A brief history of Timed CSP

Jim Davies

Department of Computer Science, University of Reading, Reading RG6 2AY, UK

Steve Schneider

Department of Computer Science, Royal Holloway, University of London, Egham, Surrey TW20 0EX, UK

Abstract This paper is a comprehensive introduction to the language of Timed CSP, proposed by Reed and Roscoe in *A timed model for communicating sequential processes,* Springer LNCS 226. A brief description of the notation is followed by a detailed survey of timed and untimed models for the language. A compositional proof system is included, together with an account of timed refinement. The paper ends with a list of the changes made to the theory in recent years, and a brief discussion of other timed process algebras.

1 Introduction

The language described in this paper is very different to the original CSP notation of [Hoa78]. The language and models of Timed CSP have undergone a gradual evolution, from [ReR86] to [DaS92]. The forthcoming text on CSP and Timed CSP should provide for some degree of standardisation; until then, we offer this document as a guide to the current state of Timed CSP.

The paper begins with a description of the language of Timed CSP, and the model of computation. In Section 3, we show how timed and untimed models for the language may be used to capture requirements and establish results about program behaviour. Two complete compositional proof systems are presented: for the untimed traces, and timed failures models. In Section 4, a notion of timed refinement is introduced, relating timed programs to untimed specifications.

In Section 5, we provide a complete list of the changes made to the language of Timed CSP since [ReR86]. The mathematical foundations provided by [ReR87, Ree88] are sufficiently robust to support such improvements without the need to restructure the semantic models. These changes have been motivated by case studies and applications, rather than by any need to modify the original intuition. The paper ends with a brief discussion of other timed process algebras.

2 The language of Timed CSP

A *program* in Timed CSP is a term in the abstract syntax, a language construct such as $a \to \mathit{Stop}$. An *observation* of a program is a record of observable behaviour during an execution. A *model* is a denotational semantic model for the language, in which each program is identified with a set of observations. The different models are named according to the type of observations made: in the timed traces model, observations are sequences of timed events.

A *process* is an element of a semantic model: a set of observations which defines a pattern of behaviour. We find it useful to maintain a distinction between programs and processes, although it is not strictly necessary—valid programs are identified with elements of the semantic model. The construction of the semantic models is influenced by the properties of our model of computation: these include

* *maximal progress*: a program will execute until it terminates or requires some external synchronisation
* *maximal parallelism*: each component of a parallel combination has sufficient resources; the speed of execution is independent of the number of programs in a parallel combination
* *finite variability*: a program may undergo only finitely many changes of state during a finite interval of time
* *synchronous communication*: each communication event requires the simultaneous participation of every program involved
* *instantaneous events*: events have zero duration

This model of computation is consistent with that employed in [Hoa85].

2.1 Untimed CSP

The language of CSP includes primitive operators for parallel composition, nondeterministic choice, and hiding. This makes for an elegant notation in which the problems of concurrency, nondeterminism, and abstraction can be addressed separately. The language also provides constructs for modelling deadlock, recursion, and program relabelling:

$$\begin{array}{rcl} P & ::= & \mathit{Stop} \;\Big|\; \mathit{Skip} \;\Big|\; a \to P \;\Big|\; P \,;\, P \;\Big|\; P \,\Box\, P \;\Big|\; P \sqcap P \;\Big| \\ & & a : A \to P_a \;\Big|\; P \rhd P \;\Big|\; P \lightning P \;\Big|\; P \triangle P \;\Big|\; f(P) \;\Big|\; P \setminus A \;\Big| \\ & & \|_{A}\, P \;\Big|\; P \;{}_{A}\|_{A}\; P \;\Big|\; P \,|||\, P \;\Big|\; P \underset{A}{\|} P \;\Big|\; \mu X \bullet F(X) \end{array}$$

The variety of operators in CSP is in contrast to other algebraic approaches to concurrency, in which much emphasis is placed upon obtaining a minimal set of operators for the syntax.

Stop is a program which will never engage in external communication; it is a broken program. *Skip* is a program which does nothing except terminate, and is ready to terminate immediately. The prefix operator $\rightarrow$ allows us to add communication events to a program description. The program $a \rightarrow P$ is initially prepared to engage in synchronisation a; if this event occurs, it immediately begins to behave as P. The sequential composition operator transfers control upon termination. In the program $P \,;\, Q$, control is passed from program P to program Q if and when P performs the termination event $\checkmark$. This event is not visible to the environment, and occurs as soon as P is ready to perform it.

$P \;\Box\; Q$ is an external choice between programs P and Q. If the environment is prepared to cooperate with P but not Q, then the choice is resolved in favour of P, and *vice versa*. $P \sqcap Q$ is a internal choice between P and Q; the outcome of this choice is nondeterministic. The program $a : A \rightarrow P_a$ offers an external choice of initial event a, drawn from a set A, which may be infinite. This construct allows us to model program input from a channel. If channel c carries values of type T, then

$$c?x : T \rightarrow P_x \quad = \quad a : \{c.v \mid v \in T\} \rightarrow P'_a$$

where $P'_{c.v} = P_v$. The program $c?x : T \rightarrow P_x$ is prepared to accept any value v of type T on channel c, and then behave accordingly. We use the expression $c!v$ to denote the output of value v on channel c. No choice construct is required in this case; the value transmitted is determined by the sending program.

The timeout program $P \rhd Q$ may behave as Q, or offers a choice between P and Q, according to whether the timeout has occurred or not. In untimed CSP, the resulting behaviour is that of a nondeterministic choice: without timing information, we cannot determine when the timeout occurs.

The interrupt program

$$P \bigtriangleup Q$$

behaves as P but may be interrupted by any initial event of Q; at this point, control is transferred to Q and process P is discarded. The transfer operator $\lightning$ passes control from one program to another after a predetermined time has elapsed. Without timing information, the first program may be interrupted at any point.

A synchronised parallel combination of a set of programs is parameterised by a corresponding set of interfaces: for each program P, we provide an interface set A_P. In the network of programs defined by

$$NETWORK \quad = \quad \|_{A_P} P$$

each event a requires the participation of every subprogram P such that $a \in A_P$. Every pair of subprograms must cooperate on each event from the intersection of their interface sets. A simple form of network is the binary parallel combination

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

in which program P may perform only those events in A, program Q may perform only those events in B, and the two programs must cooperate on events drawn from the intersection of A and B.

In an asynchronous parallel combination

$$P \quad ||| \quad Q$$

both subprograms evolve concurrently without interacting, though they must agree on termination. If both subprograms are capable of performing the same event a, then a degree of nondeterminism may be introduced. In the hybrid parallel program

$$P \underset{C}{\|} Q$$

components P and Q must synchronise upon events from set C, and on termination; and they interleave on all other events.

The relabelled program $f(P)$ has a similar control structure to P, with observable events renamed according to function f. The program $P \setminus A$ behaves as P, except that events from set A are concealed from the environment of the program. Hidden events no longer require the cooperation of the environment, and so occur as soon as P is ready to perform them.

The recursive program $\mu X \bullet F(X)$ behaves as $F(X)$, with each instance of variable X representing a recursive invocation; this program satisfies the equation $P = F(P)$. These programs have a well-defined semantics if the function F is *guarded*. In untimed CSP, a function F is guarded if every free occurrence of X in $F(X)$ is preceded by at least one observable event.

2.2 Timed CSP

The language of Timed CSP is defined by the following grammar rule:

$$\begin{array}{lll} P & ::= & \mathit{Stop} \;\big|\; \mathit{Skip} \;\big|\; \mathit{Wait}\ t \;\big|\; a \to P \;\big|\; P\, ; P \;\big|\; P \,\Box\, P \;\big|\; P \sqcap P \;\big| \\ & & a : A \to P_a \;\big|\; P \overset{t}{\rhd} P \;\big|\; P \underset{t}{\lightning} P \;\big|\; P \bigtriangleup P \;\big|\; f(P) \;\big|\; P \setminus A \;\big| \\ & & \|_{A}\, P \;\big|\; P \;{}_{A}\|_{A}\; P \;\big|\; P \,|||\, P \;\big|\; P \underset{A}{\|} P \;\big|\; \mu X \bullet F(X) \end{array}$$

In this rule, event a is drawn from the set of all synchronisations Σ, event set A ranges over the set of subsets of Σ, and t is a non-negative real number. We place no lower bound on the interval between consecutive events—this allows us to model asynchronous processes in a satisfactory fashion, without artificial constraints upon the times at which independent events may be observed.

The new operator *Wait* is a delayed form of *Skip*; it does nothing, but is ready to terminate successfully after the specified time.

As in untimed CSP, we consider events to be instantaneous; if the duration of an action is of interest, then that action may be modelled by considering the beginning and the end of the action to be separate events.

In the timeout program $P \triangleright Q$ control is transferred from P to Q if no communications have occurred. If an attempt at communication involving P is made at time t precisely, then the outcome will be nondeterministic. The situation is analogous to a bid being made as the auctioneer brings the hammer down: a satisfactory outcome cannot be guaranteed.

The timed interrupt, or transfer program

$$P \underset{t}{\lightning} Q$$

behaves as P until time t, when control is transferred to Q.

The recursive program $\mu X \bullet F(X)$ behaves as $F(X)$, with each instance of variable X representing an immediate recursive invocation. This program satisfies the equation $P = F(P)$. Again, recursive programs have a well-defined semantics if the defining function is *guarded*. In Timed CSP, a function $F(X)$ is guarded if every free occurrence of X in the body of $F(X)$ is preceded by a non-zero time delay.

3 Semantic models for Timed CSP

In [Ree88], a variety of semantic models were defined for the language of Timed CSP. In these models, programs are associated with sets of observations. We may reason about programs by reasoning about these sets: a predicate on the semantic set corresponds to a requirement upon the program. For example, in the Traces model of CSP, we may capture the requirement that program P never performs a visible action with the predicate

$$\forall\, tr \in traces(P) \bullet tr = \langle\rangle$$

In this model, the program *Stop* is associated with the singleton set $\{\langle\rangle\}$, containing only the empty trace. We may conclude that *Stop* is a program that meets this requirement.

Accordingly, a *specification* is a predicate on observations. For example, a specification in the untimed traces model $\mathcal{M}_{UT}$ is a predicate of the form $S(tr)$, where tr is an arbitrary trace. A program P *satisfies* a specification if that specification holds for every observation of an execution of P. In the traces model, we define a satisfaction relation

$$P \textbf{ sat } S(tr) \text{ in } \mathcal{M}_{UT} \quad \Leftrightarrow \quad \forall\, tr \in \mathcal{F}_{UT}[\![P]\!] \bullet S(tr)$$

where $\mathcal{F}_{UT}$ is the semantic function for the traces model. We will omit the qualification ' in $\mathcal{M}_{UT}$' where the identity of the model is obvious from the context.

3.1 Untimed models

The most abstract semantic model for the language of CSP is the traces model of [Hoa85]. It is also the most widely-used and well-understood of all the semantic models. In the traces model, each program is associated with a set of untimed finite traces—sequences of observable events. Using trace specifications, we may capture *safety conditions*—constraints that proscribe certain events or sequences of events in an execution of the program.

If we wish to capture untimed *liveness conditions*—constraints that insist that certain events become possible in an execution—we must include readiness or *refusal* information in our semantic model. In the failures model $\mathcal{M}_{UF}$ we associate each trace of a program with the set of events that may be refused afterwards. If the failure (tr, ref) is present in the semantic set of program P, then P may perform trace tr and then refuse to engage in any event from ref.

In [BrR85], observations are extended to include *divergences*. A trace of a program P is a divergence if it may be followed by an unbounded sequence of internal events. In [Ree88], we find an alternative treatment of divergence. In the stability model $\mathcal{M}_{US}$, a trace tr is associated with a stability value of ∞ if the program may diverge after performing tr, or *0* otherwise. In the failures-stability model $\mathcal{M}_{UFS}$, programs are associated with sets of triples (tr, α, ref). A stability value α is attached to each failure; if the value is zero, then the program is stable after performing trace tr: it does not diverge. An infinite stability value indicates that internal activity may continue indefinitely.

The semantic equations for each denotational model form the basis of a compositional proof system—a set of inference rules relating the properties of a program to the properties of its syntactic subcomponents. Each rule is of the form

$$\frac{\begin{array}{c} \textit{antecedent} \\ \vdots \\ \textit{antecedent} \end{array}}{\textit{consequent}} \quad [\ \textit{side condition}\ ]$$

If we establish the truth of each *antecedent*, then we can be assured of the truth of the *consequent*, providing that the *side condition* holds. Each consequent will take the form P **sat** S: these rules may be used to establish that a program meets a given specification.

3.2 An untimed proof system

The following logical rules may be derived for the untimed traces model:

$$\frac{}{P \ \mathbf{sat}\ \mathit{true}} \qquad \frac{\begin{array}{l} P \ \mathbf{sat}\ S(tr) \\ P \ \mathbf{sat}\ T(tr) \end{array}}{P \ \mathbf{sat}\ S(tr) \wedge T(tr)} \qquad \frac{\begin{array}{l} P \ \mathbf{sat}\ S(tr) \\ S(tr) \Rightarrow T(tr) \end{array}}{P \ \mathbf{sat}\ T(tr)}$$

The null specification is true of any program, each goal may be addressed separately, and we may weaken any specification already established. From the semantic equations given in [Ree88], we may derive an inference rule for each operator in the language.

$$\frac{}{\mathit{Stop}\ \mathbf{sat}\ tr = \langle\rangle} \qquad \frac{}{\mathit{Skip}\ \mathbf{sat}\ tr = \langle\rangle \vee tr = \langle\checkmark\rangle}$$

The broken program *Stop* is unable to engage in external communication: any trace of this program must be equal to the empty trace $\langle\rangle$. The program *Skip* may perform only the termination event $\checkmark$.

Any non-empty trace of the program $a \to P$ must begin with the event a, and continue with a trace of program P:

$$\frac{P \ \mathbf{sat}\ S(tr)}{\begin{array}{ll} a \to P \ \mathbf{sat} & tr = \langle\rangle \\ & \vee \\ & \exists\, tr' \bullet tr = \langle a\rangle ^\frown tr' \wedge S(tr') \end{array}}$$

We may produce a non-empty trace of $a \to P$ by catenating the singleton trace $\langle a \rangle$ with a trace tr' of program P.

A trace of the sequential composition $P \,;\, Q$ may be either a trace of P—if this program has not terminated—or the catenation of a trace of P and a trace of Q:

$$\frac{\begin{array}{l} P \textbf{ sat } S(tr) \\ Q \textbf{ sat } T(tr) \end{array}}{\begin{array}{ll} P \,;\, Q \textbf{ sat } & \checkmark \notin \sigma(tr) \wedge S(tr) \\ & \vee \\ & \exists\, tr_P, tr_Q \bullet tr = tr_P \frown tr_Q \wedge \checkmark \notin \sigma(tr_P) \wedge S(tr_P \frown \langle \checkmark \rangle) \wedge T(tr_Q) \end{array}}$$

The termination event $\checkmark$ is hidden from the environment: it is not present in the event sets of traces tr and tr_P.

An observation of a choice program must be an observation of at least one of the components:

$$\frac{\begin{array}{l} P \textbf{ sat } S(tr) \\ Q \textbf{ sat } T(tr) \end{array}}{P \sqcap Q \textbf{ sat } S(tr) \vee T(tr)} \qquad \frac{\begin{array}{l} P \textbf{ sat } S(tr) \\ Q \textbf{ sat } T(tr) \end{array}}{P \,\Box\, Q \textbf{ sat } S(tr) \vee T(tr)}$$

In establishing that a program meets a safety requirement, there is no need to distinguish between internal and external choice.

The prefix choice program $a : A \to P_a$ is initially prepared to engage in any event from A. If no events have been observed, then no event from A may be refused.

$$\frac{\forall\, a : A \bullet P_a \textbf{ sat } S_a(tr)}{a : A \to P_a \textbf{ sat } tr = \langle\rangle \vee a \in A \wedge tr = \langle a \rangle \frown tr' \wedge S_a(tr')}$$

If a is the first event observed, then a is an element of A, and the subsequent behaviour will be due to P_a.

In the timeout program $P \rhd Q$ control is transferred from P to Q at time t if no communications have occurred. Without timing information, we may infer only that the program behaves either as P, or as Q:

$$\frac{\begin{array}{l} P \textbf{ sat } S(tr) \\ Q \textbf{ sat } T(tr) \end{array}}{P \rhd Q \textbf{ sat } S(tr) \vee T(tr)}$$

The program $P \lightning Q$ behaves as P until some predetermined time, at which control is transferred to Q.

$$\frac{\begin{array}{l} P \textbf{ sat } S(tr) \\ Q \textbf{ sat } T(tr) \end{array}}{P \lightning Q \textbf{ sat } \exists\, tr_P, tr_Q \bullet tr = tr_P \frown tr_Q \wedge S(tr_P) \wedge T(tr_Q)}$$

An untimed trace of this program is simply a trace of P, followed by a trace of Q.

Similarly, the program $P \triangle Q$ behaves as P until control is passed to Q:

$$\frac{\begin{array}{l} P \ \mathbf{sat}\ S(tr) \\ Q \ \mathbf{sat}\ T(tr) \end{array}}{P \triangle Q \ \mathbf{sat}\ \exists\, tr_P, tr_Q \bullet tr = tr_P \frown tr_Q \wedge S(tr_P) \wedge T(tr_Q)}$$

We may infer only that the resulting trace is formed by concatenating two traces: one from P, and one from Q.

A parallel combination may terminate only when all programs are ready to terminate. When we consider the semantics of such a construct, we include this condition explicitly, by adding the special event $\checkmark$ to each interface set: if A is a set of events, we define the augmented set

$$A^{\checkmark} \quad = \quad A \cup \{\checkmark\}$$

Clearly, the interface will be unchanged if the termination event has already been included. In a synchronised parallel combination, each component participates in every event from its interface set. If the parallel combination is observed to perform trace tr, then each component P_i has contributed the trace $tr \upharpoonright A_i^{\checkmark}$, where A_i is the corresponding interface set, and the projection operator is defined by

$$\begin{array}{rcll} \langle\rangle \upharpoonright A & = & \langle\rangle & \\ (\langle a\rangle \frown tr) \upharpoonright A & = & \langle a\rangle \frown (tr \upharpoonright A) & \text{if } a \in A \\ & & tr \upharpoonright A & \text{otherwise} \end{array}$$

The resulting inference rule is

$$\frac{\forall\, i \in I \bullet P_i \ \mathbf{sat}\ S_i(tr)}{\|_{A_i}\, P_i \ \mathbf{sat}\ \forall\, i \in I \bullet S_i(tr \upharpoonright A_i^{\checkmark}) \wedge \sigma(tr) \subseteq \bigcup_i A_i^{\checkmark}}$$

Observe that a parallel combination is capable of performing only those events contained in the union of the interface sets: the set of events recorded $\sigma(tr)$ must be a subset of $\bigcup_i A_i$.

In the partially-interleaved parallel combination

$$P \underset{A}{\|} Q$$

the two components are required to synchronise on every event from the common interface A. If components P and Q are observed to perform traces tr_P and tr_Q, respectively, then the parallel combination may be observed to perform any trace tr from the set of interleavings $tr_P \underset{A}{\|} tr_Q$. This set is defined recursively by

$$\langle\rangle \in tr_1 \underset{A}{\|} tr_2 \quad \Leftrightarrow \quad tr_1 = tr_2 = \langle\rangle$$

$$
\begin{array}{rcl}
tr \in \langle\rangle \underset{A}{\parallel} \langle\rangle & \Leftrightarrow & tr = \langle\rangle \\
\langle a\rangle^\frown tr \in \langle b\rangle^\frown tr_2 \underset{A}{\parallel} \langle\rangle & \Leftrightarrow & a \notin A^{\checkmark} \wedge a = b \wedge tr \in tr_2 \underset{A}{\parallel} \langle\rangle \\
\langle a\rangle^\frown tr \in \langle\rangle \underset{A}{\parallel} \langle c\rangle^\frown tr_2 & \Leftrightarrow & a \notin A^{\checkmark} \wedge a = c \wedge tr \in \langle\rangle \underset{A}{\parallel} tr_2 \\
\langle a\rangle^\frown tr \in \langle b\rangle^\frown tr_P \underset{A}{\parallel} \langle c\rangle^\frown tr_Q & \Leftrightarrow & a \notin A^{\checkmark} \wedge a = b \wedge tr \in tr_P \underset{A}{\parallel} \langle c\rangle^\frown tr_Q \\
& & \qquad\quad \vee \\
& & \qquad\quad a = c \wedge tr \in tr_Q \underset{A}{\parallel} \langle b\rangle^\frown tr_P \\
& & \vee \\
& & a \in A^{\checkmark} \wedge a = b = c \wedge tr \in tr_P \underset{A}{\parallel} tr_Q
\end{array}
$$

Every event from the augmented set $A^{\checkmark}$ must appear in both component traces; other events are recorded independently. The inference rule for the partially-interleaved parallel combination is

$$
\frac{\begin{array}{l} P \;\mathbf{sat}\; S(tr) \\ Q \;\mathbf{sat}\; T(tr) \end{array}}{P \underset{A}{\parallel} Q \;\mathbf{sat}\; \exists\, tr_P, tr_Q \bullet tr \in tr_P \underset{A}{\parallel} tr_Q \wedge S(tr_P) \wedge T(tr_Q)}
$$

In the interleaved parallel combination $P \interleave Q$ both programs execute independently. However, synchronisation is required if either component is to terminate. This leads to the following inference rule for the interleaving operator:

$$
\frac{\begin{array}{l} P \;\mathbf{sat}\; S(tr) \\ Q \;\mathbf{sat}\; T(tr) \end{array}}{P \interleave Q \;\mathbf{sat}\; \exists\, tr_P, tr_Q \bullet tr \in tr_P \parallel tr_Q \wedge S(tr_P) \wedge T(tr_Q)}
$$

The image of P under relabelling f may engage in the event $f(a)$ whenever P can engage in the event a:

$$
\frac{P \;\mathbf{sat}\; S(tr)}{f(P) \;\mathbf{sat}\; \exists\, tr_P \bullet tr = f(tr_P) \wedge S(tr_P)}
$$

The effect of concealing a set of communications is simple: they disappear from the recorded trace. We define a trace hiding operator in terms of the projection operator introduced above:

$$
tr \setminus A \quad = \quad tr \upharpoonright \Sigma - A
$$

where Σ is the set of all communication events. If tr is a trace of the program $P \setminus A$, then there must be a trace tr' of P such that $tr = tr' \setminus A$. The inference rule for the hiding operator is

$$\frac{P \ \mathbf{sat}\ S(tr)}{P \setminus A \ \mathbf{sat}\ \exists\, tr_P \bullet tr = tr_P \setminus A \wedge S(tr_P)}$$

Finally, we require a recursion induction rule for reasoning about recursive program descriptions. A complete theory of recursion induction for untimed CSP is presented in [Ros82]. Here, we will content ourselves with the rule for the μ-operator:

$$\frac{X \ \mathbf{sat}\ S(tr) \ \Rightarrow\ F(X)\ \mathbf{sat}\ S(tr)}{\mu\, X \bullet F(X)\ \mathbf{sat}\ S(tr)} \quad [\ F \text{ is guarded, } S \text{ is satisfiable}\ ]$$

To show that a recursive program $\mu\, X \bullet F(X)$ satisfies a specification S, we have only to show that the specification is satisfiable, invariant under recursive calls, and that the defining function is guarded. A specification S is satisfiable if there is some program P for which P **sat** S.

3.3 Timed models

A variety of timed models have been defined for the language of CSP. Just as the untimed models recorded trace, refusal and stability information, the timed models record timed traces, timed refusals, and timed stabilities. The simplest of the timed models, $\mathcal{M}_{TT}$, associates a program with a set of timed traces. The timed failures model $\mathcal{M}_{TF}$, and the timed failures-stability model $\mathcal{M}_{TFS}$ record the events refused by a program during and after the observation of each timed trace.

The timed stability models $\mathcal{M}_{TS}$ and $\mathcal{M}_{TFS}$ include information about the presence of internal activity. The stability value of an observation is the earliest time by which all internal activity is *guaranteed* to have ceased. In the timed failures-stability model, each failure (s, X) of a program is associated with a single stability value α between *0* and ∞, inclusive. If the program exhibits the external behaviour described by (s, X), then all internal activity must cease at or before time α.

In the specification of a real-time system, internal activity is usually of only secondary importance. The correctness of a design will be expressed as a set of constraints upon the occurrence and availability of observable events or external synchronisations. This is precisely the information that may be obtained from the timed failures model $\mathcal{M}_{TF}$. Furthermore, the timed models without timed refusals are complicated by the need to record the times at which events first become available, in order to give a satisfactory semantics to the hiding operator. For these reasons, we restrict our attention to the timed failures model of CSP.

In this model, we record the times at which the program performs or refuses external events or synchronisations. An observation is an ordered pair

$$(\mathit{timed\ trace},\ \mathit{timed\ refusal})$$

in which the timed trace records the sequence of timed events observed, and the timed refusal records the set of timed events refused.

In the untimed failures model, each observation contains an untimed trace and an untimed refusal. If the observation

$$(trace,\ refusal)$$

is made of program P, then we know that P may perform the events in the $trace$, and *then* refuse to engage in any of the events from the $refusal$. In the timed failures model, the observation of a pair (s, X) corresponds to the knowledge that the program may perform the events of the trace s *while* refusing the events from the refusal set X.

As in the untimed models, we use Σ to denote the set of all observable events. Our domain of time values is the non-negative real numbers

$$TIME \quad = \quad [0, \infty)$$

A timed event is simply an event from Σ labelled with a time value from $TIME$. The set of all timed events is a Cartesian product

$$TE \quad = \quad TIME \times \Sigma$$

A timed trace is a finite sequence of timed events, such that events appear in chronological order:

$$TT \quad = \quad \{s \in \operatorname{seq} TE \mid \langle (t_1, a_1), (t_2, a_2) \rangle \preceq s \Rightarrow t_1 \leqslant t_2\}$$

where $s_1 \preceq s_2$ iff s_1 is a subsequence of s_2.

If I is a finite half-open time interval, and A is a set of events, then we say that the Cartesian product $I \times A$ is a *refusal token*, describing the refusal of a program to perform any event from A throughout interval I: every event from A is refused continuously.

We insist that the component intervals I are finite: all observations are completed in a finite time. This is a characteristic property of the timed failures model: if two programs have distinct meanings, then they may be distinguished in a finite time. The set of all refusal tokens is given by

$$RT \quad = \quad \{[t_1, t_2) \times A \mid 0 \leqslant t_1 < t_2 < \infty \wedge A \subseteq \Sigma\}$$

A timed refusal X corresponds to a record of events refused during a particular execution: timed event (t, a) is an element of X if and only if event a was refused at time t during this execution. The set of all timed refusals is given by

$$TR \quad = \quad \{\bigcup R \mid R \subseteq RT \wedge R \text{ is finite}\}$$

Any observation of refusal behaviour may be characterised as a *finite* union of refusal tokens: this is a consequence of the finite variability assumption of our computational model. The set of all timed observations, or *timed failures*, is thus

$$\mathcal{O}_{TF} \quad = \quad TT \times TR$$

We may define a semantic function $\mathcal{F}_{TF}$ from the syntax to the powerset $\mathbb{P}\,\mathcal{O}_{TF}$, associating each program with a set of timed failures. From the defining equations of this function, we may derive a complete set of inference rules, similar to the one presented above for the untimed traces model. The statement of these rules will require some additional notation for timed observations.

We define the function σ upon each type of timed observation:

$$\begin{array}{rcl} \sigma(s) & = & \{a \mid \exists\, t \bullet \langle (t,a) \rangle \preceq s\} \\ \sigma(X) & = & \{a \mid \exists\, t \bullet (t,a) \in X\} \\ \sigma(s,X) & = & \sigma(s) \cup \sigma(X) \end{array}$$

returning the set of events present in the trace, refusal or failure. Similarly, the operator '*times*' returns the set of times present in each object:

$$\begin{array}{rcl} \mathit{times}(s) & = & \{t \mid \exists\, a \bullet \langle (t,a) \rangle \preceq s\} \\ \mathit{times}(X) & = & \{t \mid \exists\, a \bullet (t,a) \in X\} \\ \mathit{times}(s,X) & = & \mathit{times}(s) \cup \mathit{times}(X) \end{array}$$

We will need to restrict our attention to the events refused during a particular interval $I \subseteq \mathit{TIME}$:

$$X \uparrow I \quad = \quad X \cap (I \times \Sigma)$$

3.4 A timed proof system

As in the case of the untimed traces model, we provide an inference rule for each operator in the language of Timed CSP, beginning with the broken program *Stop*. Any trace of this program must be equal to the empty trace,

$$\frac{}{\mathit{Stop}\ \mathbf{sat}\ s = \langle\rangle}$$

but we can infer nothing about a typical refusal set.

The program *Skip* is initially prepared to perform the termination event ✓—the only action that this program may perform:

$$\frac{}{\mathit{Skip}\ \mathbf{sat}\ (s = \langle\rangle \wedge \checkmark \notin \sigma(X)) \vee (s = \langle (t,\checkmark) \rangle \wedge \checkmark \notin \sigma(X \uparrow [0,t)))}$$

Either no events have been observed and ✓ is available, or ✓ is observed at some time t and was continuously available beforehand. A similar inference rule may be derived for the delay program:

$$\frac{}{\begin{array}{l} \mathit{Wait}\ t\ \mathbf{sat}\ s = \langle\rangle \wedge \checkmark \notin \sigma(X \uparrow [t,\infty)) \\ \qquad\qquad\quad \vee \\ \qquad\qquad\quad s = \langle (t',\checkmark) \rangle \wedge \checkmark \notin \sigma(X \uparrow [t,t')) \wedge t' \geqslant t \end{array}}$$

If no events have been observed then ✓ must be available continuously from time t onwards. Otherwise, ✓ is observed at a time $t' \geqslant t$ and made available at all times between t and t'.

To describe the behaviour of sequential composition and choice programs, we will need to refer to the time of the first and last events recorded:

$$\begin{array}{rclcrcl} begin(s) & = & \inf(times(s)) & \qquad & end(s) & = & \sup(times(s)) \\ begin(X) & = & \inf(times(X)) & \qquad & end(X) & = & \sup(times(X)) \end{array}$$

We take the infimum and supremum of the empty set of times to be ∞ and 0, respectively. This choice of values is the most convenient for the subsequent mathematics. We extend the definition of the *end* operator to timed failures:

$$end(s, X) \quad = \quad \max\{end(s), end(X)\}$$

The interval $[0, end(s, X))$ is the duration of the observation.

We define a linear addition function for timed traces, timed refusals, and timed failures, shifting each recorded time by a constant amount:

$$\begin{array}{rcl} \langle\rangle + t_0 & = & \langle\rangle \\ (\langle(t, a)\rangle^\frown s) + t_0 & = & \langle(t + t_0, a)\rangle^\frown(s + t_0) \\ X + t_0 & = & \{(t + t_0, a) \mid (t, a) \in X \land t + t_0 \geqslant 0\} \\ (s, X) + t_0 & = & (s + t_0, X + t_0) \\ (s, X) - t_0 & = & (s, X) + (-t_0) \end{array}$$

For this function to return valid traces and refusals, we require that

$$begin(s) + t_0 \geqslant 0$$

respectively; time values must be non-negative.

The event prefix operator is associated with the following rule:

$$\frac{P \ \mathbf{sat}\ S(s, X)}{\begin{array}{l} a \to P \ \mathbf{sat}\ s = \langle\rangle \land a \notin \sigma(X) \\ \qquad\qquad \lor \\ \qquad\qquad \exists\, s' \bullet s = \langle(t, a)\rangle^\frown s' \land a \notin \sigma(X \uparrow [0, t)) \land S((s', X) - t) \end{array}}$$

If the trace is empty, then a may not be refused, and is therefore absent from the refusal set X. Otherwise, the first event must be a. If a occurs at time t, we know that a may not be refused before this time; the subsequent behaviour is due to program P, and must satisfy specification S.

The behaviour of the sequential composition $P \,;\, Q$ depends upon whether the first component has terminated:

$$\frac{\begin{array}{l} P \ \mathbf{sat}\ S(s, X) \\ Q \ \mathbf{sat}\ T(s, X) \end{array}}{\begin{array}{ll} P\ ;\ Q \ \mathbf{sat} & \checkmark \notin \sigma(s) \wedge S(s, X \cup ([0, end(s,X)) \times \{\checkmark\})) \\ & \vee \\ & \exists\, s_P, s_Q, t \bullet s = s_P \frown s_Q \wedge \checkmark \notin \sigma(s_P) \wedge \\ & \qquad S(s_P \frown \langle (t, \checkmark) \rangle, X \uparrow [0, t) \cup ([0, t) \times \{\checkmark\})) \wedge \\ & \qquad T((s_Q, X) - t) \end{array}}$$

The trace s_P may be extended with a $\checkmark$ event at some time t (this event is hidden by the sequential composition operator). In the presence of the sequential composition operator, the event $\checkmark$ occurs as soon as it becomes available, so we know that it may be refused at any time before t. Hence

$$(s_P \frown \langle (t, \checkmark) \rangle, X \uparrow [0, t) \cup ([0, t) \times \{\checkmark\}))$$

must be an observation of P. The second part of the trace, together with the refusals after t, is an observation of Q.

An observation of a nondeterministic choice must be an observation of at least one of the components:

$$\frac{\begin{array}{l} P \ \mathbf{sat}\ S(s, X) \\ Q \ \mathbf{sat}\ T(s, X) \end{array}}{P \sqcap Q \ \mathbf{sat}\ S(s, X) \vee T(s, X)}$$

In the case of a deterministic choice, we may also infer that any event refused before the first event is observed must be refused by both programs:

$$\frac{\begin{array}{l} P \ \mathbf{sat}\ S(s, X) \\ Q \ \mathbf{sat}\ T(s, X) \end{array}}{\begin{array}{ll} P \,\Box\, Q \ \mathbf{sat} & S(s, X) \vee T(s, X) \\ & \wedge \\ & S(\langle\rangle, X \uparrow [0, begin(s))) \wedge T(\langle\rangle, X \uparrow [0, begin(s))) \end{array}}$$

Any observation of the form $(\langle\rangle, X)$ must be common to both alternatives.

The prefix choice program $a : A \to P_a$ is initially prepared to engage in any event from A. If no events have been observed, then no event from A may be refused.

$$\frac{\forall\, a : A \bullet P_a \ \mathbf{sat}\ S_a(s, X)}{\begin{array}{ll} a : A \to P_a \ \mathbf{sat} & s = \langle\rangle \wedge \forall\, a \in A \bullet a \notin \sigma(X) \\ & \vee \\ & \exists\, a \in A\ ;\ s' \bullet \ \ s = \langle (t, a) \rangle \frown s' \wedge \\ & \qquad \forall\, a \in A \bullet a \notin \sigma(X \uparrow [0, t)) \wedge \\ & \qquad S_a((s', X) - t) \end{array}}$$

If a is the first event observed, then a is an element of A, and the subsequent behaviour will be due to P_a.

In the timeout program $P \overset{t}{\triangleright} Q$, control is transferred to Q unless P performs an external action before time t.

$$\frac{\begin{array}{l} P \textbf{ sat } S(s,X) \\ Q \textbf{ sat } T(s,X) \end{array}}{\begin{array}{ll} P \overset{t}{\triangleright} Q \textbf{ sat} & begin(s) \leqslant t \wedge S(s,X) \\ & \vee \\ & begin(s) \geqslant t \wedge S(\langle\rangle, X \uparrow [0,t)) \wedge T((s,X) - t) \end{array}}$$

The image of P under relabelling f may engage in the event $f(a)$ whenever P can engage in the event a:

$$\frac{P \textbf{ sat } S(s,X)}{f(P) \textbf{ sat } \exists\, s' \bullet s = f(s') \wedge S(s', f^{-1}(X))}$$

The expression $f^{-1}(X)$ denotes the set $\{(t,a) \mid (t, f(a)) \in X\}$. This is the inverse image of refusal set X under function f.

To reason about the observations of $P \setminus A$, we identify the observations of P in which every event from A occurs as soon as possible. These are precisely the observations of P in which events from A may be continuously refused; if we can show that these observations satisfy a behavioural specification S' such that

$$S'(s, X \cup [0, end(s,X)) \times A) \quad \Rightarrow \quad S(s \setminus A, X)$$

then we may conclude that $P \setminus A$ satisfies S. The resulting inference rule for the hiding operator is

$$\frac{\begin{array}{l} P \textbf{ sat } ([0, end(s,X)) \times A \subseteq X \Rightarrow S'(s,X)) \\ S'(s, X \cup [0, end(s,X)) \times A) \Rightarrow S(s \setminus A, X) \end{array}}{P \setminus A \textbf{ sat } S(s,X)}$$

The second antecedent states that if S' holds for observation (s,X) when events from A are continuously refused,

$$S'(s, X \cup [0, end(s,X)) \times A)$$

then S should hold of the same observation when events from A are removed from the trace. If we are to find a suitable specification S' for program P, the external specification S must not depend upon the occurrence of events from A.

The program $P \underset{t}{\not\,} Q$ behaves as P until time t, when control is transferred to program Q.

$$\frac{\begin{array}{l} P \ \mathbf{sat}\ S(s, X) \\ Q \ \mathbf{sat}\ T(s, X) \end{array}}{\begin{array}{l} P \underset{t}{\not\,} Q \ \mathbf{sat}\ \exists\, s_P, s_Q \bullet s = s_P \frown s_Q \wedge end(s_P) \leqslant t \leqslant begin(s_Q) \\ \qquad\qquad\qquad\qquad S(s_P, X \uparrow [0, t)) \wedge T((s_Q, X) - t) \end{array}}$$

An observation of this program is simply an observation of P, ending at or before time t, followed by an observation of Q.

An observation of the interrupt program $P \bigtriangleup Q$ is an observation of P and Q in which P does nothing once Q has started to communicate. Any event refused before Q starts must be refused by Q as well as P.

$$\frac{\begin{array}{l} P \ \mathbf{sat}\ S(s, X) \\ Q \ \mathbf{sat}\ T(s, X) \end{array}}{P \bigtriangleup Q \ \mathbf{sat}\ \exists\, s_P, s_Q \bullet s = s_P \frown s_Q \wedge S(s_P, X \uparrow [0, begin(s_Q))) \wedge T(s_Q, X)}$$

If the interrupt has not taken place, then the second trace (s_Q) is empty.

To describe the behaviour of parallel combinations, we will need to restrict our attention to the occurrence and availability of events from a particular interface set. We define a projection operator on traces and refusals:

$$\begin{array}{rcll} \langle\rangle \upharpoonright A & = & \langle\rangle & \\ (\langle(t, a)\rangle \frown s) \upharpoonright A & = & \langle(t, a)\rangle \frown (s \upharpoonright A) & \text{if } a \in A \\ & & s \upharpoonright A & \text{otherwise} \\ X \upharpoonright A & = & X \cap (\mathit{TIME} \times A) & \end{array}$$

and an operator to remove the timing information from a trace:

$$\begin{array}{rcl} \text{events}(\langle\rangle) & = & \langle\rangle \\ \text{events}(\langle(t, a)\rangle \frown s) & = & \langle a\rangle \frown \text{events}(s) \end{array}$$

Suppose that (s, X) is an observation of the network

$$\|_{A_i}\, P_i$$

where the collection of programs P_i is indexed by some finite set I. For every index i, the restriction of trace s to the set $A_i^{\checkmark}$ is the trace of events performed by the corresponding program P_i. Furthermore, trace s contains only events drawn from set $\bigcup_i A_i^{\checkmark}$:

$$s \upharpoonright \bigcup_i A_i^{\checkmark} = s \quad \wedge \quad \forall\, i \bullet \exists\, s_i \bullet s \upharpoonright A_i^{\checkmark} = s_i$$

Suppose that (s_i, X_i) is the corresponding observation of component P_i. We can choose these observations such that

$$\sigma(X_i) \quad \subseteq \quad A_i^{\checkmark}$$

Any event from set $A_i^{\checkmark}$ will require the cooperation of P_i, so

$$X_i \quad \subseteq \quad X \upharpoonright A_i^{\checkmark}$$

and the inference rule for parallel combination is

$$\frac{\forall\, i \in I \bullet P_i \ \mathbf{sat}\ S_i(s, X)}{\begin{array}{ll} \|_{A_i} P_i \ \mathbf{sat}\ \forall\, i \in I \bullet \exists\, s_i, X_i \bullet & S_i(s_i, X_i) \wedge \\ & s_i = s \upharpoonright A_i^{\checkmark} \wedge \\ & \sigma(s) \subseteq \bigcup_i A_i^{\checkmark} \wedge \\ & X_i \subseteq X \upharpoonright A_i^{\checkmark} \wedge \\ & X \upharpoonright \bigcup_i A_i^{\checkmark} = \bigcup_i (X_i) \end{array}}$$

To apply this rule, we must choose a suitable behavioural specification S_i for each component program.

In the partially-interleaved parallel combination

$$P \underset{A}{\|} Q$$

the two components are required to synchronise on every event from the common interface A. If components P and Q are observed to perform traces s_P and s_Q, respectively, then the parallel combination may be observed to perform any trace s from the set of interleavings $s_P \underset{A}{\|} s_Q$, defined by

$$s \in s_P \underset{A}{\|} s_Q \quad \Leftrightarrow \quad s \in TT \wedge \forall\, t \bullet \text{events}(s \uparrow t) \in \text{events}(s_P \uparrow t) \underset{A}{\|} \text{events}(s_Q \uparrow t)$$

Any element of this set is a timed trace s such that, for any timed value t, the sequence of events in s at t is an interleaving of the events in s_P and s_Q at this time. The inference rule for the partially-interleaved parallel combination is

$$\frac{\begin{array}{l} P \ \mathbf{sat}\ S(s, X) \\ Q \ \mathbf{sat}\ T(s, X) \end{array}}{\begin{array}{ll} P \underset{A}{\|} Q \ \mathbf{sat}\ \exists\, s_P, s_Q, X_P, X_Q, t \bullet & s \in s_P \underset{A}{\|} s_Q \wedge \\ & X \upharpoonright A^{\checkmark} = X_P \upharpoonright A^{\checkmark} \cup X_Q \upharpoonright A^{\checkmark} \wedge \\ & X \setminus A^{\checkmark} = X_P \setminus A^{\checkmark} = X_Q \setminus A^{\checkmark} \wedge \\ & S(s_P, X_P) \wedge T(s_Q, X_Q) \end{array}}$$

where $X \setminus A = X \cap [\mathit{0}, \infty) \times (\Sigma \setminus A)$

The interleaving operator admits a similar rule:

$$\frac{\begin{array}{l} P \;\mathbf{sat}\; S(s, X) \\ Q \;\mathbf{sat}\; T(s, X) \end{array}}{\begin{array}{ll} P \;|||\; Q \;\mathbf{sat}\; \exists\, s_P, s_Q, X_P, X_Q, t \bullet s \in s_P \underset{\checkmark}{\|}\, s_Q \;\wedge & \\ & X \upharpoonright \{\checkmark\} = X_P \upharpoonright \{\checkmark\} \cup X_Q \upharpoonright \{\checkmark\} \;\wedge \\ & X \setminus \{\checkmark\} = X_P \setminus \{\checkmark\} = X_Q \setminus \{\checkmark\} \;\wedge \\ & S(s_P, X_P) \wedge T(s_Q, X_Q) \end{array}}$$

Once again, we require a recursion induction rule for reasoning about recursive program descriptions. A complete theory of recursion, which provides for the definition of programs by sets of mutually-recursive equations, is presented in [DaS93]. Here, we present the rule for the μ operator:

$$\frac{X \;\mathbf{sat}\; S(s, X) \;\Rightarrow\; F(X) \;\mathbf{sat}\; S(s, X)}{\mu\, X \bullet F(X) \;\mathbf{sat}\; S(s, X)} \quad [\; F \text{ is guarded, } S \text{ is satisfiable} \;]$$

To show that a recursive program $\mu\, X \bullet F(X)$ satisfies a specification S, we have only to show that the specification is satisfiable, invariant under recursive calls, and that the defining function is guarded.

4 Refinement

The semantic models for Timed CSP form a hierarchy, ordered by the information content of the semantic sets. The models are linked by projection mappings [Ree88], represented by arrows in the diagram below; the nature of these mappings ensures that results established in one model remain true as we move upwards. In reasoning about complex systems, we may use the simplest semantic model that is sufficient to express the current requirement, safe in the knowledge that the argument remains valid in the other models of the hierarchy.

$$
\begin{tikzcd}
 & & \mathcal{M}_{TFS} \arrow[ddll] \arrow[d] \arrow[ddrr] & & \\
 & & \mathcal{M}_{UFS} \arrow[dl] \arrow[dr] & & \\
\mathcal{M}_{TF} \arrow[r] \arrow[ddrr] & \mathcal{M}_{UF} \arrow[dr] & & \mathcal{M}_{US} \arrow[dl] & \mathcal{M}_{TS} \arrow[l] \arrow[ddll] \\
 & & \mathcal{M}_{UT} & & \\
 & & \mathcal{M}_{TT} \arrow[u] & &
\end{tikzcd}
$$

Figure 1: Reed's models for Timed CSP

The untimed models of CSP occupy the lowest positions in the hierarchy, with the untimed traces model $\mathcal{M}_{UT}$ at the very bottom.

To establish that a program P satisfies a specification of the form $S \wedge T$ we begin by choosing the simplest model $\mathcal{M}$ in which both S and T may be expressed. If one of these conditions—say S—can be expressed as a behavioural specification S' in a smaller model $\mathcal{M}'$, without loss of information, then we may be able to establish S by reasoning within $\mathcal{M}'$. If the two specifications are equivalent, and we can prove that P satisfies S' in $\mathcal{M}'$, then our refinement relation will allow us to deduce that P satisfies S in the larger model $\mathcal{M}$.

If the larger model is a timed model, and the smaller model is an untimed model, it may be helpful to remove the timing information from the description of program P. We define a syntactic abstraction Θ upon the syntax of Timed CSP, removing the timing information from timed operators. The definition of Θ is entirely obvious: the delay operator is the only operator without a untimed equivalent, and we define

$$\Theta(\mathit{Wait}\ t) \quad = \quad \mathit{Skip}$$

for any time t.

We are required to prove that the projection $\Theta(P)$ satisfies a condition S', equivalent to the original specification S, but expressed in terms of model $\mathcal{M}'$. For example, suppose that we wish to establish that a program P meets a conjunction of safety and liveness properties $\mathit{SAFE} \wedge \mathit{LIVE}$, and that

* the safety condition SAFE depends only upon the order of occurrence of certain events; in this case, we may be able to establish SAFE using the untimed trace model
* a proof that the program satisfies the liveness condition LIVE will require consideration of the timing properties of components of P; in this case, we must employ the timed failures model

We formalise the specification of P using the larger of the two models

$$\mathit{SAFE}(s, X) \wedge \mathit{LIVE}(s, X)$$

and define an untimed trace specification $\mathit{SAFE}'(tr)$ which is equivalent to SAFE, but is expressed in terms of the untimed traces model:

$$\forall\, s \in \mathit{TT} \quad \bullet \quad \mathit{SAFE}(s) \Leftrightarrow \mathit{SAFE}'(\text{events}\, s)$$

Our proof obligation is then reduced to showing that

$$\begin{array}{rcl} \Theta(P) & \mathbf{sat} & \mathit{SAFE}'(tr) \\ P & \mathbf{sat} & \mathit{LIVE}(s, X) \end{array}$$

This reduction is justified by a refinement proof rule from $\mathcal{M}_{UT}$ to $\mathcal{M}_{TF}$:

$$\frac{\begin{array}{l} \Theta(P)\ \mathbf{sat}\ S'(tr)\ \text{in}\ \mathcal{M}_{UT} \\ \forall\, s \in \mathit{TT} \bullet S'(\text{events}\, s) \Leftrightarrow S(s) \end{array}}{P\ \mathbf{sat}\ S(s)\ \text{in}\ \mathcal{M}_{TF}}$$

This rule follows from a more general theory of refinement, linking all of the semantic models in our hierarchy. If models $\mathcal{M}$ and $\mathcal{M}'$ are linked by a projection mapping π, then we may define a refinement relation $\sqsubseteq_\pi$ on programs, such that

$$\frac{P'\ \mathbf{sat}\ S'\ \text{in}\ \mathcal{M}'}{P\ \mathbf{sat}\ S\ \text{in}\ \mathcal{M}} \quad [\ P' \sqsubseteq_\pi P\ ]$$

where S' is an equivalent form of specification S, expressed in terms of model $\mathcal{M}'$.

5 The evolution of Timed CSP

5.1 The erosion of the delta

The most important change in the language and models of Timed CSP was the disappearance of the universal delay constant δ. In [ReR86], this delay was associated with every recursive call, and every event prefix operation. This supported the assertion that every recursive program had a valid semantics; every recursive call was guarded by a delay of at least δ.

The association of δ with event prefix prevented the simultaneous observance of causally-related events. It can be argued that, if the observance of event b is contingent upon the occurrence of event a, it should be impossible to observe both events at the same time. This treatment of timed observations meant that timed traces contained no more information than a bag—or multiset—of timed events: the ordering of events is determined only by their time of occurrence.

The multiset view of timed observations was included as an axiom in the semantic models of [Ree88]. This axiom insisted that, if a and b are observed at a single time instant, then they may be recorded in either order. In the timed failures model, for example, the axiom took the form

$$(s, X) \in S \wedge s \cong w \quad \Rightarrow \quad (w, X) \in S$$

This states that, for any process S, if (s, X) is a possible observation of S and w is trace-equivalent to s, then (w, X) is also a possible observation of S. Two traces are equivalent if they differ only in the order of appearance of simultaneous events.

However, aspects of *instant causality* are present in other aspects of the language and semantic models. In a sequential composition, control is passed to the second program at the instant the first program terminates: this does not contradict the above axiom, because the termination event is not observable, but the resulting semantics conflicts with our operational intuition.

We may improve the situation by associating a δ delay with every sequential operator, as in [Sch90] and [Dav93], but the presence of these delays makes the notation difficult to use, and the language lacks certain algebraic properties: for example, the familiar identities

$$\begin{array}{rcl} \mu X \bullet F(X) & \equiv & F(\mu X \bullet F(X)) \\ a \to Stop \mid\mid\mid b \to Stop & \equiv & (a \to b \to Stop) \\ & & \Box \\ & & (b \to a \to Stop) \end{array}$$

fail to hold in the semantic models of [Ree88].

Furthermore, a degree of instant causality is present in our treatment of refusal information. Consider the following external choice program:

$$a \to Stop \quad \Box \quad b \to Stop$$

If this program engages in event a at time t, then event b is unavailable from time t onwards. The withdrawal of the offer is instantaneous: in a timed observation, the refusal information at time t is subsequent to the trace information at time t.

By rejecting the multiset view, and adopting a more abstract view of timed observations—in which the order of simultaneous events is important—we obtain a simpler, more consistent semantics for the language. In the current models of Timed CSP, the only operator to introduce a delay is the delay operator, *Wait*. In particular, the operations of recursion and prefix are instantaneous, and the above identities are restored. The new language supports the definition of programs by sets of mutually-recursive equations. The removal of the trace equivalence axiom allows us to establish a closer relation with the untimed models of CSP, facilitating timed refinement of programs and processes.

There is a small price to pay. Without the constant delay δ, we have no guarantee that a recursive program has a valid semantics. We need to check that every recursive invocation is guarded by some non-zero delay. This is almost always a simple syntactic check upon the program in question: see [DaS93].

5.2 Infinite choice

The language of [ReR86] did not include constructs for infinite choice. In [Ree88, Sch90], the language was extended to include prefix choice: the program

$$a : A \to P_a$$

offers a choice of initial events from a possibly-infinite set A. The subsequent behaviour is dependent upon the name of the first event performed. This construct allows us to model communication along a channel. We define the operation of channel input as follows:

$$c?x \to P_x \quad = \quad c.v : \{c.v \mid v \in V\} \to P_v$$

where c is a channel, and v is an element of channel data type V.

If the choice set is infinite, it is necessary to place a restriction upon the set of options $\{a \to P_a \mid a \in A\}$. One of the assumptions of our computational model is *finite variability*, which states that a program may undergo only finitely many changes of state during a finite interval of time. In the finite timed models of CSP, we require the *bounded speed* axiom to guarantee finite variability:

$$\forall\, t \bullet \exists\, n \in \mathbb{N} \bullet s \in S \wedge end(s) < t \Rightarrow \#(s) \leqslant n$$

This states that, for any time t, there is a natural number n such that *every* trace s ending before t contains no more than n events.

To ensure that the semantics of a prefix choice program satisfies this requirement, we must check that the set of alternative programs $\{P_a \mid a \in A\}$ is uniformly bounded—that there exists a function $n : TIME \to \mathbb{N}$

$$\forall\, i : I \,;\, t : TIME \quad \bullet \quad s \in S \wedge end(s) < t \Rightarrow \#(s) \leqslant n(t)$$

The function n provides a uniform bound for every program in the set of alternatives.

The combination of infinite prefix choice and the hiding operator introduces infinite nondeterministic choice into our language. Consider the following program:

$$in?n : \mathbb{N} \rightarrow \mathit{Wait}\ n \ ; out!n \rightarrow \mathit{Stop}$$

This program is initially prepared to accept any natural number on channel in. If a number n is received on in, the program delays for n time units before offering to deliver n on channel out. If we hide all communications on channel in, we are left with a nondeterministic choice program

$$\textstyle\bigsqcap_{n \in \mathbb{N}} \mathit{Wait}\ n \ ; out!n \rightarrow \mathit{Stop}$$

It is therefore sensible to introduce an indexed nondeterministic choice operator for the language, with the requirement that the set of alternative programs is uniformly bounded.

5.3 Infinite observations

The standard models of Timed CSP are based upon finite observations. Within these models, we have only a set of finite approximations to the behaviour of a program over the entire time domain. For most applications, this presents no difficulties—finite approximations are perfectly adequate. However, if we wish to address issues like eventuality, fairness, and unbounded nondeterminism, we require a more sophisticated treatment of infinite observations.

For example, extending the nondeterministic choice program with the deadlock program *Stop* leaves the set of finite observations unchanged:

$$\textstyle\bigsqcap_{n \in \mathbb{N}} \mathit{Wait}\ n \ ; out!n \rightarrow \mathit{Stop} \quad = \quad \mathit{Stop} \sqcap \bigsqcap_{n \in \mathbb{N}} \mathit{Wait}\ n \ ; out!n \rightarrow \mathit{Stop}$$

To distinguish arbitrary waiting from infinite waiting, we must include infinite observations in semantic sets. The observation

$$(\langle\rangle, [0, \infty) \times \{out.n \mid n \in \mathbb{N}\})$$

would be associated with the right-hand program—every communication on channel out may be refused over the interval $[0, \infty)$—but not with the left-hand program. The above equality does not hold in any model which includes infinite observations.

The inclusion of infinite timed traces is also essential for modelling infinite behaviours. Consider the programs P and Q defined by

$$\begin{array}{rclcrcl} P & = & \bigsqcap_{n \in \mathbb{N}} P_n & \qquad & P_0 & = & \mathit{Stop} \\ & & & & P_{n+1} & = & a \rightarrow \mathit{Wait}\ 1 \ ; P_n \\ Q & = & a \rightarrow \mathit{Wait}\ 1 \ ; Q & & & & \end{array}$$

where a is any event from our universal alphabet. Program P is capable of performing an arbitrary number of a events, while program Q is capable of performing an infinite number. Without infinite traces, we are unable to distinguish P from $P \sqcap Q$. In an infinite model, an infinite trace of a events would be associated only with the latter.

The infinite timed model of Timed CSP, first introduced in [Sch91], has several advantages:

* the introduction of infinite observations allows us to guarantee finite variability without the use of the bounded speed condition. We may dispense with any restriction upon the use of the indexed choice operators mentioned above. Further, we obtain the result that, for any specification S, there is a least deterministic program P which satisfies S.
* unbounded timed refusals correspond more closely to untimed refusal sets. Offers and refusals recorded in untimed CSP are based upon eventualities: we consider an event to be refused if it is eventually refused forever. Infinite refusal sets allow us to express this condition, and we are able to establish a refinement relation between timed and untimed models.
* by describing complete executions, we are able to address fairness requirements in a timed context, see [DaS92], and support the temporal logic concepts of *always* and *eventually*.

The principal disadvantage of the infinite timed model lies in the complexity of the fixed point theory required to give a suitable semantics to recursive programs: the model is neither a complete metric space nor a complete partial order.

5.4 Timeout

Although the addition of *Wait* to the untimed syntax allows to simulate certain forms of timeout and interrupt behaviour, the simulation is too complicated to be practical. A more satisfactory solution comes from treating such operators as language primitives. Although the extended language is harder to reason about—there are more cases to consider—it is easier to reason *with*.

The first primitive to be added was the timeout operator $\rhd$. In the program

$$P \overset{t}{\rhd} Q$$

control is transferred from P to Q at time t if no communications have occurred. If an attempt at communication involving P is made at time t precisely, then the outcome will be nondeterministic. If either of the subprograms should terminate, then the timeout program terminates immediately.

Without the time parameter, the timeout operator is an operator of untimed CSP. The program $P \rhd Q$ may behave as Q, or offer a choice between P and Q, according to whether the timeout has occurred, or not.

$$P \rhd Q \quad = \quad (P \,\Box\, Q) \sqcap Q$$

The resulting behaviour is that of a nondeterministic choice: without timing information, we cannot determine when the timeout occurs.

It is possible to define the delay operator *Wait* using the timeout operator:

$$\mathit{Wait}\ t \quad = \quad \mathit{Stop} \overset{t}{\rhd} \mathit{Skip}$$

5.5 Interrupt

To describe systems in which a component program may be interrupted during execution, an interrupt operator was added to the language. The program

$$P \bigtriangleup Q$$

behaves as P but may be interrupted by any initial event of Q. At this point, control is transferred to Q and process P is discarded.

Alternatively, a program may be interrupted after a predetermined time has elapsed. The program

$$P \underset{t}{\lightning} Q$$

behaves as P until time t, when control is transferred to Q. Without the time parameter, the transfer operator $\lightning$ resembles the interrupt operator of [Hoa85]: the first program may be interrupted at any time.

5.6 Distributed termination

Termination in CSP and Timed CSP is modelled by the observation of the special termination event $\checkmark$. In the original language description, any component of a parallel construct can signal termination—indicating that the whole program has terminated—while others are still executing. This conflicts with our intended treatment of termination; a parallel combination should not offer to synchronise upon the event $\checkmark$ unless all components are ready to terminate. Indeed, a similar condition is placed upon parallel combinations in [Hoa85]: here, asynchronous executions may not signal termination.

In the current language of Timed CSP, parallel programs may only terminate when all their components are ready. In the parallel combinations

$$P\ {}_A\|_B\ Q \quad \text{and} \quad P \underset{C}{\|} Q$$

the special event $\checkmark$ is implicitly present in each interface set, and the interleaving operator admits the following equivalence

$$P \mathbin{|||} Q \quad = \quad P \underset{\{\checkmark\}}{\|} Q$$

where the partially-interleaved parallel operator to the right allows both components to execute independently, synchronising only upon events from the common interface.

6 Timed process algebras

In recent years, a variety of process algebras have been developed for the analysis of timed systems. Four approaches have been adopted.

The bisimulation approach

Programs are given an operational semantics—the meaning of a program is given by a tree of possible transitions describing the possible executions of the program. Programs are considered to be equivalent iff their execution trees are *bisimilar*. A bisimulation is a relation between tree structures: two nodes correspond if the sets of subsequent transitions are equivalent. The notion of equivalence depends upon the flavour of bisimilarity employed.

Specifications are programs in the process algebra, or properties of execution trees. In the first case, programs are proved correct by establishing that the semantics of the program is bisimilar to the semantics of the specification. In the second, we must establish that the semantics of the program has the specified property. These properties may be expressed in graph-theoretic terms, or within a modal logic such as Hennessy-Milner logic [HeM85]. The timed process algebras which adopt this approach include TCCS [MoT90], Timed CCS [Wan90, Lia91], CCSiT [Dan92], ATP [Nic90] and TPCCS [Han91].

The testing approach

As in the bisimulation approach, programs are given an operational semantics, but equivalence is not defined by relations on synchronisation trees. Instead, programs are characterised by their possible interactions with testing programs. Two programs are equivalent under a certain class of testing program if no test from this class can distinguish them. Furthermore, a program P may be said to refine another program Q if it passes every test that Q passes. A specification consists of a program S and a relation R, which must hold between S and any proposed implementation. This approach is taken in Hennessy and Regan's TPL [HeR91].

The algebraic approach

Program equivalence is defined directly by a complete set of algebraic laws. A specification is a program, and a proposed implementation may be verified using laws which define a refinement relation. This approach is often used in conjunction with either testing or bisimulation equivalence. Given a complete axiomatisation for the operational equivalence, both algebraic and operational techniques may be applied. This approach is taken in RTPA [BaB91], and Liang Chen's Timed CCS [Lia91].

The denotational approach

Programs are associated with elements of a denotational domain; two programs are equivalent if they are associated with the same object. A specification is a predicate upon elements of the denotational domain, and a program satisfies a specification if the defining predicate holds of its semantics. The same language may be given an operational semantics, as in APA [Jef92] and Timed CSP [Sch91], or a complete set of algebraic laws [OrF91]. The denotational approach is also taken in [BoG87], [Hoo91], and [Zwa86].

A characteristic of the use of denotational models is the separation of programs and specifications. A specification language—such as a temporal logic [Bar87]—can be given an interpretation in the denotational domain, and used to capture program requirements in a property-oriented fashion. As an example, consider the requirement that two events a and b occur alternately, beginning with an occurrence of event a. In CSP, this can be expressed as a untimed trace specification

$$\text{alternate}\ tr \quad = \quad tr \upharpoonright \{a, b\} \leqslant \langle a, b \rangle^{\omega}$$

where $\langle a, b\rangle^{\omega}$ is the sequence $\langle a, b, a, b, a, b, \ldots\rangle$. If we were to express this requirement as a program in untimed CSP, we might employ a mutual recursion

$$\begin{aligned} P_a &= (a \rightarrow P_b) \mathrel{\Box} (c \rightarrow P_a) \sqcap \mathit{Stop} \\ P_b &= (b \rightarrow P_a) \mathrel{\Box} (c \rightarrow P_b) \sqcap \mathit{Stop} \end{aligned}$$

Any other events which are possible for the program must be included in the program description; in the above description, we have allowed for the possibility of a third event c.

The requirement does not insist that either of the events is performed—no liveness condition is present—so we must allow for the possibility that an implementation may cease to perform a's and b's at any time; this is the reason for the nondeterministic choice above. More generally, the requirement might be described by a state machine

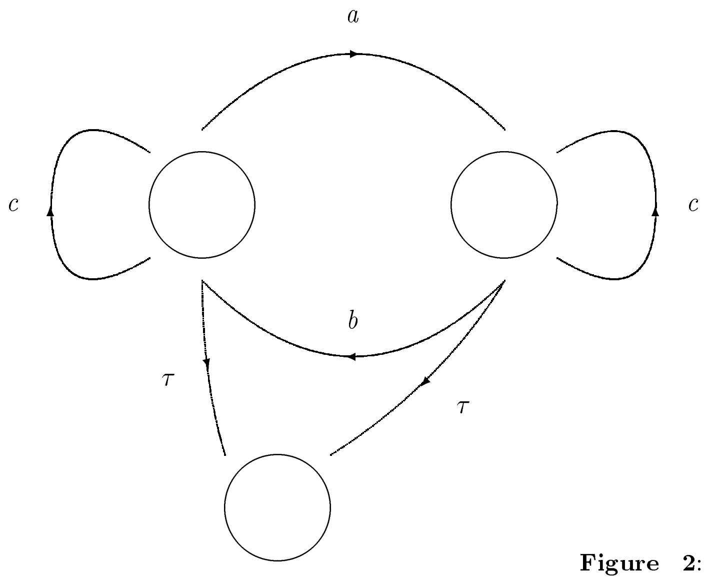

Figure 2: An alternating state machine

This machine alternates between two live states, according to whether a or b is proscribed, but may move to a deadlocked state at any time by performing the internal action τ.

The practice of using programs to capture requirements is more successful at higher levels of abstraction. As more information is added to the semantics of the language, using programs in this way leads to over-specification—additional requirements are placed upon the implementation—or complicated expressions which are difficult to relate to the original intention.

Further, it is difficult to combine programs which represent requirements in an intuitive and compositional manner; the result of combining two such programs may not correspond to the desired combination of requirements. This problem is eliminated by the use of a denotational semantic model for specification. The nature of denotational semantics guarantees compositionality.

We will often wish to examine the behaviour of the same program at different levels of abstraction. In the operational approach, this may be done by considering different notions of program equivalence. In the denotational approach, a different semantic model is required. In Timed CSP, different semantic models may be used in the same specification: Reed's hierarchy supports a uniform theory of program verification at several levels of abstraction.

Acknowledgements

This paper was written when both authors were at Oxford Programming Research Group. We gratefully acknowledge the insightful comments and useful criticism offered by Bill Roscoe, Mike Reed, Tony Hoare, Gavin Lowe, Dave Jackson, Karen Seidel, Mike Mislove, Alan Jeffrey, and the anonymous referees.

Jim Davies received support from DRA Malvern, and would like in particular to thank Colin O'Halloran. Steve Schneider was funded by the Esprit BRA SPEC and a SERC Postdoctoral fellowship B91/RFH/312. Support was also received from the Esprit BRA REACT, and the US Office of Naval Research.

References

[BaB91] J. C. M. Baeten and J. A. Bergstra, *Real time process algebra,* Formal Aspects of Computing, Volume 3, Number 2, 1991

[Bar87] H. Barringer, *The use of temporal logic in the compositional specification of concurrent systems,* in *Temporal Logics and their Applications*, Academic Press 1987

[BrR85] S. D. Brookes and A. W. Roscoe, *An improved failures model for communicating sequential processes,* Proceedings of Pittsburgh Seminar on Concurrency, Springer LNCS 197, 1985

[BoG87] A. Boucher and R. Gerth, *A timed model for extended communicating sequential processes,* Proceedings of ICALP '87, Springer LNCS 267, 1987

[Dan92] M. Daniels, *Modelling real-time behaviour with an interval time calculus,* Proceedings of Formal Techniques in Real-time and Fault-tolerant Systems, Springer LNCS 571, 1992

[DaS92] J. Davies and S. A. Schneider, *Using CSP to verify a timed protocol over a fair medium,* in proceedings of CONCUR 92, Springer LNCS 630, 1992

[DaS93] J. Davies and S. A. Schneider, *Recursion induction for real-time processes*, Formal Aspects of Computing 5 (6), 1993

[Dav93] J. Davies, *Specification and proof in real-time CSP*, Cambridge University Press 1993.

[Han91] H. A. Hansson, *Time and probability in formal design of distributed systems,* doctoral dissertation, Uppsala University, 1991

[HeM85] M. Hennessy and R. Milner, *Algebraic laws for nondeterminism and concurrency,* Journal of the ACM 32, 1985

[HeR91] M. Hennessy and T. Regan, *A process algebra for timed systems,* Technical Report 5–91, School of Cognitive and Computing Sciences, University of Sussex 1991

[Hoa78] C. A. R. Hoare, *Communicating Sequential Processes,* Communications of the ACM 21, 1978

[Hoa85] C. A. R. Hoare, *Communicating Sequential Processes,* Prentice-Hall International 1985

[Hoo91] J. Hooman, *Specification and compositional verification of real-time systems*, Springer LNCS 558, 1991

[Jef92] A. S. Jeffrey, *Observation spaces and timed processes,* Oxford University D.Phil thesis 1992

[Lia91] Liang Chen, *An interleaving model for real-time systems*, LFCS report series, ECS-LFCS-91-184, University of Edinburgh 1991

[MoT90] F. Moller and C. Tofts, *A temporal calculus of communicating systems*, Proceedings of CONCUR 90, Springer LNCS 458, 1990

[Nic90] X. Nicollin, J.-L. Richier, J. Sifakis and J. Voiron, *ATP: an algebra for timed processes,* Proceedings of the IFIP Working Conference on Programming Concepts and Methods, 1990

[OrF91] Y. Ortega-Mallen and D. de Frutos-Escrig, *A complete proof system for timed observations*, Proceedings of TAPSOFT 91, Springer LNCS 493, 1991

[Ree88] G. M. Reed, *A uniform mathematical theory for real-time distributed computing,* Oxford University D.Phil thesis 1988

[ReR86] G. M. Reed and A. W. Roscoe, *A timed model for communicating sequential processes,* Proceedings of ICALP'86, Springer LNCS 226, 1986

[ReR87] G. M. Reed and A. W. Roscoe, *Metric spaces as models for real-time concurrency,* Proceedings of the Third Workshop on the Mathematical Foundations of Programming Language Semantics, Springer LNCS 298, 1987

[ReR91] G. M. Reed and A. W. Roscoe, *Analysing* TM_{FS}*: a study of nondeterminism in real-time concurrency*, Springer LNCS 491, 1991

[Ros82] A. W. Roscoe, *A mathematical theory of communicating processes,* Oxford University D.Phil thesis 1982

[Sch90] S. A. Schneider, *Correctness and communication in real-time systems,* Programming Research Group Monograph PRG-84, Oxford University 1990

[Sch91] S. A. Schneider, *Unbounded nondeterminism in Timed CSP,* Esprit SPEC project deliverable 1991

[Wan90] Wang Yi, *Real-time behaviour of asynchronous agents,* Proceedings of CONCUR 90, Springer LNCS 458, 1990

[Zwa86] A.E. Zwarico, *A formal model of real-time computing,* University of Pennsylvania Technical Report 1986