip* strips the timing information from a trace:

$$\begin{array}{rcl}
tstrip(\langle\rangle) & \widehat{=} & \langle\rangle \\
tstrip(\langle(t,a)\rangle^\frown s) & \widehat{=} & \langle a\rangle^\frown tstrip(s)
\end{array}$$

We use *thstrip* to denote the composition of these two functions.

Alphabet

We define an operator σ on traces, which yields the set of events present in the trace:

$$\sigma(s) \;\widehat{=}\; \{a \in \Sigma \mid \exists\, t \bullet \langle (t, a)\rangle \text{ in } \mathit{hstrip}(s)\}$$

Note that we discard the 'hat' information when considering which events are present in a trace.

Shifting and Counting

We define a temporal shift operator:

$$\begin{array}{rcll} \langle\rangle \dot{-} t & \widehat{=} & \langle\rangle & \\ (\langle (t_1, a)\rangle^\frown s) \dot{-} t & \widehat{=} & \langle (t_1 - t, a)\rangle^\frown (s \dot{-} t) & \text{if } t_1 \geqslant t \\ (\langle (t_1, a)\rangle^\frown s) \dot{-} t & \widehat{=} & s \dot{-} t & \text{otherwise} \end{array}$$

and a count operator, $\downarrow$, which returns the number of occurrences of events from a given set:

$$s \downarrow A \;\widehat{=}\; \#(s \restriction A)$$

In the case that $A = \{a\}$ for some event a, we omit the brackets.

Hiding, Equivalence and Parallel Combination

The following functions are used in conjunction with the corresponding *TCSP* operators. We define a simple hiding operator on traces, with the effect of removing hidden events:

$$s \setminus A \;\widehat{=}\; s \restriction (\Sigma - A)$$

and an equivalence relation $\cong$ on traces as follows: $u \cong v$ if and only if u is a permutation of v. As both are timed traces, only events occurring at the same time may be interchanged.

Finally, we define two parallel operators on traces, corresponding to the effect of parallel composition in Timed CSP:

$$\begin{array}{rcl} u\ {}_X\|_Y\ v & \widehat{=} & \{s \mid s \restriction X = u \wedge s \restriction Y = v \wedge s \restriction (X \cup Y) = s\} \\ u \mathbin{|||} v & \widehat{=} & \{s \mid \forall\, t \bullet s \uparrow t \cong (u \uparrow t)^\frown (v \uparrow t)\} \end{array}$$

Examples

As an example, consider the timed trace s, where

$$s \mathrel{\widehat{=}} \langle (1,a),(2,b),(2,\hat{a}),(3,c)\rangle$$

we observe that:

$$\begin{aligned}
last(s) &= c \\
begin(s) &= 1 \\
s \uparrow [2,3) &= \langle (2,b),(2,\hat{a})\rangle \\
s \upharpoonright 2 &= \langle (1,a),(2,b),(2,\hat{a})\rangle \\
s \upharpoonright \{a\} \uparrow 1 &= \langle (2,\hat{a})\rangle \\
\sigma(s) &= \{a,b,c\} \\
hstrip(s) &= \langle (1,a),(2,b),(2,a),(3,c)\rangle \\
tstrip(s) &= \langle a,b,\hat{a},c\rangle \\
thstrip(s) &= \langle a,b,a,c\rangle \\
s \downarrow a &= 2 \\
s \dot{-} 2 &= \langle (0,b),(0,\hat{a}),(1,c)\rangle \\
s \setminus a &= \langle (2,b),(3,c)\rangle
\end{aligned}$$

To illustrate the use of the parallel operators, we make the following definitions:

$$\begin{aligned}
u &\mathrel{\widehat{=}} \langle (1,a),(2,b),(2,c)\rangle \\
v &\mathrel{\widehat{=}} \langle (1,d),(2,b),(2,c)\rangle \\
w &\mathrel{\widehat{=}} \langle (2,b),(3,c)\rangle \\
X &\mathrel{\widehat{=}} \{a,b,c\} \\
Y &\mathrel{\widehat{=}} \{b,c,d\}
\end{aligned}$$

and observe that:

$$\begin{aligned}
u\ {}_X\|_Y\ v &= \{\langle (1,a),(1,d),(2,b),(2,c)\rangle, \langle (1,d),(1,a),(2,b),(2,c)\rangle\} \\
u\ {}_X\|_Y\ w &= \emptyset \\
u \mathbin{|||} w &= \{\langle (1,a),(2,b),(2,b),(2,c),(3,c)\rangle, \\
&\quad\ \langle (1,a),(2,b),(2,c),(2,b),(3,c)\rangle\}
\end{aligned}$$

A.2 Timed Refusals

A number of the above operators have a similar action when applied to timed refusal sets. If we make the definition

$$I(\aleph) \mathrel{\widehat{=}} \{t \mid \exists\, a \bullet (t,a) \in \aleph\}$$

then we can define *begin* and *end* on refusals:

$$\begin{array}{rcll} begin(\aleph) & \mathrel{\widehat=} & \infty & \text{if } \aleph = \emptyset \\ begin(\aleph) & \mathrel{\widehat=} & inf(I(\aleph)) & \text{otherwise} \\ end(\aleph) & \mathrel{\widehat=} & 0 & \text{if } \aleph = \emptyset \\ end(\aleph) & \mathrel{\widehat=} & sup(I(\aleph)) & \text{otherwise} \end{array}$$

Before, After and During

The *before*, *after*, and *during* operators can be defined on refusals:

$$\begin{array}{rcl} \aleph \upharpoonright t & \mathrel{\widehat=} & \aleph \cap ([0, t) \times \Sigma) \\ \aleph \upharpoonleft t & \mathrel{\widehat=} & \aleph \cap ([t, \infty) \times \Sigma) \\ \aleph \uparrow [t_1, t_2) & \mathrel{\widehat=} & \aleph \cap ([t_1, t_2) \times \Sigma) \end{array}$$

Recalling that Σ denotes the set of all events, we see that these restrict a refusal set to events that may be refused before, during, and after the specified times.

Restriction and Hiding

We overload the $\upharpoonright$ symbol to denote set restriction:

$$\aleph \upharpoonright A \mathrel{\widehat=} \aleph \cap ([0, \infty) \times A)$$

with hiding defined in the obvious way:

$$\aleph \setminus A \mathrel{\widehat=} \aleph \upharpoonright (\Sigma - A)$$

Shifting

We define a temporal shift operator on refusals:

$$\aleph \dot{-} t \mathrel{\widehat=} \{(t_1 - t, a) \mid (t_1, a) \in \aleph \wedge t_1 \geqslant t\}$$

Alphabet

We define an alphabet operator σ:

$$\sigma(\aleph) \mathrel{\widehat=} \{a \in \Sigma \mid \exists\, t \bullet (t, a) \in \aleph\}$$

Examples

To illustrate the use of these operators, we make the definition:

$$\aleph \;\widehat{=}\; ([0,2) \times \{a,b\}) \cup ([1,5) \times \{c,d\})$$

and observe that:

$$\begin{aligned}
I(\aleph) &= [0,5) \\
begin(\aleph) &= 0 \\
end(\aleph) &= 5 \\
\aleph \upharpoonright 3 &= ([0,2) \times \{a,b\}) \cup ([1,3) \times \{c,d\}) \\
\aleph \upharpoonleft 3 &= [3,5) \times \{c,d\} \\
\sigma(\aleph) &= \{a,b,c,d\} \\
\aleph \upharpoonright a &= [0,2) \times \{a\} \\
\aleph \dot{-} 2 &= [0,3) \times \{c,d\}
\end{aligned}$$

A.3 Failures

For convenience, we extend some of the above definitions to individual failures: timed (*trace*, *refusal*) pairs:

$$\begin{aligned}
begin(s,\aleph) &\;\widehat{=}\; min\{begin(s), begin(\aleph)\} \\
end(s,\aleph) &\;\widehat{=}\; max\{end(s), end(\aleph)\} \\
(s,\aleph) \uparrow I &\;\widehat{=}\; (s \uparrow I, \aleph \uparrow I) \\
(s,\aleph) \upharpoonright t &\;\widehat{=}\; (s \upharpoonright t, \aleph \upharpoonright t) \\
(s,\aleph) \upharpoonleft t &\;\widehat{=}\; (s \upharpoonleft t, \aleph \upharpoonleft t) \\
\sigma(s,\aleph) &\;\widehat{=}\; \sigma(s) \cup \sigma(\aleph) \\
(s,\aleph) \dot{-} t &\;\widehat{=}\; (s \dot{-} t, \aleph \dot{-} t) \\
(s,\aleph) \upharpoonright A &\;\widehat{=}\; (s \upharpoonright A, \aleph \upharpoonright A) \\
(s,\aleph) \setminus A &\;\widehat{=}\; (s \setminus A, \aleph \setminus A)
\end{aligned}$$

A.4 Processes

We extend the alphabet operator to *TCSP* processes:

$$\sigma(P) \;\widehat{=}\; \bigcup\{\sigma(s) \mid s \in traces(P)\}$$

and observe that this differs from the *alphabet* concept used in earlier versions of CSP.

Factorizing Proofs in Timed CSP[1]

Jim Davies and Steve Schneider

Oxford University Computing Laboratory
Programming Research Group
11 Keble Road
Oxford OX1 3QD

Abstract. A simple notion of specification is introduced, and a complete set of inference rules given, for reasoning about real-time processes. The notation of Timed Communicating Sequential Processes is employed, and the strongest possible specification of a process is discussed. A proof of correctness of a simple protocol is given to illustrate the method of verification.

1 Introduction

Timed CSP is an extension of Communicating Sequential Processes [H85] which includes timing information. It can be used to model time-dependent properties of concurrent systems. An algebraic notation is employed in the definition of processes, capturing the behaviour of a system in a clear and intuitive manner. A uniform hierarchy of semantic models for this notation is presented in [Re88]. Each semantic model identifies a process with a set of possible behaviours: by reasoning about these sets, we may establish properties of the corresponding processes.

[1]The material presented in this paper will appear in the proceedings of the Fifth Conference on the Mathematical Foundations of Programming Semantics (1989): Springer-Verlag LNCS.

In untimed CSP we have a number of algebraic laws that preserve the semantics of a process. These laws allow us to rewrite a process definition to facilitate such reasoning; if necessary, we may eliminate the abstraction and parallel operators. This is not possible in Timed CSP. The semantics of the timed models are necessarily complicated, but we may use the semantic equations to derive a number of useful laws relating processes to predicates on behaviour.

These laws are central to the application of Timed CSP to the design and analysis of complex systems. We can capture the requirements of the specification using the notation of the semantic model, and formalise our intended solution in the process algebra. This solution allows us to move towards an implementation, should this be our aim. In any case, we are obliged to show that our proposed solution meets our requirements; we must verify it.

We consider a verification of a Timed CSP process to be a demonstration that all its possible behaviours meet a proposed specification, expressed as a predicate on a typical element of its semantics. In this case, we say that the process *satisfies* the specification. A specification in TM_{FS}, the most expressive model, can often be written as a conjunction of constraints in the simpler models; the process can then be shown to satisfy each of these independently.

Even within the simpler models, TM_F, TM_S and TM_T, the construction of such a proof directly from the semantics may be difficult and laborious. If we are to reason about complex time-critical distribnted systems, we require a method of translating a proof obligation on a process into proof obligations on its syntactic subcomponents. This method will employ a number of rules grounded in the semantic mappings introduced in [RR86], [RR87] and [Re88].

In this paper, we present the notion of behavioural specifications: correctness conditions on the possible behaviours of a process. We then give a complete set of inference rules for translating such a specification on a compound process into requirements upon its subprocesses. The soundness of each rule can be established from the semantic equations for the relevant operators; example proofs are included as an appendix. To illustrate the use of these rules we present a verification of a simple stop-and-wait protocol in the Timed Failures model, TM_F.

2 Notation

In this section, we present the notation of Timed CSP, the process algebra and the semantic models, as defined in [Re88]. We then explain our concept of specification and introduce the additional notation required for this paper.

2.1 Timed CSP

Timed CSP is a simple extension of CSP [H85]. The process algebra, *TCSP*, is given in Backus-Naur form below:

$$\begin{aligned} P \ ::= \ & \perp \mid STOP \mid SKIP \mid WAIT\ t \mid a \rightarrow P \mid \\ & P \,\Box\, P \mid P \sqcap P \mid P \parallel P \mid P\ {}_A\|_B\ P \mid P \mid\mid\mid P \mid \\ & P\,;P \mid P \setminus A \mid f^{-1}(P) \mid f(P) \mid \mu X \bullet F(X) \end{aligned}$$

These operators are given interpretations in a hierarchy of semantic models, as detailed in [Re88]. These models allow us to write process specifications: a predicate on the semantics of a process corresponds to a requirement on its possible behaviours.

The semantic model TM_F consists of sets of pairs $(s, \aleph)$ satisfying the seven healthiness conditions given in [Re88]. We refer to a pair $(s, \aleph)$ as a timed failure. The semantic function $\mathcal{F}_T$ is defined on elements of *TCSP*, mapping them to failure sets in TM_F.

The first component of a timed failure represents a possible timed trace of the process: a sequence of timed observable events. The second component, $\aleph$, represents a finite union of refusal tokens, each refusal token being the product of a half-open finite time interval and a subset of the set of all events, Σ. This component denotes the (*time*, *event*) pairs that may be refused if the process performs the trace s.

2.2 Specification

We consider a specification to be a predicate on a typical behaviour of a process: an arbitrary element of its semantics. If this predicate holds of all possible behaviours of a process, we say that the process satisfies the specification. We define the satisfaction operator **sat** for a process P and a specification $S(s, \aleph)$:

$$P \textbf{ sat } S(s,\aleph) \ \ \widehat{=} \ \ \forall (s,\aleph) \in \mathcal{F}_T[\![P]\!] \bullet S(s,\aleph)$$

From this definition, we can establish a number of simple inference rules:

$$\frac{}{P \textbf{ sat } true} \qquad \frac{\begin{array}{l} P \textbf{ sat } S(s,\aleph) \\ P \textbf{ sat } T(s,\aleph) \end{array}}{P \textbf{ sat } S(s,\aleph) \wedge T(s,\aleph)} \qquad \frac{\begin{array}{l} P \textbf{ sat } S(s,\aleph) \\ S(s,\aleph) \Rightarrow T(s,\aleph) \end{array}}{P \textbf{ sat } T(s,\aleph)}$$

Using the **sat** operator, we can capture any requirement that corresponds to a condition upon *all* of the possible behaviours of a process. The resulting predicate upon the *TCSP* process we call a behavioural specification. In [Re88], Reed defines specifications as predicates on the semantic set of a process, we define predicates on a typical element of that set. Behavioural specifications form a subset of Reed's specifications.

Reed's specifications permit a more detailed analysis of the process representation; ours are more suited to the capture of general requirements upon a process.

For example, the predicate

$$(\langle(1,a),(2,b)\rangle,\emptyset) \;\in\; \mathcal{F}_T[\![P]\!]$$

cannot be written as a behavioural specification. It states that $(\langle(1,a),(2,b)\rangle,\emptyset)$ is a possible behaviour of P, and to decide upon its truth we need to examine the whole of the semantic set.

We are interested in the correctness of processes. Behavioural specifications reflect this: they insist that every possible behaviour is acceptable. To state that a process *may* participate in a certain event at a certain time, or refuse a certain event at another, without further information, is of little use. We are interested in what can be guaranteed about a process behaviour.

2.3 Notation

For convenience, we define a number of operators on timed failures, timed traces and timed refusals [2]. We define two functions on traces:

$$\begin{aligned} last(s\frown\langle(t,a)\rangle) &\;\hat{=}\; a \\ tstrip(\langle\rangle) &\;\hat{=}\; \langle\rangle \\ tstrip(\langle(t,a)\rangle\frown s) &\;\hat{=}\; \langle a\rangle\frown tstrip(s) \end{aligned}$$

The first returns the last event in the trace, the second merely strips the time information from the trace.

We define the *before*, *after*, and *during* operators on refusals:

$$\begin{aligned} \aleph \upharpoonright t &\;\hat{=}\; \aleph \cap ([0,t)\times\Sigma) \\ \aleph \upharpoonleft t &\;\hat{=}\; \aleph \cap ([t,\infty)\times\Sigma) \\ \aleph \uparrow [t_1,t_2) &\;\hat{=}\; \aleph \cap ([t_1,t_2)\times\Sigma) \end{aligned}$$

Recalling that Σ denotes the set of all events, we see that these restrict a refusal set to events that may be refused before, during, and after the specified times.

[2]From now on, we will omit the prefix 'timed' as all subsequent specifications will be drawn from TM_F

We define a subtraction operator on traces and refusals, translating through time:

$$\begin{aligned} \langle\rangle \dot{-} t &\mathrel{\widehat{=}} \langle\rangle \\ (\langle(t_1, a)\rangle^\frown s) \dot{-} t &\mathrel{\widehat{=}} \begin{cases} s \dot{-} t & \text{if } t_1 < t \\ \langle(t_1 - t, a)\rangle^\frown (s \dot{-} t) & \text{otherwise} \end{cases} \\ \aleph \dot{-} t &\mathrel{\widehat{=}} \{(t_1 - t, a) \mid (t_1, a) \in \aleph \wedge t_1 \geqslant t\} \end{aligned}$$

We define an operator α on traces and refusals, yielding the set of events that occur in each. For convenience, we extend the definition of α to cover failures and processes; in the latter case, the result is the set of events in which the process may participate. Observe that this operator differs from the *alphabet* concept used in earlier versions of CSP.

$$\begin{aligned} \alpha(s) &\mathrel{\widehat{=}} \{a \in \Sigma \mid \exists\, t \bullet \langle(t, a)\rangle \text{ in } s\} \\ \alpha(\aleph) &\mathrel{\widehat{=}} \{a \in \Sigma \mid \exists\, t \bullet (t, a) \in \aleph\} \\ \alpha(s, \aleph) &\mathrel{\widehat{=}} \alpha(s) \cup \alpha(\aleph) \\ \alpha(P) &\mathrel{\widehat{=}} \bigcup\{\alpha(s) \mid s \in \mathit{traces}(P)\} \end{aligned}$$

Similarly, we extend the definition of *end* in [Re88]:

$$\mathit{end}(s, \aleph) \mathrel{\widehat{=}} \mathit{max}\{\mathit{end}(s), \mathit{end}(\aleph)\}$$

Finally, for use with the hiding operator, we define a predicate on failures, indexed by a set of events A:

$$\widehat{A} \mathrel{\widehat{=}} ([0, \mathit{end}(s, \aleph)) \times A) \subseteq \aleph$$

This predicate holds exactly when the failure $(s, \aleph)$ is *activated* on set A.

3 Abstraction and Concurrency

As an introduction to our method of verifying processes, we consider two operators central to the language of Timed CSP: the hiding and parallel operators.

3.1 Hiding

In applying Timed CSP to complex systems, we use the hiding operator to abstract away from internal behaviour. To prove our description correct, we may need to reason about this behaviour. Hiding a set of events A from the environment of a process P restricts the set of possible behaviours to those in which P is forced to perform events from A as soon as they become available: the *A-activated* behaviours.

The events in A are no longer observable from the environment, and we may not mention them in reasoning about $P \setminus A$. Instead, we identify the A-activated behaviours of the process, establishing results that may involve events from A. These results may then be used to derive a specification that is independent of events from A. This specification is then satisfied by $P \setminus A$.

In the untimed version of CSP, we can use algebraic laws to eliminate the hiding operator from a process description: these laws preserve the equivalent set of behaviours. It would be possible to derive similar laws for Timed CSP, but their complexity would render them unusable: consider the identity below, which corresponds to the simplest non-trivial case of hiding over deterministic choice.

$$(a \to STOP \Box b \to SKIP) \setminus \{a\} \;\equiv\; ((SKIP \Box b \to SKIP)\,;SKIP) \parallel b \to SKIP$$

Our approach offers a simple, systematic solution to the problem of hiding.

We defined the $\hat{}$ operator in the previous section: $\hat{A}$ holds precisely when $(s, \aleph)$ is an A-activated failure. The following inference rule illustrates the relationship between the failures of P and those of $P \setminus A$:

$$\frac{P \textbf{ sat } (\hat{A}(s,\aleph) \wedge \alpha(\aleph') \subsetneq A) \Rightarrow S(s \setminus A, \aleph - \aleph')}{P \setminus A \textbf{ sat } S(s,\aleph)}$$

This follows from the semantic equation for the hiding operator given in [Re88], and transforms a proof obligation on $P \setminus A$ into one on P.

3.2 Parallelism

Timed CSP has three parallel operators: alphabeticised parallel, synchronised parallel and interleaving. The latter operators can be viewed as particular instances of the first. In this paper, we will illustrate the use of the most general form of parallel operator.

The alphabeticised parallel operator places a restriction on the events communicable by each argument: in the parallel combination $P \;{}_X\!\|_Y\; Q$, process P may perform only those events in set X. Similarly, Q is restricted to those in Y. The two processes must co-operate on events common to both sets.

As in the case of hiding, it would be impractical to eliminate alphabeticised parallelism using algebraic laws. As an illustration, consider the identity below, which holds for untimed CSP.

$$a \rightarrow P \,|||\, b \rightarrow Q \;\equiv\; a \rightarrow (P \,|||\, b \rightarrow Q) \,\Box\, b \rightarrow ((a \rightarrow P) \,|||\, Q)$$

This is no longer true for Timed CSP processes because of the delay δ introduced by the prefix operator. This can arise whenever a process contains a form of parallelism which is not completely synchronised.

This means that, except for the simple case of completely synchronised parallelism, we cannot transform a process in a semantics-preserving fashion and alter the degree of parallelism present. However, our inability to do this need not detract from the applicability of the formalism; time-critical systems with communication delays have a minimum degree of parallelism. We can derive rules to allow us to establish properties of such systems.

As an introduction to the operator, consider the special case that is synchronised parallelism. The following inference rule can be derived for this operator:

$$\frac{\begin{array}{l} P_1 \textbf{ sat } S_1(s, \aleph) \\ P_2 \textbf{ sat } S_2(s, \aleph) \\ S_1(s, \aleph_1) \wedge S_2(s, \aleph_2) \Rightarrow S(s, \aleph_1 \cup \aleph_2) \end{array}}{P_1 \parallel P_2 \textbf{ sat } S(s, \aleph)}$$

To establish that a parallel combination meets a given specification S, it is sufficient to find two specifications, one for each component, that yield S for a combination of behaviours. More precisely, a typical failure of $P_1 \parallel P_2$ must satisfy:

- any trace of $P_1 \parallel P_2$ is a trace of each component.
- any refusal set of $P_1 \parallel P_2$ will be the union of two refusal sets: one from each of the component processes.

The parallel combination refuses to participate in an event e whenever either or both of its components refuses e.

The rule for the alphabeticised parallel operator is necessarily more complicated:

$$\frac{\begin{array}{l} P_1 \textbf{ sat } S_1(s, \aleph) \\ P_2 \textbf{ sat } S_2(s, \aleph) \\ (\alpha(s_1, \aleph_1) \subseteq X \wedge \alpha(s_2, \aleph_2) \subseteq Y \wedge \alpha(\aleph_3) \subseteq \Sigma - (X \cup Y) \wedge S_1(s_1, \aleph_1) \\ \qquad \wedge S_2(s_2, \aleph_2) \wedge s_3 \in s_1 \, {}_X\|_Y \, s_2) \Rightarrow S(s_3, \aleph_1 \cup \aleph_2 \cup \aleph_3) \end{array}}{P_1 \, {}_X\|_Y \, P_2 \textbf{ sat } S(s, \aleph)}$$

As before, we must find two specifications, one for each component, that yield S for a combination of behaviours. This time, a failure of the parallel combination must satisfy:

- any trace of $P_1 \ {}_X\|_Y \ P_2$ must be the parallel combination of a trace from each component.
- any refusal set of $P_1 \ {}_X\|_Y \ P_2$ must be the union of three refusal sets: one from each component, and an arbitrary refusal set whose alphabet lies outside $X \cup Y$.

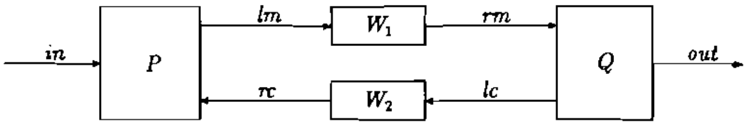
Recall that the parallel operator on traces produces a set of traces: sequences of events drawn from $X \cup Y$, whose restriction to the sets X and Y produces the first and second arguments of the operator, respectively.

These conditions lead to the third antecedent of the rule, which allows us to transform a predicate on the failures of a parallel combination into requirements on the corresponding failures of the component processes. Together with the hiding rule, this is sufficient to treat the example of the next section.

4 A Simple Protocol

A protocol is a distributed algorithm for facilitating the communication of messages between processes. CSP is particularly suitable for the specification of protocols; the enhancements introduced in Timed CSP allow us to address the timing considerations that are often necessary for the correctness of the protocol. Using Timed CSP, we can describe and analyse processes that include *timeouts*, *interrupts* and *time-critical synchronisation.*

In this section, we consider the specification of a simple 'stop-and-wait' protocol, similar to the one described in [PS88]. This consists of two processes, P and Q, communicating across two wires: W_1 and W_2. Together, they control the flow of data between two external processes. This may be represented pictorially as follows:



In general, protocols allow for unreliable channels, by duplicating data or requiring acknowledgements: such behaviour is easily modelled in Timed CSP. However, our purpose is to illustrate the use of the inference rules; we need not concern ourselves with these complications. Our protocol addresses only dataflow considerations, and we assume that the wires W_1 and W_2 are *reliable*: for every input, there is a corresponding output.

4.1 Specifications

There are many requirements that we could place upon the protocol, but we will consider just one: that if a message is input, then output is ready within two seconds. Formally, we wish our protocol $PROT$ to meet the following timed failures specification:

$$SPEC(s, \aleph) \ \widehat{=}\ last(s) = in \Rightarrow out \notin \alpha(\aleph \uparrow (end(s) + 2))$$

We give conditions on the components of the protocol, and verify that they are sufficient to ensure that the protocol exhibits this behaviour.

The sending process P should meet the following specification: it should perform the three events in, lm, rc in strict rotation; after performing an event, it should be prepared to perform the next within a certain time; initially, it should be ready to receive an input. We capture these requirements in the timed failures specification $SPEC_P$:

$$\begin{aligned} SPEC_P(s, \aleph) \ \widehat{=}\ & tstrip(s) \leqslant \langle in, lm, rc\rangle^* \wedge \\ & last(s) = in \Rightarrow lm \notin \alpha(\aleph \uparrow (end(s) + 2\delta)) \wedge \\ & last(s) = lm \Rightarrow rc \notin \alpha(\aleph \uparrow (end(s) + 2\delta)) \wedge \\ & last(s) = rc \Rightarrow in \notin \alpha(\aleph \uparrow (end(s) + 2\delta)) \wedge \\ & s = \langle\rangle \Rightarrow in \notin \alpha(\aleph) \end{aligned}$$

After accepting and transmitting a message, the sending process must await confirmation from the receiving process before accepting another. The receiving process will send a confirmation signal once the previous message has been output. Initially, the system is empty. Hence we wish the receiving process Q to satisfy $SPEC_Q$:

$$\begin{aligned} SPEC_Q(s, \aleph) \ \widehat{=}\ & tstrip(s) \leqslant \langle rm, out, lc\rangle^* \wedge \\ & last(s) = rm \Rightarrow out \notin \alpha(\aleph \uparrow (end(s) + 2\delta)) \wedge \\ & last(s) = out \Rightarrow lc \notin \alpha(\aleph \uparrow (end(s) + 2\delta)) \wedge \\ & last(s) = lc \Rightarrow rm \notin \alpha(\aleph \uparrow (end(s) + 2\delta)) \wedge \\ & s = \langle\rangle \Rightarrow rm \notin \alpha(\aleph) \end{aligned}$$

The wires W_1 and W_2 have a propagation delay of 1 second, and will not be required to transmit more than one message at a time. However, each must be ready to accept another input almost immediately after output. They satisfy the specifications $SPEC_{W_1}$ and $SPEC_{W_2}$ respectively, where

$$\begin{aligned} SPEC_{W_1}(s,\aleph) \;\widehat{=}\; & tstrip(s) \leqslant \langle lm, rm\rangle^* \wedge \\ & last(s) = lm \Rightarrow rm \notin \alpha(\aleph \upharpoonright (end(s)+1)) \wedge \\ & last(s) = rm \Rightarrow lm \notin \alpha(\aleph \upharpoonright (end(s)+2\delta)) \wedge \\ & s = \langle\rangle \Rightarrow lm \notin \alpha(\aleph) \end{aligned}$$

$$\begin{aligned} SPEC_{W_2}(s,\aleph) \;\widehat{=}\; & tstrip(s) \leqslant \langle lc, rc\rangle^* \wedge \\ & last(s) = lc \Rightarrow rc \notin \alpha(\aleph \upharpoonright (end(s)+1)) \wedge \\ & last(s) = rc \Rightarrow lc \notin \alpha(\aleph \upharpoonright (end(s)+2\delta)) \wedge \\ & s = \langle\rangle \Rightarrow lc \notin \alpha(\aleph) \end{aligned}$$

The protocol is a combination of the sending process, the receiving process, and the wires. We combine these in $TCSP$ by way of the alphabeticised parallel operator, and hide the internal detail. If we define the sets

$$\begin{aligned} X &\;\widehat{=}\; \{in, lm, rc\} \\ Y &\;\widehat{=}\; \{out, rm, lc\} \\ C &\;\widehat{=}\; \{lc, rc\} \\ M &\;\widehat{=}\; \{lm, rm\} \\ A &\;\widehat{=}\; M \cup C \end{aligned}$$

then the protocol may be defined:

$$PROT \;\widehat{=}\; ((P \;{}_X\|_Y\; Q) \;{}_{X\cup Y}\|_{M\cup C}\; (W_1 \;{}_M\|_C\; W_2)) \setminus A$$

4.2 Verification

Having formalised our requirements, we can now use the inference rules given in section 3 to demonstrate that the protocol $PROT$ will meet the specification $SPEC$. We wish to establish that:

$$PROT \quad \textbf{sat} \quad SPEC(s,\aleph)$$

The definition of $PROT$ involves the hiding operator at the outermost level, so we must first apply the hiding rule. This reduces the proof requirement to:

$$\begin{aligned} (P \;{}_X\|_Y\; Q) \;{}_{X\cup Y}\|_{M\cup C}\; (W_1 \;{}_M\|_C\; W_2) \quad \textbf{sat} \quad & \alpha(\aleph') \subseteq A \wedge ([0, end(s,\aleph)) \times A) \subseteq \aleph \\ & \Rightarrow SPEC(s \setminus A, \aleph - \aleph') \end{aligned}$$

This is a proof requirement on a parallel combination, so we apply the rule for the parallel operator. We have then to find specifications S_1 and S_2 such that:

$$\begin{array}{lll} P \;{}_X\|_Y\; Q & \textbf{sat} & S_1(s, \aleph) \\ W_1 \;{}_M\|_C\; W_2 & \textbf{sat} & S_2(s, \aleph) \end{array}$$

$$\left.\begin{array}{l} \alpha(s_1, \aleph_1) \subseteq (X \cup Y) \wedge \alpha(s_2, \aleph_2) \subseteq (M \cup C) \\ \alpha(\aleph_3) \subseteq \Sigma - (X \cup Y \cup M \cup C) \\ S_1(s_1, \aleph_1) \wedge S_2(s_2, \aleph_2) \wedge s_3 \in s_1 \;{}_{X\cup Y}\|_{M\cup C}\; s_2 \\ \aleph = \aleph_1 \cup \aleph_2 \cup \aleph_3 \\ \alpha(\aleph') \subseteq A \wedge ([0, end(s_3, \aleph)) \times A) \subseteq \aleph \end{array}\right\} \Rightarrow SPEC(s_3 \setminus A, \aleph - \aleph')$$

Before we continue, we note that the specification $SPEC$ is independent of the hidden set of events A, for consider the definition:

$$SPEC \;\; \widehat{=} \;\; last(s) = in \Rightarrow out \notin \alpha(\aleph \upharpoonright (end(s) + 2))$$

Formally, we can show that

$$SPEC(s, \aleph \upharpoonright (\Sigma - A)) \;\; \Rightarrow \;\; SPEC(s, \aleph)$$

This concurs with our intuition: the correctness of the protocol may be dependent upon hidden interactions, but our formal description of the service provided (the specification $SPEC$) should abstract away from internal detail.

Taking this in conjunction with the alphabet conditions upon the failure sets, we may reduce the third proof obligation to

$$\left.\begin{array}{l} \alpha(s_1, \aleph_1) \subseteq (X \cup Y) \wedge \alpha(s_2, \aleph_2) \subseteq (M \cup C) \\ \alpha(\aleph_3) \subseteq \Sigma - (X \cup Y) \\ S_1(s_1, \aleph_1) \wedge S_2(s_2, \aleph_2) \wedge s_3 \in s_1 \;{}_{X\cup Y}\|_{M\cup C}\; s_2 \\ ([0, end(s_3, \aleph_1 \cup \aleph_2 \cup \aleph_3)) \times A) \subseteq \aleph_1 \cup \aleph_2 \end{array}\right\} \Rightarrow SPEC(s_3 \setminus A, \aleph_1)$$

To identify S_1 we apply the parallel rule once again. We are then required to find S_4 and S_5 such that:

$$\begin{array}{lll} P & \textbf{sat} & S_4(s, \aleph) \\ Q & \textbf{sat} & S_5(s, \aleph) \end{array}$$

$$\left.\begin{array}{l} \alpha(s_4, \aleph_4) \subseteq X \wedge \alpha(s_5, \aleph_5) \subseteq Y \\ \alpha(\aleph_6) \subseteq \Sigma - (X \cup Y) \\ S_4(s_4, \aleph_4) \wedge S_5(s_5, \aleph_5) \wedge s_6 \in s_4 \;{}_X\|_Y\; s_5 \end{array}\right\} \Rightarrow S_1(s_6, \aleph_4 \cup \aleph_5 \cup \aleph_6)$$

We already have specifications for the components P and Q. Substituting these for S_4 and S_5, and using the alphabet conditions upon the traces and refusals, we can reduce this proof obligation to:

$$\left.\begin{array}{l} SPEC_P(s \upharpoonright X, \aleph \upharpoonright X) \\ SPEC_Q(s \upharpoonright Y, \aleph \upharpoonright Y) \\ \alpha(s) \subseteq (X \cup Y) \end{array}\right\} \Rightarrow S_1(s, \aleph)$$

This yields a suitable instantiation for S_1: the antecedent of the above expression. In a similar fashion, we arrive at the following instantiation for S_2:

$$\begin{array}{l} SPEC_{W_1}(s \upharpoonright M, \aleph \upharpoonright M) \wedge \\ SPEC_{W_2}(s \upharpoonright C, \aleph \upharpoonright C) \wedge \\ \alpha(s) \subseteq (M \cup C) \end{array}$$

Our proof requirement can then be written as follows:

$$\left.\begin{array}{l} \alpha(s_1, \aleph_1) \subseteq (X \cup Y) \wedge \alpha(s_2, \aleph_2) \subseteq (M \cup C) \\ \alpha(\aleph_3) \subseteq \Sigma - (X \cup Y) \\ SPEC_P(s_1 \upharpoonright X, \aleph_1 \upharpoonright X) \wedge SPEC_Q(s_1 \upharpoonright Y, \aleph_1 \upharpoonright Y) \\ SPEC_{W_1}(s_2 \upharpoonright M, \aleph_2 \upharpoonright M) \wedge SPEC_{W_2}(s_2 \upharpoonright C, \aleph_2 \upharpoonright C) \\ ([0, end(s_3, \aleph_1 \cup \aleph_2 \cup \aleph_3)) \times A) \subseteq \aleph_1 \cup \aleph_2 \\ s_3 \in s_1 \,{}_{X \cup Y}\|_{M \cup C}\, s_2 \end{array}\right\} \Rightarrow SPEC(s_3 \setminus A, \aleph_1)$$

The alphabet conditions in S_1 and S_2 are subsumed in the first two conditions above.

We have reduced the proof obligation to a predicate on traces and refusal sets: the verification may be completed using simple properties of sets and sequences: assuming the conjuncts in the above antecedent, we are trying to establish that

$$last(s_3 \setminus A) = in \Rightarrow out \notin \alpha(\aleph_1 \rceil (end(s_3 \setminus A) + 2))$$

From $SPEC_P$, $SPEC_Q$, $SPEC_{W_1}$, $SPEC_{W_2}$, and the properties of sequences, we can deduce that

$$s_3 \leqslant \langle in, lm, rm, out, lc, rc \rangle^*$$

We then proceed by case analysis on the identity of the last event in s_3, given that $last(s_3 \setminus A) = in$, there are three possibilities.

Case: $last(s_3) = in$

By $SPEC_P$,	$lm \notin \alpha((\aleph_1 \upharpoonright X) \uparrow (end(s_1 \upharpoonright X) + 2\delta))$
In this case	$end(s_1) = end(s_1 \upharpoonright X)$
and we know that	$lm \notin Y$
Hence	$lm \notin \alpha(\aleph_1 \uparrow (end(s_3) + 2\delta))$
Similarly, as	$s_3 \in s_1 \; {}_{X \cup Y}\|_{M \cup C} \; s_2$,
$SPEC_{W_1}$ implies that	$lm \notin \alpha(\aleph_2 \uparrow (end(s_3) + 2\delta))$
Hence	$lm \notin \alpha((\aleph_1 \cup \aleph_2 \cup \aleph_3) \uparrow (end(s_3) + 2\delta))$
However,	$([0, end(s, \aleph_1 \cup \aleph_2 \cup \aleph_3)) \times A) \subseteq (\aleph_1 \cup \aleph_2 \cup \aleph_3)$
and	$lm \in A$
So	$end(\aleph_1 \cup \aleph_2 \cup \aleph_3) \leqslant end(s_3) + 2\delta$
But $\delta \ll 1$, so	$(\aleph_1 \cup \aleph_2 \cup \aleph_3) \uparrow (end(s_3) + 2)) = \{\}$
We condnde that	$out \notin \alpha((\aleph_1 \cup \aleph_2 \cup \aleph_3) \uparrow end(s_3 \setminus A + 2))$

Case: $last(s_3) = lm$

We establish that $end(s_3) \leqslant end(s_3 \setminus A) + 2\delta$: that the lm event occurred within time 2δ of the last inpnt.

Assume otherwise:	$end(s_3) > end(s_3 \setminus A) + 2\delta$
If we let t be the time	$(end(s_3 \setminus A) + end(s_3) + 2\delta)/2$
Then we know that	$last(s_3 \upharpoonright t) = in$
By the previous case	$lm \notin \alpha(((\aleph_1 \cup \aleph_2 \cup \aleph_3) \upharpoonright t) \uparrow (end(s_3 \upharpoonright t) + 2\delta))$
From our assumptions	$([0, end(s_3, \aleph_1 \cup \aleph_2 \cup \aleph_3)) \times A) \subseteq \aleph_1 \cup \aleph_2$
And	$end(s_3 \upharpoonright t) + 2\delta = end(s_3 \setminus A) + 2\delta < t$
Hence	$lm \in \alpha(((\aleph_1 \cup \aleph_2 \cup \aleph_3) \upharpoonright t) \uparrow (end(s_3 \upharpoonright t) + 2\delta))$

Forcing a contradiction.

We can show, with a similar argument to the first case, in which the event rm replaces lm, that $end(\aleph_1 \cup \aleph_2 \cup \aleph_3) \leqslant end(s_3) + 1$. From above, $end(s_3) \leqslant end(s_3 \setminus A) + 2\delta$: the result follows.

Case: $last(s_3) = rm$

By a similar argument, we can establish that the event rm must occur no later than $1 + 2\delta$ after the last input. We then appeal to the specification of Q, and the result follows immediately. □

The treatment of hiding in Timed CSP is central to the construction of the above proof; the hidden events *lm* and *rm* must occur as soon as possible. Our method of proof allowed us to include these events in our reasoning, by eliminating the hiding operator from our proof obligation.

4.3 Other Requirements

Only at the final stage of the proof did we identify the protocol requirement $SPEC$. To establish that another property holds of the above protocol, it would not be necessary to perform the whole proof again. We have characterised the behaviour of the protocol in terms of the known properties of its components. To prove that the protocol satisfies an arbitrary specification S, we have only to show that the following predicate is true:

$$\left.\begin{array}{l}
\alpha(s_1, \aleph_1) \subseteq (X \cup Y) \wedge \alpha(s_2, \aleph_2) \subseteq (M \cup C) \\
\alpha(\aleph_3) \subseteq \Sigma - (X \cup Y) \\
SPEC_P(s_1 \upharpoonright X, \aleph_1 \upharpoonright X) \wedge SPEC_Q(s_1 \upharpoonright Y, \aleph_1 \upharpoonright Y) \\
SPEC_{W_1}(s_2 \upharpoonright M, \aleph_2 \upharpoonright M) \wedge SPEC_{W_2}(s_2 \upharpoonright C, \aleph_2 \upharpoonright C) \\
\aleph = \aleph_1 \cup \aleph_2 \cup \aleph_3 \wedge s_3 \in s_1 \; {}_{X \cup Y}\|_{M \cup C} \; s_2 \\
\alpha(\aleph') \subseteq A \wedge ([0, end(s_3, \aleph)) \times A) \subseteq \aleph
\end{array}\right\} \Rightarrow S(s_3 \setminus A, \aleph - \aleph')$$

For a particular specification S, we will be able to discard most of the conditions in the antecedent: the residual proof requirement is often easy to discharge.

5 Recursion and Delay

The inference rules presented in section 3 were sufficient for the example proof above. If we wish to provide implementations for the components mentioned in the previous section, we will require other $TCSP$ operators; to verify these implementations, we will require other inference rules.

5.1 Prefixing

The simplest $TCSP$ process is deadlock, or $STOP$. It cannot engage in any event, so any trace must be empty. It may refuse any event at any time, so there are no restrictions upon refusal set $\aleph$:

$$\frac{}{STOP \ \mathbf{sat} \ (s = \langle\rangle)}$$

This process will be useful in showing that certain specifications are *satisfiable*: that there is a process that will satisfy them.

More interesting processes will be able to perform events: for these, we will require the *prefix* operator. In Timed CSP, this operator introduces a delay, corresponding to the time taken to recover from participation:

$$\frac{\begin{array}{l} P \ \mathbf{sat}\ T(s,\aleph) \\ \left.\begin{array}{l} s = \langle\rangle \wedge a \notin \alpha(\aleph) \\ \vee \\ s = \langle (t,a)\rangle ^\frown s' \wedge a \notin \alpha(\aleph \upharpoonright t) \wedge T(s' \dot{-} (t+\delta), \aleph \dot{-} (t+\delta)) \end{array}\right\} \Rightarrow S(s,\aleph) \end{array}}{(a \rightarrow P)\ \mathbf{sat}\ S(s,\aleph)}$$

Any behaviour of the process $a \rightarrow P$ must involve the non-refusal of event a until it has been performed. If event a occurs at time t, the subsequent behaviour will be that of process P, but starting at time $t + \delta$ instead of time 0. If process P meets the specification $T(s, \aleph)$, then these subsequent behaviours will be described by the predicate $T(s' \dot{-} (t + \delta), \aleph \dot{-} (t + \delta))$.

5.2 Recursion

Almost any application of *TCSP* will involve repetitive behaviour: to model this, we can use the recursion operator μ. If F is a function defined on *TCSP* processes, we define the function:

$$\begin{array}{rcl} C_{\hat{F}} & : & TM_F \rightarrow TM_F \\ C_{\hat{F}}(X) & \hat{=} & \mathcal{F}_T[\![WAIT\,\delta\,; F(X)]\!] \end{array}$$

The process $\mu X \bullet F(X)$ behaves as the fixed point of $C_{\hat{F}}$ in the model TM_F:

$$\mu X \bullet F(X) \ \equiv \ F(WAIT\,\delta\,; \mu X \bullet F(X))$$

The recursion induction theorem introduced by Roscoe in [Ro82], developed by Reed in [Re88], provides the basis for an inference rule for recursively-defined processes:

$$\frac{\begin{array}{l} \forall X : TCSP \bullet X \ \mathbf{sat}\ S(s,\aleph) \Rightarrow F(WAIT\,\delta\,; X)\ \mathbf{sat}\ S(s,\aleph) \\ \exists P : TCSP \bullet P\ \mathbf{sat}\ S(s,\aleph) \end{array}}{\mu X \bullet F(X)\ \mathbf{sat}\ S(s,\aleph)}$$

The topological result from which this rule is derived requires that the predicate "**sat** $S(s,\aleph)$" be both continuous and satisfiable on *TCSP* processes. It is a consequence of the definition of the **sat** operator that all such predicates are continuous; this leaves the rule with only one side condition: included as the second antecedent above. We also require that $C_{\hat{P}}$ is a contraction mapping on TM_F, and that the specification $S(s, \aleph)$ is preserved by each recursive call. The first of these follows from the continuity of all basic *TCSP* operators, the second becomes the first antecedent of the rule.

It is possible that the specification $S(s, \aleph)$ may only be satisfiable by a recursive process. In this case, the side condition cannot be established without a separate inductive proof. By extending the contraction mapping that corresponds to F, we can produce a rule that does not have this problem:

$$\frac{\forall X \bullet X \ \mathbf{sat}\ S(s,\aleph) \Rightarrow F(\mathit{WAIT}\,\delta\,;X)\ \mathbf{sat}\ S(s,\aleph)}{\mu X \bullet F(X)\ \mathbf{sat}\ S(s,\aleph)}$$

This follows from the same topological result as the previous rule, given a simple extension to the semantic function $\mathcal{F}_T$, as detailed in appendix B. We have eliminated the second antecedent. The first antecedent is stronger: we may no longer assume that the semantics of X satisfies the axioms of TM_F: we have lost the implicit assumption that X is a *TCSP* process.

The set of inference rules in this paper is independent of the axioms of the model TM_F, so each rule may be applied to arbitrary sets of failures: they can therefore be used to establish the new antecedent. Further, the fact that all of our specifications are behavioural means that this rule is no weaker than the recursion rule in TM_F.

5.3 Delay

Finally, we will need to reason about the behaviours of processes involving delays. We may derive a simple rule from the inference rules for the sequential composition and delay operators:

$$\frac{\begin{array}{l} P\ \mathbf{sat}\ T(s,\aleph) \\ \left.\begin{array}{l} s = \langle\rangle \wedge end(\aleph) \leqslant t \\ \vee \\ begin(s) \geqslant t \wedge T(s \dot{-} t, \aleph \dot{-} t) \end{array}\right\} \Rightarrow S(s,\aleph) \end{array}}{\mathit{WAIT}\ t\ ; P\ \mathbf{sat}\ S(s,\aleph)}$$

The inclusion of arbitrary refusals $\aleph'$ before time t reflects the fact that *WAIT* $t ; P$ may refuse any event before time t.

Whenever we apply the recursion rule, we will be left with a proof obligation on $WAIT\,\delta\,;X$, given that X satisfies a certain specification. In this case, an alternative form of the above rule will be more useful:

$$\frac{P \text{ sat } S(s,\aleph)}{WAIT\ t\,;P \text{ sat } \begin{array}{l} s = \langle\rangle \wedge end(\aleph) \leqslant t \\ \vee \\ begin(s) \geqslant t \wedge S(s \dot{-} t, \aleph \dot{-} t) \end{array}}$$

No event may occur before time t, and the subsequent behaviours are simply the failures of process P translated through time t.

We have now presented all the rules required to verify a simple implementation of the protocol specified in section 4.

6 Implementing the Protocol

In section 4 we used Timed CSP to establish the correctness of a simple protocol: this result was dependent upon the correct behaviour of each component of the protocol. We now propose *TCSP* implementations of the components, and use the inference rules given in section 5 to demonstrate that they meet the appropriate specifications.

6.1 Implementation

The protocol consists of two components, transmitter P and receiver Q, communicating across two wires W_1 and W_2. The transmitter process should accept an input on channel *in*, and be prepared to transmit it along W_1, via channel *lm*. After this transmission has occurred, P waits for a confirmation event from wire W_2, on channel *rc*, before repeating this behaviour. Our intuition suggests the following as an implementation:

$$P \;\widehat{=}\; \mu X \bullet in \rightarrow lm \rightarrow rc \rightarrow X$$

We have yet to establish that this implements our requirements: that it meets the formal specification $SPEC_P$.

A similar set of conditions applies to the receiving process Q. It should be prepared to receive a signal from wire W_1, on channel *rm*, before offering output on channel *out*. It should then send a confirmation signal along wire W_2, on channel *lc*, before returning to its initial state. Our proposed solution:

$$Q \;\widehat{=}\; \mu Y \bullet rm \rightarrow out \rightarrow lc \rightarrow Y$$

Again, we will have to verify that this is an implementation of the specification $SPEC_Q$.

We could also model wires W_1 and W_2 in the $TCSP$ process algebra. Consider wire W_1: the propagation delay, the delay between input on channel lm and availability of output on channel rm, should be no more than one second. There will be a very small ($O(\delta)$) recovery time after output has occurred. In the context of [S88], it behaves as a stable one-place timed buffer:

$$\begin{aligned} W_1 &\mathrel{\hat{=}} \mu X \bullet lm \rightarrow WAIT\,(1-\delta)\,;\, rm \rightarrow X \\ W_2 &\mathrel{\hat{=}} \mu Y \bullet lc \rightarrow WAIT\,(1-\delta)\,;\, rc \rightarrow Y \end{aligned}$$

Note that the explicit delay between the occurrence of lm and the availability of rm is shortened by δ to allow for the delay introduced by the prefix operator: the time taken to recover from performing an event. Although we would not wish to implement wires in this fashion, the $TCSP$ description could be used to produce a software simulation of their behaviour.

6.2 Verification

We wish to show that the transmitting process P meets the specification placed upon it:

$$\mu X \bullet in \rightarrow lm \rightarrow rc \rightarrow X \quad \textbf{sat} \quad SPEC_P(s, \aleph)$$

This is a recursive process; the second recursion rule requires us to find a specification $S(s, \aleph)$ such that:

$$\begin{aligned} X \textbf{ sat } S(s,\aleph) &\Rightarrow in \rightarrow lm \rightarrow rc \rightarrow (WAIT\,\delta\,;\,X) \textbf{ sat } S(s,\aleph) \\ S(s,\aleph) &\Rightarrow SPEC_P(s,\aleph) \end{aligned}$$

Our strategy for finding such a specification would be to consider S to be $SPEC_P \wedge S'$, strengthening S' until the conjunction, which must still be satisfiable, is preserved by the recursive call. In this example, the specification $SPEC_P$ is strong enough to be preserved by the recursion, and no other conditions are required. We instantiate S with $SPEC_P$. We have then to show that:

$$X \textbf{ sat } SPEC_P(s,\aleph) \Rightarrow in \rightarrow lm \rightarrow rc \rightarrow (WAIT\,\delta\,;\,X) \textbf{ sat } SPEC_P(s,\aleph)$$

Assume that X **sat** $SPEC_P(s,\aleph)$. We wish to establish that:

$$in \rightarrow lm \rightarrow rc \rightarrow (WAIT\,\delta\,;\,X) \quad \textbf{sat} \quad SPEC_P(s,\aleph)$$

Applying the prefix rule three times transforms this proof obligation to the following requirement: we must find a specification $U(s, \aleph)$ such that:

$$
\begin{array}{l}
\mathit{WAIT}\ \delta\,; X \ \mathsf{sat}\ U(s, \aleph) \\
\left.\begin{array}{l}
s = \langle\rangle \wedge in \notin \alpha(\aleph) \\
\vee \\
s = \langle(t, in)\rangle^\frown s' \wedge in \notin \alpha(\aleph \upharpoonright t_1) \wedge \\
\quad s' \dot{-} (t_1 + \delta) = \langle\rangle \wedge lm \notin \alpha(\aleph \dot{-} (t_1 + \delta)) \\
\quad \vee \\
\quad s' \dot{-} (t_1 + \delta) = \langle(t_2, lm)\rangle^\frown s'' \wedge lm \notin \alpha(\aleph \dot{-} (t_1 + \delta) \upharpoonright t_2) \wedge \\
\qquad s'' \dot{-} (t_2 + \delta) = \langle\rangle \wedge rc \notin \alpha(\aleph \dot{-} (t_1 + t_2 + 2\delta)) \\
\qquad \vee \\
\qquad s'' \dot{-} (t_2 + \delta) = \langle(t_3, rc)\rangle^\frown s''' \wedge \\
\qquad\quad rc \notin \alpha(\aleph \dot{-} (t_1 + t_2 + 2\delta) \upharpoonright t_3) \wedge \\
\qquad\quad U(s''' \dot{-} (t_1 + t_2 + t_3 + 3\delta), \aleph \dot{-} (t_1 + t_2 + t_3 + 3\delta))
\end{array}\right\} \Rightarrow SPEC_P(s, \aleph)
\end{array}
$$

With a suitable choice of τ_1, τ_2, τ_3, this can be transformed to:

$$
\begin{array}{l}
\mathit{WAIT}\ \delta\,; X \ \mathsf{sat}\ U(s, \aleph) \\
\left.\begin{array}{l}
s = \langle\rangle \wedge in \notin \alpha(\aleph) \\
\vee \\
s = \langle(\tau_1, in)\rangle \wedge \in \notin \alpha(\aleph \upharpoonright \tau_1) \wedge lm \notin \alpha(\aleph \upharpoonleft \tau_1 + \delta) \\
\vee \\
s = \langle(\tau_1, in), (\tau_2, lm)\rangle \wedge in \notin \alpha(\aleph \upharpoonright \tau_1) \\
\qquad\qquad\qquad\qquad\quad \wedge\ lm \notin \alpha(\aleph \uparrow [\tau_1 + \delta, \tau_2)) \\
\qquad\qquad\qquad\qquad\quad \wedge\ rc \notin \alpha(\aleph \upharpoonleft \tau_2 + \delta) \\
\vee \\
s = \langle(\tau_1, in), (\tau_2, lm), (\tau_3, rc)\rangle^\frown u \wedge in \notin \alpha(\aleph \upharpoonright \tau_1) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \wedge\ lm \notin \alpha(\aleph \uparrow [\tau_1 + \delta, \tau_2) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \wedge\ rc \notin \alpha(\aleph \uparrow [\tau_2 + \delta, \tau_3)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \wedge\ U(u \dot{-} (\tau_3 + \delta), \aleph \dot{-} (\tau_3 + \delta))
\end{array}\right\} \Rightarrow SPEC_P(s, \aleph)
\end{array}
$$

Applying the second form of the delay rule, we can instautiate U as follows:

$$U(s, \aleph) \ \equiv\ SPEC_P(s \dot{-} \delta, \aleph \dot{-} \delta) \wedge begin(s) \geqslant \delta$$

Having discharged the first proof obligation, the proof can be completed with a simple case analysis on trace s. This becomes clear when we recall the form of specification $SPEC_P$:

$$\begin{array}{rcl} SPEC_P & \triangleq & tstrip(s) \leqslant \langle in, lm, rc\rangle^* \wedge \\ & & last(s) = in \Rightarrow lm \notin \alpha(\aleph \upharpoonright (end(s) + 2\delta)) \wedge \\ & & last(s) = lm \Rightarrow rc \notin \alpha(\aleph \upharpoonright (end(s) + 2\delta)) \wedge \\ & & last(s) = rc \Rightarrow in \notin \alpha(\aleph \upharpoonright (end(s) + 2\delta)) \wedge \\ & & s = \langle\rangle \Rightarrow in \notin \alpha(\aleph) \end{array}$$

The only non-trivial case corresponds to $s = \langle(\tau_1, in), (\tau_2, lm), (\tau_3, rc)\rangle^\frown u$. Here we require two arguments, one for each of the cases: $u = \langle\rangle$, $u \neq \langle\rangle$. Expanding the specification $SPEC_P$ makes the solution obvious.

This completes the verification of our transmitter process P. It will not be necessary to perform a similar proof for the receiver Q; we can exploit the symmetry present in our descriptions.

6.3 Renaming

The operator f in *TCSP* allows ns to relabel the events performed by a process. In the case of injective functions, this allows us to re-use a process description. By renaming events, we can transform processes while retaining their structure. The relationships between different events are maintained: given that a particular result holds for all the behaviours of a process, we can infer a corresponding result about the behaviours of the image of that process under such a transformation:

$$\frac{\begin{array}{l} P \textbf{ sat } S_1(s, \aleph) \\ S_1(s, \aleph) \Rightarrow S(f(s), f(\aleph)) \end{array}}{f(P) \textbf{ sat } S(s, \aleph)}$$

For example, we can use the result of the previous section to establish that Q **sat** $SPEC_Q$, by defining injective function f such that:

$$\begin{array}{rcl} f(in) & \triangleq & rm \\ f(lm) & \triangleq & out \\ f(rc) & \triangleq & lc \end{array}$$

We then observe that:

$$\begin{array}{rcl} SPEC_P(s, \aleph) & = & SPEC_Q(f(s), f(\aleph)) \\ Q & = & f(P) \end{array}$$

The inference rule allows us to conclude that:

$$Q \quad \textbf{sat} \quad SPEC_Q(s, \aleph)$$

Which completes our verification of the protocol.

This method of re-using implementation/specification pairs helps to eliminate redundant verifications: by observing and exploiting symmetry, we can re-use process components and their specifications.

7 Completing the Picture

The laws presented above, together with the others in the appendix, are complete with respect to the semantics: any specification provable from the semantics is provable using these laws. This becomes clear when we consider the *strongest specification* of a process.

7.1 Strongest Specifications

The identification of a process with the strongest specification that it can satisfy has been discussed before. It provides an alternative method for eliminating the process algebra from our proof obligations. The inference rules presented in this paper are more flexible in this: our specification may reflect only one of the properties of the system. Using our intuition, we need consider only the relevant properties of each component: those necessary to establish that the system meets the specification. As an example, consider the law:

$$\frac{\begin{array}{l} P_1 \textbf{ sat } S_1(s, \aleph) \\ P_2 \textbf{ sat } S_2(s, \aleph) \\ S_1(s, \aleph_1) \wedge S_2(s, \aleph_2) \Rightarrow S(s, \aleph_1 \cup \aleph_2) \end{array}}{P_1 \parallel P_2 \textbf{ sat } S(s, \aleph)}$$

For $P_1 \parallel P_2$ to meet specification S, we require that P_1 and P_2 meet specifications S_1 and S_2 respectively. These need only be strong enough to fulfil the third antecedent of the rule.

If we lack this intuition, we can use the strongest specifications of P_1 and P_2 as instantiations for S_1 and S_2. If suitable instantiations exist, they can be no stronger than these: any property of a process is a logical consequence of its strongest specification. We write $SS[\![P]\!]$ to denote the strongest specification of process P. For example, the strongest specification of deadlock is given by:

$$SS[\![STOP]\!](s, \aleph) \equiv s = \langle\rangle$$

This is all we can possibly know about the behaviours of $STOP$, we can draw no conclusions about the refusal set: $STOP$ may refuse any event at any time.

For a compound process, the strongest specification is defined in terms of the strongest specifications of its proper syntactic subcomponents:

$$\begin{array}{rcl} \mathcal{SS}[\![a \to P]\!](s,\aleph) & \equiv & s = \langle\rangle \wedge a \notin \alpha(\aleph) \\ & & \vee \\ & & \exists\, s', t \bullet (s = \langle(t,a)\rangle^\frown s') \wedge a \notin \alpha(\aleph \upharpoonright t) \wedge \\ & & \qquad \mathcal{SS}[\![P]\!](s', \aleph \dot{-} (t+\delta)) \end{array}$$

These definitions are equivalent to the semantic equations for the model TM_F. The equivalence

$$\mathcal{SS}[\![P]\!](s,\aleph) \;\equiv\; (s,\aleph) \in \mathcal{F}_T[\![P]\!]$$

can be established by structural induction upon process P.

Strongest specifications may he used to reduce the proof requirement on a compound process to a predicate on traces and refusals, similar to the one at the end of 4.2. The inference rules given in this paper may provide a much simpler predicate; we can discard unnecessary information. But strongest specifications provide a more *mechanical* method; there are no choices to be made, even in the case of recursion.

$$\begin{array}{rcl} \mathcal{SS}[\![\mu X \bullet F(X)]\!](s,\aleph) & \equiv & \bigwedge_{i\in\mathbb{N}}(end(s,\aleph) < \delta i \Rightarrow \mathcal{SS}[\![\widehat{F}^i(STOP)]\!](s,\aleph)) \\ & & \text{where } \widehat{F}(X) = WAIT\,\delta\,;F(X) \end{array}$$

We consider the recursive process $\mu X \bullet F(X)$ to be the limit of the finite approximations $\widehat{F}^i(STOP)$. A given behaviour of the recursive process must be a behaviour of all the finite approximations involving a sufficient number of recursions. If the behaviour in question is described by the failure $(s,\aleph)$, then all of the approximations $\widehat{F}^i(STOP)$, where $i > end(s,\aleph)/\delta$, must also exhibit that behaviour.

Hence the strongest specification of $\mu X \bullet F(X)$ can be written as the conjunction of the strongest specifications of its finite approximations, guarded by an applicability condition $end(s,\aleph) < \delta i$. We are spared the task of finding a sufficient specification that will be preserved by each recursive call.

Strongest specifications provide a complete description of the possible behaviours of a process. To decide whether a component is adequate for use in a given situation, we can use the inference rules in this paper to confirm that it meets the requirements. If a component is to be re-used in different systems, then it should be supplied with its strongest specification. The comprehensive nature of strongest specifications also allows us to demonstrate that the inference rules presented in this paper are *complete* with respect to the semantics.

7.2 Completeness

The inference rules presented in this paper are easily seen to be *sound*; example proofs are presented in appendix B. If we can use the rules to show that process P satisfies a specification $S(s, \aleph)$ then predicate $S(s, \aleph)$ must hold for all behaviours of P: it must be true of all the elements of the set of failures corresponding to P in the semantic model TM_F.

These rules also form a *complete* set. If a predicate $S(s, \aleph)$ holds for all behaviours of P, then we can use the rules to establish that P **sat** $S(s, \aleph)$. We can demonstrate this by showing that the rules preserve strongest specifications: they yield the strongest specification of a compound process in terms of the strongest specifications of its components. For example, consider the case of the parallel operator.

Suppose that the parallel combination $P_1 \parallel P_2$ meets the specification $S(s, \aleph)$. In our proof, we would employ the following inference rule:

$$\frac{\begin{array}{l} P_1 \textbf{ sat } S_1(s, \aleph) \\ P_2 \textbf{ sat } S_2(s, \aleph) \\ S_1(s, \aleph_1) \wedge S_2(s, \aleph_2) \Rightarrow S(s, \aleph_1 \cup \aleph_2) \end{array}}{P_1 \parallel P_2 \textbf{ sat } S(s, \aleph)}$$

This requires that we exhibit specifications S_1 and S_2 for which the three antecedents of the rule hold. The first two antecedents insist that these are no stronger than the corresponding strongest specifications, so if the third is also to hold, it must hold with the following instantiation:

$$\mathcal{SS}[\![P_1]\!](s, \aleph_1) \wedge \mathcal{SS}[\![P_2]\!](s, \aleph_2) \quad \Rightarrow \quad S(s, \aleph_1 \cup \aleph_2)$$

However, as $S(s, \aleph)$ is true of all behaviours of $P_1 \parallel P_2$, it can be no stronger than the strongest specification of that process, i.e.

$$\mathcal{SS}[\![P_1 \parallel P_2]\!](s, \aleph) \quad \Rightarrow \quad S(s, \aleph)$$

But the strongest specification is given by:

$$\mathcal{SS}[\![P_1 \parallel P_2]\!](s, \aleph) \quad \equiv \quad \exists \aleph_1, \aleph_2 \bullet \mathcal{SS}[\![P_1]\!](s, \aleph_1) \wedge \mathcal{SS}[\![P_2]\!](s, \aleph_2) \wedge \aleph = \aleph_1 \cup \aleph_2$$

So the inference rule is sufficient to establish that $P_1 \parallel P_2$ **sat** $S(s, \aleph)$. The same is true for the other operators, and we have shown that the equivalences that define our strongest specifications are no weaker than the semantic equations: we lose no information. Hence our inference rules form a complete set with respect to the semantics.

8 Stability

In this paper, we have been working within the Timed Failures model of TCSP, TM_F. Timed CSP identifies a further aspect of a process's behaviour: the *stability* value corresponding to each (trace, refusal) pair. In [Re88], this is defined to be the earliest time at which it can be guaranteed that the process can make no further internal progress. This notion has been refined by Blamey in [Bl89]. Here, he associates with each (trace,refusal) pair an "instability" set rather than a stability value: the set of times at which the process might not be stable.

One advantage of this approach is that it allows us to extend the work in this paper to models which include stability. Using instability sets, we can express the behaviour of a compound process in terms of the behaviours of its components. This is not possible in the original stability models, TM_S and TM_{FS}: to see why, consider the processes defined below.

$$\begin{aligned} P_1 &\mathrel{\widehat{=}} a \rightarrow STOP \\ P_2 &\mathrel{\widehat{=}} a \rightarrow WAIT\,1\,; STOP \end{aligned}$$

The stabilities associated with this process are given by:

$$\begin{aligned} \mathcal{S}_T[\![P_1]\!] &\equiv \{(\langle\rangle, 0)\} \cup \{(\langle(t,a)\rangle, t+\delta) \mid t \geqslant 0\} \\ \mathcal{S}_T[\![P_2]\!] &\equiv \{(\langle\rangle, 0)\} \cup \{(\langle(t,a)\rangle, t+1+\delta) \mid t \geqslant 0\} \end{aligned}$$

Now consider the behaviours of the process $P_1 \,|||\, P_2$, given the semantic equation for the interleaving operator:

$$\begin{aligned} \mathcal{S}_T[\![P_1 \,|||\, P_2]\!] \mathrel{\widehat{=}} & \; SUP\{(s, max\{\alpha_1, \alpha_2\}) \mid \\ & \quad \exists (s_1, \alpha_1) \in \mathcal{S}_T[\![P_1]\!], (s_2, \alpha_2) \in \mathcal{S}_T[\![P_2]\!] \bullet s \in Tmerge(s_1, s_2)\} \end{aligned}$$

The compound process can engage in a single a event, from each of its components, and give rise to a stability value that cannot be inferred from the properties of a typical behaviour of either process acting independently. The trace $\langle(0,a)\rangle$ has a stability value of $1+\delta$: this can only be deduced by considering *all* of the stability values associated with that trace.

However, if we identify instability sets rather than stability values, no such difficulties arise. The properties of a typical instability set of a compound process behaviour can be deduced from the properties of arbitrary behaviours of the component processes. As with the timed failures model, we can restrict our attention to a typical element of the semantics. We can thus formulate a set of inference rules for reasoning about specifications involving stability conditions. As an example, we can derive the following rule for the nondeterministic choice operator:

$$\frac{\begin{array}{l} P_1 \ \mathbf{sat}\ S_1(s,\gamma,\aleph) \\ P_2 \ \mathbf{sat}\ S_2(s,\gamma,\aleph) \\ (S_1(s,\gamma,\aleph) \vee S_2(s,\gamma,\aleph)) \Rightarrow S(s,\gamma,\aleph) \end{array}}{P_1 \sqcap P_2 \ \mathbf{sat}\ S(s,\gamma,\aleph)}$$

In the above specifications γ represents an arbitrary instability value, and the **sat** operator is extended in the obvious way. The rule illustrates that an instability value of $P_1 \sqcap P_2$ must be an instability value for one of the components P_1, P_2. The converse is also true; this is not the case in TM_{FS}, in which an arbitrary behaviour requires more information. Similar results are obtained for the other *TCSP* operators.

9 Conclusions

In this paper, we have shown how we can factor out the complexity inherent in reasoning about timed distributed systems. We introduced behavioural specifications, capturing correctness conditions as simple predicates on a typical element of the semantics. We have given inference rules, derived from the semantic mappings, for reasoning about these specifications. These rules allow us to reduce proof obligations on a composite Timed CSP process to requirements on the syntactic subcomponents.

The lack of sufficient algebraic laws means that we cannot construct a proof system for Timed CSP similar to the one developed in [Br83], but we can produce a complete set of inference rules for proofs of correctness. Further, we have presented the rules in such a form as to make their application completely mechanical: an automated proof assistant could be developed similar to the one employed in [D87].

As an illustration of the use of the rules, we have presented a verification of a simple flow control protocol, whose definition involved both abstraction and concurrency. The correctness of this example depends upon the subtle treatment of hiding in Timed CSP: any hidden events are forced to occur as soon as they become available. An implementation of the protocol was proposed and verified; this required a useful result about the properties of recursive processes.

We have exhibited strongest specifications for Timed CSP processes and used these to verify that our rules form a complete set with respect to the semantics. Our intention is to work towards a specification-oriented semantics for Timed CSP, similar to the one described in [OH83], using the enhanced timed failures-stability model and the hierarchy of lower models. This will allow us to work towards a powerful specification and development methodology for real-time concurrent systems.

Acknowledgements

The authors would like to thank Bill Roscoe and Mike Reed for their advice and encouragement; Stephen Blamey, Steve Brookes, Jim Woodcock, and Mike Goldsmith for their suggestions; Alice King-Farlow and Elizabeth Schneider for further inspiration. We are also indebted to our colleagues in the Programming Research Group for their friendship. This work was snpported by grants from SERC and BP.

References

[Bl89] S. R. Blamey, *TCSP Processes as Predicates,* (to appear) Oxford 1989.

[Br83] S. D. Brookes, *A Model for Communicating Sequential Processes,* Oxford University D.Phil thesis 1983.

[D87] J. W. Davies, *Assisted Proofs for Communicating Sequential Processes,* Oxford University M.Sc. thesis 1987.

[H85] C. A. R. Hoare, *Communicating Sequential Processes,* Prentice-Hall International 1985.

[OH83] E.R. Olderog and C.A.R. Hoare, *Specification-oriented Semantics for Communicating Processes* Springer LNCS **154** 1983, **561–572**. (Also, Acta Informatica 23 1986, 9–66.).

[PS88] K. Paliwoda and J.W. Sanders, *The Sliding-Window Protocol in CSP,* Oxford University Programming Research Group Technical Monograph 1988, **66**.

[Re88] G. M. Reed, *A Uniform Mathematical Theory for Real-time Distributed Computing,* Oxford University D.Phil thesis 1988.

[RR86] G. M. Reed and A.W. Roscoe, *A Timed Model for Communicating Sequential Processes* Proceedings of ICALP'86, Springer LNCS 226 (1986), **314–323**; Theoretical Computer Science **58** 1988, **249–261**.

[RR87] G. M. Reed and A.W. Roscoe, *Metric Spaces as Models for Real-time Concurrency* Proceedings of the Third Workshop on the Mathematical Foundations of Programming Language Semantics, LNCS **298** 1987, **331–343**.

[Ro82] A.W. Roscoe *A Mathematical Theory of Communicating Processes* Oxford University D.Phil thesis 1982.

[S88] S.A. Schneider *Communication in Timed Distributed Computing* Oxford University M.Sc. thesis 1988.

A Inference Rules

In this appendix, we present a complete set of inference rules for behavioural specifications. A rule is presented for each TCSP operator.

Rule *STOP*

$$\frac{}{\mathit{STOP}\ \mathbf{sat}\ (s = \langle\rangle)}$$

Rule $\perp$

$$\frac{}{\perp\ \mathbf{sat}\ (s = \langle\rangle)}$$

Rule *SKIP*

$$\frac{}{\begin{array}{ll} \mathit{SKIP}\ \mathbf{sat} & (s = \langle\rangle \wedge \checkmark \notin \alpha(\aleph)) \\ & \vee \\ & (s = \langle(t, \checkmark)\rangle \wedge \checkmark \notin \alpha(\aleph \upharpoonright t) \wedge t \geqslant 0) \end{array}}$$

Rule *WAIT t*

$$\frac{}{\begin{array}{ll} \mathit{WAIT}\ t\ \mathbf{sat} & s = \langle\rangle \wedge \checkmark \notin \alpha(\aleph \upharpoonleft t) \\ & \vee \\ & s = \langle(t', \checkmark)\rangle \wedge t' \geqslant t \wedge \checkmark \notin \alpha(\aleph \uparrow [t, t')) \end{array}}$$

The following rules apply to compound processes. When a process variable is present, it is more convenient to match proof obligations to consequents: the form in which the rules are presented makes this possible.

Rule $a \rightarrow P$

$$\frac{\begin{array}{l} P\ \mathbf{sat}\ T(s, \aleph) \\ \left.\begin{array}{l} s = \langle\rangle \wedge a \notin \alpha(\aleph) \\ \vee \\ s = \langle(t, a)\rangle^\frown s' \wedge a \notin \alpha(\aleph \upharpoonright t) \wedge T(s' \mathbin{\dot{-}} (t + \delta), (\aleph \mathbin{\dot{-}} (t + \delta)) \end{array}\right\} \Rightarrow S(s, \aleph) \end{array}}{(a \rightarrow P)\ \mathbf{sat}\ S(s, \aleph)}$$

Rule $P_1 \Box P_2$

$$\begin{array}{l}
P_1 \textbf{ sat } S_1(s,\aleph) \\
P_2 \textbf{ sat } S_2(s,\aleph) \\
\left.\begin{array}{l}
(S_1(s,\aleph) \lor S_2(s,\aleph) \\
\land \\
S_1(\langle\rangle, \aleph \restriction begin(s)) \land S_2(\langle\rangle, \aleph \restriction begin(s))
\end{array}\right\} \Rightarrow S(s,\aleph) \\
\hline
P_1 \Box P_2 \textbf{ sat } S(s,\aleph)
\end{array}$$

Rule $a : A \rightarrow P_a$

$$\begin{array}{l}
\forall a \in A \bullet P_a \textbf{ sat } S_a(s,\aleph) \\
\left.\begin{array}{l}
(\aleph \cap ([0, begin(s)) \times A) = \emptyset \\
\land \\
\forall a \in A \bullet (s = \langle(t,a)\rangle^\frown s') \Rightarrow S_a(s' \dot{-} (t+\delta), \aleph \dot{-} (t+\delta))
\end{array}\right\} \Rightarrow S(s,\aleph) \\
\hline
a : A \rightarrow P_a \textbf{ sat } S(s,\aleph)
\end{array}$$

Rule $P_1 \sqcap P_2$

$$\begin{array}{l}
P_1 \textbf{ sat } S_1(s,\aleph) \\
P_2 \textbf{ sat } S_2(s,\aleph) \\
S_1(s,\aleph) \lor S_2(s,\aleph) \Rightarrow S(s,\aleph) \\
\hline
P_1 \sqcap P_2 \textbf{ sat } S(s,\aleph)
\end{array}$$

Rule $P_1 \parallel P_2$

$$\begin{array}{l}
P_1 \textbf{ sat } S_1(s,\aleph) \\
P_2 \textbf{ sat } S_2(s,\aleph) \\
S_1(s,\aleph_1) \land S_2(s,\aleph_2) \Rightarrow S(s,\aleph_1 \cup \aleph_2) \\
\hline
P_1 \parallel P_2 \textbf{ sat } S(s,\aleph)
\end{array}$$

Rule $P_1 \; {}_X\|_Y \; P_2$

$$\begin{array}{l}
P_1 \textbf{ sat } S_1(s,\aleph) \\
P_2 \textbf{ sat } S_2(s,\aleph) \\
\left.\begin{array}{l}
(\alpha(s_1,\aleph_1) \subseteq X \land \alpha(s_2,\aleph_2) \subseteq Y \\
\land \\
\land\, \alpha(\aleph_3) \subseteq \Sigma - (X \cup Y) \\
\land \\
S_1(s_1,\aleph_1) \land S_2(s_2,\aleph_2) \land s_3 \in s_1 \; {}_X\|_Y \; s_2)
\end{array}\right\} \Rightarrow S(s_3, \aleph_1 \cup \aleph_2 \cup \aleph_3) \\
\hline
P_1 \; {}_X\|_Y \; P_2 \textbf{ sat } S(s,\aleph)
\end{array}$$

Rule $P_1 \,|||\, P_2$

$$\frac{\begin{array}{l} P_1 \textbf{ sat } S_1(s,\aleph) \\ P_2 \textbf{ sat } S_2(s,\aleph) \\ (s \in \mathit{Tmerge}(u,v) \wedge S_1(u,\aleph) \wedge S_2(v,\aleph)) \Rightarrow S(s,\aleph) \end{array}}{P_1 \,|||\, P_2 \textbf{ sat } S(s,\aleph)}$$

Rule $P_1\,;P_2$

$$\frac{\begin{array}{l} P_1 \textbf{ sat } S_1(s,\aleph) \\ P_2 \textbf{ sat } S_2(s,\aleph) \\ (\checkmark \notin \alpha(s) \;\wedge \forall I \in \mathit{TINT} \bullet S_1(s, \aleph \cup (I \times \{\checkmark\}))) \Rightarrow S(s,\aleph) \\ \left.\begin{array}{l} s \cong s_1{}^\frown(s_2 + t) \wedge \checkmark \notin \alpha(s_1) \wedge \mathit{end}(\aleph_1) \leqslant t \\ \wedge \\ S_1(s_1{}^\frown\langle(t,\checkmark)\rangle, \aleph_1 \cup ([0,t) \times \{\checkmark\})) \wedge S_2(s_2,\aleph_2) \end{array}\right\} \Rightarrow S(s, \aleph_1 \cup (\aleph_2 + t)) \end{array}}{(P_1\,;P_2) \textbf{ sat } S(s,\aleph)}$$

Rule $P \setminus A$

$$\frac{P \textbf{ sat } \widehat{A}(s,\aleph) \wedge \alpha(\aleph') \subseteq A \Rightarrow S(s \setminus A, \aleph - \aleph')}{P \setminus A \textbf{ sat } S(s,\aleph)}$$

Rule $f^{-1}(P)$

$$\frac{\begin{array}{l} P \textbf{ sat } S_1(s,\aleph) \\ S_1(f(s), f(\aleph)) \Rightarrow S(s,\aleph) \end{array}}{f^{-1}(P) \textbf{ sat } S(s,\aleph)}$$

Rule $f(P)$

$$\frac{\begin{array}{l} P \textbf{ sat } S_1(s,\aleph) \\ S_1(s, f^{-1}(\aleph)) \Rightarrow S(f(s),\aleph) \end{array}}{f(P) \textbf{ sat } S(s,\aleph)}$$

Rule $\mu X \bullet F(X)$

$$\frac{\forall X \bullet X \textbf{ sat } S(s,\aleph) \Rightarrow F(\mathit{WAIT}\,\delta\,; X) \textbf{ sat } S(s,\aleph)}{\mu X \bullet F(X) \textbf{ sat } S(s,\aleph)}$$

B Example Proofs

In this appendix we present a proof of soundness for the prefixing rule. We then extend the semantic function $\mathcal{F}_T$ to permit a proof of the second recursion rule given in section 5. We verify that the proposed extension is consistent with the original formulation, and provide a simple proof of the hiding rule.

B.1 Prefixing

Rule

$$\frac{P \textbf{ sat } T(s,\aleph) \qquad \left.\begin{array}{l} s = \langle\rangle \wedge a \notin \alpha(\aleph) \\ \vee \\ s = \langle(t,a)\rangle^\frown s' \wedge a \notin \alpha(\aleph \upharpoonright t) \wedge T(s' \dot{-} (t+\delta), \aleph \dot{-} (t+\delta)) \end{array}\right\} \Rightarrow S(s,\aleph)}{(a \to P) \textbf{ sat } S(s,\aleph)}$$

Semantics

$$\begin{array}{rcl} \mathcal{F}_T[\![a \to P]\!] & \widehat{=} & \{(\langle\rangle,\aleph) \mid a \notin \alpha(\aleph)\} \\ & & \cup \\ & & \{(\langle(t,a)\rangle^\frown(s + (t+\delta)), \aleph_1 \cup \aleph_2 \cup (\aleph_3 + (t+\delta))) \mid \\ & & \quad t \geqslant 0 \wedge (I(\aleph_1) \subseteq [0,t) \wedge a \notin \alpha(\aleph_1)) \\ & & \quad \wedge\, I(\aleph_2) \subseteq [t, t+\delta) \wedge (s,\aleph_3) \in \mathcal{F}_T[\![P]\!]\} \end{array}$$

Proof

$$P \textbf{ sat } T(s,\aleph)$$

$$\begin{array}{rl} (s,\aleph) \in \mathcal{F}_T[\![a \to P]\!] \Rightarrow & s = \langle\rangle \wedge a \notin \alpha(\aleph) \\ & \vee \\ & \exists\, \aleph_1, \aleph_2, \aleph_3, s' \bullet s = \langle(t,a)\rangle^\frown(s' + t + \delta) \\ & \quad \wedge\, \aleph = \aleph_1 \cup \aleph_2 \cup (\aleph_3 + t + \delta) \wedge t \geqslant 0 \\ & \quad \wedge\, I(\aleph_1) \subseteq [0,t) \wedge a \notin \alpha(\aleph_1) \\ & \quad \wedge\, I(\aleph_2) \subseteq [t, t+\delta) \wedge (s', \aleph_3) \in \mathcal{F}_T[\![P]\!] \end{array}$$

$$\begin{array}{llll} \vdash & \forall (s, \aleph) \in \mathcal{F}_T[\![a \rightarrow P]\!] & \bullet & s = \langle\rangle \wedge a \notin \alpha(\aleph) \\ & & & \vee \\ & & & s = \langle (t, a)\rangle ^\frown (s' + t + \delta) \wedge t' \geqslant 0 \wedge a \notin \alpha(\aleph \upharpoonright t) \\ & & & \qquad \wedge (s', \aleph \dot{-} (t + \delta)) \in \mathcal{F}_T[\![P]\!] \end{array}$$

$$\begin{array}{lrl} \vdash & a \rightarrow P \textbf{ sat} & s = \langle\rangle \wedge a \notin \alpha(\aleph) \\ & & \vee \\ & & s = \langle (t, a)\rangle ^\frown (s' + t + \delta) \wedge t' \geqslant 0 \wedge a \notin \alpha(\aleph \upharpoonright t) \\ & & \qquad \wedge T(s', \aleph \dot{-} (t + \delta)) \end{array}$$

The inference rule for prefixing follows immediately, by a simple property of the **sat** operator (see the third inference rule given in section 2). We conclude that the rule rests soundly upon the semantics.

B.2 The Semantic Function $\overline{\mathcal{F}}_T$

As mentioned in section 5, we obtain a more powerful rule for reasoning about the behaviour of recursive processes if we extend the semantic function $\mathcal{F}_T$. First, we must define the type of failure sets, $\overline{TF}$:

$$\overline{TF} \;\widehat{=}\; \mathbb{P}(T\Sigma^*_\leqslant \times RSET)$$

where $T\Sigma^*_\leqslant$ and $RSET$ are as defined in [Re88]. We then extend the syntax of Timed CSP:

$$TCSP^+ \;::=\; TCSP \mid X_E$$

where E ranges over the whole of $\overline{TF}$. Finally, we extend the semantic function $\mathcal{F}_T$ in the following fashion:

$$\begin{array}{rcl} \overline{\mathcal{F}}_T[\![X_E]\!] & \widehat{=} & E \\ \overline{\mathcal{F}}_T[\![P \setminus A]\!] & \widehat{=} & \{(s \setminus A, \aleph - \aleph') \mid (s, \aleph) \in \overline{\mathcal{F}}_T[\![P]\!] \wedge \hat{A}(s, \aleph) \wedge \alpha(\aleph') \subseteq A\} \end{array}$$

The remaining clauses are entirely similar to the defining equations for $\mathcal{F}_T$. To show that the new semantic function is an extension of $\mathcal{F}_T$ we must demonstrate that the two functions agree on the intersection of their domains: *TCSP*. A simple structural induction will suffice: the only non-trivial case is that of the hiding operator. In this case, recalling the relevant semantic equations

$$\begin{array}{rcl} \mathcal{F}_T[\![P \setminus A]\!] & \widehat{=} & \{(s \setminus A, \aleph) \mid (s, \aleph \cup ([0, end(s, \aleph)) \times A)) \in \mathcal{F}_T[\![P]\!]\} \\ \overline{\mathcal{F}}_T[\![P \setminus A]\!] & \widehat{=} & \{(s \setminus A, \aleph - \aleph') \mid (s, \aleph) \in \overline{\mathcal{F}}_T[\![P]\!] \wedge \hat{A}(s, \aleph) \wedge \alpha(\aleph') \subseteq A\} \end{array}$$

and the definition

$$\hat{A}(s,\aleph) \;\widehat{=}\; ([0, end(s,\aleph)) \times A) \subseteq \aleph$$

we proceed as follows:

Assume that $\mathcal{F}_T[\![P]\!] = \overline{\mathcal{F}}_T[\![P]\!]$ and that P is a process.

$$(s,\aleph) \in \mathcal{F}_T[\![P \setminus A]\!]$$

$$\begin{array}{ll} \vdash & \exists\, s_1, \aleph_1, \aleph_2 \bullet s = s_1 \setminus A \wedge (s_1, \aleph \cup ([0, end(s_1,\aleph)) \times A)) \in \mathcal{F}_T[\![P]\!] \\ & \quad \wedge\ \aleph_1 = \aleph \cup ([0, end(s_1,\aleph)) \times A) \\ & \quad \wedge\ \aleph = \aleph_1 - \aleph_2 \wedge \alpha(\aleph_2) \subseteq A \end{array}$$

$$\begin{array}{ll} \vdash & \exists\, s_1, \aleph_1, \aleph_2 \bullet s = s_1 \setminus A \wedge \aleph = \aleph_1 - \aleph_2 \wedge (s_1, \aleph_1) \in \overline{\mathcal{F}}_T[\![P]\!] \\ & \quad \wedge\ [0, end(s_1,\aleph_1)) \times A \subseteq \aleph_1 \wedge \alpha(\aleph_2) \subseteq A \end{array}$$

since $end(s_1,\aleph) = end(s_1,\aleph_1)$

$$\vdash \quad \exists\, s_1, \aleph_1, \aleph_2 \bullet s = s_1 \setminus A \wedge \aleph = \aleph_1 - \aleph_2 \wedge (s_1 \setminus A, \aleph_1 - \aleph_2) \in \overline{\mathcal{F}}_T[\![P \setminus A]\!]$$

$$\vdash \quad (s,\aleph) \in \overline{\mathcal{F}}_T[\![P \setminus A]\!]$$

Conversely,

$$(s,\aleph) \in \overline{\mathcal{F}}_T[\![P \setminus A]\!]$$

$$\begin{array}{ll} \vdash & \exists\, s_1, \aleph_1, \aleph_2 \bullet s = s_1 \setminus A \wedge \aleph = \aleph_1 - \aleph_2 \wedge (s_1, \aleph_1) \in \overline{\mathcal{F}}_T[\![P]\!] \\ & \quad \wedge\ [0, end(s_1,\aleph_1)) \times A \subseteq \aleph_1 \wedge \alpha(\aleph_2) \subseteq A \end{array}$$

$$\begin{array}{ll} \vdash & \exists\, s_1, \aleph_1, \aleph_2 \bullet s = s_1 \setminus A \wedge (s_1, \aleph \cup ([0, end(s_1,\aleph_1)) \times A)) \in \mathcal{F}_T[\![P]\!] \\ & \quad \wedge\ \aleph_1 = \aleph \cup ([0, end(s_1,\aleph_1)) \times A) \\ & \quad \wedge\ \aleph = \aleph_1 - \aleph_2 \wedge \alpha(\aleph_2) \subseteq A \end{array}$$

by Axiom 6 of TM_F

$$\vdash \quad \exists\, s_1 \bullet s = s_1 \setminus A \wedge (s_1, \aleph \cup ([0, end(s_1,\aleph)) \times A)) \in \mathcal{F}_T[\![P]\!]$$

by Axiom 6 again, since $end(s_1,\aleph) \leqslant end(s_1,\aleph_1)$

$$\vdash \quad (s,\aleph) \in \mathcal{F}_T[\![P \setminus A]\!]$$

□

B.3 Hiding

Having verified that $\overline{\mathcal{F}}_T$ is an extension of $\mathcal{F}_T$, we can easily establish the soundness of the rule for the hiding operator:

$$\frac{P \text{ sat } \widehat{A}(s,\aleph) \wedge \alpha(\aleph') \subseteq A \Rightarrow S(s \setminus A, \aleph - \aleph')}{P \setminus A \text{ sat } S(s,\aleph)}$$

Given the semantic equation

$$\overline{\mathcal{F}}_T[\![P \setminus A]\!] \;\; \widehat{=} \;\; \{(s \setminus A, \aleph - \aleph') \mid (s,\aleph) \in \overline{\mathcal{F}}_T[\![P]\!] \wedge \widehat{A}(s,\aleph) \wedge \alpha(\aleph') \subseteq A\}$$

we proceed as follows:

$$P \text{ sat } \widehat{A}(s,\aleph) \wedge \alpha(\aleph') \subseteq A \Rightarrow S(s \setminus A, \aleph - \aleph')$$

$$\begin{aligned} (s,\aleph) \in \overline{\mathcal{F}}_T[\![P \setminus A]\!] \;\; \Rightarrow \;\; & \exists s_1, \aleph_1, \aleph_2 \bullet s = s_1 \setminus A \wedge \aleph = \aleph_1 - \aleph_2 \wedge \\ & \widehat{A}(s_1,\aleph_1) \wedge (s_1,\aleph_1) \in \overline{\mathcal{F}}_T[\![P]\!] \\ & \wedge \alpha(\aleph_2) \subseteq A \end{aligned}$$

$$\begin{aligned} \vdash \;\; \forall (s,\aleph) \in \overline{\mathcal{F}}_T[\![P \setminus A]\!] \;\; \bullet \;\; & \exists s_1, \aleph_1, \aleph_2 \bullet s = s_1 \setminus A \wedge \aleph = \aleph_1 - \aleph_2 \\ & \wedge S(s_1 \setminus A, \aleph_1 - \aleph_2) \end{aligned}$$

$$\vdash \qquad P \setminus A \text{ sat } S(s,\aleph)$$

B.4 Recursion

Finally, we establish the result that provides the motivation for the extension to the semantics: the second inference rule for recursion:

$$\frac{\forall X \bullet X \text{ sat } S(s,\aleph) \Rightarrow F(\mathit{WAIT}\,\delta\,; X) \text{ sat } S(s,\aleph)}{\mu X \bullet F(X) \text{ sat } S(s,\aleph)}$$

We begin by extending the topology on TM_F defined in [Re88] to $\overline{TF}$ in the obvious way: Reed's proof that all of the basic *TCSP* operators are non-expanding is independent of the axioms. That all basic $TCSP^+$ operators are non-expanding follows immediately.

If F is a function on $TCSP^+$ composed of basic operators, there is a corresponding function C_F defined on $\overline{TF}$ by:

$$C_F(E) \;\; \widehat{=} \;\; \overline{\mathcal{F}}_T[\![F(X_E)]\!]$$

From the above result, it follows that F is non-expanding, and that if any of the components of F are contracting, then so is F. The function $WAIT\,\delta\,;X$ is always contracting; if we define

$$\widehat{F}(X) \;\widehat{=}\; F(WAIT\,\delta\,;X)$$

then, for any F, the function $\widehat{F}$ will be contracting; the corresponding mapping on $\overline{TF}$, $C_{\widehat{F}}$, will be a contraction mapping on a metric space: it will have a unique fixed point. This fixed point is the semantics of $\mu\,X \bullet F(X)$.

If we consider the sequence $\{E_n\}$, where

$$E_n \;\widehat{=}\; C^n_{\widehat{F}}(\emptyset)$$

we observe that

$$\lim_{n\to\infty}(E_n) \;=\; \overline{\mathcal{F}}_T[\![\mu\,X \bullet F(X)]\!]$$

The antecedent of the recursion rule

$$\forall\,X \bullet X \;\textbf{sat}\; S(s,\aleph) \;\Rightarrow\; \widehat{F}(X)\;\textbf{sat}\; S(s,\aleph)$$

allows us to conclude that

$$\forall\,X,n \bullet X \;\textbf{sat}\; S(s,\aleph) \;\Rightarrow\; \widehat{F}^n(X)\;\textbf{sat}\; S(s,\aleph)$$

However, it is easy to show that $\forall\,S \bullet X$ **sat** $S(s,\aleph)$, and so

$$\forall\,n \;\bullet\; \widehat{F}^n(X)\;\textbf{sat}\; S(s,\aleph)$$

and it can be shown that all predicates of the form **sat** $S(s,\aleph)$ correspond to closed predicates in $\overline{TF}$: if such a predicate holds of all the elements of a sequence, it must hold of the limit. Hence

$$\mu\,X \bullet F(X) \quad \textbf{sat} \quad S(s,\aleph)$$

Hence the recursion rule is sound with respect to the new semantics. □