SCS&E Report 9308
June, 1993

A Comparison of Two Real-time Description Techniques

John J. Zic

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

Abstract

A new real-time description language based on Hoare's CSP is proposed, and compared to the more usual Timed CSP in its effectiveness in describing a buffer with differing input and output rates and transit delay requirements. It is found that the new notation offers a concise, natural way of formulating complex timing relationships.

1 Introduction

There has been considerable effort recently in extending Hoare's CSP [6] and Milner's CCS [7, 8] to allow formal reasoning about real-time systems. Examples of such systems are commonly found in communication protocols where the response to a message is required before the message becomes obsolete, or where message outputs need to be spaced so as to avoid overflow conditions at the receiving end.

The author proposed some informal time (and probability) extensions to CSP in [11] where a special $DELAY\ n$ process allowed temporal separation between any two successive events by n time units. At the time, the author felt that this should have been sufficient to capture most of the timing constraints within a system. Timed CSP [10] introduced a similar special process $WAIT\ n$. In addition, Reed and Roscoe provided a complete timed semantics for their timed failures model. Gerber, Lee and Zwarico [5] introduced a *timed action* operation to temporally separate adjacent events into their timed acceptances model.

Quemada and Fernandez [9] proposed an extension to the LOTOS specification language [2] by associating an enabling interval with each event. This time interval represents the time over which a process may engage the event.

Recently the author in [13] proposed an extended CSP by not only associating an enabling interval with each event, but also allowing this interval to be expressed as a function of one or more *marker events.* In the author's experience, this combination of expressing enabling intervals in terms of marker events allow particularly concise descriptions of some realistic real-time systems. Some of the examples considered in [13] were various clocks (accurate and inaccurate), buffers, multichannel multiplexers, a TCP timer and a stop and wait communication protocol.

This paper is organised as follows. First, we present the problem, which is the modelling of a store-and-forward communication system with specific quality of service requirements. Next, an algebraic description of the buffer is sought. Two approaches to providing an algebraic description of the buffer are presented. The first is based purely on a $WAIT$ operation, the second using the author's extensions of CSP: event enabling intervals which are expressions of a set of marker variables. The paper concludes by noting that using purely a $WAIT$ based system at worst cannot be used to describe the buffer behaviour. At best, the notation leads to clumsy, unnatural descriptions. On the contrary, using the combination of marker variables to determine event enabling intervals leads to natural, concise descriptions of real-time systems.

The solutions then are presented. The first buffer developed considers only the input and output rates, and the second introduces the transit delay as well as the input and output rate constraints.

2 The problem

A store-and-forward communication network may be abstractly represented by a message *buffer*. Messages injected into the network at a particular node appear some time later at another node in the same order as they were sent (assuming that the networking protocols are correctly handling any message losses and reordering).

Besides this most abstract functionality of order preservation, a communication system may also need to provide end users with some real-time performance. For example, maximum

Time	In	Out	Buffer contents	Oldest age
0	a	.	$[(0.a)]$	0
1	b	.	$[(0.a),(1.b)]$	1
2	c	a	$[(1.b),(2.c)]$	1
3	d	.	$[(1.b),(2.c),(3.d)]$	2
4	e	b	$[(2.c),(3.d),(4.e)]$	2
5	f	.	$[(2.c),(3.d),(4.d),(5.f)]$	3
6	g	c	$[(3.d),(4.e),(5.f),(6.g)]$	3
7	h	.	$[(3.d),(4.e),(5.f),(6.g),(7.h)]$	4
8	i	d	$[(4.e),(5.f),(6.g),(7.h),(8.i)]$	4
9	j	.	$[(4.e),(5.f),(6.g),(7.h),(8.i),(9.j)]$	5
10	k	e	$[(5.f),(6.g),(7.h),(8.i),(9.j),(10.k)]$	5
> 10	Cannot meet both transit delay and output timing requirements			

Table 1: Buffer timing assuming earliest initial message output

and average message delays, throughput, reliability, probability of loss of a message, and other client requirements may be important [4, 1].

This paper attempts to describe a communication system with the following characteristics given by a (somewhat) naive client:

- up to 128 messages in transit at any time,
- message latency in the range $[2, 5]$ time units,
- message input rate set to 1 message per time unit., and
- message output rate of one messages per two time units.

2.1 Initial timing analysis

As the informal timing constraints stand, there will be problems with any implementation. Firstly, the fact that the output message rate is half that of the message input rate means that the any finite sized buffer will eventually either overflow or not meet the message transit delay (or latency) requirements. This is best illustrated in Tables 1 and 2.

Both tables assume that the buffer starts accepting inputs at time 0. The sequence of messages to be input is $a, b, c, d, e, f, g, h, \ldots$ and the times at which each of these successive inputs may appear are at times 0, 1, 2, 3, ...etc if the input timing requirements are to be met. Tables 1 and 2 give the two extremes in timing behaviour. [1] The timing behaviour after the first input event is dependent upon the time at which the first output action is engaged by the buffer. This is in turn is determined by the transit delay bounds only, and the two tables represent the maximum and minimum transit delays for the first output. After the first output, the system must satisfy both the output timing (outputs need to be spaced at exactly two time units) and the transit delay (which lies in the range $[2, 5]$) requirements.

[1] The buffer contents is displayed *after* input and output operations have been performed.

Time	In	Out	Buffer contents	Oldest age
0	a	.	$[(0.a)]$	0
1	b	.	$[(0.a),(1.b)]$	1
2	c	.	$[(0.a),(1.b),(2.c)]$	2
3	d	.	$[(0.a),(1.b),(2.c),(3.d)]$	3
4	e	.	$[(0.a),(1.b),(2.c),(3.d),(4.e)]$	4
5	f	a	$[(1.b),(2.c),(3.d),(4.e),(5.f)]$	4
6	g	.	$[(1.b),(2.c),(3.d),(4.e),(5.f),(6.g)]$	5
7	h	b	$[(2.c),(3.d),(4.e),(5.f),(6.g),(7.h)]$	5
> 7	Cannot meet both transit delay and output timing requirements			

Table 2: Buffer timing assuming latest initial message output

Eventually both conditions cannot be simultaneously satisfied, and the system must fail (in some manner).

Nonetheless, let us examine some of the other difficulties encountered in trying to describe such a system. These other difficulties arise not because of some mistaken timing specification, but rather are a result of the description language limitations.

3 The search for a solution

3.1 Using the delay based approach

Consider a simple one-place buffer (with input channel *in* and output channel *out*) which has no timing constraints. The simplest implementation possible is given by

$$\mu X \circ in?x \rightarrow out!x \rightarrow X \tag{1}$$

where an input is immediately output before allowing a further input.

If we introduce time into the above process, then it is possible to interpret the lack of any explicit temporal separation in Equation (1) between two successive events (such as *in* then *out* communications) in at least two ways.

In the first view, the lack of explicit timing may be interpreted as allowing successive events to occur at the same time while maintaining any sequencing order. For example, a sequence such as $a \rightarrow b \rightarrow \ldots$ is differentiated from the sequence $b \rightarrow a \rightarrow \ldots$, despite both events being observed at the same global time (according to the observer's watch, say). If both a and b are observed at time 1, the former ($a \rightarrow b \rightarrow \ldots$) has a trace $\langle 1.a, 1.b, \ldots \rangle$ while the latter has a trace $\langle 1.b, 1.a, \ldots \rangle$.

In the second view, the lack of explicit timing is interpreted as allowing events to occur at any time, again provided that any sequencing is preserved. A sequence such as $a \rightarrow b \rightarrow \ldots$ where the a occurs at time 0 for example would allow the b event to follow at any time taken from the half-open interval $[0, \infty)$ after the a.

The proposed extended CSP (called *CSP+T*) uses this latter view. Most other algebras adopt the former view, and use a temporal operation or process to provide the required interevent delay.

We start, then, with the Timed CSP model first proposed by Reed and Roscoe [10], which has been subsequently modified to eliminate the system delay constant [3] so that any event timing must be explicitly described using a $WAIT$ process.

Producing a buffer which delays each message by the required delay is straightforward in this model:

$$\mu X \circ in?x \rightarrow WAIT\ I\ ;\ out!x \rightarrow X \tag{2}$$

with the interval $I = [2, 5]$. This buffer accepts an input, then delays by an amount taken from the interval I, and then outputs the message. Notice that there is an asymmetry in this process. Despite ensuring that the input and outputs are correctly timed, there is an inherent zero spacing between an output and a following input under the maximal progress model. Further, inputs are separated from successive inputs (and outputs from successive outputs) by an amount determined by the input to output separation. These timings are therefore *dependent on each other.* This buffer not only spaces (temporally) inputs to outputs, but it also spaces outputs to successive outputs, and inputs to successive inputs.

This very same buffer may be used to space inputs to inputs, or outputs to outputs. In the case of inputs to inputs

$$\mu X \circ in?x \rightarrow WAIT\ 1\ ;\ out!x \rightarrow X \tag{3}$$

will satisfy the timing requirement that inputs are separated by one time unit.

However, buffer

$$\mu X \circ in?x \rightarrow out!x \rightarrow WAIT\ 2\ ;\ X \tag{4}$$

does not delay any message by any fixed amount across input to output but spaces its inputs and outputs by two time units.

Note that each of the buffers expressed in Equations (2), (3), and (4) implement only a single part of the required behaviour. Further, there is no message storage—only one single message is ever "in transit". Achieving the three goals simultaneously (specific message transit delay, differing input and output rates) cannot be done with a single process which is based on (1). A composite process needs to be found, with each component defining the input, output and transit delay timings.

A natural decomposition would be to use separate buffers for input and output determining the input and output rates. These are composed in parallel with a third (intermediate) buffer, forming a *chain*. Unfortunately, the required timing may only be accomplished if the intermediate buffer itself provides the necessary input and output timings. Since it is impossible to achieve this goal, it is pointless trying to introduce a third factor, the specific transit delay.

So we abandon the use of single place buffers in seeking a solution to this problem and move onto finite size buffers, which are based on the infinite buffer of [6, p138, X9]. These buffers may input several messages in sequence without having any intervening outputs, or, alternatively, produce several outputs without accepting any further intervening inputs. [2]

$$Buff \mathrel{\widehat{=}} X_{\langle\rangle}$$

[2] This paper adopts the conventional notation **if** b **then** P **else** Q **fi** for the CSP conditional $P \lhd b \rhd Q$ where b is a boolean value, P and Q processes.

$B_{\tau in}$	
Subsequence	**Time Difference**
$\langle in, in \rangle$	1
$\langle in, out \rangle$	1
$\langle out, in \rangle$	0
$\langle out, out \rangle$	0

Table 3: Buffer with input delay $t_i = 1$, and output delay $t_o = 0$

where

$$
\begin{array}{lcl}
X_{\langle\rangle} & = & in?x \rightarrow X_{\langle x \rangle} \\
X_{\langle y \rangle^\frown S} & = & \textbf{if } \#(\langle y \rangle^\frown S) < 128 \\
 & & \textbf{then} \quad in?x \rightarrow X_{\langle y \rangle^\frown S^\frown \langle x \rangle} \\
 & & \qquad \Box \quad out!y \rightarrow X_S \\
 & & \textbf{else } out!y \rightarrow X_S \\
 & & \textbf{fi}
\end{array}
\tag{5}
$$

This buffer is either empty, or it is full and is holding 128 messages. If it is empty, then the only possible action is to accepts inputs. If it is full, the only possible action is the output of the head of the queue of messages. [3]

The above buffer is used as a basis for the implementation of a buffer B_τ which has delays on both input (t_i) and output (t_o).

$$B_\tau \mathrel{\widehat{=}} Z_{\langle\rangle}$$

where

$$
\begin{array}{lcl}
Z_{\langle\rangle} & = & in?x \rightarrow WAIT\ t_i \,;\, Z_{\langle x \rangle} \\
Z_{\langle y \rangle^\frown S} & = & \textbf{if } \#(\langle y \rangle^\frown S) < 128 \\
 & & \textbf{then} \quad in?x \rightarrow WAIT\ t_i \,;\, Z_{\langle y \rangle^\frown S^\frown \langle x \rangle} \\
 & & \qquad \Box \quad out!y \rightarrow WAIT\ t_o \,;\, Z_S \\
 & & \textbf{else } out!y \rightarrow WAIT\ t_o \,;\, Z_S \\
 & & \textbf{fi}
\end{array}
\tag{6}
$$

There are four possible "interesting" communication event subsequences. If we consider only on which channel the event occured, rather than the content, we have: $\langle in, in \rangle$, $\langle in, out \rangle$, $\langle out, in \rangle$ and $\langle out, out \rangle$.

A buffer $B_{\tau in}$ which has an input delay only ($t_i = 1$) and no output delay ($t_o = 0$) results in the time differences given by Table 3. Table 4 gives the time difference for a buffer $B_{\tau out}$ which has no input delay ($t_i = 0$) and an output delay given by $t_o = 2$.

However, these two tables do not account for the storage time associated with each item. As these are finite buffers (possibly holding 128), each item may be stored in the buffer or immediately output. For example, suppose $B_{\tau in}$ engages in 128 successive inputs (with no

[3] This buffer would seldom be realised, since it is prone to *chatter* when it gets full. When full, it may enter a cycle of outputting a message, then accepting an input, becoming full again, outputting, accepting an input, etc. Realistic buffers incorporate some form of hysteresis which prevents this type of behaviour. See [11] for the specification of a buffer with hysteresis.

$B_{\tau out}$	
Subsequence	**Time Difference**
$\langle in, in \rangle$	0
$\langle in, out \rangle$	0
$\langle out, in \rangle$	2
$\langle out, out \rangle$	2

Table 4: Buffer with input delay $t_i = 0$ and output delay $t_o = 2$

Buffer	t_i	t_o	**Transit delay range**
$B_{\tau in}$	1	0	$[1, 128]$
$B_{\tau out}$	0	2	$[0, 256]$

Table 5: Buffer transit delay

intervening outputs) followed by 128 successive outputs (with no intervening inputs). Each particular item in the buffer will encounter a delay of 128×1 time units from the time it was stored in the buffer to the time it was output. Similarly, $B_{\tau out}$ will lead to a delay of 128×2 time units if it engages in 128 successive inputs, followed by 128 successive outputs. This case represents an absolute maximum delay. The minimum delay is 1 for $B_{\tau in}$ and 0 for $B_{\tau out}$.

Again, because of the interdependence of input, output and transit delay times, these buffers are once again not suited to implementing a single buffer with independent input output, and transit delay times. The *WAIT* process cannot be successfully used in specifying interevent timings for this buffer.

An alternative method is required for specifying interevent timings.

3.2 Using the extended CSP algebra

3.2.1 A brief description of the extensions

The CSP+T syntax is a superset of the basic untimed deterministic CSP syntax presented by Hoare [6]. The fundamental changes to the untimed algebra are that:

- A new event operator $\bowtie$ is introduced so that (informally) writing $ev \bowtie v$ means that the time at which the event ev is observed in a process execution is recorded in the variable v. The scope of all such variables is restricted to being solely within a single sequential process definition. If the process definition involves other process definitions (e.g. recursion, or other process calls *within a process body*), that variable may be used by these processes. The marker variable may not be referenced by a peer process or a process which contains the process.

- Each event is associated with a time interval. This time interval expresses the time since some preceding event that the current event is *enabled*. These intervals are defined

in terms of functions over a set (including the empty set) of marker variables. If not explicitly mentioned with an event, the interval $[0, \infty)$ is assumed. That is, the event associated with this interval is allowed to occur at any time since the immediately preceding event. In this sense, then, processes are *not* maximal progress *unless they are specified to be so* by setting the enabling interval to $[0, 0]$.

- 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 the algebra (and in the corresponding traces model).
- Only *deterministic* processes can be described in the algebra. Nondeterministic timings lead us to situations where it may be impossible to say anything about the process timing whatsoever.

The compositional semantics of these extensions will not be presented in this paper. Rather, a small example introduces the notation by providing an informal operational semantics prior to considering the buffer specification proper.

3.2.2 A small digression: specifying real-time clocks

An analogue mechanical clock may be abstracted to only ever engage in a single event: *tick*. If we do not care about the sense of the passage of time, then that abstraction is sufficient, and thus the clock may be described by

$$UntimedClk \;\widehat{=}\; \mu\, X \circ tick \rightarrow X. \tag{7}$$

However, the most common interest in clocks is that they do represent the passage of time, in which case the above process is insufficient. It does not capture the temporal relationships between the *tick* events.

One way to express this temporal relationship is to assume a maximal progress model, and then introduce an explicit *WAIT* process.

Another way is to define the times at which the *tick* events may be engaged by the process by using a suitable event enabling interval in a least specified process model. If we require that our new clock *tick* once per second (say), then each *tick* is enabled precisely one second since its predecessor (which, of course, was another *tick*). This is expressed by

$$TimedClk \;\widehat{=}\; \mu\, X \circ [1, 1].tick \rightarrow X. \tag{8}$$

It should be noted that the recursive call does not take any time in this extended model.

When does the first *tick* occur in this process definition? This question is resolved by introducing a special event, called the *process instantiation event* (denoted by $\star$). This event represents the time at which a process or system of processes is "powered up" or instantiated. This event has no enabling interval associated with it by definition; the time associated with it represents the "global time" at which the process instantiation event occurs.

Hence our timed clock given in (8) only starts to *tick* once it is wound up at some time *st* (which can be thought of as instantiating the clock):

$$RealClock \;\widehat{=}\; st.\star \rightarrow TimedClk. \tag{9}$$

The *RealClock* engages in its first *tick* at (absolute, global time) $st + 1$, the second *tick* at $st + 2$, and so on.

Notice that changing the above enabling interval allows the specification of a clock which *tick*s once per every two seconds by writing $[2, 2].tick$ for the enabling interval in *TimedClk*. Similarly, it may be made to tick once every third of a second by rewriting the enabling interval to $[1/3, 1/3]$.

A clock which has some degree of slack in its mechanism will lead to a tick event occurring approximately once per second. This can be specified by allowing the enabling interval to move from a point to a narrow width interval such as $[0.99, 1.01]$.

The same clocks may be expressed using *marker events*. The *TimedClk* above may be written as

$$TimedClk' \;\widehat{=}\; \mu\, X \circ E.tick \bowtie v \rightarrow X \tag{10}$$

where

$$E = \{s | s = rel(1, v)\}. \tag{11}$$

and the *rel* function is defined as follows. If the preceding event occurred at time t_0, the expression $rel(x, v)$ denotes $x + v - t_0$. This convention allows us to combine conditions expressed relative to several different marker events in the definition of a single enabling interval.

Again, the process does not start to *tick* until it is instantiated. By changing the enabling interval expression, the clock timing characteristics are readily changed. For example, the inaccurate clock may have its enabling interval expressed as

$$E = \{s | rel(0.99, v) \leq s \leq rel(1.01, v)\}.$$

Notice that since the enabling interval is specified by an expression involving marker variables it may be defined by an event which occurred many events events in the past. That is, there is no longer the requirement that an event's timing is solely dependent upon its immediate predecessor. For example, a clock which engages in *tick* events once per second and a *tock* event 0.25 seconds after a *tick* event may be expressed by

$$\mu\, X \circ E_1.tick \bowtie v \rightarrow E_2.tock \rightarrow X$$

where

$$\begin{aligned} E_1 &= \{s | s = rel(1, v)\} \\ E_2 &= \{t | t = rel(0.25, v)\}. \end{aligned}$$

3.2.3 Using CSP+T to specify the buffer

Since we do require that the buffer hold more than one item, we start again using the buffer given by Equation 5, and consider the specification of the input and output timings.

The buffer engages in only two events. Either it inputs a value and places it at the end of the queue, or it outputs the head of the queue. We therefore associate a marker variable with the input and output to capture their respective event enabling intervals. The enabling interval function for input is solely a function of the input marker variable. The enabling interval for output is similarly expressed in terms of the output marker variable.

Let E_{in} represent the input enabling interval, and E_{out} represent the output enabling interval. Then we set

$$\begin{aligned} E_{in} &= \{s | s = rel(1, v_i)\} \\ E_{out} &= \{t | t = rel(2, v_o)\}. \end{aligned}$$

The buffer B with two separate input and output rates may be defined by the set of recursive equations

$$B \mathrel{\widehat{=}} X_{\langle\rangle}$$

where

$$\begin{aligned} X_{\langle\rangle} &= E_{in}.in?x \bowtie v_i \rightarrow X_{\langle x\rangle} \\ X_{\langle y\rangle^\frown S} &= \textbf{if } \#(\langle y\rangle^\frown S) < 128 \\ &\quad \textbf{then} \quad E_{in}.in?x \bowtie v_i \rightarrow X_{\langle y\rangle^\frown S^\frown \langle (val(v_i),x)\rangle} \\ &\qquad \Box \quad E_{out}.out!y \bowtie v_o \rightarrow X_S \\ &\quad \textbf{else } E_{out}.out!y \bowtie \{v_i, v_o\} \rightarrow X_S \\ &\quad \textbf{fi} \end{aligned} \tag{12}$$

Suppose that the buffer is instantiated at time 0. At this time, the marker variables are also initialised (to be 0). The first input to the buffer must occur at time 1, while the first output must occur at time 2. However, at the same time as the first output, the buffer expects to perform an input.

The buffer will fill at time 255, after which the only possible action is that it output a message. However, it is at this point that problems arise. These problems are not a result of the notation. They arise simply because of the characteristics of this buffer. Notice that these issues never arose with the *WAIT* notation since most of the effort was directed at trying to achieve the independent input and output timings.

At time 255, the buffer fills. At time 256, the front of the queue is output and the buffer contents reduced so that the buffer is no longer full. The only action when the buffer is full is to output the head of the queue. Both the v_i and v_o marker variables now hold the time that this action was performed. Therefore, since the buffer is no longer full, at time 257 the next input occurs, filling the buffer. The buffer then may only output a message at time 258. This cycle continues ad infinitum, with the buffer filling, then emptying one message, accepting one message, filling, and so on.

However, the input timing restrictions have now been violated, and inputs are only allowed to be accepted every two time units (tracking the output timing).

This may be an acceptable behaviour in some cases. More usual, however, is to signal that the buffer is full, stop any further inputs until the buffer empties out completely, then start the inputs again. This complicates the buffer slightly and highlights a problem in the specification. The original specification did not point out what input and output timings are acceptable when the buffer is full.

Alternative solutions to this problem are known to exist. Commonly, buffers may display hysteresis. This is implemented by having a "high water" and "low water" values within the buffer, as well as the empty and full indications. The buffer accepts inputs until it reaches the length of the queued items reaches the "high water" value. It then blocks any further inputs, and proceeds to produce outputs until the length of the queued items reaches the "low water" value. At this time, the buffer is free to accept any further inputs.

The buffer presented in (12) does not address the transit delay constraint. We change the conditional so that either the buffer produces an output only once if it is full or if the head of the queue of messages has been delayed by an amount in the interval $[2,5]$. During this interval, the buffer may also accept further inputs. Thus the buffer $Buff'$ now can be described by

$$Buff' \mathrel{\widehat{=}} Y_{\langle\rangle}$$

where

$$
\begin{array}{lcl}
Y_{\langle\rangle} & = & E_{in}.in?x \bowtie v_i \rightarrow Y_{\langle\langle val(v_i),x\rangle\rangle} \\
Y_{\langle\langle t,y\rangle\rangle^\frown S} & = & \textbf{if}\ \#(\langle\langle t.y\rangle\rangle^\frown S) < 128 \\
 & & \quad \textbf{then}\ \ \textbf{if}\ oldest(\langle\langle t,y\rangle\rangle^\frown S) < 2 \\
 & & \quad\quad \textbf{then}\ E_{in}.in?x \bowtie v_i \rightarrow Y_{\langle\langle t,y\rangle\rangle^\frown S^\frown\langle\langle val(v_i),x\rangle\rangle} \\
 & & \quad\quad \textbf{else}\ \ \textbf{if}\ 2 \leq oldest(\langle\langle t,y\rangle\rangle^\frown S) \leq 5 \\
 & & \quad\quad\quad \textbf{then}\ \ E_{in}.in?x \bowtie v_i \rightarrow Y_{\langle\langle t,y\rangle\rangle^\frown S^\frown\langle\langle val(v_i),x\rangle\rangle} \\
 & & \quad\quad\quad\quad \Box\ \ E_{out}.out!y \bowtie v_o \rightarrow Y_S \\
 & & \quad\quad\quad \textbf{else}\ TooOld \rightarrow STOP \\
 & & \quad\quad\quad \textbf{fi} \\
 & & \quad\quad \textbf{fi} \\
 & & \quad \textbf{else}\ Full \rightarrow STOP \\
 & & \quad \textbf{fi}
\end{array}
\tag{13}
$$

where the function *oldest* takes returns the "age" of the head of the queue of messages by comparing the time at which it arrived to the current time. Items are queued as pairs $(time, event)$ representing the time at which they arrive. The events *TooOld* and *Full* are used to signal the error conditions encountered due to the finite size of the buffer and the differing input and output timing constraints. The function *val* returns the value of a variable.

This buffer will break when the head of the queue of messages is too old, or if the buffer is full. However, it is possible for some small value of n that the combination of transit delay, input and output rate restrictions will cause it to break after only a few messages have been sent.

4 Conclusions

Using a *WAIT* operation to describe interevent timings may be semantically elegant, and offer a sound and complete model of some systems. However, its use in specifying a commonly found realistic system such as that presented in this paper has been found to be inadequate (at worst) or clumsy (at best). In the case presented, the primary issues of separate input, output and throughput measures are clouded by having to deal with an awkward notation.

On the other hand, the use of event enabling intervals which are functions of a set of marker events allows arbitrary event timings to be easily and naturally described. The example presented in this paper highlights the ease of use of the notation. It has been the author's experience that these extensions allow concise and natural descriptions of a wide range of systems (clocks, time-out timers, multiplexers, other buffers, and a real-time limited stop and wait protocol. [13, 12]).

However, the notation does not offer a complete semantics as yet, and only partial results may be formulated. This clearly defines the task ahead if this notation is to develop into a useful real-time Formal Description Technique. The primary priority is the development of a complete semantic model for the extensions. Secondly, if the notation is to be used in describing performance issues (such as "a message will be delivered within five seconds of being sent with a 95 % probability" or "a database query will fail only 1 % of all transactions") then the notation needs to allow event enabling intervals to be *stochastic functions of a set of marker variables.* Again, this is the subject of ongoing work presented in the author's thesis [13].

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. A brief history of Timed CSP. Technical report, Programming Research Group, Oxford University, Oxford OX1 3QD UK, 1992.

[4] D. Ferrari. Client requirements for real-time communication services. Published as part of the Internet Network Working Group Request for Comments (RFC), number 1193 (RFC1193)., 1990 November.

[5] R. Gerber, I. Lee, and A. Zwarico. A complete axiomatization of real-time processes. Technical Report MS-CIS-88-88, Dept. of Computer and Information Science, School of Engineering and Applied Sciences, Uni. of Pennsylvania PA 19104, November 1988.

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

[8] 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.

[9] Juan Quemada and Angel Fernandez. Introduction of quantitative relative time into LOTOS. In H. Rudin and C.H. West, editors, *Protocol Specification, Testing, and Verification, VII*, pages 105–121. Elsevier Science Publishers B.V., 1987.

[10] 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.

[11] J.J. Zic. A New Communication Protocol Specification and Analysis Technique. Technical Report TR287, Basser Department of Computer Science, July 1986.

[12] John J. Zic. Using CSP+T to describe a stop-and-wait protocol. Submitted to the 1993 International Conference on Network Protocols (ICNP'93), Feburary 1993.

[13] 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.