Time-Constrained Buffer Specifications in CSP+T and Timed CSP

JOHN J. ŽIC
University of New South Wales

A finite buffer with time constraints on the rate of accepting inputs, producing outputs, and message latency is specified using both Timed CSP and a new real-time specification language, CSP+T, which adds expressive power to some of the sequential aspects of CSP and allows the description of complex event timings from within a single sequential process. On the other hand, Timed CSP encourages event-timing descriptions to be built up in a constraint-oriented manner with the parallel composition of several processes. Although these represent two complementary specification styles, both provide valuable insights into the specification of complex event timings.

Categories and Subject Descriptors: B.4.4 [**Input / Output and Data Communications**]: Performance Analysis and Design Aids—*format models*; D.2.1 [**Software Engineering**]: Requirements/Specification—*languages*

General Terms: Design, Languages, Performance

Additional Key Words and Phrases: Real-time algebraic languages

1. INTRODUCTION

There has been considerable effort recently in extending Hoare's [1985] CSP and Milner's [1980; 1989] CCS 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.

We propose some informal time (and probability) extensions to CSP in Žic [1986] where a special *DELAY* process allowed temporal separation between any two successive events. At approximately the same time, Reed and Roscoe [1986] introduced a similar special process *WAIT* into CSP and provided a complete semantics based on timed failures for their Timed CSP language. Gerber et al. [1988] introduced a *timed-action* operation (effectively, a time was associated with each prefix operation) to separate adjacent events temporally into their extended CSP. They also provided a Timed-Acceptance semantics for the language. Quemada and Fernandez [1987] proposed an extension

Author's address: School of Computer Science and Engineering, University of New South Wales, NSW 2052, Australia; email: johnz@cs.unsw.oz.au.

to the LOTOS specification language [Brinksma 1987] by associating an enabling interval with each event. This time interval represents the time over which a process may engage the event.

There are difficulties in describing and specifying complex system timings using this type of construction, where an event timing is defined solely by its immediate predecessor. Complex timings and the ability to define the future behavior of a system will inevitably rely on a set with more than one element of preceding events in the process' execution. Furthermore, these events may have occurred a long time in the past execution of the process. To address these problems, Žic [1991] proposed an extended CSP called CSP + T that associated an enabling interval with each event and allowed this interval to be expressed as a function of one or more *marker events*.

It will be seen that the CSP + T approach differs fundamentally from the Timed CSP solutions. Timed CSP captures timing constraints by the parallel composition of a set of processes, each of which describes a specific timing constraint. These timing constraints are viewed as representing timed refinements of the system and facilitate the specification and proof of system timings. However, the algebraic manipulation of complex timing requirements from a parallel composed system into, essentially, a simpler sequential system may, in some cases, be impossible without extending Timed CSP to allow a way of recording and using the time at which specific events occurred in the system's execution.

On the other hand, CSP + T describes timing constraints of sequential processes in sequential terms as much as possible, reducing the need for adding any parallel processes to express timing constraints. Furthermore, the additional expressive power of the language now allows some complex parallel systems to be rendered as sequential processes, something that could not be done with only interevent timing constructs such as *DELAY*.

This article is organized as follows. First, we present the problem, which is the modeling of a store-and-forward communication system with a specific quality of service requirements. This is done by representing the system as a finite-length buffer with appropriate input, transit, and output timing requirements. Second, we present the CSP extensions for CSP + T. Third, we develop a solution in terms of existing Timed CSP algebra. Finally, we present an alternative solution using the CSP + T notation.

2. THE PROBLEM

A store-and-forward communication network may be abstractly represented by a finite-length 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.

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 and average message delays, throughput, reliability, probability of loss of a message, and other client requirements may be important [WG6 1986; Ferrari 1990]. This article attempts to describe a communication system with the following characteristics given by a (somewhat) naive client:

(1) up to 128 messages in transit at any time;
(2) message latency in the range of 2 to 5 time units;
(3) message input rate set to 1 message per time unit; and
(4) message output rate of 1 message per two time units.

As these timing constraints stand, there will be problems with any implementation. First, 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 requirements. Second, the output must simultaneously satisfy both the transit delay requirements and the output timing requirement for a given (fixed) input timing. Eventually, all of these conditions cannot be simultaneously satisfied, and the system in some way fails.

3. A BRIEF DESCRIPTION OF THE EXTENSIONS

The CSP + T *syntax* is a superset of the basic untimed deterministic CSP syntax presented by Hoare [1985]. The fundamental changes to the untimed algebra are the following:

(1) A new event, ★, is introduced to denote process instantiation into both the algebra and traces model.
(2) A new event operator $\bowtie$ is introduced, which is used in conjunction with a variable to record the time at which an event occurs. These times are taken from the set of positive Real numbers, with successive event times forming a monotonic nondecreasing sequence. We allow any number of successive events in a single process trace to have the same time. It is the designer's responsibility to mention explicitly any limitations on the number of computations done over any particular time period, including zero.
(3) Each event now has a time interval associated with it. This time interval represents a choice at which times the event must occur. These intervals are continuous and are usually expressed *relative* to a set of events.
(4) Only deterministic timed processes can be described in the algebra to date. A semantics for processes making an internal or nondeterministic choice is currently being formulated.

The major change to the traces model is that events are now *pairs*, *t.e*, where *t* is the global *absolute* time at which event *e* is observed. This is the same as Timed CSP traces model with the addition of the new instantiation event. We consider each of the above items in turn.

3.1 The Process Instantiation Event

Each system of process definitions requires that it is *instantiated* before it can execute. As such, a special process instantiation event denoted ★ (star) is introduced into the algebra and the corresponding traces model. This event is unique in that it must be associated with a unique, global time. It represents the global time at which the system of processes may start.

Consider a process that engages in a single event a, then breaks. This process may be defined in untimed CSP by $P = a \rightarrow Stop$. The traces set of this process is

$$\{\langle\rangle, \langle a\rangle\}.$$

In CSP + T, we prefix this process with the instantiation event in order to allow it to execute. If the timed version of P, called P', is instantiated at time 1, then we have

$$P' = 1.\bigstar \rightarrow a \rightarrow Stop,$$

and the timed traces set of P' is

$$\{\langle\rangle, \langle 1.\bigstar\rangle, \langle 1.\bigstar, s.a\rangle\},$$

where $s \in [1, \infty)$. Notice that event a occurs only once in the interval $[1, \infty)$ for any particular process execution. Also, the times in traces descriptions are *absolute*, where the time intervals in the process description are *relative*.

3.2 The Time Capture Operator

A new event operator $\bowtie$ is introduced so that 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. So, for example, if we have a process instantiated at time 1, which behaves as $1.\bigstar \rightarrow a \bowtie v \rightarrow Stop$, then the variable v holds the *global* time at which event a occurs. In this case, the time at which event a occurs, and hence, the time recorded in the variable v, is going to be greater than or equal to the process instantiation time of 1.

Variables associated with the time capture operator, or *marker variables*, have their scope limited to a strictly sequential process. They cannot be referenced or accessed in any other way across parallel processes. Further limitations occur in the use of the variable if the event (associated with a marker variable) is the subject of a restriction operation.

Finally, marker variables may be initialized by using the time capture operator on any event that has a well-defined, global time associated with it. Typically, this is the process instantiation event. For example,

$$1.\bigstar \bowtie \{u, v, x, y\} \rightarrow a \bowtie v \rightarrow Stop$$

initializes variables u, v, x, and y to the process instantiation time of 1. The variable v is then used to reference the time at which event a occurs. The references to the instantiation event as it is initialized is replaced by the reference to the time at which event a is observed.

3.3 Event-Enabling Interval

Each event in CSP + T is associated with a time interval, whether or not it is explicitly used. This represents the time over which the current event is regarded as being available to the process and its environment, relative to some preceding event from its current execution. In effect, the event-enabling interval may be regarded as equivalent to a deterministic choice construction, with event labels drawn from the dense (Real number) interval defining the times at which the event may occur. Furthermore, if the event has not occurred by the end of its enabling interval, the process withdraws effectively its offer to engage in that event. The process behaves as *Stop* if it cannot engage in an alternative event, after the "expiration" of the current event.

For example, an event a that has an enabling time interval of $[0, 1]$ is written as $[0, 1].a$ and must occur only *once* in the specified time interval, between 0 and 1 time units after a's preceding event. Another event $[1, 1].b$ must occur precisely at one time unit immediately after its preceding event. An example process that uses event-enabling intervals is $0.\bigstar \rightarrow [1, 2].a \rightarrow$ *Stop*. This is a process that will engage in a single a event only between 1 and 2 time units since it was instantiated (at time 0), and then broken.

If a time interval is not explicitly mentioned with an event, the least defined interval $[0, \infty)$ is assumed. That is, the event associated with this interval is allowed to occur at any time after the immediately preceding event.

These intervals are defined in terms of functions over a set, including the empty set, of marker variables. When there are no marker variables referenced, then the enabling interval is defined for the immediately preceding event, as above. More typically, however, the expression is given in terms of one or more marker variables. For example, a clock is instantiated at time 0, then *ticking* once every time unit after that, may be defined by

$$RealClk \mathrel{\hat{=}} 0.\bigstar \bowtie v \rightarrow TimedClk$$
$$TimedClk = \mu X \bullet E.tick \bowtie v \rightarrow X$$

where $E = \{s | s = rel(1, v)\}$, and the *rel* function is defined as follows. If the preceding either event, reference or marker, occurs at time t_0, then $rel(x, v)$ 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.

The use of these enabling intervals presents two major sets of questions regarding parallel and sequential composition. For example, how can processes that use these enabling intervals be composed with each other in parallel? Related to this, what other influences may determine when an event occurs within its enabling interval? We consider each of these points in the following discussion.

3.3.1 *Process Synchronization.* Consider the two simple processes $P = E_1.a \rightarrow P$ and $Q = E_2.a \rightarrow Q$, with identical untimed alphabets. Now, suppose we compose these two in parallel as $0.\bigstar \rightarrow P \| Q$. The semantics of this composition are dependent on whether the values taken by E_1 and E_2 are identical, partially overlapping, or disjoint. If two intervals are identical, the composite process engages in a single synchronized event a within the interval. There are two possible behaviors for the parallel composition when the intervals become partially overlapping. We may choose to have the processes engage independently on events outside the intersection of intervals E_1 and E_2 and synchronized within the intersection. Alternatively, we could completely disallow any events to be engaged that lie outside the intersection of the two enabling intervals. This latter case was selected in the original language design because it offers a simpler semantics. In summary:

(1) If $E_1 = E_2$, the processes synchronize on the a event once during this interval.
(2) If $E_1 \neq E_2 \wedge E_1 \cap E_2 \neq \{\}$, then the processes only synchronize during the interval $E_1 \cap E_2$. The system withdraws offers of engaging in events outside this time.
(3) If $E_1 \cap E_2 = \{\}$, then there are no times at which the processes may synchronize, and the composition behaves as *Stop*.

Additionally, we observe the following about synchronization between processes:

(1) If the process' untimed alphabets are disjoint, the event-enabling intervals play no direct role except to note that the resulting interleavings must be ordered so that the sequence of times in any timed trace are monotonic and nondecreasing. For convenience, we call this property **mndt**.
(2) If the untimed alphabets are partially disjoint, synchronization depends both on the set of events in the intersection of the untimed alphabets and consideration of the event-enabling intervals. Events outside the intersection set of the untimed alphabet lead to interleavings possessing the **mndt** property.
(3) Explicit communication between processes via a channel may occur only if the sender and receiver processes have enabling intervals that intersect. For example,

$$(E_1.c!v \rightarrow P) \| (E_2.c?x \rightarrow Q(x))$$

will lead to a communication event $c.v$ in the interval defined by $E_1 \cap E_2$. If E_1 and E_2 are disjoint, with no other events offered, then the communication fails, and the system stops. By setting the enabling interval on reception to $[0, \infty)$ and the sending process to E, the system's timing behavior is set by the sending process, and the communication occurs during E.

3.3.2 *Sequential Concerns*

3.3.2.1 *Prefix.* Consider the process

$$P = a \bowtie v \rightarrow b \rightarrow E.c \rightarrow Stop,$$

where $E = \{t | rel(3, v) \leq t \leq rel(5, v)\}$. Because the c event must occur at any time from 3 time units to 5 time units after the occurrence of the a event, the time at which event b occurs must also be less than or equal to the maximum of c's enabling interval. If this is not the case, the process breaks immediately after engaging the b because there is no way that c could be engaged by the process. Hence, this process has two separate behaviors dependent on the time at which event b occurs relative to the event c's enabling interval. Let $\lceil I \rceil$ represent the upper bound on an interval I; then process P may be rewritten as:

$$\begin{aligned} P = a \bowtie v \rightarrow {} & [0, \lceil E \rceil].b \rightarrow E.c \rightarrow \mathit{Stop} \\ & \Box \\ & (\lceil E \rceil, \infty).b \rightarrow \mathit{Stop} \end{aligned}$$

Although it is possible to transform the generalizations of this case, such constructions should be avoided. As a general design principle, the timing intervals of a purely sequential process, consisting of prefix and choice operations, should be such that the entire expression does not abort due to enabling intervals alone. Any deliberate expression abortion will be due to other causes, such as parallel composed processes or explicitly designed timing exceptions.

3.3.2.2 *Choice.* The prefix operation is regarded as a base case of the more general deterministic, or menu, choice operations. Choice in the CSP + T algebra selects from a finite subset of events from the untimed alphabet. However, as each event is associated with an enabling interval, this choice is between a possibly infinite set of timed events.

Choice between a set of events with disjoint enabling intervals is made according to the natural time order. That is, a given choice set at one point in time reduces to a smaller subset as time progresses. Events "expire" and are removed from the choice set. For example, assume that the original choice set at time 0 is $\{[1,1].a, [2,3).b, [4,5].c\}$. If the process does not engage in event a at time 1, then the choice set is reduced to $\{[2,3).b, [4,5].c\}$. Then, if the process does not engage in event b, the choice set is reduced to $\{[4,5].c\}$. If time progresses, with the process failing to engage in event c during the specified time interval, then no further choices may be made, and the process behaves as *Stop* from that point on. This should be regarded as a mistaken construction. Deterministic timed choice requires that at least one of the choices is taken during any process execution. Furthermore, each of the choices should be *distinct*, paralleling the untimed CSP model. Distinct timed events have either disjoint enabling intervals or disjoint untimed events, or both.

3.4 Deterministic Process Descriptions

At present, CSP + T can only describe deterministic processes, and describing nondeterministic process behaviors is part of ongoing work. The lack of nondeterminism means that the current language is limited as a *specification* technique, but not as an *implementation* technique. It is well known that nondeterministic process descriptions may be viewed as a way of underspecifying process behaviors, (whether timed or untimed). On the other hand, implementations are seldom nondeterministic:

of course, $\sqcap$ is not intended as a useful operator for *implementing* a process... The main advantage of nondeterminism is in *specifying* a process [Hoare 1985, p. 102]

4. DESCRIBING THE SYSTEM USING TIMED CSP

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

$$\mu X \bullet in?x \to out!x \to X \tag{1}$$

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 Eq. (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 $a \to b \to \cdots$ is differentiated from the sequence $b \to a \to \ldots$, despite both events being observed at the same global time, say, according to the observer's watch. If both a and b are observed at time 1, the former ($a \to b \to \cdots$) 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 $a \to b \to \ldots$ where event a occurs at time 0, for example, would allow event b to follow at any time taken from the half-open interval $[0, \infty)$ after the a, such as $\langle 1.a, 1.b \rangle$, $\langle 1.a, 25.b \rangle$, or even $\langle 1.a, (\pi \times 10^{129}).b \rangle$.

The proposed extended CSP (CSP + T) as well as Reed and Roscoe's [1986] Timed CSP use this latter view. 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, which has been subsequently modified to eliminate the system delay constant [Davies and Schneider 1992] so that any event timing must be explicitly described using a *WAIT* process.

Producing a buffer that delays each message by the required delay is straightforward in the following model:

$$SPB = \mu X \bullet in?x \to WAIT\, I \,;\, out!x \to X \tag{2}$$

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 to output timing is correctly defined, the spacing between an output and a following input is defied to occur at any time in the interval $[0, \infty)$. Timings among input to input, output to output, and input to output are *dependent on each other*.

The timings between inputs may be defined by *constraining* the above process by composing *SPB* in parallel with

$$IN = \mu X \bullet in?x \rightarrow WAIT\ 1\ ;\ IN \tag{3}$$

and similarly, the output may be constrained by the parallel composition of *SPB* with

$$OUT = \mu X \bullet out!x \rightarrow WAIT\ 2\ ;\ OUT. \tag{4}$$

Note that each of the buffers expressed in Eqs. (2), (3), and (4) implement only a single part of the required behavior. Furthermore 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 based on (1). Instead, we use a finite-size buffer as a starting point. The buffer presented, *Buff*0, is based on the infinite buffer of example X9 in Hoare [1985, p. 138]:[1]

$$Buff0 \triangleq W_{\langle\rangle}$$

where

```
$W_{\langle\rangle} = in?x \rightarrow W_{\langle x \rangle}$                                    (5)
$W_S =$ if $\#S < 128$
     then $(in?x \rightarrow W_{S \frown \langle x \rangle} \square out!S_0 \rightarrow W_{S'})$
     else $out!S_0 \rightarrow W_{S'}$
     fi
```

This buffer, like any other implementation, must resolve which actions to take under "error" conditions such as buffer filling or messages arriving at an incorrect rate. The original naive specification did not point out which buffer behavior and event timings are acceptable when the buffer encounters these error conditions. Any correct implementation would resolve these issues at the specification stage.

In view of this discussion, let us modify the behavior of Eq. (5) so that once it fills, it engages in a *Full* event, presumably signalled to the buffer's environment, and then breaks:

$$Buff1 \triangleq X_{\langle\rangle}$$

where

```
$X_{\langle\rangle} = in?x \rightarrow X_{\langle x \rangle}$                                    (6)
$X_S =$ if $\#S < 128$
     then $(in?x \rightarrow X_{S \frown \langle x \rangle} \square out!S_0 \rightarrow X_{S'})$
     else $Full \rightarrow Stop$
     fi
```

[1]This article adopts the conventional notation if b then P else Q fi for the CSP conditional $P \not< b \not> Q$ where b is a boolean value, and P and Q are processes.

The input and output timing requirements for this buffer using Timed CSP are captured by using the parallel composition of processes defined in Eqs. (3) and (4). The transit delay constraint may be met by introducing another process into the parallel composition with the *Buff*1 process. This spaces inputs to outputs using the parallel composition

$$TD = in?x \rightarrow (WAIT[2,5]\ ;\ out!x \rightarrow Stop) \,|||\, TD \tag{7}$$

with the parallel composition done using an interleaving to prevent the system from deadlocking at the very first recursive call. Therefore, the system of processes

$$Buff1 \| IN \| OUT \| TD$$

gives the required timing characteristics for the buffer *provided that the buffer is not full.*

4.1 Discussion

This specification style is commonly referred to as being *constraint oriented*, and it is used extensively in both timed and untimed Formal Description Methods. Each constraint may be regarded as representing a refinement step moving from an untimed model to a timed model. Although this method is attractive in simple timing descriptions, it is our experience that the use of the *WAIT* construct may lead to awkward and unnatural formulations of complex timing relationships. Furthermore, the analysis of parallel composed systems, both timed and untimed, may require them to be reduced to equivalent sequential systems. However, processes such as

$$(a \rightarrow p) \| (WAIT\, n\ ;\ b \rightarrow Q) \tag{8}$$

cannot be reduced to a unique sequential process from within a model that defines timing properties solely using a WAIT-like operation. Rewriting Eq. (8) to a sequential form requires the specification of the future behavior of a process determined by the times at which preceding events occurred. In Eq. (8), the process behavior is determined by the time at which the first event occurs, and so recording this time in some manner is important for the "serialization" of the system. This is identified both by Schneider [1992] and Fidge [1993]. Schneider proposed a new operator to do this, and created a prenormal form for a Timed CSP language. Fidge defined a way of labeling events such that causal relationships may be expressed as directed graphs. This leads to a true concurrency semantics for a real-timed process calculus based on CCS.

The approach taken in CSP + T differs from these. Rather than attempting to reduce a set of concurrent processes to some simpler sequential forms, the proposed extensions provide more expressive power to the sequential aspect of CSP. This reduces the need for introducing additional parallel constraining processes that may be difficult to analyze and allows some algebraic manipulation of processes. Thus, Eq. (8) may be rewritten as:

$$\begin{array}{l} a \bowtie u \rightarrow (P \,\|\, (E_1(u).b \rightarrow Q)) \\ \square \\ b \bowtie v \rightarrow ((E_2(v).a \rightarrow P) \,\|\, Q). \end{array}$$

Note that the enabling intervals E_1 and E_2 have not been specified in this example, since we mean to illustrate that the future behavior of the system depends on specific events in the system's execution. Of course, this is only one of many interpretations that may be expressed by this notation. The enabling intervals may also be defined solely in terms of a single marker variable. Or, alternatively, they may both be functions of both marker variables. We now move back to the description of the store-and-forward communication system.

5. DESCRIBING THE SYSTEM USING CSP+T

Recall that we are trying to describe a finite buffer with specific input, output, and transit delay requirements. Because we do require that the buffer hold more than one item, we start again by using the buffer given by Eq. (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. Therefore, we associate a marker variable with the input and output in order to capture their respective event-enabling intervals. The enabling interval function for input is solely a function of the input marker variable. Similarly, the enabling interval for output is expressed in terms of the output marker variable. Let us assume again that the buffer engages in a final *Full* action before stopping when full.

Let E_ι represent the input enabling interval, and let E_o represent the output enabling interval. Then, we set

$$E_\iota = \{s \mid s = rel(1, v_\iota)\}$$
$$E_o = \{t \mid t = rel(2, v_o)\}.$$

The buffer stores input events as a queue of timestamped events of the form $(t_\iota.x)$, where t_i represents the time at which a message contained in x was received. Let *eventof* be a function that strips out the time component of any such message.

A buffer that implies the appropriate inputs and output timings is

$$Buff2 \triangleq Y_{\langle\rangle}, \tag{9}$$

where

```
$Y_{\langle\rangle} = E_\iota.in?x \bowtie v_\iota \rightarrow Y_{\langle v_\iota\, x \rangle}$
$Y_S =$ if $\#S < 128$
    then $E_\iota.in?x \bowtie v_\iota \rightarrow Y_{S \frown \langle v_\iota\, x \rangle}$
         $\square$
         $E_o.out!eventof(S_0) \bowtie v_o \rightarrow Y_{S'}$
    else $Full \rightarrow Stop$
    fi.
```

The transit delay constraint means all of the messages held in the buffer differ between two and five time units since the time that they were input to the time they are outputted, relative to the current time. We now add this constraint to *Buff* 2. We define *age* to be a function that returns the age of the message at the head of the queue, by comparing the message time with the current time:

$$Buff3 \triangleq Z_{\langle\rangle},$$

where

```
$Z_{\langle\rangle} = E_i.in?x \bowtie v_i \to Z_{\langle v_i\ x\rangle}$
$Z_S$ = if $\#S < 128$
        then if $age(S) < 2$
              then $E_i.in?x \bowtie v_i \to Z_{S \frown \langle v_i\ x\rangle}$
              else if $2 \leq age(S) < 5$
                    then $E_i.in?x \bowtie v_i \to Z_{S \frown \langle v_i\ x\rangle}$
                          $\Box$
                          $E_o.out!eventof(S_0) \bowtie v_o \to Z_{S'}$
                    else if $age(S) = 5$
                          then $[0, 0].out!eventof(S_0) \bowtie v_o \to Z_{S'}$
                          else $Stop$
                          fi
                    fi
              fi
        else $Full \to Stop$
        fi.
```

Notice that this description is more *prescriptive* and *sequential* in nature than is the previously given Timed CSP specification. Timed CSP specifications tend to be presented as parallel compositions of component processes with each component representing a separate timing constraint. The above CSP + T specification is more concrete (or less abstract) than the Timed CSP specification given earlier, since it gives the buffer's behavior under overflow conditions and messages being held in the buffer for too long.

An alternative, more *descriptive*, or abstract, specification in CSP + T could have used the transit delay constraint given in Eq. (7) in parallel with the *Buff* 2 process of Eq. (9):

$$Buff4 = Buff2 \parallel TD.$$

In this system, we use the marker events and enabling intervals to specify the input and output timing, and the transit delay requirement is specified as a constraining process on the *Buff* 2 process.

6. CONCLUSIONS

Timed CSP and similarly related specification techniques define system timing within a sequential process by using a specific interevent delay. More complex timing relationships can be described with processes that use delays. By composing them in parallel with each other, we produce a set of independent timing constraints. However, there are some systems that cannot be described in this way because the timing relationships are not independent of either one another or preceding events. Such systems of processes cannot be converted into equivalent sequential forms, as in Eq. (8). To deal with such systems, we need to increase the expressiveness of the specification language.

This article introduced a new real-time description language, CSP + T, which addresses these problems. CSP + T extends the untimed CSP language in two ways. First, all events have an enabling time interval, over which the event is expected to be observed only once during any particular execution. The second extension is that these time intervals may be expressed in terms of a set of arbitrary marker events within a process' execution.

In order to focus the discussion, we specified a naively defined time-constrained buffer, first in Timed CSP, then in CSP + T. As was observed in this example, the two specification styles differ. Timed CSP encourages the use of constraining processes composed in parallel to define event-timing relationships, whereas CSP + T encourages the use of a single, sequential process to define event timing.

We feel that although these styles are complementary, careful use of both approaches will prove beneficial in specifying complex system timings.

ACKNOWLEDGMENTS

I wish to thank Colin Fidge, Steve Schneider, Asis Goswami, Jacek Olszewski, and the anonymous referees for providing useful comments and insights on earlier drafts of this article.

REFERENCES

BRINKSMA, E. 1987. An introduction to LOTOS. In *Protocol Specification, Testing, and Verification*, H. Rudin and C. West, Eds. Vol. 7. Elsevier Science Publishers B.V., Amsterdam.

DAVIES, J. AND SCHNEIDER, S. 1992. A brief history of Timed CSP. Tech. Rep., Programming Research Group, Oxford Univ., Oxford, U.K.

FERRARI, D. 1990. Client requirements for real-time communication services. Internet RFC 1193. Network Working Group, Univ. of California at Berkeley. Available via ftp@archie.au:/rfc.rfc1193.txt.gz.

FIDGE, C. 1993. A constraint-oriented real-time process calculus. In *Formal Description Techniques*, M. Diaz and R. Groz, Eds. Vol. 5. North-Holland, New York, 363–378.

GERBER, R., LEE, L., AND ZWARICO, A. 1988. A complete axiomatization of real-time processes. Tech. Rep. MS-CIS-88-88 (Nov.), Dept. of Computer and Information Science, School of Engineering and Applied Sciences, Univ. of Pennsylvania, University Park, Pa.

HOARE, C. 1985. *Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall International Ltd., Hertfordshire, U.K.

MILNER, R. 1980. *A Calculus of Communicating Systems*. Lecture Notes in Computer Science, vol. 92. Springer-Verlag, New York.

MILNER, R. 1989. *Communication and Concurrency*. International Series in Computer Science. Prentice-Hall International Ltd., Hertfordshire, U.K.

QUEMADA, J. AND FERNANDEZ, A. 1987. Introduction of quantitative relative time into LOTOS. In *Protocol Specification, Testing, and Verification*, H. Rudin and C. West, Eds., Vol. 7. Elsevier Science Publishers B.V., Amsterdam, 105–121.

REED, G. AND ROSCOE, A. 1986. A timed model for communicating sequential processes. In *Automata, Languages, and Programming, 13th International Colloqium Proceedings*. Lecture Notes in Computer Science, vol. 226. Springer-Verlag, New York.

SCHNEIDER, S. 1992. Unbounded nondeterminism for real-time processes. Tech. Rep. TR-12 (July), Programming Research Group, Oxford Univ., Oxford, U.K.

WG6. 1986. Information processing systems—Open Systems Interconnection—transport service definition—Connectionless mode transmission. Standard ISO-8072-1986-Addendum 1 ISO, New York.

Žic, J. 1986. A new communication protocol specification and analysis technique. Tech Rep TR287 (July), Basser Dept. of Computer Science, Univ. of Sydney, Sydney, Aus.

Žic, J. J. 1991. CSP+T: A formalism for describing real-time systems Ph.D. Thesis, Basser Dept of Computer Science, Univ. of Sydney, Sydney, Aus

Received March 1993; revised April 1994; accepted June 1994