SCS&E Report 9309
July, 1993

Using CSP+T to Describe a Timing Constrained Stop-and-Wait Protocol

John J. Zic

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING
THE UNIVERSITY OF NEW SOUTH WALES

Abstract

This paper presents a novel description of a time-constrained stop and wait protocol using an extended CSP model. The timing constraints examined include the usual message transit delay, as well as message input rate limitations and message timeouts.

The extended CSP model used for this example is based on associating finite time intervals with each event the process engages. These time intervals in turn are functions over a set of markers events.

1 Introduction

A complex system, whether it be a chemical plant control system, a fly-by-wire aircraft control system, a data communications suite, or a piece of medical electronics needs to be carefully and methodically developed. Rigourous design, specification and implementation techniques should be used in order to ensure that the product finally delivered is functionally correct for its given specification, and that the specification itself captures accurately the understanding of the product design.

To achieve this rigour requires the use of some of the many types of Formal Description Techniques, such as LOTOS [2], VDM [8], Z [20], Petri Nets [13], Estelle [7], the observational congruence model of CCS [11, 12] and the failures model of CSP [6]. Typically these Formal Description Techniques (or FDTs) specify system behaviour by using mathematical systems of Logic, Set Theory, or Process Algebras with rules enabling the correct construction of a system. Furthermore, a FDT also includes construction and reduction rules allowing the system to be broken into subcomponents.

These specifications (expressed in a FDT) are deliberately abstract from any particular implementation. This gives the implementor freedom to choose between varying implementations according to their own knowledge and skills.

The FDTs mentioned all deal only with functionality, ignoring performance. In particular, time is not discussed. There are many examples in communication systems software where speed is essential, and so must be mentioned in a specification. The requirements include maximum processing delay for any particular message, minimum spacing between successive transmissions to prevent flooding, and even conditional demands (e.g. "if packets are received too quickly, send a choke"). Specifying a system's temporal characteristics requires extensions to the above mentioned FDTs. Amongst the earliest proposed extensions were to the Petri Nets to develop the Timed Transition Petri Nets [14] and (less commonly) Timed Place Petri Nets [4, 19]. More recently, the Z specification language has been extended to allow the use of continuous real functions to model timed histories [10], and Hoare's CSP has been extended (by several workers [15, 3, 17, 9]) to allow real-time system specifications.

This paper presents another set of extensions to CSP, which allow succinct real-time specifications often displaying similar structures to the untimed specifications. The motivation for these extensions was that it was found that other Timed CSP models could describe some simple systems only with difficulty, and the resulting process description seemed unnatural [22]. For example, a (multi-place) buffer which has differing input and output rates, and provides each message with a specific delay could not be easily expressed in any existing technique. This was a result of each model being able to specify only delays between any two successive events within a sequential process. In order to allow description of the above system, the specification technique must be able to specify delays between arbitrary events in the process execution. As will be demonstrated, the proposed description technique (called CSP+T) achieves this latter requirement.

The paper is divided as follows. First a brief introduction to the CSP+T is presented, including a discussion on the algebraic extensions as well as the traces model. Next, an algebraic description is developed. Following this, the relevant timing constraints are described and formalised in some detail. These timing constraints are developed independently of the algebra and hence any particular algebraic implementation. A proof outline follows, which, when a full trace semantics for the algebra is developed, will be able to demonstrate the correctness of the implementation and the component specifications. The paper concludes

with some issues which are concerns for current and future work.

2 A brief introduction to CSP+T

2.1 Overview

Hoare's Communicating Sequential Processes (CSP) is a well-known formalism for describing sets of communicating concurrent processes and their interactions. CSP can be divided into two distinct but related parts. First, the *traces* model allows *non-constructive* process descriptions of one or more deterministic processes. Second, the *algebra* allows *constructive* process descriptions. These two are closely tied together by a complete semantics which allows one to show that a particular algebraic description formally satisfies its specification expressed in the traces model.

However, neither the traces model nor the algebra can be used to describe the *timing properties* of any process, since it was never intended to do so. The basic premise of CSP is that it allows description of a process' behaviour in terms of the sets of sequences of observed events. Sequencing information on its own does not carry any timing information.

The author in [24] proposed some extensions to both the algebra and the traces model which would allow the description of process timing properties. A summary of this work is now presented which will serve as an introduction to the notation and concepts used in the example.

2.2 The Algebra

The CSP+T syntax is a superset of the basic untimed deterministic CSP syntax presented by Hoare [6]. The syntactic extensions and a partial trace semantics are presented in the author's thesis, and will not be reproduced here. A computational semantics for the algebra is the subject of continuing research.

The fundamental changes to the untimed algebra are that:

- A new event operator $\bowtie$ is introduced so that writing $ev \bowtie var_1$ means that the time at which the event ev is observed in a process execution is recorded in the variable var_1. The events associated with this operator are called *marker events*. Variables used in conjunction with this operator are called marker variables. The scope of all such variables is restricted to being solely within a single sequential process definition, with the exception that if the process definition involves calling other processes (eg mutually recursively, plain recursion, or other process calls *within a process body*), that variable may be used by these processes. Outside of these cases, the marker cannot be used by other processes.

- Each event is associated with a time interval. This time interval expresses the time since the preceding event that the current event is *enabled*. Relative expressions are introduced to express constraints that depend on the time of events other than the most recent. These expressions use marker variables.

- Each process definition requires that it is *instantiated* before it can execute. As such, a special process instantiation event denoted by "$\star$" is introduced into both the algebra and the traces model.

- Only *deterministic* processes can be described in the algebra.

Consider the following clock (presented in [6]):

$CLOCK = tick \rightarrow CLOCK$

This clock engages in a single event: *tick*, and it does so forever. However, this clock only gives us sequencing information. We cannot say anything about the *temporal relationships* between *tick* events.

Suppose that this clock is meant to engage in a *tick* event every time unit. The process may be specified by using enabling intervals in one of two methods. Both methods have enabling intervals that may be open, closed or half-open, with $[0, \infty)$ representing the least specified enabling period (for an event that can occur at any time).

The first method uses an enabling interval of $[1, 1]$ on each *tick* event.

$CLK = [1, 1].tick \rightarrow CLK$

This means that a *tick* event occurs precisely one time unit after the immediately preceding *tick* provided that this process is *isolated*. An *isolated process* is a process which is either:

- the only executing process within an environment, or
- if there are one or more other processes executing within the environment along with the $CLOCK$ process, that all events from these other processes are *hidden* from the environment (and consequently, the $CLOCK$ process).

The reason for requiring process isolation is that it is possible that the preceding event (to the *tick*) may have been due to another process, in which case the clock's timing will no longer be regular. This example shows that if the function representing the enabling interval does not explicitly mention any specific marker variables, then the event timing is defined strictly in terms of the immediately preceding event time. This is the usual timing that other Timed CSP algebras adopt (by defining successive interevent delays).

The need for process isolation is reason to bring in the second method, where the enabling interval associated with the *tick* event is dependent solely upon the preceding *tick* time. This is achieved by specifying the enabling interval in terms of the marker variable associated with the tick event:

$CLK = [rel(1, v), rel(1, v)].tick \bowtie v \rightarrow CLK$

where *rel* is defined as follows: if the preceding event occurred at some absolute time t_0 and the value held in a variable v is x, then the expression $rel(x, v)$ denotes $x + v - t_0$. This expression is defined to allow the expression of a single enabling interval in terms of several different different marker variables.

Besides the event-based extensions, some basic processes are also defined: $STOP$, the "broken" process; $SKIP$, the successfully terminating process; and $DELAY(n)$, which also terminates successfully, but only after the specified period n has elapsed since process instantiation. Each of these processes, as all other processes, requires that it be instantiated prior to attempting to be executed. This implies that each process engages in at least a process instantiation event (denoted by $\star$), and that the timed traces of even the $STOP$ process are not empty—they must include the $\star$ event. Other than this difference, the semantic meaning

of these processes is similar to the Reed-Roscoe Timed CSP processes $STOP$, $SKIP$, and $WAIT\ n$, respectively. [15, 16, 17]

Additionally, this paper denotes a process alphabet by Ev, and the set of interconnecting channels is denoted by Ch

2.3 The Traces Model

The traces model is extended by associating a time stamp (drawn from the set of positive real numbers, $\mathbb{R}_+$) with each event in the process' execution, so that the process may be characterised by the set of finite length sequences of pairs called *timed traces*. For example, a timed trace

$$\langle (0.\star), ((0.5).a), (1.b), ((1.21).c) \rangle$$

records that the process was instantiated at time 0, then an event a was observed at time 0.5, event b at time 1, and event c at time 1.21.

Definition 1 *A timed event is written as a pair, $(t.e)$, where $t \in \mathbb{R}_+$, and e is an arbitrary event taken from the untimed process alphabet. Event times are absolute with respect to an observer's system clock.*

There is no restriction on the minimum event separation in CSP+T. However, the model does require that sequence of times in any timed trace must not decrease. That is, any timed trace must be *monotonic in time.*

Definition 2 (Event monotonicity) *If the sequence of event time stamps is a monotonically nondecreasing sequence taken from the set of positive real numbers $\mathbb{R}_+$, then the sequence is* monotonic in time.

Notice that this definition does not exclude timed traces which may have events with identical time stamps, but which are ordered by the sequencing information (perhaps defined in the algebra). This means that timed traces such as $\langle 0.a, 0.b \rangle$ is regarded as *distinct* from one such as $\langle 0.b, 0.a \rangle$.

2.3.1 Timed executions and traces

It was found that the most common method for describing process evolution in the algebra was in terms of relative timings, rather than absolute timings. Accordingly, the timed traces model accommodates this by dividing a set of timed traces into a (special) singleton instantiation event (written $\star$) and the set of timed sequences called *timed executions*. Since these timed executions are affected by the values of the process' variables, we incorporate these into the specification of a process' timed executions by listing all the variables and their currently bound values in the process in an *association list*. An empty association list is written as [] and a singleton association list with variable v_1 with a value t_1 as $[v_1 = t_1]$. In general, an association list of n values is written as $[v_1 = t_1, v_2 = t_2, \ldots, v_n = t_n]$ where each variable name is distinct, but each value does not have to be distinct. Two or more variables can be bound to the same value, e.g. $[v_1 = t_1, v_2 = v_3 = v_4 = t_2]$.

Definition 3 (Timed Executions) *For a process expression P, the set of timed executions observable when the variable values are given by an association list A and process instantiation occurs at time st is written*

$$exec(P, A, st).$$

Further, since process expressions now involve explicitly named variables (and seldom if ever use subscription) the usual manner of providing process specifications exclusively in terms of the trace properties cannot be used alone. Typical specifications therefore consist of the conjunction of the trace properties and a set of predicates on the variable values used explicitly.

The entire set of timed traces of any process may be considered to be a union over all possible instantiation times of the process executions, each prefixed with an instantiation event at the appropriate time.

Definition 4 (Timed traces) *For all instantiation times st:*

$$timedtraces(P) = \bigcup\{x \mid x = \langle\rangle \vee (\mathsf{hd}\, x = (st.\star) \wedge \mathsf{t}|\, x \in exec(P, A, st))\}$$

As an example, consider a bell that is instantiated, then goes *ding* after waiting for a period given by a variable x, and then goes *dong* 1 time unit after that, and can then perform *clack* at any time thereafter. If the value of x is 5 and the process was instantiated at time 0, the execution set is given by $exec(bell, [x = 5], 0)$ and includes many sequences:

$$\langle(5.ding), (6.dong), (8.clack)\rangle,$$

$$\langle(5.ding), (6.dong), (11.clack)\rangle,$$

and even

$$\langle(5.ding), (6.dong)\rangle,$$

since it is possible that the observer will cease recording events while the process is still running. This set is represented usually by a declarative description, rather than by listing its members since there is a possibility that the set of process executions may be infinite, despite each member being finite. It is preferable to write $exec(bell, [x = x_0], st)$ to represent the union of $\{\langle((st + x_0).ding), ((st + x_0 + 1).dong), (y.clack)\rangle \mid y \geq st + x_0 + 1\}$ together with all the prefixes of members of this set (including the null execution, $\langle\rangle$).

However, timed executions such as $\langle(6.ding), (8.dong)\rangle$ and $\langle(7.ding)\rangle$ cannot occur. The bell must engage the *ding* event only at time 5, and the *dong* event exactly one time unit after that, that is, at time 6.

As pointed out earlier, the author has some partial trace semantics for the algebra (which for the sake of brevity are not presented here). Work is currently in progress to ensure that the trace semantics is complete.

We now move on from the outline of the CSP+T formalism to presenting the protocol proper.

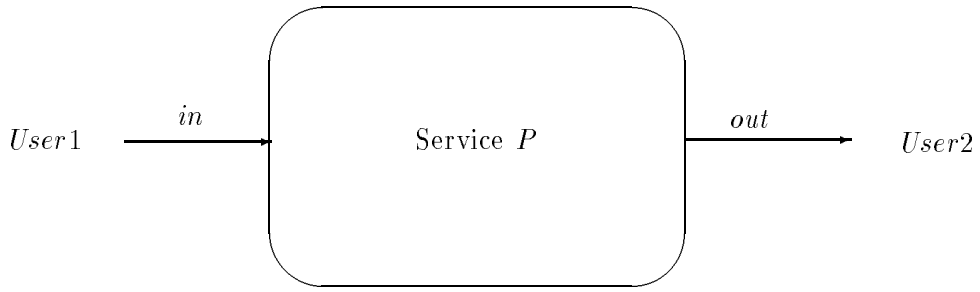

Figure 1: Service diagram

3 The protocol

The following example shows how the proposed extensions to CSP allow the components of a simple communication system to be described. In this example, there is a unidirectional message transfer between two agents $User1$ and $User2$ via a communication service P (refer to Figure 1). However, the service itself must be implemented using unreliable communication links and so a service protocol is required to ensure that as far as the users are concerned, they have a reliable communication link between them. This is done by the use of an unbounded sequence number stop and wait protocol with an appropriate timeout mechanism.

Let P be a protocol service process, with $\{in, out\}$ the channels connecting the two user entities $(User1, User2)$ to the service.

First consider the untimed characteristics of this service.

Any message sent by $User1$ will be eventually received in the correct order by $User2$. The usual buffer condition applies for any timed trace of the system t:

Protocol-spec $\triangleq$
$t \downarrow out \leq t \downarrow in.$

This says that the sequence of contents of messages on the *out* channel form a prefix of the sequence of contents of messages on the *in* channel.

Further requirements might be put on the service in that it must meet the following real-time specifications as well:

- Average throughput of $THRU$ messages per second which may be expressed as a predicate *Thru-spec*,
- Message input rate of R messages per second (given by a predicate *Input-rate*) and
- each message sent through the system will encounter a transit delay laying within a range of values $[\mathcal{T}_{min}, \mathcal{T}_{max}]$ (which may be expressed as a predicate *TD-spec*)

The complete system behaviour is then the conjunction of these timed predicates [1] and the untimed predicate:

[1] These predicates are not given in this paper for the sake of brevity. They are available from the author if required.

Protocol-spec $\wedge$ *Input-rate* $\wedge$ *Thru-spec* $\wedge$ *TD-spec*.

There are many implementations that will meet the above specification, each being selected on the resources available to the implementor. There are some resources that preclude any implementation using these resources ever meeting this specification since they do not possess satisfactory timing properties. In this well studied example, it is common to assume that the underlying connections between processes may only nondeterministically lose messages; a typical (and quite common) implementation for unreliable channels is that there is a nondeterministic choice made on whether to accept input messages or whether to lose them. However, as the proposed CSP+T allows descriptions of only *deterministic* processes, the usual unreliable buffer descriptions cannot be used. For the purposes of this example, it is assumed that the unreliable channels may be described by a fair, deterministic message loss mechanism. The usual nondeterministic behaviour in such a system is rejected since it leads to the possibility that nothing can be said about the system timing which, after all, defeats the purpose of introducing timing at all.

We now turn our attention to a description of the service provider processes.

3.1 Overall view of the service provider processes

The system provider process is composed of a transmitter process TX, a forward and reverse channel FWD and REV respectively, and a receiver RX. These component processes are interconnected as shown in Figure 2 and operate in a parallel composition to produce the required service $P = TX \;||\; FWD \;||\; RX \;||\; REV$. Messages are sent from the $User1$ client to the transmitter TX. The transmitter then forwards the message onto the buffer FWD, representing an unreliable channel with a finite message delay.

The receiver, on correct reception of a message, will pass it on to the $User2$ process and send the transmitter an acknowledgment message via the reverse channel. If the acknowledgment sent by the receiver on the reverse channel does not reach the transmitter within a specific time, the transmitter resends the current message. This is repeated until the correct acknowledgment is received.

To ensure that the service correctly reassembles the messages at the receiver (including removal of any duplicate messages), each service message is labelled with a sequence number taken from the set of natural numbers $\mathbb{N}_1 = \{1, 2, 3, \ldots\}$. This is done in the transmitter. Thus each event sent on by the transmitter to the channel and on to the receiver (and *vice versa*) is a 4-tuple $(t.seq.ch.ev)$ where $t \in R_+$ denotes the time at which the message ev is observed, $seq \in \mathbb{N}_1$ is the sequence number associated with the message, $ch \in Ch$ is the channel name (drawn from the set of channel names for this system) that the message appears upon.

Each of the component processes is now described. We commence with the simplest (the channel), then proceed on to the receiver, and finally the most complex (the transmitter). The overall system is then placed in a parallel composition to provide the service required.

3.2 The channel processes

The forward and reverse channels are implemented using a single type of buffer with the channel names appropriately relabelled. Call the basic buffer process LB (for lossy buffer). LB has a single input in and a single output out. Thus the forward and reverse buffers are:

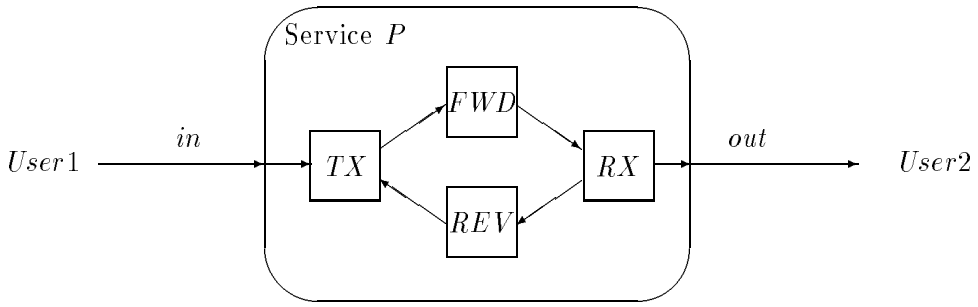

Figure 2: An implementation of the service process

$FWD = LB[in/f1, out/f2]$
$REV = LB[in/r1, out/r2]$.

The LB process either inputs a message on the left, and outputs the message after a specific delay, or it inputs the message and loses it. The message loss mechanism is *deterministic* and is determined by some *coin tossing* experiment, with two outcomes: *head* and *tail*.[2] This coin toss immediately follows any input event (and hence the enabling interval is set to $\varepsilon = [0, 0]$). Should the coin toss turn up a *head*, the message is delayed before passing onto the output channel. Conversely, if the coin toss produces a *tail*, that received message is lost.

A suitable simple implementation for LB using an enabling interval specified as a function of the marker event $in?x$) on the input and output events is:

$$LB = \mu X \cdot E.in?x \bowtie inTime \rightarrow \left(\begin{array}{ll} & \varepsilon.head \rightarrow E.out!x \rightarrow X \\ | & \varepsilon.tail \rightarrow X \end{array} \right)$$

where

$E_1 = \{s \mid s = rel(D, inTime)\}$.

3.3 The receiver process

The receiver operates as follows. If a message is received on its input at any time, and it is in the correct order, it is stripped of its sequence number and passed onto the client. An acknowledgment is sent back to the transmitter process via the reverse channel, indicating that the expected (in sequence) message was correctly received. Should the message appear out of sequence, it is discarded and an acknowledgment is sent back with the sequence number of the last correctly received message. The sequence number (se in the following descriptions) increments only if a correct message that is in sequence is received. Each of these actions are assumed to occur immediately following the reception, and so each event's enabling interval is ε.

[2]This paper deliberately abstracts out the way in which these two events are generated.

$$RX = se := 0; RXS$$

$$RXS = f2?x \rightarrow \left(\begin{array}{c} TO\text{-}CLIENT\text{-}AND\text{-}ACK \\ \not< \ (GetSeq(x) = se) \ \not> \\ RESEND\text{-}ACK \end{array} \right)$$

The *TO-CLIENT-AND-ACK* process outputs an acknowledgment and the message is output to the client, then increments the expected sequence number.

$$TO\text{-}CLIENT\text{-}AND\text{-}ACK = \left(\ \varepsilon.r1!(se.ack) \rightarrow \varepsilon.out!GetEvent(x) \rightarrow \varepsilon.(se := se + 1); RXS \ \right)$$

The *RESEND-ACK* process retransmits an acknowledgment with the current sequence number.

$$RESEND\text{-}ACK = \varepsilon.r2!(se.ack) \rightarrow RXS$$

3.4 The transmitter process

The transmitter process implementation is based on the following *untimed* transmitter, UTX. The transmitter can either accept inputs from the client process and queue them for delivery, accept acknowledgments from the reverse channel, or send messages to the receiver via the forward channel by removing them from the message queue. Clearly it cannot be expected to do anything else except accept inputs from the client if there are no messages queued for delivery.

$$UTX = seq := 0; S := \langle\rangle; \mu X \cdot \left\{ \begin{array}{l} (in?x \rightarrow USTORE; X) \\ \quad \not< \ S = \langle\rangle \ \not> \\ \left(\begin{array}{l} \quad in?x \rightarrow USTORE; X \\ \mid \ r2?y \rightarrow UW(y) \\ \mid \ timeout \rightarrow out!(\mathsf{hd}\, S) \rightarrow X \\ \mid \ TD\text{-}exceeded \rightarrow STOP \end{array} \right) \end{array} \right\}$$

UTX starts with the initial value of the message sequence number being zero ($seq = 0$), which is incremented each time a message is successfully enqueued for transmission (the queue is represented by the variable S). Messages from the $User1$ client are queued by the transmitter's internal buffer, after having been tagged with their appropriate sequence number. This is done by the *USTORE* process:

$$USTORE = seq := seq + 1; S := S^\frown\langle(seq.x)\rangle.$$

If there is nothing queued to be sent on, then the only possible action that the process may engage is the further input of a new message from the $User1$ client. However, if it is not empty, then it may either input a new message and store it, or it may output the head of the queue of messages to be sent off. Should the latter be chosen, the transmitter must firstly wait for an acknowledgment message to be received before proceeding to do anything else. If the acknowledgment arrives and is in the correct sequence, then the current message is removed from the queue, and the process is ready to either input a new message or output the new head of the queue. If the acknowledgment comes back with an incorrect sequence number or a timeout is indicated, the current message is resent and the queue is not shortened. The UW process is used to implement this.

$$UW(x) = \quad (S := (\mathsf{tl}\, S)); out!(\mathsf{hd}\, S) \rightarrow X \;\; \not< \; GetSeq(x) = GetSeq(\mathsf{hd}\, S) \; >\!\!\not\; \; f1!(\mathsf{hd}\, S) \rightarrow X$$

Using this untimed transmitter as a basis, we formulate the *timed* transmitter by finding suitably "interesting" events for use as markers.

First, the transmitter may *timeout*. This occurs if the time difference between the sending of a message on the forward channel and the reception of its corresponding acknowledgment on the reverse channel is greater than the expected message round-trip time. The timeout mechanism occurs only because there has been an excessive time period from the transmission of the message; it is not dependent upon the time at which the message acknowledgment is received. This means that the marker event is the transmission of the message.

Further, the transmitter can accept input messages at a specific rate of R messages per unit time. Thus input messages need to be separated by a time difference of $1/R$. Secondly, there is the possibility that a message may take longer than the specified transit delay, in which case the transmitter is deemed to be "dead", and thus behave as $STOP$. In both of these cases the marker event is the client input.

In summary, the marker events are the

- client input and
- output the sequence numbered message onto the forward channel.

We now need to consider the event enabling intervals. As mentioned earlier, the timeout needs an enabling interval expressed as a function of the time at which the current message was sent forward onto the channel. Secondly, the client input events need an enabling interval expressed as a function of previous input events. Third, acknowledgments are expected to occur within a specific time of the forward message transmission. Finally, the transmitted will break should the transit delay be excessive: this is determined from the time at which the client message was accepted by the transmitter.

Thus the timed transmitter process adds relative enabling intervals to each of the choice events in the above UTX as well as appropriate marker variables to the input and output events. The $USTORE$ and UW processes are now the timed processes $STORE$ and W:

$$\begin{array}{l} TX = \\ \quad seq := 0; \\ \quad S := \langle\rangle; \\ \quad \mu X \cdot \left\{ \begin{array}{l} \quad (E_1.in?x \bowtie inTime \rightarrow STORE; X) \\ \quad\quad \not< \; S = \langle\rangle \; >\!\!\not\; \\ \left(\begin{array}{l} \quad E_1.in?x \bowtie inTime \rightarrow STORE; X \\ \mid \quad E_2.r2?y \rightarrow W(y) \\ \mid \quad E_3.timeout \rightarrow out!(\mathsf{hd}\, S) \rightarrow X \\ \mid \quad E_4.TD\text{-}exceeded \rightarrow STOP \end{array} \right) \end{array} \right\} \end{array}$$

The $STORE$ process stores incoming messages as time and event pairs, with the sequence number is incremented each time a new message is enqueued.

$$STORE = \varepsilon.(seq := seq + 1); \varepsilon.(S := S^\frown \langle inTime.seq.x \rangle)$$

The W process has a similar structure to the untimed UW process.

$$W(x) = \varepsilon.(S := \mathsf{tl}\, S); \varepsilon.f1!GetEvent(\mathsf{hd}\, S) \bowtie outTime \rightarrow X \\ \not< GetSeq(x) = GetSeq(\mathsf{hd}\, S) \not> \\ \varepsilon.f1!GetEvent(\mathsf{hd}\, S) \bowtie outTime \rightarrow X$$

The enabling conditions for the input, output, reception of an acknowledgment and timeout are, respectively:

- The input is enabled subject to the requirement that inputs be spaced at R messages per time unit. This means that the time difference between successive inputs must be $1/R$.
$$E_1 = \{r \mid r = rel(1/R, inTime)\}$$
- After each message has been sent, the process expects to receive an acknowledgment within the round-trip time for a message
$$E_2 = \left\{ t \;\middle|\; rel(2 \times \mathcal{T}_{min}, outTime) \leq t < rel(2 \times (\mathcal{T}_{max}), outTime) \right\}.$$
- The timeout period occurs at the end of the acknowledgment period, and excludes it. It occurs also if the age of the oldest message in the message queue exceeds some fraction of the transit delay, but is still less than the transit delay:
$$E_3 = \left\{ s \;\middle|\; \begin{array}{c} s = rel(2 \times (\mathcal{T}_{max}), outTime) \;\vee \\ \left(\begin{array}{c} rel(k_{age} \times (\mathcal{T}_{max}), \mathsf{hd}\, times) \\ \leq s < \\ rel(\mathcal{T}_{max}, \mathsf{hd}\, times) \end{array} \right) \end{array} \right\}.$$
- The transmitter breaks once the oldest message's age exceeds the transit delay:
$$E_4 = \{u \mid u \geq rel(\mathcal{T}_{max}, \mathsf{hd}\, times)\}.$$

4 Required timing relationships

We now turn our focues from describing the system *algebraically* to capturing the timing relationships in the timed traces model. The timing requirements developed are *independent* of any particular algebraic implementation, and attempt to capture the essential invariant timing relationships which hold during the system's execution.

The following formalisations of each of the service components' timing requirements assume that s, t and u are arbitrary timed traces of a process, with e_1, e_2 and so on representing distinct events.

4.1 Channels

Both FWD and REV share the timing characteristics of LB. The timed trace specification for this process is the conjunction of

$$HEAD \triangleq \\ s^\frown\langle t_1.head\rangle^\frown u \;\Rightarrow\; \mathsf{last}\, s = t_1.in.e_1 \wedge ((u = \langle\rangle) \vee (\mathsf{hd}\, u = (t_1 + D).out.e_1))$$

and

$$TAIL \triangleq \\ s^\frown\langle t_1.tail\rangle^\frown u \;\Rightarrow\; \mathsf{last}\, s = t_1.in.e_1 \wedge ((u = \langle\rangle) \vee (\mathsf{hd}\, u = t_1.in.e_2.))$$

4.2 Receiver

The receiver as presented in Section 3.3 has the following untimed behaviour. Any client output message implies several conditions have been satisfied. First, an output with value e_1 should be observed only if the last event on the input channel was an e_1. Second, there should only have been one such message received before being output. This ensures that there are no repeated messages passed onto the client. Third, the first event in any subsequent process trace will either be empty or will have a new message (with the sequence number incremented) coming in on the receiver input channel.

Formalising this discussion, then:

$$
\begin{array}{l}
OUT \triangleq \\
\quad s^\frown\langle out.e_1\rangle^\frown t \;\Rightarrow\; GetEvent(\mathsf{last}\,(s \downarrow f2)) = e_1 \wedge \\
\qquad\qquad\qquad\qquad (\mathsf{last}\,(s \downarrow f2)) \neg\,\mathsf{in}\; (\mathsf{butlast}\,(s \downarrow f2)) \wedge \\
\qquad\qquad\qquad\qquad (u = \langle\rangle \vee StripTime(\mathsf{hd}\,u) = (seq+1).f2.e_2)
\end{array}
$$

When an acknowledgment is lost for a particular message ev with sequence number seq, any trace will be a prefix of the sequence $\langle seq.f2.e, seq.r1.ack\rangle^n$ for some n:

$$
\begin{array}{l}
LOST\text{-}ACK \triangleq \\
\quad \exists n \in \mathbb{N}_1 \cdot out.ev \neg\,\mathsf{in}\; t \;\Rightarrow\; StripTime^*(t) \leq \langle seq.f2.ev, seq.r1.ack\rangle^n
\end{array}
$$

Neither of these two specifications mentions any explicit timings restrictions. Introducing the timing specifications strengthens the receiver specification.

Should an out-of-sequence message be received, the resulting acknowledgment is sent as soon as possible with no output message.

$$
\begin{array}{l}
UNEXPECTED\text{-}MESSAGE\text{-}TIMING \triangleq \\
\quad (out.ev \neg\,\mathsf{in}\; t \wedge GetEvent(\mathsf{last}\,(t \downarrow r1)) = ack \;\Rightarrow \\
\quad GetTime(\mathsf{last}\,(t \downarrow r1)) - GetTime(\mathsf{last}\,(t \downarrow f2)) = \varepsilon
\end{array}
$$

If a message is correctly received, the message is output and the acknowledgment sent as soon as possible after this message is received:

$$
\begin{array}{l}
EXPECTED\text{-}MESSAGE\text{-}TIMING \triangleq \\
\quad \mathsf{last}\,t = out.ev \;\Rightarrow\; timeof(\mathsf{last}\,t) - timeof(\mathsf{last\,butlast}\,(t \downarrow r1)) = \varepsilon
\end{array}
$$

The receiver specification is the conjunction of the timed and untimed specifications:

$$
\begin{array}{l}
OUT \wedge LOST\text{-}ACK \wedge UNEXPECTED\text{-}MESSAGE\text{-}TIMING \wedge \\
EXPECTED\text{-}MESSAGE\text{-}TIMING
\end{array}
$$

4.3 Transmitter

The transmitter is the most complex of any of the service components. Its function is to ensure the correct sequencing of messages, possibly buffering incoming messages, and addressing the problems of timeout and message acknowledgment.

We first consider the untimed transmitter behaviour.

4.3.1 The untimed constraints

Let e_{tx} represent an arbitrary timed execution of the transmitter process, with $e_{tx} \downarrow f1 = f1$, $e_{tx} \downarrow r2 = r2$ and $e_{tx} \downarrow in = in$.

It is possible that due to the transmitter timing out, a message may be resent a number of (consecutive) times onto the forward $f1$ channel. That is, the messages on the $f1$ channel, if stripped of their timing and sequencing information and with any repeat sequences flattened, forms a prefix order on that of the messages already input on the *in* channel:

$squash(\, GetEvent^*(f1)) \leq in.$

The sequence numbers in the messages sent on channels $f1$ and $r2$ are just incrementing sequences of numbers drawn from $\mathbb{N}_1$, with the sequences on $f1$ consisting of a finite number of repeats of any particular sequence number. Applying a sequence flattening function *squash* to any arbitrary channel trace $f1$, results in a totally ordered sequence of numbers from $\mathbb{N}_1$.

$squash(\, GetSeq^*(f1)) \leq [1, 2, 3, \ldots].$

Similarly, the sequence numbers on the reverse channel $r2$ also form a totally ordered sequence of numbers from $\mathbb{N}_1$.

$squash(\, GetSeq^*(r2)) \leq [1, 2, 3, \ldots].$

Thirdly, the acknowledgment sequence numbers are 1-prefixes of those messages sent forward on the $f1$ channel (with repeats flattened once again). Only one acknowledgment is expected per message (or group of repeating messages), since the repeats are generated due to missed acknowledgments.

$squash(\, GetSeq^*(r2)) \leq^1 squash(GetSeq^*(f1))$

Should the current message's acknowledgment not arrive prior to the timeout period, the current message is resent on the forward channel $f1$.

$\exists u, v \in GetEvent^*(e_{tx}) \cdot$
$\quad (u \frown \langle timeout \rangle \frown v \ \wedge \ v \neq \langle\rangle) \ \Rightarrow \ \mathsf{last}\ u = \mathsf{hd}\ v$

Notice that none of these statements deal with the timing constraints—they are identical to the untimed specifications since the times are removed from each event. As in the receiver, introducing the timing constraints on input message rate, timeout period value and transit delay strengthen the transmitter specification.

4.3.2 The timing constraints

The input message rate may be defined as an ensemble measure. For any timed input sequence, the time difference between the first and the last event of this sequence will be the length of the sequence divided by constant, R..

MR-spec' $\triangleq$
$\quad timeof(\mathsf{last}\ in) - timeof(\mathsf{hd}\ in)) = \#in/R$

Alternatively, the message rate may have been defined as a time difference between two successive inputs. For all $tt \leq timedtraces(TX)$,

MR-spec $\triangleq$
$\quad timeof(\mathsf{hd}\ tt) - timeof(\mathsf{hd}\ \mathsf{tl}\ tt) = 1/R.$

We adopt the latter definition (*MR-Spec*)for simplicity, even though the ISO has used the ensemble measure in its Quality of Service specification [1].

The timeout value is dependent upon two factors. First, the timeout must be set so to be greater than the maximum expected single return trip time if one assumes that there

is no message loss. The time between the transmission of a message and the time its acknowledgment is received must be at least the maximum expected single return trip time (message forward and acknowledgment back). Second, a timeout may occur if the oldest message in the queue of messages is older than some fraction of the expected transit delay. Once a message ages past the specified transit time, the transmitter signals *TD-exceeded* and breaks. This action of deliberately breaking a connection and then reconnecting is done in some protocols since it may lead to an improvement in the service quality. Each of these two factors is now considered in detail.

A good estimate of the round trip time for the transmitter will ensure that a message is correctly received. If a poor estimate is chosen, the protocol throughput tends to zero as the timeout period approaches the round trip time. Choosing a static range of values simplifies the model, however, if the network topology changes, these values need to be adjusted. A dynamic estimate of the round trip time may be possible in some networks, or, as pointed out by Zhang [21], impossible in others where there is the possibility of message loss or reordering. The choice must be made on an engineering basis, and each network and service characterised separately.

This example considers that no messages from $User1$ to $User2$ will experience delays outside the specified range $[\mathcal{T}_{min}, \mathcal{T}_{max}]$. The channels provide a fraction k of this delay. Thus, setting the timeout value to be the sum of the maximum of the forward and reverse message delays. Since they are equal, the maximum expected (single) round trip time is of the order of $2 \times \mathcal{T}_{max}$.

A typical timed execution where the transmitter times out due to the message acknowledgment not being received within specified period would be

$$\langle \ldots, t_1.n_1.f1.a, t_2.n_1.r2.ack, t_3.n_2.f1.b, t_4.timeout, t_5.n_2.f1.b \rangle.$$

For this process, the time difference between the first message (a) being sent and the reception of the acknowledgment of the first message occurred within the timeout period. The process then proceeded to transmit the second message (with sequence number $n_2 = n_1 + 1$), however there was no acknowledgment received within the timeout period. Once the timeout event has occurred, the transmitter can immediately resend the message ($t_5 = t_4$).

The second factor affecting the timeout is excessive message age. This is a direct result of the messages being queued from the $User1$ client, and not being passed on to the $User2$ client due to excessive repeats within the service trying to recover from lost messages. Thus, in order to ensure that the transmitter tries to clear the oldest messages as quickly as possible, it engages a timeout event once the age of the oldest message exceeds some fraction of the transit delay. The channel used in this example is purely deterministic. Recall the channel gives both alternatives, one which provides for a delay of D, and another which causes message loss. This allows a simple way of determining the age of a message. The age of a message is proportional to the sum of the number of lost messages and acknowledgments. This is done by counting the number of *tail* events due to the forward buffer, and adding it to the number of *tail* events due to the reverse buffer. In reality, message loss mechanisms are probabilistic, and it may not be possible to count special loss events such as *tail* above . Thus determining the best value for this fraction needs knowledge of the underlying channel characteristics: typically expected error rate probabilities, distributions of channel delays and other probabilistic notions. Unfortunately CSP+T has no formalism for describing these mechanisms at present, although work is in progress to allow such descriptions.

Formalising this discussion:

$$
\begin{array}{l}
\mathit{Timeout\text{-}spec} \triangleq \\
\quad u^\frown\langle \tau.\mathit{timeout}\rangle^\frown v \ \Rightarrow \\
\quad (v = \langle\rangle) \ \vee \\
\quad (\tau - \mathit{GetTime}(\mathsf{last}\,(u \downarrow \mathit{out})) = 2 \times \mathcal{T}_{max}) \ \vee \\
\quad \left(\begin{array}{c} (k_{age} \times (\mathcal{T}_{max}) \leq (\tau - \mathit{GetTime}(\mathsf{last}\,(u \downarrow \mathit{in})) < \mathcal{T}_{min}) \ \wedge \\ \mathit{GetTime}(\mathsf{hd}\,(v \downarrow \mathit{out})) = \tau \ \wedge \\ \mathit{GetEvent}(\mathsf{last}\,(u \downarrow \mathit{out})) = \mathit{GetEvent}(\mathsf{hd}\,(v \downarrow \mathit{out}))). \end{array} \right)
\end{array}
$$

Since all acknowledgments lie in the range determined by the expected round trip times:

$$
\begin{array}{l}
\mathit{Ack\text{-}expected\text{-}spec} \triangleq \\
\quad u^\frown\langle t_{ack}.\mathit{seq}_n.r2.\mathit{ack}\rangle \ \Rightarrow \\
\quad\quad (\mathsf{last}\ u = \tau.\mathit{seq}_n.f1.\mathit{msg}) \wedge (2 \times \mathcal{T}_{min} < (\tau - t_{ack}) < 2 \times (\mathcal{T}_{max}))
\end{array}
$$

In conclusion, the service is deemed unusable if any messages are queued for more than the specified transit delay. The transmitter engages the *TD-exceeded* event and breaks:

$$
\begin{array}{l}
\mathit{Break\text{-}service\text{-}spec} \triangleq \\
\quad s^\frown\langle \tau.\mathit{TD\text{-}exceeded}\rangle \ \wedge \mathit{TD\text{-}exceeded}\neg\ \mathsf{in}\ s \ \Rightarrow \\
\quad\quad (\tau - \mathit{GetTime}(\mathsf{last}\, s \downarrow \mathit{in})) > \mathcal{T}_{max}.
\end{array}
$$

It should be noted here that a timeout cannot occur during the time that the transmitter is expecting an acknowledgment. Similarly, it cannot be expecting an acknowledgment during the timeout period. This should be kept in mind when selecting a value for k_{age}.

The conjunction of all of these specifications gives us the specification for the whole transmitter.

5 Discussion and conclusions

As the aim of this paper was to demonstrate that complex timing relationships may be readily expressed in the CSP+T algebra, the final stage of showing that the above algebraic implementations and timed trace specifications do behave as required is not demonstrated. Indeed, it cannot be demonstrated yet since any proof requires a complete compositional trace semantics, as well as rules and reductions within the algebra. As mentioned earlier, this is the subject of ongoing research by the author.

However, a generalised proof tactic may be as follows. In order to demonstrate the system trace description does satisfy the specification requirement expressed as the conjunction

$$\mathit{Protocol\text{-}spec} \ \wedge \ \mathit{Input\text{-}rate} \ \wedge \ \mathit{Thru\text{-}spec} \ \wedge \mathit{TD\text{-}spec}$$

the proof task would be divided into three stages.

First the composition of the untimed component specifications would be shown to behave as a buffer. Each of the timed component specifications would then be shown to satisfy each of *Input-rate*, *Thru-spec* and *TD-spec* respectively. Third, each process component (expressed in the algebra) needs to be shown to meet its respective component specification. Once done, the composition of the processes can be shown to indeed meet the required timing constraints on the protocol.

Besides completing the formal semantics for the extensions, there are some issues that need to be addressed in future work. Notably, the process semantics of parallel and interleaved composition allow processes to produce non-causal traces. This indicates that the complete semantics for these two operations requires some additional features, such as a

latest possible time for any next action indicator. The enabling intervals provide a means for describing process actions (process "birth") whereas the latter feature allows description of process failure (or "death").

Second, the model presented allows for purely deterministic process descriptions.

Finally (and related to the above) if this method is to be adopted for use in a more general performance specification, the enabling interval functions need to allow stochastic functions of marker variables. For example, it may be necessary to specify that an event occurs with a specific probability distribution over a given time interval since some preceding event. In addition to this probabilistic extension, the algebra may need to incorporate probabilistic choice. This was proposed informally by the author in [23]. Lowe [9] and Seidel [18] have developed two differing semantic models for a probabilistic CSP using such an operator. Hansson [5] also has defined a probabilistic choice operator in his Timed Probabilistic Calculus of Communicating Systems (TPCCS).

In conclusion, the addition of marker variables and event enabling intervals is felt to increase the expressive power of the CSP algebra, allowing processes to carry over their untimed structure to their timed versions. This is attractive since it allows a degree of economy in process descriptions which other methods (relying purely on delay operations or processes) cannot offer.

A Timed trace operations

A general (binary) relation R on two real numbers is defined by

$$R : \mathbb{R}_+ \times \mathbb{R}_+ \to \mathbb{B}$$
$$R(x, y) \triangleq xRy$$

Examples of R are the usual relations "$\leq$", "$=$" etc.

The head of a trace is changed from Hoare's notation tt_0 to $\mathsf{hd}\, tt$, and similarly, the tail of a trace is written as $\mathsf{tl}\, tt$ rather than tt'.

$$\mathsf{hd} : \mathbb{R}_+ \times Ev^* \to \mathbb{R}_+ \times Ev$$
$$\mathsf{hd}\,(tt) \triangleq tt_0$$

$$\mathsf{tl} : \mathbb{R}_+ \times Ev^* \to \mathbb{R}_+ \times Ev^*$$
$$\mathsf{tl}\,(tt) \triangleq tt'$$

The *timeof* and *eventof* functions are the first and second projections of a timed event:

$$timeof : \mathbb{R}_+ \times Ev \to \mathbb{R}_+$$
$$timeof(e) \triangleq \mathsf{proj1}\,(e)$$

$$eventof : \mathbb{R}_+ \times Ev \to Ev$$
$$eventof(e) \triangleq \mathsf{proj2}\,(e)$$

The last event of a timed trace is obtained by applying the last function.

$$\mathsf{last} : (\mathbb{R}_+ \times Ev)^* \to \mathbb{R}_+ \times Ev$$

$$\mathsf{last}\,(tt) \triangleq \mathsf{hd}\,\overline{tt}$$

where $\overline{a}$ represents the reverse of a, as usual.
The function butlast returns a list with the last element removed:

$$\mathsf{butlast} : (\mathbb{R}_+ \times Ev)^* \to (\mathbb{R}_+ \times Ev)^*$$

$$\mathsf{butlast}\,(tr) \triangleq \overline{\mathsf{tl}\,\overline{tr}}$$

A sequence such as $\langle a, a, a, b, c, c, d\rangle$ may have the runs of consecutive symbols replaced by a single symbol from that run by the *squash* function. In this case,

$$squash(\langle a, a, a, b, c, c, d\rangle) = \langle a, b, c, d\rangle.$$

The function is defined by the recursive formulae:

$$squash(\langle\rangle) = \langle\rangle$$

$$squash(\langle a\rangle^\frown l) = \begin{cases} squash(l) & \text{if } \mathsf{hd}\, l = a \\ \langle a\rangle^\frown squash(l) & \text{if } \mathsf{hd}\, l \neq a \end{cases}$$

As in CSP, $(t \upharpoonright A)$ denotes the trace restriction of a timed trace t to the set of (timed) events in the set A. For example,

$$\langle(0.a), (1.b), (3.b), (4.c)\rangle \upharpoonright \{(0.a), (4.c)\} = \langle(0.a), (4.c)\rangle$$

Events are removed from a trace when they are disjoint from the restriction set times:

$$\langle(0.a), (1.b), (3.b), (4.c)\rangle \upharpoonright \{(1.a), (1.b), (9.c)\} = \langle(1.b)\rangle.$$

If the events in A have no explicit time attached to them, then the trace restriction keeps *any* of those events in the trace despite their times.

$$\langle(0.a), (1.b), (3.b), (4.c)\rangle \upharpoonright \{b\} = \langle(1.b), (3.b)\rangle$$

It is quite common to have events on a particular channel in a system which are explicitly labelled with a sequence number taken from the set of natural numbers. The following functions are defined for such sequenced timed events.

The following functions rename the first,second and third projection functions on timed, sequenced, channel events.

$$GetTime : \mathbb{R}_+ \times \mathbb{N}_1 \times Ev \to \mathbb{R}_+$$

$$GetTime(x) \triangleq \mathsf{proj1}\,(x)$$

$$GetSeq : \mathbb{R}_+ \times \mathbb{N}_1 \times Ch \times Ev \to \mathbb{N}_1$$

$$GetSeq(x) \triangleq \mathsf{proj2}\,(x)$$

$$GetEvent : \mathbb{R}_+ \times \mathbb{N}_1 \times Ev \to Ev$$

$$GetEvent(x) \triangleq \mathsf{proj3}\,(x)$$

$$StripTime : \mathbb{R}_+ \times \mathbb{N}_1 \times Ev \rightarrow \mathbb{N}_1 \times Ev$$

$$StripTime(x) \triangleq \mathsf{proj2}\,(x).\mathsf{proj3}\,(x)$$

References

[1] ISO/TC 97/SC 16/ WG 6. Information Processing Systems – Open Systems Interconnection – Transport Service Definition – Connectionless mode transmission. Standard ISO-8072-1986-Addendum1, ISO, 1986.

[2] Ed. Brinksma. An Introduction to LOTOS. In H. Rudin and C.H. West, editors, *Protocol Specification, Testing, and Verification, VII*. Elsevier Science Publishers B.V., Amsterdam, May 1987.

[3] Jim Davies and Steve Schneider. An Introduction to Timed CSP. Technical monograph PRG-75, Oxford University Computing Laboratory, Programming Research Group, 11 Keble Rd Oxford OX1 3QD England, August 1989.

[4] Y.W. Han. Performance evaluation of a digital system using a Petri Net like approach. In *Proceedings of the National Electronics Conference*, volume 32, pages 166–172, Oct 1978.

[5] Hans A. Hansson. *Time and Probability in Formal Design of Distributed Systems*. PhD thesis, Department of Computer Science, Uppsala University, September 1991.

[6] C.A.R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall International (UK) Ltd, 66 Wood Lane End, Hemel Hempstead, Hertfordshire HP2 4RG UK, 1985.

[7] ISO. Information Processing Systems - Open Systems Interconnection - Estelle, a Formal Description Technique based on an Extended State Transition Model. Standard DIS 9074, ISO, Geneva, July 1987.

[8] Cliff B. Jones. *Systematic Software Development Using VDM*. International Series in Computer Science. Prentice-Hall International (UK) Ltd, 66 Wood Lane End, Hemel Hempstead, Hertfordshire HP2 4RG UK, 1986.

[9] Gavin Lowe. Prioritized and probabilistic models of timed CSP. Technical Report PRG-TR-24-91, Programming Research Group, Oxford University Computing Laboratory, 11 Keble Rd Oxford 0x1 3QD, 1991.

[10] Brendan Mahony and Ian Hayes. Using continous real functions to model timed histories. Technical report, Department of Computer Science, University of Queensland 4072, 1991.

[11] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin–Heidelberg–New York, 1980.

[12] Robin Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice-Hall International (UK) Ltd, 66 Wood Lane End, Hemel Hempstead, Hertfordshire HP2 4RG UK, 1989.

[13] J.L. Peterson. *Petri Net Theory and the modelling of systems.* Prentice-Hall, Inc., Eaglewood Cliffs, NJ 7632, 1981.

[14] C. Ramchandani. *Analysis of asynchronous concurrent systems by Timed Petri Nets.* PhD thesis, MIT, September 1973.

[15] G.M. Reed and A.W. Roscoe. A Timed Model for Communicating Sequential Processes. In *Automata, Languages, and Programming , 13th Intl. Colloqium Proceedings, Lecture Notes in Computer Science*, Berlin–Heidelberg–New York, 1986. Springer-Verlag.

[16] A.W. Roscoe. Two papers on csp. Technical Monograph PRG-67, Oxford University Computing Laboratory, Programming Research Group, 8-11 Keble Rd Oxford OX1 3QD, July 1988.

[17] Steve Schneider. Correctness and Communication in Real-time Systems. Technical Monograph PRG-84, Oxford University Computing Laboratory, Programming Research Group, 8-11 Keble Rd Oxford OX1 3QD, March 1990.

[18] Karen Seidel. *Probabilistic Communicating Processes.* PhD thesis, Oxford University, Oxford OX1 3QD, UK, 1992.

[19] J. Sifakis. Use of Petri Nets for performance evaluation. In Beilner and Gelenbe, editors, *Measuring, Modelling and Evaluating Computer Systems.* North Holland, 1977.

[20] J.M. Spivey. *The Z notation: a reference manual.* Prenctice-Hall International, 1989.

[21] Lixia Zhang. Why TCP timers don't work well. In *SIGCOMM '86 Symposium Communications Architectures and Protocols*, volume 16 of *Computer Communication Review*, pages 397–405, 11 West 42nd St, New York NY10036, August 1986. SIGCOMM, ACM.

[22] John J. Zic. Exercises in real-time buffer specification. Submitted for publication to the ACM LOPLAS, Feb 1993.

[23] John J. Zic. Extensions to Communicating Sequential Processes to allow protocol performance specification. In *SIGCOMM '87 Workshop on Frontiers in Computer Communication Technology*, volume 17 of *Computer Communication Review*, 11 West 42nd St, New York NY10036, August 1987. SIGCOMM, ACM.

[24] John J. Zic. *CSP+T: a formalism for describing real-time systems.* PhD thesis, Basser Department of Computer Science, University of Sydney, NSW 2006, July 1991.