Introducing Monitoring Events to Timed-CSP

Nicholaos Petalidis
Riverstone Networks
Unit 1, Heron Industrial Estate
Reading
RG7 1PG, UK
n.petalidis@computer.org

Deshinder S. Gill
University of Brighton
School of Engineering
Cockcroft Building
BN2 4GJ, UK
d.s.gill@brighton.ac.uk

Abstract

Timed-CSP is a process algebra designed to help in the modelling and analysis of real-time concurrent systems. Timed-CSP can handle synchronisation events that require the cooperation of all the interested parties and broadcasting events which do not require any cooperation from the environment. This paper argues that, still, there are scenarios that can not be adequately modelled in Timed-CSP and proposes an extension that would allow the modelling of more advanced communication mechanisms, like multi-casting.

1 Introduction

In most of the established process algebras, like CSP [**?**] and CCS [**?**], and specification languages based on process algebras, like LOTOS [**?**], the occurrence of an event requires the participation of all the interested parties. This approach allows only for synchronisations between peer processes.

The typical example used when introducing the synchronisation mechanisms of process algebras is that of a vending machine. The system includes a customer and a vending machine which are usually modelled by two different processes running in parallel. A customer needs to press a button in order to receive a cup of coffee. The process algebra approach in synchronisation is that the occurrence of the event "pressing of a button" requires cooperation by both the customer and the vending-machine processes and can not take place if either of them is not ready to perform that event.

Unfortunately, this approach is inadequate for modelling of communication mechanisms like broadcasting and multicasting. Broadcasting is a mechanism that is usually employed by the lower levels of communication protocols. It is, actually, a "one to everyone" communication, where there is a single sender who transmits information without being interested on whether this information will be received by anybody. Multicasting, on the other hand, is a feature of a number of industrial communication protocols, like Foundation Fieldbus [**?**], and other newer generation protocols, like XTP [**?**]. A multicast communication is a "one to many" or a "many to many" communication where information is usually transmitted by one or more nodes and must be received by a number of other nodes. Moreover, there can be a set of nodes which may or may not receive the information but whose presence is not necessary for the communication to take place.

The inadequacy of process algebras in the modelling of broadcasting has so far been addressed by only a small number of theoretical models. In [**?**] a calculus is presented where messages are broadcasted and received instantaneously by all nodes, whereas in [**?**, **?**] an extension of the process algebra Timed-CSP is proposed that allows the modelling of broadcasting signals. Signals are events that propagate through the parallel combination of processes but do not require any external synchronisation in order to occur.

Unfortunately, none of the current algebraic models can adequately support the multicast communication mechanism, i.e. a communication where m particular processes must agree for an event to take place, and $n \geqslant m$ processes eventually participate in that event. Therefore, in this paper, an extension to Timed-CSP is proposed that is able to handle multicast communications.

The rest of the paper is organised as follows: section **??** presents the rationale behind the introduction of monitor events and argues that the specification of a class of problems can benefit from it. An informal presentation of the extended Timed-CSP language appears in section **??** whereas the formal semantics of the operators can be found in section **??**. In section **??** it is shown that the model is an extension of the basic Timed-CSP Timed Failures model. Examples of how the new operators can be employed are presented in section **??** and, finally, conclusions are drawn in section **??**.

2 Monitors in Timed-CSP

The class of problems that the proposed extension attempts to address can be summarised in the following example.

A system that consists of a vending machine that sells coffee and a number of customers that occasionally order coffee needs to be modelled. In the classical approach of Timed-CSP and CSP this system is usually denoted by two processes, say *Customers* and *VMachine*, which communicate through events *button* and *coffee*. In the Timed-CSP terminology this system can be specified as:

$$System \stackrel{def}{=} Customers \ |[A \,|\, B]| \ VMachine$$

The definition above specifies that the system consists of two processes. Process *Customers* models the set of customers that use the vending machine and process *VMachine* models the vending machine. These processes run in parallel and must cooperate on the events drawn from the intersection of the two sets A and B that appear inside the parallel operator "$\|$". It is assumed here that both sets are the same and consist only of the events $\{button, coffee\}$. In fact, because in this case the sets are identical, the notation:

$$System \stackrel{def}{=} Customers \parallel VMachine$$

can be used, that denotes that the two processes synchronise in every event.

Event *button* models the pressing of a button and event *coffee* models the delivery of coffee by the vending machine. According to the semantics of Timed-CSP, in order for the event *button* to take place, both processes must agree to offer that event. If any of the two processes is not ready to offer that event, then no synchronisation can take place.

Assume now that the system needs to be extended so that the company that distributes the vending machine is taken into account. Moreover, assume that the company needs to do surveys from time to time in order to produce statistics about the usage of its vending machines. Thus it needs to *monitor* occasionally the events *coffee* and *button*. A solution similar to the one presented before can not really capture the new requirements:

$$System \stackrel{def}{=} (Customers \parallel VMachine) \ |[A \,|\, C]| \ Company$$

Here the new process *Company* models the presence of the company that monitors the sales, and C is some set of actions that the company is performing that includes event *coffee*. The introduction of the new process changes the requirement that only two processes are needed for the occurrence of the event *coffee*. Now all three processes are required in order for the event to take place, even though process *Company* only occasionally logs the sales and does not actively participate in the interactions between the customer and the vending machine.

Of course, a solution can be found to accommodate the new situation but this solution would require introduction of events that do not appear in the informal specification of the problem. Such events add complexity in the model and give a non-intuitive formal description of the system. For example, in the signal-model of Timed-CSP that allows broadcasting, the specification of the vending machine can be amended to emit a signal, whenever a transaction takes place. This signal can then trigger a synchronisation

inside process *Company*:

$$VMachine \stackrel{def}{=} button;\ coffee;\ \widehat{monitor};\ VMachine$$

$$Customers \stackrel{def}{=} button;\ coffee;\ Customers$$

$$Company \stackrel{def}{=} \ldots;\ monitor;\ \ldots$$

$$System \stackrel{def}{=} (Customers \parallel VMachine)\ |[A \mid C]|\ Company$$

where $A \stackrel{def}{=} \{button, coffee, \widehat{monitor}\}$ and $C \stackrel{def}{=} \{\ldots, monitor, \ldots\}$.

The problem with the above specification is that the introduction of the signal *monitor* in the description of the vending machine is alien to the machine's functionality; a vending machine should simply sell products. The specification of *VMachine* looks even stranger if it is re-used in environments where no monitoring is required. In such cases, the broadcasting of signal *monitor* would be totally incomprehensible.

The main drawback of the signal-model approach is that the monitoring of events has to be explicitly allowed by the components that participate in these events. An equivalent requirement in electronic-circuit design, for example, would be that the monitoring of the voltage across the circuit can not be performed unless explicitly allowed by special hardware on the circuit. In real life, however, the monitoring of events does not have to be explicitly allowed: events can be monitored by anyone unless explicitly hidden.

Thus, a more natural approach would be to leave the specification of the vending machine unaltered and simply specify that the process *Company* monitors the event *coffee*.

$$VMachine \stackrel{def}{=} button;\ coffee;\ VMachine$$

$$Customers \stackrel{def}{=} button;\ coffee;\ Customers$$

$$Company \stackrel{def}{=} \ldots;\ \overline{coffee};\ \ldots$$

$$System \stackrel{def}{=} (Customers \parallel VMachine)\ |[A \mid C]|\ Company$$

where $A \stackrel{def}{=} \{coffee, button\}$ and $C \stackrel{def}{=} \{\ldots, \overline{coffee}, \ldots\}$.

Unlike the signal-based approach, the specification of the process *VMachine* remains the same and it is only specified that process *Company* monitors event *coffee*, an operation denoted as $\overline{coffee}$. The intention behind the introduction of the special event $\overline{coffee}$ is that its presence is not necessary for the event *coffee* to take place. The parallel combination of *VMachine* and *Customers* can go ahead even if process *Company* is not ready to participate in the synchronisation of *coffee*. However, if event *coffee* takes place and process *Company* is ready to monitor it, then event $\overline{coffee}$ will also take place. This approach is more natural than the signal-based one: as long as an event takes place and it is not hidden, then it can be monitored; no other actions need to be taken by the participating components.

It should be pointed out here that the vending machine problem described above is not an artificial one but it appeared a number of times during the formalisation of systems using process-algebraic based approaches. For example, in [**?**] the specification of a real-life fieldbus scheduler was described and the problem of specifying a process that monitors link activity, in a way similar to the example above, appeared. The multicast mechanisms of XTP, a transport layer protocol, also present the same problem when they propose policies where n entities "listen" to a transmission but only $m \leqslant n$ entities are actually required in order for the transmission to take place. Note here that *m particular* processes must participate.

In order to be able to model the sort of multicast communications presented above an extension to Timed-CSP is proposed. This extension is developed in a manner similar to Timed-CSP's extension with signals that was presented in [**?**, **?**]. However, this time the semantics of the operators are different.

3 The language of Timed-CSP extended with monitors

Timed-CSP is a process algebra that can describe the behaviour of systems. A system is "an entity that may be considered to possess behaviour, to perform actions" [**?**]. Timed-CSP assumes that a system is made up of a number of *processes*, which may themselves be further divided into other sub-processes. The term *process* is used to denote the behaviour of an object, expressed in terms of the occurrence and availability of a set of *events*. This set of events is referred to as the process' *alphabet*. An *event* denotes an instantaneous interaction between a process and its environment.

Timed-CSP assumes that all events are drawn from a universal alphabet Σ. In order to accommodate the use of monitor events, another alphabet is added, the alphabet of monitor events, $\bar{\Sigma}$. Event $\bar{a} \in \bar{\Sigma}$ *monitors* event $a \in \Sigma$. Thus, the alphabet of the extended language of Timed-CSP is considered to be the set:

$$\check{\Sigma} \stackrel{def}{=} \Sigma \cup \bar{\Sigma} \cup \{\checkmark\}$$

The special event $\checkmark$ denotes successful termination and it is considered that does not belong to either Σ or $\bar{\Sigma}$. A different version of Timed-CSP with monitors could have considered an event $\bar{\checkmark}$ that could enable a process P to monitor the successful termination of another process Q.

From now on, sets denoted as $\check{A}, \check{B}, \ldots$ will contain events from both Σ and $\bar{\Sigma}$, sets denoted as $\bar{A}, \bar{B}, \ldots$ will contain only events from $\bar{\Sigma}$ and sets denoted as $A, B, \ldots$ will contain only events from Σ. Specifically if $\check{A} \subseteq \check{\Sigma}$ then the sets A and $\bar{A}$ will be such that:

$$A \stackrel{def}{=} \check{A} \cap \Sigma \qquad\qquad \bar{A} \stackrel{def}{=} \check{A} \cap \bar{\Sigma}$$

Timed-CSP has been successfully used in a number of cases (e.g. [**?**, **?**, **?**, **?**, **?**]) and therefore it would be desirable that any extension to it would not invalidate specifications written in the original language. Therefore, the extended language of Timed-CSP, which will be referred as $TCSP_{+mon}$, uses the same syntax and operators as with the original language with the extra addition of monitor events. The syntax and the operators are defined by the following BNF grammar rule:

$$\begin{aligned} P ::= & \mathit{Stop} \mid \mathit{Skip} \mid \mathit{Wait}\ t \mid a \rightarrow P \mid P;\ P \mid \\ & P \,\Box\, P \mid P \sqcap P \mid P \stackrel{t}{\triangleright} P \mid \\ & P \mid\mid [\check{A} \mid \check{B}] \mid\mid P \mid P \mid\mid\mid P \mid \\ & P \setminus \check{A} \mid f[P] \mid f^{-1}[P] \mid \\ & X \mid \mu X \bullet P \mid \\ & \bar{a} \rightarrow P \end{aligned}$$

By applying the above BNF rule, all of the traditional Timed-CSP *terms*, as presented in [**?**, **?**] can be derived. By adding the extra rule $\bar{a} \rightarrow P$ the terms of $TCSP_{+mon}$ can also be produced. The interpretation of most of the operators remains unchanged from the original Timed-CSP language.

Stop represents a process that has either deadlocked or livelocked. In the model considered here these two situations can not be distinguished. The *Wait t* term represents a process that just waits for time t and then exits successfully. The time domain is that of the non-negative real numbers. The successful termination of a process is denoted by the special event $\checkmark$. *Skip* is equivalent to process *Wait* 0.

The operator "$\rightarrow$" is the event prefix operator and denotes that process $a \rightarrow P$ is initially ready to participate in the event a and then behave as process P. It is assumed that this operator can be applied to monitor events as well. $\bar{a} \rightarrow P$ denotes that a process is monitoring event a. If the environment of the process performs event a then the process will start behaving as P. A form of these two prefix operators that introduces a delay can also be defined: $a \stackrel{t}{\rightarrow} P$ will delay for t time units to behave like P after it has performed a. A similar interpretation holds for $\bar{a} \stackrel{t}{\rightarrow} P$.

Processes are combined sequentially using the sequential composition operator "; ". Process $P;\ Q$ behaves initially like P. When P terminates successfully, control is passed to process Q.

The two operators "$\Box$" and "$\sqcap$" represent the notion of a choice. Process $P \Box Q$ represents a process that has the choice of behaving either as P or as Q. This choice can be influenced by the environment: if process P initially offers event a, whereas Q does not, then if the environment also offers event a, then it is process P that will be executed. On the other hand $P \sqcap Q$ represents, too, a choice between P and Q but here the choice is non-deterministic, i.e. it can not be influenced by the environment. Thus, in the previous example even if the environment had offered event a and nothing else it would not be guaranteed that process P would have been executed.

Timed-CSP also provides a timeout operator. $P \stackrel{t}{\triangleright} Q$ denotes a process which will behave like Q if P does not perform a synchronisation within time t. In case process P is offered a synchronisation just as time t expires, the choice between performing the synchronisation or behaving like Q is non-deterministic.

Processes can be combined in parallel using either of the parallel operators $\|$ and $|||$. The construct $P\ |[\check{A}\,|\,\check{B}]|\ Q$ denotes a process P which runs in parallel with process Q. The two parameters $\check{A}$ and $\check{B}$ are both subsets of the alphabet $\check{\Sigma}$, i.e. $\check{A} \subseteq \check{\Sigma}$, $\check{B} \subseteq \check{\Sigma}$. Process P can only engage in synchronisation of events that are present in the set $\check{A}$, whereas process Q can only engage in events from the set $\check{B}$. If the two sets $\check{A}, \check{B}$ are equal a shorthand notation $P\ \|\ Q$ may be used, whereas if their intersection is the empty set, the notation $P\ |||\ Q$ may be used.

In the traditional Timed-CSP, the two processes must agree on synchronisations that are drawn from the intersection of the two sets, $\check{A}$ and $\check{B}$. In the extended language that is presented here, such a synchronisation is required only if the events from $\check{A}$ and $\check{B}$ are drawn from the alphabet Σ. In traditional Timed-CSP, if the combination:

$$a \stackrel{1}{\rightarrow} b \stackrel{4}{\rightarrow} \mathit{Stop}\ |[\,\{a,b\}\,|\,\{a,b\}\,]|\ a \stackrel{2}{\rightarrow} b \stackrel{5}{\rightarrow} \mathit{Stop}$$

performs a at time t_0, it will not perform event b at time $t_0 + 1$ although one of the processes offers event b one time unit after the occurrence of a. If the environment of the parallel combination is willing to perform b at time $t_0 + 2$, then event b will occur at that time. One of the aims of this extension is to leave the semantics of the original algebra unchanged, so in $TCSP_{+mon}$ the interpretation of the parallel combination presented above will remain the same. However, the semantics of the operator will change if one of the

events involved comes from the alphabet $\bar{\Sigma}$. Assume the parallel combination of three processes:

$$(a \xrightarrow{1} b \xrightarrow{4} \mathit{Stop} \ |[\,\{a,b\}\,|\,\{a,b\}\,]|\ a \xrightarrow{2} b \xrightarrow{5} \mathit{Stop}) \ |[\,\{a,b\}\,|\,\{a,\bar{b}\}\,]|\ a \xrightarrow{5} \bar{b} \xrightarrow{5} \mathit{Stop}$$

In this case, if event a takes place at time t_0 and the environment is ready to offer event b at time t_0+2 then event b will take place at that time without waiting to synchronise with event $\bar{b}$. In other words, the trace:

$$\langle (t_0, a), (t_0+2, b) \rangle$$

is a possible trace of the parallel combination. Thus in the extended model presented here synchronisation events can go ahead without waiting for their corresponding monitor event.

On the other hand, in the parallel combination

$$a \xrightarrow{2} b \xrightarrow{4} \mathit{Stop} \ |[\,\{a,b\}\,|\,\{a,\bar{b}\}\,]|\ a \xrightarrow{1} \bar{b} \xrightarrow{5} \mathit{Stop}$$

if the environment is ready to offer event b at time t_0+2 then event b will take place; the occurrence of this event will trigger the monitor $\bar{b}$. One of the questions that arises is whether the parallel combination above should be considered as performing two events at time t_0+2, namely events b and $\bar{b}$, or just one, namely event b. In the first case, the trace would be:

$$\langle (t_0, a), (t_0+2, b), (t_0+2, \bar{b}) \rangle \tag{1}$$

whereas in the second case the trace would be:

$$\langle (t_0, a), (t_0+2, b) \rangle \tag{2}$$

In this model, the second option is chosen. The reason is that the inclusion of the monitor event, in the trace **??**, does not really give any more information about the process. This is because the presence of the timed-event (t_0+2, b) suggests that any monitor-events $\bar{b}$ that could occur at time t_0+2 did take place. Another way to see this is to consider the parallel combination of these processes with another process P. The inclusion of the timed-event $(t_0+2, \bar{b})$ can not give rise to any synchronisations or deadlocks that could not have occurred without this timed-event in the trace. For these reasons, it would be considered that in the case of the parallel combination the occurrence of a synchronisation event "conceals" the occurrence of the corresponding monitor event.

A different situation occurs when two parallel components are monitoring the same event. The intention is that they do not have to synchronise with each other and therefore one component may proceed even when the other does not monitor the event at the same times. However, if they both happen to monitor an event at the same time t and the corresponding synchronisation also takes place at time t then both of them will be able to monitor that event. Thus in the example below

$$(\bar{a} \xrightarrow{1} b \xrightarrow{4} \mathit{Stop} \ |[\,\{\bar{a},b\}\,|\,\{\bar{a},c\}\,]|\ \bar{a} \xrightarrow{2} c \xrightarrow{5} \mathit{Stop}) \ |[\,\{\bar{a},b,c\}\,|\,\{a,d\}\,]|\ a \xrightarrow{5} d \xrightarrow{5} \mathit{Stop}$$

the trace

$$\langle (t_0, a), (t_0+1, b), (t_0+2, c), (t_0+5, d) \rangle$$

is a possible trace of the parallel combination.

The interleaving operator "|||" is much simpler and it denotes that two processes evolve concurrently and no synchronisation is required between them.

Timed-CSP also provides a hiding operator that conceals certain events. The process $P \setminus \check{A}$ behaves exactly like process P but events in the set $\check{A} \subseteq \check{\Sigma}$ are hidden and occur as soon as possible, i.e. they do not require synchronisation from the environment. An event $\bar{a}$ that is hidden is constrained to monitor events only within process P.

The operators $f(P)$ and $f^{-1}(P)$ are renaming operators. Processes $f(P)$ and $f^{-1}(P)$ have the same structure as process P but events in P are renamed according to the alphabet mapping function f. It is assumed that the function does not rename monitor events to synchronisation events and vice versa.

Repeated behaviours can be specified by the recursion operator $\mu X \bullet F(X)$. This operator describes a process that behaves as $F(X)$, where X is the unique solution of the equation $X = F(X)$. Such a unique solution exists whenever recursive calls can not be made without some form of a positive time delay.

4 The semantics of $TCSP_{+mon}$

The operators of the original model have been given both a denotational semantics [**?**] as well as an operational semantics [**?**]. The denotational method will be followed in this report. In fact only minor modifications will be made to the semantics of the operators of the original Timed-CSP.

4.1 The denotational model

A number of semantic models have been defined for the language of Timed-CSP [**?**]. For most applications, however, where only safety and liveness conditions need to be captured, the timed failures model, M_{TF}, is adequate. In this model, a process is associated with a set of *traces* and a set of *refusals*. A trace records the events in which a process participated together with the times they occurred, and a refusal records the events which the process refused together with the periods for which these events were denied. For example, a process P that has performed events a and b at times 1 and 2 respectively, whereas it refused event b up until time 2, will include the pair $(\langle(1,a),(2,b)\rangle, [0,2) \times \{b\})$ in its semantic definition. In this pair, $\langle(1,a),(2,b)\rangle$ is the timed trace and $[0,2) \times \{b\}$ is the timed refusal. Therefore, in this model two processes are different if they have a difference in their traces or their refusals. A pair of a timed trace and a timed refusal will be called a *behaviour*.

In the language of $TCSP_{+mon}$ the alphabet of the original language has been extended, so it seems logical to extend the model M_{TF} in order to be able to tell when a process has monitored a particular event. Thus, the notion of a trace will be extended to include monitor events as well. The notion of a refusal will not change: monitor events occur only when their corresponding synchronisation occurs. In other words, a monitor event is refused only if its corresponding synchronisation is refused as well. Therefore the notion of a refusal does not need to change as it can easily be deduced that if event a was refused at time t then event $\bar{a}$ was also refused.

Note here that the initial motivation behind the introduction of monitor events was the ability to model processes that can monitor a system without altering its behaviour. The extension of the semantic model

does not invalidate this motivation. Indeed, it will be shown that the behaviour of $P \ |[\bar{\Sigma} \mid \Sigma]| \ Q$ will be indistinguishable from that of Q. The terms of the language of $TCSP_{+mon}$ will therefore be mapped in the model $M_{T\check{F}}$, an extension of the original timed failures model, M_{TF}, that cares for monitor events. Formal definitions for this model can be found in Table **??**.

In Table **??** the timed failures model is defined as a subset of the set of all possible processes, $S_{T\check{F}}$. This subset includes exactly those processes $S \in S_{T\check{F}}$ that satisfy a set of "sanity" criteria, the same criteria that hold for the original Timed-CSP processes. This set is reproduced in Table **??**. The only difference between the axioms of the original Timed-CSP and the ones presented below is in the fourth one, the so called *finite variability axiom*, which has been extended to care for the possibility of monitor events in the trace. Thus the axiom now states that there is always a refusal set $\aleph'$ that includes all of the refusal information for a process up to a certain time t. Moreover, if a timed event (t', a) does not belong to this refusal set and $t' \leqslant t$ then the trace of the process can be extended either by monitoring or by synchronising on event a at time t'. In general, the finite variability axiom restricts a process to change only a finite number of states in a finite amount of time.

The model $M_{T\check{F}}$ is a complete metric space under the distance function defined by Davies [**?**]:

$$\forall S_1, S_2 \in M_{T\check{F}} \bullet \check{d}(S_1, S_2) \stackrel{def}{=} inf(\{2^{-t} \mid S_1 \upharpoonright t = S_2 \upharpoonright t\} \cup \{1\}), t \in \mathbb{R}^+$$

The distance function presented above depends on the time at which two processes first exhibit different behaviours. The projection $S \upharpoonright t$ is defined as follows:

$$\forall t \in \mathbb{R}^+; \ S \in M_{T\check{F}} \bullet S \upharpoonright t \stackrel{def}{=} \{(s, \aleph) \mid (s, \aleph) \in S \wedge end(s, \aleph) \leqslant t\}$$

$S \upharpoonright t$ includes all the possible behaviours of S that do not extend after time t.

In Table **??** as well as in the following sections the usual operators applied to Timed-CSP traces and refusals will be used:

The $\aleph$ symbol denotes a refusal set, i.e. $\aleph \in TR$. The "$\frown$" operator denotes concatenation of traces. The "*begin*" operator returns the time of the occurrence of the first event in a timed trace and the operator "*end*" returns the time at which the last event of a trace took place. These two operators can also be extended in order to be applicable to refusal sets as well as timed traces. The *before* operator, "$\restriction$", restricts a trace, or a refusal, to the events before a certain time t. Similarly, the *after* operator, "$\upharpoonright$", restricts a trace or a refusal to events after a certain time, whereas the *during* operator, "$\Updownarrow$", returns the part of a trace or a refusal that lies in a specific time interval. The statement $s \cong w$ denotes that w is a possible permutation of s. The "$\#$" operator returns the number of events in a trace. The alphabet operator σ returns the events present in a timed trace or a refusal. The sub-trace relation of traces, "$\preccurlyeq$", relates a trace s_1 to a trace s_2 when all of the timed events of s_1 also belong to s_2. Finally, the restriction operator, "$\restriction$", restricts a trace or a refusal to a closed set of events,$[a, b]$. A strict version of the operator, $\Updownarrow$, can be used to restrict a trace or refusal to a half-open set of events,i.e. $[a, b)$. Table **??** provides the formal definitions for these operators.

Sometimes time-shift operators, like "$+$" or "$-$", will also be used. These operators shift a trace or a timed refusal by a specified amount of time.

4.2 The semantics of the operators

The semantics of the operators presented in section **??** can now be given. In order to accommodate the presence of variables the usual treatment of process variable bindings will be made. Thus, it is assumed that all variables are evaluated in a particular context, the environment of the variable. If *Env* is the domain of all mappings from variables to $M_{T\check{F}}$, i.e. *Env* includes all possible environments, then the function $F_{T\check{F}} \in TCSP_{+mon} \rightarrow Env \rightarrow M_{T\check{F}}$ will give the semantics of the operators of $TCSL_{+mon}$. The notation $F_{T\check{F}}[\![P]\!]\rho$ will denote the semantics of term P where all free variables X are substituted by the value they possess in environment ρ.

$TCSP_{+mon}$ has essentially the same operators as the original Timed-CSP. In fact only the semantics of the relabelling, hiding and parallel operators need to be amended and new semantics for the monitor event prefix operator need to be defined. Table **??** presents the semantics of the operators that remain unchanged from the original Timed-CSP.

Recursion is treated as in the usual language of Timed-CSP, where $X = F(X)$ defines X to be the (unique) solution of the equation $X = F(X)$. In particular, it can be proved [**?**] that such a solution exists for all time-guarded recursive equations, i.e. recursions which require at least time $t > 0$ in order to unfold. This solution is the unique fixed point of $X = F(X)$ in the metric space $M_{T\check{F}}$.

The monitor event prefix operator The monitor-event prefix operator $\bar{a} \rightarrow P$ is used to introduce monitor events in the description of a process. Process $\bar{a} \rightarrow P$ is monitoring event a. If this event occurs then it will start behaving like process P. Thus, the presence of monitor event $\bar{a}$ in the trace of this process would indicate that it successfully monitored that event. Like the prefix operator of the original Timed-CSP, this operator, too, is associated with a non-zero delay δ that corresponds to the time it takes for the process to change states.

$$F_{T\check{F}}[\![\bar{a} \rightarrow P]\!]\rho \stackrel{def}{=} \{(\langle\langle\rangle, \aleph) \mid a \notin \sigma(\aleph)\} \cup \{(\langle\langle(t,\bar{a})\rangle \frown s, \aleph) \mid t \geqslant 0 \wedge a \notin \sigma(\aleph \Vdash t) \wedge (s,\aleph) - (t+\delta) \in F_{T\check{F}}[\![P]\!]\rho\}$$

The relabelling operators The relabelling operators should not rename synchronisation events to monitor events and vice versa:

$$a \in \Sigma \wedge \bar{a} \in \bar{\Sigma} \Leftrightarrow f(a) \in \Sigma \wedge f(\bar{a}) \in \bar{\Sigma}$$

The semantics of $f(P)$ and $f^{-1}(P)$ can now be given:

$$F_{T\check{F}}[\![f^{-1}(P)]\!]\rho \stackrel{def}{=} \{(s,\aleph) \mid (f(s), f(\aleph)) \in F_{T\check{F}}[\![P]\!]\rho\}$$
$$F_{T\check{F}}[\![f(P)]\!]\rho \stackrel{def}{=} \{(f(s),\aleph) \mid (s, f^{-1}(\aleph)) \in F_{T\check{F}}[\![P]\!]\rho\}$$

The relabelling function respects monitoring and thus it can be assumed that:

$$f(\bar{a}) = \overline{f(a)}$$

The hiding operator The hiding operator is used to conceal certain events from the environment of a process. Process $P \setminus \check{A}$ behaves like process P but events from the set $\check{A}$ are hidden. Moreover events that belong to A occur as soon as P is ready to offer them. Thus, the behaviour of the hiding operator for synchronisation events is identical to that of the traditional Timed-CSP. If a monitoring event belongs to the set $\check{A}$ it is simply removed from the trace of the process. In this way, hiding of monitor events does not alter the behaviour of ordinary processes but restricts the scope of monitoring.

Note here that if an event $a \in \check{A}$ is hidden all events that monitor a will also occur as soon as possible, since a occurs as soon as possible.

The semantics of the hiding operator can now be given:

$$F_{T\check{F}}[\![P \setminus \check{A}]\!]\rho \stackrel{def}{=} \{(s \setminus \check{A}, \aleph) \mid (s, \aleph \cup ([0, end(s,\aleph)) \times A)) \in F_{T\check{F}}[\![P]\!]\rho\}$$

In the above definition, $s \setminus \check{A}$ denotes a trace s where all events from the set $\check{A}$ have been removed. Formally, this operator can be defined as:

$$\langle\rangle \setminus \check{A} \stackrel{def}{=} \langle\rangle$$

$$(\langle(t,a)\rangle \frown s) \setminus \check{A} \stackrel{def}{=} \begin{cases} s \setminus \check{A}, & a \in \check{A} \\ \langle(t,a)\rangle \frown (s \setminus \check{A}), & a \notin \check{A} \end{cases}$$

$$(\langle(t,\bar{a})\rangle \frown s) \setminus \check{A} \stackrel{def}{=} \begin{cases} s \setminus \check{A}, & \bar{a} \in \check{A} \\ \langle(t,\bar{a})\rangle \frown (s \setminus \check{A}), & \bar{a} \notin \check{A} \end{cases}$$

The requirement that $(s, \aleph \cup ([0, end(s,\aleph)) \times A)) \in F_{T\check{F}}[\![P]\!]\rho$ ensures that all synchronisation events from the set A are constantly refused and thus they occur as soon as possible. To see this, assume that hidden event a becomes available at time t. By the finite variability hypothesis, which is stated by axioms 4 and 6 in Table **??**, the number of events made available by P at time t should be finite. Since event a occurs as soon as possible, it is always on offer by the environment. Therefore P will perform all the possible synchronisations of a at time t. But then, since all synchronisations of a already took place at time t, any more offers at that time should be refused.

The parallel operator It is intended that synchronisation events will behave as in the original Timed-CSP but at the same time they will be propagated to all monitor events; monitor events will be triggered by their corresponding synchronisations. The occurrence of a synchronisation event a will allow any process that monitors that event to proceed. If a monitor event appears in the interface between two processes and the corresponding synchronisation of the monitor event is received then the occurrence of the monitor event will be concealed. This behaviour is analogous to that of the parallel operator in the signal model of Timed-CSP [**?**, **?**], $TS_{\check{F}}$, where a synchronisation event can be concealed by the occurrence of its corresponding signal event.

The notation $P \ |[\check{A} \,|\, \check{B}]| \ Q$ denotes that processes P and Q run in parallel and that P can engage in synchronisations from the set $\check{A}$ whereas Q can engage in synchronisations from the set $\check{B}$. If the sets $\check{A}$ and $\check{B}$ contain monitor events, then the set of events that can be monitored is also defined. It will be assumed

here that a process can not monitor and synchronise on the same event. For example, if sets $\check{A}$ and $\check{B}$ are defined as following:

$$\check{A} \stackrel{def}{=} \{\bar{a}, \bar{b}, c\} \qquad\qquad \check{B} \stackrel{def}{=} \{\bar{a}, b, c\}$$

then process P monitors events a and b whereas process Q monitors only event a. If Q performs event b then process P will be able to monitor it. However, event $\bar{b}$ will not appear in the trace of $P \ |[\check{A} \,|\, \check{B}]|\ Q$; only event b will. Monitor events can not synchronise with each other, so the monitoring of event a by process P is independent of the monitoring of the same event by process Q. The monitoring event $\bar{a}$ will be propagated to the environment of the two processes. If their environment offers at some point the synchronisation a, and both P and Q are ready to monitor that event, then both of them will. Finally, processes P and Q must synchronise on event c in the same way they do in the original model.

In order to define the semantics of the parallel operator, it is convenient to define beforehand the following operators:

$$\begin{aligned}
monit(\langle\rangle) &\stackrel{def}{=} \langle\rangle \\
monit(\langle(t,a)\rangle \frown s) &\stackrel{def}{=} \langle(t,\bar{a})\rangle \frown monit(s) \\
monit(\langle(t,\bar{a})\rangle \frown s) &\stackrel{def}{=} \langle(t,\bar{a})\rangle \frown monit(s) \\
monit(\check{A}) &\stackrel{def}{=} \{\bar{a} \in \bar{\Sigma} \mid a \in \check{A} \vee \bar{a} \in \check{A}\}
\end{aligned}$$

Operator *monit* returns the set of monitor events involved as either synchronisations or monitors.

$$sync(\check{A}) \stackrel{def}{=} \{a \in \Sigma \mid a \in \check{A} \vee \bar{a} \in \check{A}\}$$

$sync(\check{A})$ denotes the set of the synchronisation events present in a set $\check{A}$.

Note that:

$$\begin{aligned}
monit(B) &= \{\bar{b} \mid b \in B\} \\
monit(\bar{B}) &= \bar{B} \\
sync(A) &= A \\
sync(\bar{A}) &= \{a \mid \bar{a} \in \bar{A}\}
\end{aligned}$$

The semantic equation for the parallel combination $P \ |[\check{A} \,|\, \check{B}]|\ Q$ can now be given. First, it is required that each process does not offer the same event for both monitoring and synchronisation:

$$\begin{aligned}
&\nexists a \bullet a \in \check{A} \wedge \bar{a} \in \check{A} \\
&\nexists b \bullet b \in \check{B} \wedge \bar{b} \in \check{B}
\end{aligned}$$

The behaviour of the parallel operator when only synchronisation events are involved should remain identical to that of the original Timed-CSP. Thus, any synchronisation common to both components must be performed by both of them and any synchronisation that is exclusive to one component will be observed, if it is performed by that component. If s_P is a trace of P, s_Q is a trace of Q and s is a trace of the parallel

combination, then:

$$s \upharpoonright A = s_P \upharpoonright A \tag{3}$$

$$s \upharpoonright B = s_Q \upharpoonright B \tag{4}$$

Monitor events of P that are also synchronisation events of Q are removed from the trace and can only occur when Q offers the corresponding event. An analogous situation holds for monitor events of Q.

$$s_P \upharpoonright (\bar{A} \cap monit(B)) \subseteq monit(s_Q \upharpoonright (B \cap sync(\bar{A}))) \tag{5}$$

$$s_Q \upharpoonright (\bar{B} \cap monit(A)) \subseteq monit(s_P \upharpoonright (A \cap sync(\bar{B}))) \tag{6}$$

If $\bar{a}$ is a monitor event, then $(t, \bar{a})$ may appear in s_P only if (t, a) appears in s_Q. This will be true whenever $(t, \bar{a})$ is in the trace $monit(s_Q \upharpoonright B)$. Note that the fact that a single synchronisation event a at time t can cause more than one monitoring event to take place at the same time, excludes the use of equalities above.

If a monitor event appears in one of the sets $\check{A}$ or $\check{B}$ and it is not hidden by a corresponding synchronisation event, then it will also appear in the trace of the parallel combination. Any monitor events that are common to both components can occur independently of one another.

$$s \upharpoonright (\bar{A} \cup \bar{B}) \in s_P \upharpoonright (\bar{A} \setminus monit(B)) \;|||\; s_Q \upharpoonright (\bar{B} \setminus monit(A)) \tag{7}$$

In the above equation, the notation $s \;|||\; w$ denotes the set of all possible interleavings of the two traces s and w.

Events outside the sets $\check{A}$ and $\check{B}$ are proscribed.

$$s = s \upharpoonright (\check{A} \cup \check{B}) \tag{8}$$

A trace s belongs to the parallel combination of the traces of P and Q, $s_P \;|[\check{A} \,|\, \check{B}]|\; s_Q$, when all of the above conditions are met, i.e.:

$$s \in s_P \;|[\check{A} \,|\, \check{B}]|\; s_Q \Leftrightarrow (\mathbf{??}) \wedge (\mathbf{??}) \wedge (\mathbf{??}) \wedge (\mathbf{??}) \wedge (\mathbf{??}) \wedge (\mathbf{??})$$

Now that the exact form of the traces of the parallel combination was decided, the equations for the refusal sets must be derived.

Any synchronisation event that is present in set A should be refused by the parallel combination if it is refused by component P. A similar situation holds for events from the set B. Moreover, if a synchronisation event that is present in the set A or the set B was refused by the parallel combination then it must have been refused by at least one component:

$$\aleph \upharpoonright (A \cup B) = (\aleph_P \upharpoonright A) \cup (\aleph_Q \upharpoonright B) \tag{9}$$

All the synchronisation events that correspond to a monitor event and are exclusive to one component should be refused if that component refused them. The synchronisations that correspond to monitor events and are exclusive to component P belong to the set $sync(\bar{A}) \setminus sync(\check{B})$, whereas the synchronisations that correspond to monitor events and are exclusive to component Q belong to the set $sync(\bar{B}) \setminus sync(\check{A})$. On the other hand a synchronisation that corresponds to a monitor event is refused if it is not a synchronisation already in the sets A and B and

1. it is exclusive to one component and refused by that component, or

2. it is common to both components and refused by both of them

Let M be the set of synchronisations that correspond to monitor events and do not appear in the sets A and B, i.e. $M \stackrel{def}{=} ((sync(\bar{A} \cup \bar{B})) \setminus (A \cup B))$. Then

$$\begin{aligned} \aleph \upharpoonright M = & \aleph_P \upharpoonright (sync(\bar{A}) \setminus sync(\check{B})) \cup \aleph_Q \upharpoonright (sync(\bar{B}) \setminus sync(\check{A})) \\ & \cup (\aleph_P \cap \aleph_Q \upharpoonright (sync(\bar{A}) \cap sync(\bar{B}))) \end{aligned} \tag{10}$$

It is of course assumed that any events outside the two sets $\check{A}$ and $\check{B}$ are refused. It can now be stated that a refusal $\aleph$ is a refusal of the parallel combination of $\aleph_P$ and $\aleph_Q$ when:

$$\aleph \in \aleph_P \ |[\check{A} \,|\, \check{B}]|\ \aleph_Q \Leftrightarrow (\mathbf{??}) \wedge (\mathbf{??})$$

It is assumed that whenever a synchronisation takes place, all of the monitor events that monitor that particular synchronisation should be triggered. As explained in the section describing the hiding operator this will occur if and only if the timed synchronisation can be added to the refusal set of the process that contains the monitor events. Thus a behaviour $(s_P, \aleph_P)$ will be a behaviour of P in which all the available monitor events $\bar{a}$ are triggered at time t iff the timed event (t, a) may be added to the refusal set $\aleph_P$. Thus, another requirement governing the refusal sets of the two components is that:

$$\forall t \in \mathbb{R}^+ \bullet \sigma(s_Q \upharpoonright (B \cap sync(\bar{A})) \uparrow t) \subseteq \sigma(\aleph'_P \uparrow t) \tag{11}$$

$$\forall t \in \mathbb{R}^+ \bullet \sigma(s_P \upharpoonright (A \cap sync(\bar{B})) \uparrow t) \subseteq \sigma(\aleph'_Q \uparrow t) \tag{12}$$

The synchronisation events performed by Q at time t must be refused by component P at time t, if they are contained in the refusal set of process P at time t.

The semantic equation for the parallel operator can now be given.

$$\begin{aligned} F_{T\check{F}}[\![P \ |[\check{A} \,|\, \check{B}]|\ Q]\!]\rho \stackrel{def}{=} \{(s, \aleph) \mid & \exists s_P, s_Q, \aleph_P, \aleph_Q, \aleph'_P, \aleph'_Q \bullet \\ & s \in s_P \ |[\check{A} \,|\, \check{B}]|\ s_Q \wedge \aleph \in \aleph_P \ |[\check{A} \,|\, \check{B}]|\ \aleph_Q \\ & \wedge (\mathbf{??}) \wedge (\mathbf{??}) \\ & \wedge (s_P, \aleph_P \cup \aleph'_P) \in F_{T\check{F}}[\![P]\!]\rho \\ & \wedge (s_Q, \aleph_Q \cup \aleph'_Q) \in F_{T\check{F}}[\![Q]\!]\rho\} \end{aligned}$$

The interleaving operator, "|||", has a much simpler semantics:

$$\begin{aligned} F_{T\check{F}}[\![P \ ||| \ Q]\!]\rho \stackrel{def}{=} \{(s, \aleph) \mid & \exists s_P, s_Q \bullet s \in s_P \ ||| \ s_Q \\ & \wedge (s_P, \aleph) \in F_{T\check{F}}[\![P]\!]\rho \\ & \wedge (s_Q, \aleph) \in F_{T\check{F}}[\![Q]\!]\rho\} \end{aligned}$$

For the interleaving operator it is allowed that the alphabet of a process [**?**] may contain the same synchronisation and monitor events. The synchronisations and monitoring that the processes can perform are independent from one another. A monitor event is not triggered by its corresponding synchronisation event.

5 Monitors as an extension of the original Timed-CSP

It should initially be stated that the semantic equations are well-formed since they do respect the axioms of the model.

Theorem 1 *The semantic equations respect the axioms of the model*

Proof Most of the semantic equations of the extended algebra remain unchanged and identical to the operators of the original algebra. The proof for the operators that changed can proceed by structural induction. For example, assuming that P and Q are valid processes in the new model, it can be easily proved that $\langle\langle\rangle, \{\}\rangle \in F[\![P \ |[\check{A} \,|\, \check{B}]|\ Q]\!]$. □

Monitor events were introduced in order to simplify the specification of processes that "listen" to certain synchronisations but not actively participate in them. Therefore, if a process only monitors events then it should not alter the behaviour of the processes that monitors. Assume that a process P is such a process and that it monitors the synchronisations that another process, Q, undertakes. It would then be desirable that the behaviour of $P \ |[\bar{A} \,|\, A]|\ Q$ is identical to that of Q.

Lemma 1 $F_{T\check{F}}[\![P \ |[\bar{A} \,|\, A]|\ Q]\!]\rho = F_{T\check{F}}[\![Q]\!]\rho$, *where P is a process that can engage only in events from the set $\bar{A}$ and Q is a process that can engage only in events from the set A.*

Proof According to the semantic definition of the parallel operator that was given in the previous section, the traces of the parallel combination are determined by:

$$s \in s_P \ |[\bar{A} \,|\, A]|\ s_Q \Leftrightarrow (\mathbf{??}) \wedge (\mathbf{??}) \wedge (\mathbf{??}) \wedge (\mathbf{??}) \wedge (\mathbf{??}) \wedge (\mathbf{??})$$

Equation (**??**) is trivially true. The same holds for equation (**??**). From equation (**??**) it follows that:

$$s \upharpoonright \bar{A} \in \{\langle\rangle\}$$

and thus from equations (**??**) and (**??**) follows that

$$s = s \upharpoonright A = s_Q \upharpoonright A \tag{13}$$

Finally, from equation (**??**) it is required that:

$$s_P \upharpoonright \bar{A} \subseteq monit(s_Q \upharpoonright A) \tag{14}$$

On the other hand, the refusal sets of the parallel combination are these sets $\aleph$ such that:

$$\aleph \in \aleph_P \ |[\bar{A} \,|\, A]|\ \aleph_Q \Leftrightarrow (\mathbf{??}) \wedge (\mathbf{??})$$

Equation (**??**) is trivially true, whereas from equation (**??**) it follows that:

$$\aleph \upharpoonright A = \aleph_Q \upharpoonright A \tag{15}$$

Finally, equation (**??**) is trivially true, whereas from equation (**??**) it is required that:

$$\forall t \in \mathbb{R}^+ \bullet \sigma(s_Q \upharpoonright A \uparrow t) \subseteq \sigma(\aleph'_P \uparrow t) \tag{16}$$

Equations (**??**) and (**??**) show that the behaviour of process $P \ |[\bar{A}\,|\,A]|\ Q$ depends only on s_Q and $\aleph_Q$ and that it is indistinguishable from that of process Q.

Equations (**??**) and (**??**) denote which traces and refusals of a process P should be considered in order to create process $P \ |[\bar{A}\,|\,A]|\ Q$.

The above equations show that any behaviour of $P \ |[\bar{A}\,|\,A]|\ Q$ is also a behaviour of Q, i.e.

$$F[\![P \ |[\bar{A}\,|\,A]|\ Q]\!] \subseteq F[\![Q]\!]$$

We also need to show that the reverse is true, that any behaviour of Q is also a behaviour of $P \ |[\bar{A}\,|\,A]|\ Q$, i.e.

$$F[\![Q]\!] \subseteq F[\![P \ |[\bar{A}\,|\,A]|\ Q]\!]$$

Let P, Q be valid processes of the model, where P consists only of monitoring events and Q only of synchronisation events. For any behaviour $(s, \aleph)$ of Q, we need to find behaviours $(s_P, \aleph_P) \in F[\![P]\!]$ and $(s_Q, \aleph_Q) \in F[\![Q]\!]$ such that $(s, \aleph) = (s_P, \aleph_P) \ |[\bar{A}\,|\,A]|\ (s_Q, \aleph_Q)$.

This is not difficult. Let $s_Q = s$ and $\aleph_Q = \aleph$. We can construct the appropriate $s_P, \aleph_P$ by following the axioms of the model.

By the first axiom $(\langle\rangle, \{\}) \in F[\![P]\!]$. For every (t, a) that apears in trace s_Q, the empty behaviour of P can be extended so that $(t, a) \in \aleph_P$. This is due to the fourth and the fifth axioms. If $(t, a) \notin \aleph_P$ by the fourth axiom the empty behaviour can be extended so that $(t, \bar{a}) \in s_P$. But by the fifth axiom this cannot be repeatead infinitely, and there is a point where no more events can be added at time t to the trace. So, there must also be a point where P must refuse event a at time t, or $(t, a) \in \aleph_P$.

If the behaviour of P constructed previously is considered, then it is clear that equation **??** is true. But then equation **??** is also true by construction, since we started with the empty behaviour and we only added to s_P those events that also appear in s_Q. This means that:

$$(s_P, \aleph_P) \ |[\bar{A}\,|\,A]|\ (s_Q, \aleph_Q) = (s_Q, \aleph_Q) = (s, \aleph) \tag{17}$$

and the proof is concluded. □

In fact it can be shown that the extended model is equivalent to the original model if no mention of monitor events is made in the definition of a process. The proof is similar to the proof that the signal model is an extension model presented in [**?**] and thus it will be only outlined here.

Definition 1 *A term P of $TCSP_{+mon}$ is monitor-free if it is constructed only by using the operators of the original Timed-CSP and all of the free variables in P are eventually substituted by monitor-free terms.*

Lemma 2 *If P is a monitor-free term, then $F_{T\check{F}}[\![P]\!]\rho = F_{TF}[\![P]\!]\rho$, where ρ is an environment where variables can not be substituted by terms that refer to monitor events.*

Proof Firstly, it should be noted that if a process satisfies the axioms of the extended model, then, if it is monitor free, it also satisfies the axioms of the original model.

The rest of the proof can proceed by structural induction. The base case is when $P \equiv Stop$. The theorem is trivially true since the semantics of the *Stop* operator are identical in both models. The inductive step can

be made by considering each of the possible operators. For example for the case of the parallel operator consider that $P \stackrel{def}{=} P_1 \ |[A \,|\, B]|\ P_2$. Then

$$s \in s_{P_1} \ |[A \,|\, B]|\ s_{P_2} \Leftrightarrow (\mathbf{??}) \wedge (\mathbf{??}) \wedge (\mathbf{??}) \wedge (\mathbf{??}) \wedge (\mathbf{??}) \wedge (\mathbf{??})$$

But since P is constructed only by using the original Timed-CSP operators, the sets A and B do not contain any monitor events, thus $\bar{A} = \bar{B} = \emptyset$. Therefore, by $(\mathbf{??})$ it follows that s contains only synchronisations and moreover $s \upharpoonright A = s_{P_1}$ $s \upharpoonright B = s_{P_2}$ and $s \upharpoonright (A \cup B) = s$.

Similarly it can be deduced that $\aleph_{P_1} \subseteq \aleph \upharpoonright A$, $\aleph_{P_2} \subseteq \aleph \upharpoonright B$ and $\aleph \upharpoonright (A \cup B) = \aleph_{P_1} \cup \aleph_{P_2}$. Thus all the traces and the refusals are the same with the traces and refusals of the original parallel operator and therefore $F_{T\tilde{F}}[\![P_1 \ |[A \,|\, B]|\ P_2]\!]\rho = F_{TF}[\![P_1 \ |[A \,|\, B]|\ P_2]\!]\rho$. □

6 Examples

In this section, examples that demonstrate the usefulness of monitor events in the description of real-life protocols will be given. The first example is taken from the specification of the real-time scheduler of the Foundation Fieldbus protocol [**?**]. Similar schedulers can be found in the ISA Fieldbus protocols [**?**] as well as in the WorldFIP Fieldbus protocols [**?**].

Fieldbuses are time-critical, local area networks that are used at the process control level of a factory. They follow a layered architecture where only three of the seven layers of the OSI model are adopted: the physical layer, the data link layer and the application layer. Fieldbuses are usually token-based networks where a node needs permission to transmit before actually doing so. The scheduler lies at the data link layer and is responsible for distributing the token across the network's nodes.

The scheduler can be decomposed into two different sub-levels. The lower sublevel is responsible for communication with the physical layer and for passing any incoming packets to the upper sublevel and vice versa. The upper sublevel is responsible for distributing a token to the link's nodes according to a schedule that is downloaded to it via extra-protocol means. The scheduler does not only distribute the token but it is also responsible for link maintenance, time distribution, schedule distribution and so on.

The physical layer communicates with the lower sublevel via a series of primitives:

`Start`: The reception of this primitive indicates to the lower sublevel that reception of data is about to start.

`Data`: This primitive indicates the reception of a portion of a Data Link Layer Protocol Data Unit (DLPDU)

`End`: The reception of this primitive indicates the conclusion of a transmission and the reception of a whole DLPDU.

Normally, the physical layer issues a `Start` primitive followed by a number of `Data` primitives and ending with a single `End` primitive. It is assumed that the rate of transmission is constant and each primitive is separated by the other, by a time value of t_{octet}. The receiving process of the lower sublevel can then be

simply specified in Timed-CSP as:

$$RECV \stackrel{def}{=} start \stackrel{t_{octet}}{\rightarrow} DATA$$
$$DATA \stackrel{def}{=} data \stackrel{t_{octet}}{\rightarrow} DATA$$
$$\Box\ end \stackrel{t_{octet}}{\rightarrow} RECV$$

On the other hand, the upper sublevel of the scheduler is responsible for a number of functions. Distributing the token requires that the scheduler forms a token and passes it to the lower level for transmission. The upper level should then check if the transmission of the token created any activity on the link. If no activity is recorded after the transmission of the token, the scheduler assumes that the token is lost and is free to create and send another token to a different node. Another function of the scheduler is that of the distribution of the current time. In this case the scheduler forms a packet containing the time and forwards it to the lower sublevel which later transmits it. Exactly which function the upper level of the scheduler will perform depends on a schedule that is downloaded to it by external means. Since the detailed timing of the functions is not of importance here, the decision of which function should be performed will be non-deterministic and will be modelled by using the internal choice operator $\sqcap$. The scheduler can non-deterministically choose to start the token-passing function, via the $pt!on$ event or the time-distribution function via the $td!on$ event. When the token-passing function concludes, a $pt!off$ event is issued and the scheduler can again decide on the next function it should perform. Similarly a $td!off$ event indicates the conclusion of the time-distribution function. Here the usual notation of Timed-CSP was used where "!" denotes the output of a value and "?' the reception of a value.

$$SCHED \stackrel{def}{=} (pt!on \rightarrow pt!off \rightarrow SCHED)$$
$$\sqcap\ (td!on \rightarrow td!off \rightarrow SCHED)$$

Two different processes can also be used to model the two different functions the scheduler performs. Process *TOKENP* can be used to model the function that implements token passing and process *TIMEP* can be used to model the function that implements time distribution. These two processes are independent of one another and so they can interleave. Figure **??** describes the decomposition of the data-link layer into two levels and shows how the components at each level interact. The figure also shows graphically how process *FUNC* monitors the event *start*.

$$FUNC \stackrel{def}{=} TOKENP \ ||| \ TIMEP$$

If it is assumed that the upper and lower levels of the scheduler communicate via event *mac* and abstract away from the details of how a DLPDU is formed, then these two processes can be specified as follows:

$$TIMEP \stackrel{def}{=} td!on \rightarrow mac!TimePacket \rightarrow td!off \rightarrow TIMEP$$

TimePacket is assumed to be a DLPDU that contains the time. *TOKENP* can be defined as follows:

$$TOKENP \stackrel{def}{=} pt!on \rightarrow mac!Token \rightarrow$$
$$((\overline{start} \rightarrow mac?DLPDU \rightarrow pt!off \rightarrow TOKENP) \stackrel{timeout}{\triangleright}$$
$$(pt!off \rightarrow TOKENP))$$

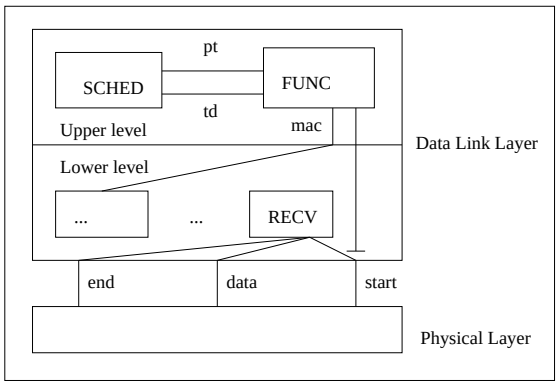

Figure 1: Representation of the data-link layer functions

Here, *Token* models the token of the fieldbus. Once the process gets an instruction from *SCHED* to commence the token passing activities, it forwards the *Token* to the lower level and then listens for the presence of link activity. If such presence is detected, i.e. a `Start` is received, the process is waiting for a reply from the node the token was passed to. When a reply is received, control is passed to the scheduler. If no activity is heard within *timeout* time units, control is passed to the scheduler without waiting for a reply.

The whole upper level of the scheduler can now be defined as:

$$UPPER \stackrel{def}{=} (SCHED \ |[\, pt, td \,|\, pt, td, mac, \overline{start} \,]|\ FUNC) \setminus \{pt, td\}$$

Assuming that the processes of the lower level that handle the reception of a DLPDU from the upper level are not important here, the lower level can be specified as:

$$LOWER \stackrel{def}{=} (\ldots \parallel \ldots \parallel RECV \parallel \ldots)$$

The whole system can now be specified as:

$$SCHEDULER \stackrel{def}{=} (UPPER \ |[\, mac, \overline{start} \,|\, mac, start, data \,]|\ LOWER) \setminus \{mac\}$$

and the scheduler's operation together with the physical layer can be specified as:

$$SYSTEM \stackrel{def}{=} (SCHEDULER \ |[\, start, data \,|\, start, data \,]|\ PHYSICAL) \setminus \{start, data\}$$

where *PHYSICAL* is a process that models the operations of the physical layer.

Note here that, if monitor events were not available, it would be impossible to state that process *TOKENP* monitors event *start*. Had the specification of *TOKENP* been:

$$\begin{aligned} TOKENP \stackrel{def}{=} & pt!on \rightarrow mac!Token \rightarrow \\ & ((start \rightarrow mac?DLPDU \rightarrow pt!off \rightarrow TOKENP) \stackrel{timeout}{\triangleright} \\ & (pt!off \rightarrow TOKENP)) \end{aligned}$$

the lower level would have been able to synchronise on event *start* only if the upper level was also willing to participate in that event. This behaviour is of course not desirable. An alternative behaviour could of

course be written in Timed-CSP to compensate for the cases *TIMEP* is active instead of *TOKENP*. Such a solution however would be non-intuitive and it would result in an over-specification of the initial problem.

Another real life application of monitor events is in the modelling of multicast services. In [**?**] the formal specification of the Foundation Fieldbus data link layer service is presented. The service includes provision for multicast communication, where there might be an unlimited number of receivers. In this kind of connections, the publisher usually places a packet simultaneously at all of the receivers' queues. If *PUB* is a process that represents the publisher and *SUBS* is a process that represents all the potential subscribers in the system, then the above requirement can be captured in:

$$SYSTEM \stackrel{def}{=} (PUB \ |[\, multi \,|\, multi \,]|\ SUBS) \setminus \{multi\}$$

Event *multi* models the simultaneous transmission from the publisher to all of the subscribers' queues. Since this is the formal specification of a service definition, the description must be as abstract as possible, avoid any references on how exactly a subscriber actually joins the connection and model the totality of the possible combinations of the system. Thus, *SUBS* should include all the *potential* subscribers and provide a simple intuitive method for changing the status of a subscriber from being a potential one to being an active one. Process *SUBS* can be described as:

$$SUBS \stackrel{def}{=} SUB_1 \ |[\, a_1, multi \,|\, a_2, multi \,]|\ SUB_2 \ldots SUB_{n-1} \ |[\, a_{n-1}, multi \,|\, a_n, multi \,]|\ SUB_n$$

where ... should be replaced by the maximum number of subscribers the system can afford. Each of the *SUB* processes models a potential subscriber that awaits a command from the service user in order to participate in the transaction:

$$SUB_i \stackrel{def}{=} a_i!MultiON \rightarrow ACTSUB,\ i = 1, \ldots, n$$

Here the event $a!MultiON$ denotes that the subscriber waits for the particular command *MultiON* at the service access point a. Then the subscriber can behave as an active subscriber, modelled by the process *ACTSUB*. This process can be specified as:

$$ACTSUB \stackrel{def}{=} multi \rightarrow ACTSUB$$

The problem with this approach is that unless every subscriber becomes active, then event *multi* can not take place at all. A solution would be to amend *SUB* so that it either waits for a command or a synchronisation at gate *multi*:

$$SUB_i \stackrel{def}{=} a_i!MultiON \rightarrow ACTSUB \,\Box\, multi \rightarrow SUB_i,\ i = 1, \ldots, n$$

but such a solution seems non-intuitive. On the contrary, using monitor events the whole specification becomes easier and more closely resembles the informal description:

$$\begin{aligned}
SYSTEM &\stackrel{def}{=} (PUB \ |[\, multi \,|\, \overline{multi} \,]|\ SUBS) \setminus \{multi, \overline{multi}\} \\
SUBS &\stackrel{def}{=} (SUB_1 \ |[\, a_1, \overline{multi} \,|\, a_2, \overline{multi} \,]|\ SUB_2 \ldots SUB_{n-1} \ |[\, a_{n-1}, \overline{multi} \,|\, a_n, \overline{multi} \,]|\ SUB_n) \\
SUB_i &\stackrel{def}{=} a_i!MultiON \rightarrow ACTSUB,\ i = 1, \ldots, n \\
ACTSUB &\stackrel{def}{=} \overline{multi} \rightarrow ACTSUB
\end{aligned}$$

In this case, the use of monitor events and hiding emulates the signal model of Timed-CSP.

Finally, a third example is taken from [**?**]. In there, a service definition is provided for the XTP protocol and a formal specification is presented. The service includes multicast connections and an example scenario is shown where there is a multicast communication with one transmitter and two receivers that communicate at a hidden event M. This event serves the same purpose as event *multi* in the previous example. However, the XTP protocol provides for more elaborate schemas with complex multicast group manipulations. Such an example is a multicast communication where there are more than one transmitter and more than one receiver. Using monitor events the system can be simply specified as:

$$SYSTEM \stackrel{def}{=} ((PUB \ |[M\,|\,M]|\ PUB \ldots ...) \ |[M\,|\,\bar{M}]|\ SUBS) \setminus \{M, \bar{M}\}$$

7 Conclusions

An extension to the original Timed-CSP was presented here that allows for the modelling of monitoring events. The extension was developed in a manner similar to that of [**?**, **?**], where an extension of Timed-CSP with signal events was presented. Monitoring events are triggered by their corresponding synchronisations without, however, imposing any restriction upon the occurrence of these synchronisations. A particular case where such events can prove very useful is in the modelling of communication mechanisms like multicasting. It was shown in the examples of the previous section that monitoring events can be easily used to model multicast communications where m particular processes must communicate. Other algebras either require the introduction of extra events to model such communications, as for example does Timed-CSP with signals, or are unable to insist that particular processes must communicate, e.g. E-LOTOS[**?**]. When monitoring events are used, specifications are more intuitive than the specifications written with other algebras and closer resemble the informal descriptions. In this respect, this paper contributes towards the simplification of real-life modelling problems.

The languages of CSP as well as that of Timed-CSP have not stopped evolving and they have recently undergone a revision [**?**]. The notation used in this paper has been consistent with that of [**?**] rather than that of older references such as [**?**]. The semantic model of Timed-CSP presented in [**?**] is slightly different from the one presented here since it also takes account of the possibility of infinite traces during the observation of a process and prefixing is not associated with delta delays. The results of this paper, however, are equally applicable to the new model as they are not affected by the possibility of an infinite behaviour.

8 Acknowledgements

The authors would like to acknowledge the contribution of the anonymous referees that made critical remarks and improved significantly the results presented herein.

$T\check{\Sigma} \stackrel{def}{=} \mathbb{R}^+ \times \check{\Sigma}$	The set of timed events
$T\check{T} \stackrel{def}{=} \{s \in \operatorname{seq} T\check{\Sigma} \mid (t, \check{a}) \text{ precedes } (t', \check{a}') \text{ in } s \Rightarrow t \leqslant t'\}$	The set of timed traces
$RT \stackrel{def}{=} \{[t_{start}, t_{end}) \times A \mid 0 \leqslant t_{start} < t_{end} < \infty \wedge A \subseteq \Sigma\}$	The set of refusal tokens
$TR \stackrel{def}{=} \{\bigcup C \mid C \subseteq RT \wedge C \, finite\}$	The set of timed refusals
$T\check{F} \stackrel{def}{=} T\check{T} \times TR$	The set of possible observations
$S_{T\check{F}} \stackrel{def}{=} \mathbb{P} T\check{F}$	The set of all possible processes
$M_{T\check{F}} \subset S_{T\check{F}}$	The timed failures model

Table 1: The timed failures model, M_{TF}, extended with monitors

1. $(\langle\rangle, \{\}) \in S$
2. $(s \frown w, \aleph) \in S \Rightarrow (s, \aleph \Vdash begin(w)) \in S$
3. $(s, \aleph) \in S \wedge s \cong w \Rightarrow (w, \aleph) \in S$
4. $(s, \aleph) \in S \wedge t \geqslant 0 \Rightarrow \exists \aleph' : TR \bullet \aleph \subseteq \aleph' \wedge (s, \aleph') \in S$
   $\wedge ((t' \leqslant t \wedge (t', a) \notin \aleph') \Rightarrow$
   $(s \upharpoonright t' \frown \langle (t', a) \rangle, \aleph' \Vdash t') \in S$
   $\vee (s \upharpoonright t' \frown \langle (t', \bar{a}) \rangle, \aleph' \Vdash t') \in S)$
5. $\forall t : \mathbb{R}^+ \bullet \exists n_t : \mathbb{N} \bullet (s, \aleph) \in S \wedge end(s) \leqslant t \Rightarrow \#(s) \leqslant n_t$
6. $\forall \aleph' : TR \bullet (s, \aleph) \in S \wedge \aleph' \subseteq \aleph \Rightarrow (s, \aleph') \in S$

Table 2: The timed-failures model axioms

$begin(\langle\rangle) \stackrel{def}{=} \infty$	$begin(\aleph) \stackrel{def}{=} inf(\{t \mid \exists a \bullet (t,a) \in \aleph\})$
$begin(\langle(t,\check{a})\rangle \frown s) \stackrel{def}{=} t$	$begin(\{\}) \stackrel{def}{=} \infty$
$end(\langle\rangle) \stackrel{def}{=} 0$	$end(\aleph) \stackrel{def}{=} sup(\{t \mid \exists a \bullet (t,a) \in \aleph\})$
$end(s \frown \langle(t,\check{a})\rangle) \stackrel{def}{=} t$	$end(\{\}) \stackrel{def}{=} 0$
$\langle\rangle \uparrow I \stackrel{def}{=} \langle\rangle$	$\aleph \uparrow I \stackrel{def}{=} \aleph \cap (I \times \Sigma)$
$(\langle(t,\check{a})\rangle \frown s) \uparrow I \stackrel{def}{=} \begin{cases} \langle(t,\check{a})\rangle \frown (s \uparrow I), & t \in I \\ s \uparrow I, & t \notin I \end{cases}$	
$s \upharpoonright t \stackrel{def}{=} s \uparrow [0,t]$	
$s \upharpoonleft t \stackrel{def}{=} s \uparrow [t,\infty)$	$\aleph \upharpoonleft t \stackrel{def}{=} \aleph \uparrow [t,\infty)$
$s \Vdash t \stackrel{def}{=} s \uparrow [0,t)$	$\aleph \Vdash t \stackrel{def}{=} \aleph \uparrow [0,t)$
$s \Vdash t \stackrel{def}{=} s \uparrow (t,\infty)$	
$\sigma(s) \stackrel{def}{=} \{\check{a} \in \check{\Sigma} \mid \exists t \bullet \langle(t,\check{a})\rangle \, in \, s\}$	$\sigma(\aleph) \stackrel{def}{=} \{a \in \Sigma \mid \exists t \bullet (t,a) \in s\}$
$s_1 \subseteq s_2 \Leftrightarrow \forall t, \check{a} \bullet \langle(t,\check{a})\rangle \in s_1 \Rightarrow \langle(t,\check{a})\rangle \in s_2$	
$\langle\rangle \upharpoonright \check{A} \stackrel{def}{=} \langle\rangle$	$\aleph \upharpoonright \check{A} \stackrel{def}{=} \{(t,\check{a}) \mid (t,\check{a}) \in \aleph \wedge \check{a} \in \check{A}\}$
$(\langle(t,\check{a})\rangle \frown s) \upharpoonright \check{A} \stackrel{def}{=} \begin{cases} \langle(t,\check{a})\rangle \frown (s \upharpoonright \check{A}), & \check{a} \in \check{A} \\ s \upharpoonright \check{A}, & \check{a} \notin \check{A} \end{cases}$	

Table 3: Notation used in the definitions of the semantics of $TCSP_{+mon}$

$$
\begin{array}{rcl}
F_{T\check{F}}[\![\mathit{Stop}]\!]\rho & \stackrel{def}{=} & \{(\langle\rangle, \aleph) \mid \aleph \in \mathit{TR}\} \\
F_{T\check{F}}[\![\mathit{Wait}\ t]\!]\rho & \stackrel{def}{=} & \{(\langle\rangle, \aleph) \mid \checkmark \notin \sigma(\aleph \restriction t)\} \cup \{(\langle(t', \checkmark)\rangle, \aleph) \mid t' \geqslant t \wedge \checkmark \notin \sigma(\aleph \uparrow [t, t'))\} \\
F_{T\check{F}}[\![\mathit{Skip}]\!]\rho & \stackrel{def}{=} & F_{T\check{F}}[\![\mathit{Wait}\ 0]\!]\rho \\
F_{T\check{F}}[\![a \rightarrow P]\!]\rho & \stackrel{def}{=} & \{(\langle\rangle, \aleph) \mid a \notin \sigma(\aleph)\} \cup \{(\langle(t, a)\rangle \frown s, \aleph) \mid t \geqslant 0 \wedge a \notin \sigma(\aleph \Vdash t) \wedge \\
 & & \quad (s, \aleph) - (t + \delta) \in F_{T\check{F}}[\![P]\!]\rho\} \\
F_{T\check{F}}[\![P;\ Q]\!]\rho & \stackrel{def}{=} & \{(s, \aleph) \mid \checkmark \notin \sigma(s) \wedge \forall\, t_0, t_1 \in \mathbb{R}^+ \\
 & & \bullet\ (s, \aleph \cup ([t_0, t_1) \times \{\checkmark\})) \in F_{T\check{F}}[\![P]\!]\rho\} \\
 & & \quad \cup \{(s \frown w, \aleph) \mid \exists\, t \bullet \checkmark \notin \sigma(s) \wedge (w, \aleph) - t \in F_{T\check{F}}[\![Q]\!]\rho \\
 & & \qquad \wedge (s \frown \langle(t, \checkmark)\rangle, \aleph \Vdash t \cup ([0, t) \times \{\checkmark\})) \in F_{T\check{F}}[\![P]\!]\rho\} \\
F_{T\check{F}}[\![P \,\Box\, Q]\!]\rho & \stackrel{def}{=} & \{(s, \aleph) \mid (s, \aleph) \in F_{T\check{F}}[\![P]\!]\rho \cup F_{T\check{F}}[\![Q]\!]\rho \\
 & & \quad \wedge (\langle\rangle, \aleph \Vdash \mathit{begin}(s)) \in F_{T\check{F}}[\![P]\!]\rho \cap F_{T\check{F}}[\![Q]\!]\rho\} \\
F_{T\check{F}}[\![P \sqcap Q]\!]\rho & \stackrel{def}{=} & F_{T\check{F}}[\![P]\!]\rho \cup F_{T\check{F}}[\![Q]\!]\rho \\
F_{T\check{F}}[\![P \stackrel{t}{\triangleright} Q]\!]\rho & \stackrel{def}{=} & \{(s, \aleph) \mid \mathit{begin}(s) \leqslant t \wedge (s, \aleph) \in F_{T\check{F}}[\![P]\!]\rho\} \\
 & & \quad \cup \{(s, \aleph) \mid \mathit{begin}(s) \geqslant t + \delta \wedge (\langle\rangle, \aleph \Vdash t) \in F_{T\check{F}}[\![P]\!]\rho \\
 & & \qquad \wedge (s, \aleph) - (t + \delta) \in F_{T\check{F}}[\![Q]\!]\rho\} \\
F_{T\check{F}}[\![\mu X \bullet F(X)]\!]\rho & \stackrel{def}{=} & \text{the unique solution of } X = F(X)
\end{array}
$$

Table 4: Semantics of the original Timed-CSP language