Timed CSP: Theory and Practice

Oxford University Timed CSP Group[1]

Programming Research Group
Oxford University Computing Laboratory
11 Keble Road
Oxford OX1 3QD

Abstract. Over the past five years, G. M. Reed and A. W. Roscoe have directed a large group of research staff and graduate students at Oxford University in a comprehensive study of their theory of Timed CSP. This theory has now matured with associated proof systems, temporal logics, and refinement methods, and it has been employed in several realistic case studies. This paper presents an overview of the Oxford work on Timed CSP.

Keywords: Concurrency, Real-Time CSP, Temporal Logic, Timewise Refinement, Specification and Verification, Robotics, Telephone Switching, Control Software

Contents

[1]The work reported in this paper is due to S.R. Blamey, J. Davies, D.M. Jackson, A. Kay, M.W. Mislove, G.M. Reed, J.N. Reed, A.W. Roscoe, B. Scattergood, S.A. Schneider, R. Stamper, S. Superville, and A. Wallace. [The paper itself was written by Davies, Jackson, the Reeds, Roscoe, and Schneider, who take responsibility for any errors.] This work has been supported by the U.S. Office of Naval Research, Esprit BRA SPEC, Esprit REX, SERC, RSRE, Rolls Royce, BP, and Formal Systems (Europe).

1 Introduction

In [ReR86, ReR87, Ree88, Ree90, ReR91] G. M. Reed and A. W. Roscoe developed a hierarchy of timed and untimed models for CSP. This mathematical hierarchy supports a uniform treatment of concurrent processes at different levels of abstraction: 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.

The links between the various untimed and timed models in the hierarchy are well-established, and a large group of academic staff, research staff, and graduate students at Oxford under the direction of Reed and Roscoe are exploring the use of this hierarchy in the design and specification of real-time processes. This group includes: J.N. Reed (senior research associate); J. Davies, A. Kay, and S.A. Schneider (research officers); D.M. Jackson, G. Lowe, and B. Scattergood (doctoral students).

Members of the above group have (1) made the explicit addition of channel communication [Sch90]; (2) developed a complete behavioural proof system [DaS89, Sch90, Dav91]; (3) developed a theory of timewise refinement, using relating timed processes to untimed ones in a manner that supports the promotion of correctness results within the hierarchy of semantic models [Sch90, Dav91]; (4) developed a temporal logic consistent with the existing timed semantics [Jac90], and produced a complete temporal logic proof system [Jac91]; (5) constructed an operational semantics for the language [Sch92]; (6) developed a new fixed-point theory based on the operational semantics, and used this fixed-point theory to construct a model based on infinite behaviours, extending Roscoe's work on unbounded nondeterminism in untimed CSP to timed processes [Sch91, MRS91]; (7) constructed the first timed probabilistic model for CSP [Low91]; (8) extended the theory to include an element of broadcast concurrency [Dav91, DJS92].

The theory of Timed CSP has been successfully applied to the usual examples: the alternating bit protocol, a sliding window protocol, and a watchdog timer, see [Ja+90, Sc+90]. More significantly, the theory has been applied by members of this group—and others at Oxford—to the design of control software for aircraft engines [Jac89], to real-time robotics [Sca90, Sta90, Wal91], to the specification of a realistic telephone switching network [KaR90, Su91], and to the verification of a local area network protocol [Dav91].

In this paper, we outline the theory of Timed CSP and examine some common themes from the above applications.

2 The Language of Timed CSP

The language of Timed CSP introduced in [ReR86] is a simple extension of Hoare's Communicating Sequential Processes. The word *process* is used to denote the behaviour pattern of an object, viewed through the occurrence and availability of certain *events*—atomic communications between an object and its environment. Every event is drawn from a universal alphabet of synchronisations Σ. The language of untimed CSP consists of several process constructors, including primitives for parallel composition, nondeterministic choice and hiding.

Note: In this paper, we will present the language of Timed CSP and we will limit our discussion primarily to the Timed Failures Model. We are currently preparing the definitive text on Timed CSP, in which a complete discussion of the language and its hierarchy of semantic models will be given.

2.1 Syntax

In Timed CSP, each of the untimed CSP operators is interpreted in a timed context, and two timing operators are added: delay and timeout. The syntax of *TCSP* terms is given by the following BNF rule:

$$\begin{aligned} P \ ::= \ & STOP \mid SKIP \mid WAIT\, t \mid a \xrightarrow{t} P \mid P\,;P \mid \\ & P \,\Box\, P \mid P \sqcap P \mid P \overset{t}{\triangleright} P \mid P \,{}_A\|_A\, P \mid P \,|||\, P \mid \\ & f(P) \mid P \setminus A \mid \mu X \circ F(X) \end{aligned}$$

In the above 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 term *STOP* represents a broken process which will never engage in external communication, corresponding to the undesirable phenomena of deadlock and divergence. The term *SKIP* represents a process which does nothing except terminate immediately. As in untimed CSP, the special event $\surd$ is used to indicate that a process has terminated. The term *WAIT* t is a delayed form of *SKIP*. It represents a process which does nothing except terminate successfully after time t.

The prefix operator $\rightarrow$ allows us to introduce communication events into the behaviour pattern of a process. In untimed CSP, the term $a \rightarrow P$ models a system which is initially prepared to engage in event a, and then eventually behaves as P. To model real-time systems, we must be able to model constraints upon the time between the observation of an a and the onset of P.

The process $a \xrightarrow{t} P$ will behave as P precisely t time units after the synchronisation a is observed. The relation between prefix and delay is an obvious one:

$$a \xrightarrow{t_1+t_2} P \ \equiv \ a \xrightarrow{t_1} WAIT\, t_2\,;P$$

For example, a process that makes event b available *exactly* four seconds after event a is observed, only to halt two seconds after b is performed, would be written:

$$PROC \ == \ a \xrightarrow{4} b \xrightarrow{2} STOP$$

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.

The sequential composition operator provides a means of transferring control on termination. In the construct $P\,;Q$, control is passed from P to Q if and when P performs

the termination event $\checkmark$. This event is not visible to the environment, and occurs as soon as it becomes available.

Timed CSP provides two forms of choice: external and internal. An external choice $P \,\Box\, Q$ may be resolved by the environment. If the environment is prepared to cooperate with P, but not with Q, then the choice behaves as P: the choice is resolved by the first communication. In contrast, the environment has no influence over an internal choice $P \sqcap Q$: the outcome of such a choice is nondeterministic.

The timeout operator $P \overset{t}{\rhd} Q$ transfers control from P to Q if no communications occur before time t. As above, this time may be any non-negative real number. If an attempt at communication involving P is made at time t precisely, then the outcome will be nondeterministic. If either of the components should terminate, then the entire timeout construct terminates immediately.

In Timed CSP, the parallel combination of two terms P and Q is parametrised by two sets of events. In the construct

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

term P may perform only those events in A, term Q may perform only those events in B, and the two terms must cooperate on events drawn from the intersection of A and B. The asynchronous parallel combinator, $|||$, allows both components to evolve concurrently without interacting.

The hiding operator provides a mechanism for abstraction in Timed CSP. The term $P \backslash A$ behaves as P except that events in A are concealed from the environment. Concealed events no longer require the cooperation of the environment, and so occur as soon as P is ready to perform them. The relabelled term $f(P)$ has a similar control structure to term P, with observable events renamed according to function f.

We use the expression $\mu X \circ F(X)$ to denote the unique fixed point of the semantic domain mapping represented by F. Only recursions that admit such fixed points are allowed.

2.2 The Timed Failures Model TM_F

We will now provide a denotational semantic model to facilitate a formal description of the language, based upon the Timed Failures model presented in [Ree88]. In our model, each piece of process algebra will be associated with a set of observations. Each observation is represented by a pair $(s, \aleph)$. The first component is a timed trace s: a record of timed events observed. The second is a timed refusal $\aleph$: a record of timed events refused.

A timed event is a pair (t, a), where t is a time value and a is a communication event. The domain of time values is defined to be

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

The set of all timed events is thus

$$T\Sigma \;\; == \;\; TIME \times \Sigma$$

A timed trace is a chronologically ordered sequence of timed events; the set of all timed traces is given by:

$$T\Sigma^*_{\leqslant} \;==\; \{s \in \mathrm{seq}\, T\Sigma \mid (t,a) \text{ precedes } (t',a') \text{ in } s \Rightarrow t \leqslant t'\}$$

The presence of a timed event (t, a) in a timed trace will correspond to the observation of the synchronisation a at time t.

A timed refusal set is a set of timed events, corresponding to a finite union of refusal tokens. Each token is a Cartesian product set of the form $I \times A$, where I is a half-open time interval and A is a set of events. If we take $TINT$ to be the set of all such intervals, and $RTOK$ to be the set of all refusal tokens, then the set of all timed refusal sets is defined by

$$RSET \;==\; \{\bigcup C \mid C \subseteq_{\mathrm{fin}} RTOK\}$$

where

$$\begin{aligned} TINT &== \{[b,e) \mid 0 \leqslant b < e < \infty\} \\ RTOK &== \{I \times A \mid I \in TINT \wedge A \in \mathbb{P}\Sigma\} \end{aligned}$$

The presence of a timed event (t, a) in refusal set $\aleph$ corresponds to the refusal of the process to engage in synchronisation a at time t.

The set of possible observations, or *timed failures*, is given by TF, where

$$TF \;==\; T\Sigma^*_{\leqslant} \times RSET$$

2.3 Notation

We inherit the following operators from [Hoa85]:

$\frown$	concatenation of traces	**in**	contiguous subsequence
$\langle\rangle$	the empty trace	$\#$	length of a sequence
$\leqslant$	trace prefix	$<$	strict trace prefix

The predicate s_1 **in** s_2 holds precisely when trace s_1 is a contiguous subsequence of s_2. If s is a timed trace, then $\#(s)$ returns the number of timed events in that trace.

The *begin* operator returns the time of occurrence of the first event in a timed trace, refusal, or failure:

$$\begin{aligned} begin(s) &== \min(\{t : TIME \mid \exists\, a : \Sigma \bullet (t,a) \in s\} \cup \{\infty\}) \\ begin(\aleph) &== \inf(\{t \mid \exists\, a \bullet (t,a) \in \aleph\} \cup \{\infty\}) \\ begin(s,\aleph) &== \min\{begin(s), begin(\aleph)\} \end{aligned}$$

Similarly, the *end* operator returns the time of occurrence of the last event in a timed trace, refusal, or failure:

$$\begin{aligned} end(s) &== \max(\{t : TIME \mid \exists\, a : \Sigma \bullet (t,a) \in s\} \cup \{0\}) \\ end(\aleph) &== \sup(\{t \mid \exists\, a \bullet (t,a) \in \aleph\} \cup \{0\}) \\ end(s,\aleph) &== \max\{begin(s), begin(\aleph)\} \end{aligned}$$

We define the *during* ($\uparrow$) operator on timed traces and refusals, returning the part of the trace or refusal that lies in a specified time interval:

$$\begin{array}{rcll} \langle\rangle \uparrow I & == & \langle\rangle & \\ (\langle(t,a)\rangle^\frown s) \uparrow I & == & \langle(t,a)\rangle^\frown(s \uparrow I) & \text{if } t \in I \\ & & (s \uparrow I) & \text{otherwise} \\ \aleph \uparrow I & == & \aleph \cap (I \times \Sigma) & \end{array}$$

where I is a set of real numbers. In the case that $I = \{t\}$ for some time t, we may omit the set brackets.

It proves convenient to define a *before* operator on traces and refusals:

$$\begin{array}{rcl} s \upharpoonright t & == & s \uparrow [0,t] \\ \aleph \upharpoonright t & == & \aleph \uparrow [0,t) \end{array}$$

The definition of $\upharpoonright$ on refusal sets differs from that on timed traces. For traces, $s \upharpoonright t$ includes events at t; in the case of refusals, such events are excluded. This choice of definitions is the most convenient for timed failures specifications.

We define an operator to strip the timing information from a timed trace, yielding the corresponding trace of untimed events:

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

Finally, we define an operator that shifts traces, refusals and failures through time:

$$\begin{array}{rcl} \langle\rangle + t & == & \langle\rangle \\ (\langle(t_1,a)\rangle^\frown s) + t & == & (\langle(t_1+t,a)\rangle^\frown(s+t)) \uparrow [0,\infty) \\ \aleph + t & == & \{(t_1+t,a) \mid (t_1,a) \in \aleph\} \uparrow [0,\infty) \\ (s,\aleph) + t & == & (s+t, \aleph+t) \end{array}$$

2.4 The Domain TM_F

We define the Timed Failures model TM_F to be those subsets S of TF which satisfy the following set of healthiness conditions:

1. $(\langle\rangle, \{\}) \in S$
2. $(s^\frown w, \aleph) \in S \Rightarrow (s, \aleph \upharpoonright begin(w)) \in S$
3. $\forall \aleph' : RSET \bullet (s,\aleph) \in S \wedge \aleph' \subseteq \aleph \Rightarrow (s,\aleph') \in S$
4. $\forall t : [0,\infty) \bullet \exists n(t) : \mathbb{N} \bullet (s,\aleph) \in S \wedge end(s) \leqslant t \Rightarrow \#(s) \leqslant n(t)$

5. $(s, \aleph) \in S \Rightarrow$
$\exists \aleph' : RSET \bullet \aleph \subseteq \aleph' \wedge (s, \aleph') \in S \wedge$
$\forall (t, a) \in T\Sigma \bullet (t, a) \notin \aleph' \Rightarrow (s \restriction t^\frown\langle (t, a)\rangle, \aleph' \restriction t) \in S$
$\wedge$
$((t > 0 \wedge \nexists \epsilon > 0 \bullet ((t - \epsilon, t) \times \{a\} \subseteq \aleph')) \Rightarrow$
$(s \restriction t^\frown\langle (t, a)\rangle, \aleph' \restriction t) \in S)$

2.5 The Complete Metric on TM_F

We define a distance metric d on TM_F by considering the earliest time after which the elements of two sets may be distinguished.

$$S \restriction t \quad == \quad \{(s, \aleph) \mid (s, \aleph) \in S \wedge end(s, \aleph) \leqslant t\}$$

If S is a element of TM_F then $S \restriction t$ is the set of elements of S which do not extend beyond time t. We may now define the complete metric:

$$d(S, T) \quad == \quad \inf(\{2^{-t} \mid S \restriction t = T \restriction t\} \cup \{1\})$$

2.6 The Semantic Function $\mathcal{F}_T$

We require the following additional operators on traces and refusals:

$$\begin{array}{rcll}
\sigma(s) & == & \{a : \Sigma \mid \exists t \bullet (t, a) \textbf{ in } s\} & \\
\sigma(\aleph) & == & \{a : \Sigma \mid \exists t \bullet (t, a) \in \aleph\} & \\
\aleph \uparrow t & == & \aleph \restriction [t, \infty) & \\
\aleph \downarrow A & == & \aleph \cap [0, \infty) \times A & \\
\langle\rangle \downarrow A & == & \langle\rangle & \\
(\langle (t, a)\rangle^\frown s) \downarrow A & == & \langle (t, a)\rangle^\frown (s \downarrow A) & \text{if } a \in A \\
 & & s \downarrow A & \text{otherwise}
\end{array}$$

The semantic function

$$\mathcal{F}_T : TCSP \rightarrow TM_F$$

is defined by the following set of equations:

$$\begin{array}{rcl}
\mathcal{F}_T[\![STOP]\!] & == & \{(\langle\rangle, \aleph) \mid \aleph \in RSET\} \\
\mathcal{F}_T[\![SKIP]\!] & == & \{(\langle\rangle, \aleph) \mid \checkmark \notin \sigma(\aleph)\} \\
 & & \cup \\
 & & \{(\langle (t, \checkmark)\rangle, \aleph) \mid t \geqslant 0 \wedge \checkmark \notin \sigma(\aleph \restriction [0, t))\} \\
\mathcal{F}_T[\![WAIT\ t_0]\!] & == & \{(\langle\rangle, \aleph) \mid \checkmark \notin \sigma(\aleph \uparrow t_0)\} \\
 & & \cup \\
 & & \{(\langle (t, \checkmark)\rangle, \aleph) \mid t \geqslant t_0 \wedge \checkmark \notin \sigma(\aleph \restriction [t_0, t))\}
\end{array}$$

$$\mathcal{F}_T[\![a \xrightarrow{t_0} P]\!] == \{(\langle\rangle, \aleph) \mid a \notin \sigma(\aleph)\}$$
$$\cup$$
$$\{(((t,a)\rangle^\frown s, \aleph) \mid t \geqslant 0 \wedge a \notin \sigma(\aleph \restriction t) \wedge begin(s) \geqslant t + t_0 \wedge (s, \aleph) - (t + t_0) \in \mathcal{F}_T[\![P]\!]\}$$

$$\mathcal{F}_T[\![P\,;Q]\!] == \{(s,\aleph) \mid \checkmark \notin \sigma(s) \wedge (s, \aleph \cup ([0, end(s,\aleph)) \times \{\checkmark\})) \in \mathcal{F}_T[\![P]\!]$$
$$\vee$$
$$s = s_P{}^\frown s_Q \wedge \checkmark \notin \sigma(s_P) \wedge (s_Q, \aleph) - t \in \mathcal{F}_T[\![Q]\!] \wedge (s_P{}^\frown\langle(t,\checkmark)\rangle, \aleph \restriction t \cup ([0,t) \times \{\checkmark\})) \in \mathcal{F}_T[\![P]\!]\}$$

$$\mathcal{F}_T[\![P \sqcap Q]\!] == \mathcal{F}_T[\![P]\!] \cup \mathcal{F}_T[\![Q]\!]$$

$$\mathcal{F}_T[\![P \,\Box\, Q]\!] == \{(\langle\rangle, \aleph) \mid (\langle\rangle, \aleph) \in \mathcal{F}_T[\![P]\!] \cap \mathcal{F}_T[\![Q]\!]\}$$
$$\cup$$
$$\{(s,\aleph) \mid s \neq \langle\rangle \wedge (s,\aleph) \in \mathcal{F}_T[\![P]\!] \cup \mathcal{F}_T[\![Q]\!] \wedge (\langle\rangle, \aleph \restriction begin(s)) \in \mathcal{F}_T[\![P]\!] \cap \mathcal{F}_T[\![Q]\!]\}$$

$$\mathcal{F}_T[\![f(P)]\!] == \{(f(s), \aleph) \mid (s, f^{-1}(\aleph)) \in \mathcal{F}_T[\![P]\!]\}$$

$$\mathcal{F}_T[\![P \setminus A]\!] == \{(s \setminus A, \aleph) \mid (s, \aleph \cup ([0, end(s,\aleph) \times A) \in \mathcal{F}_T[\![P]\!]\}$$

$$\mathcal{F}_T[\![P \;{}_A\|_B\; Q]\!] == \{(s, \aleph_P \cup \aleph_Q \cup \aleph_R) \mid \exists\, s_P, s_Q \bullet \sigma(\aleph_P) \subseteq A \wedge \sigma(\aleph_Q) \subseteq B \wedge \sigma(\aleph_R) \subseteq \Sigma - (A \cup B) \wedge s \in (s_P \;{}_A\|_B\; s_Q) \wedge (s_P, \aleph_P) \in \mathcal{F}_T[\![P]\!] \wedge (s_Q, \aleph_Q) \in \mathcal{F}_T[\![Q]\!]\}$$

$$\mathcal{F}_T[\![P \,|||\, Q]\!] == \{(s,\aleph) \mid \exists\, s_P, s_Q \bullet s \in s_P \,|||\, s_Q \wedge (s_P, \aleph) \in \mathcal{F}_T[\![P]\!] \wedge (s_Q, \aleph) \in \mathcal{F}_T[\![Q]\!]\}$$

$$\mathcal{F}_T[\![P \overset{t_0}{\rhd} Q]\!] == \{(s,\aleph) \mid begin(s) \leqslant t_0 \wedge (s,\aleph) \in \mathcal{F}_T[\![P]\!]\}$$
$$\cup$$
$$\{(s,\aleph) \mid begin(s) \geqslant t_0 \wedge (\langle\rangle, \aleph \restriction t_0) \in \mathcal{F}_T[\![P]\!] \wedge (s,\aleph) - t_0 \in \mathcal{F}_T[\![Q]\!]\}$$

$$\mathcal{F}_T[\![\mu\, X \circ F(X)]\!] == \text{the unique fixed point of the mapping corresponding to } F \text{ on the semantic domain, if this mapping is a contraction mapping under metric } d$$

where the auxiliary functions on timed traces are defined as follows:

$$s_P \;{}_A\|_B\; s_Q == \{s \in T\Sigma^*_{\leqslant} \mid s \lfloor A = s_P \wedge s \lfloor B = s_Q \wedge s \lfloor (A \cup B) = s\}$$

$$s_P \,|||\, s_Q \;==\; \{s : T\Sigma^*_{\leqslant} \mid \forall t : TIME \bullet \forall a : \Sigma \bullet \\ s \uparrow t \downarrow \{a\} = s_P \uparrow t \downarrow \{a\}^\frown s_Q \uparrow t \downarrow \{a\}\}$$

2.7 Remarks on Differences with Previous Versions

The Timed Failures Model presented above differs from that originally given in [ReR87, Ree88] in several respects. In particular, there is no constant delay δ required for prefixing or the unwinding of recursions. All time delays are now explicit, and include 0. Recursion is only valid for those recursive functions representing contraction mappings on the complete metric space domain. Note that in [Dav91], Davies shows that valid recursions in the current model can be checked by syntactic analysis alone.

We believe that the current model is more applicable in its greater abstraction. Of course, the original model is now simply one of many possible interpretations.

3 Specification and Verification

A specification of a system is a formal description of its intended behaviour. We say that a program P meets a specification S in a denotational model M if every possible behaviour of P in that model meets the corresponding specification condition. In this case we say that P satisfies S, written P **sat** S. The task of verification is to establish that the specification S is indeed satisfied by process P.

We will describe two styles of specification. The first is mathematical, allowing any predicate S on behaviours as a specification. This approach has the advantage of expressivity, but its generality makes fully formal verification difficult. The second gives a grammar for a temporal logic specification language. This has the advantage of allowing formal verification. It is not so expressive, but nonetheless contains a large class of useful specifications.

Each of these styles of specification has an associated proof system, which reduces the proof that P satisfies S to a number of smaller derivations on the (syntactically simpler) subcomponents of P. Ultimately, the verification task is reduced to tautology checking of statements written in the specification language.

A further method of verification is that of timewise refinement. This approach uses the links between various timed and untimed models for CSP in order to verify specifications in the simpler models, and then translate the result to the more complex model. This may be done in cases where the specification in question is not time-critical, so the additional timing information available in the more complex model is unnecessary for the verification. The higher level of abstraction in the simpler model generally makes reasoning easier.

3.1 Mathematical specification

The satisfaction relation is defined between processes and specifications in the failures model as follows:

$$P \text{ sat } S \;==\; \forall (s, \aleph) \in \mathcal{F}_T[\![P]\!] \bullet S$$

We say that a specification S is satisfiable if there is a process that satisfies it. We identify necessary conditions for a specification to be satisfiable (see [Dav91]); specifications should be checked against these conditions before searching for an implementation:

Lemma 3.1 If S is satisfiable, then $S[(\langle\rangle, \{\})/s, \aleph]$. ♡

Since all processes may exhibit the minimal behaviour, any implementation P may do so; since S must hold for all possible behaviours of P, it must in particular hold for $(\langle\rangle, \{\})$. This condition rules out any requirement that insists that a certain timed event appears in the trace or refusal, without a qualifying assumption.

A more surprising result, which has no analogue in the untimed models, is that a specification may not insist that a certain timed event is absent from the refusal set.

Lemma 3.2 If S is a behavioural specification such that

$$\exists e : T\Sigma \bullet S \Rightarrow e \notin \aleph$$

then S is not satisfiable ♡

This result holds because any process may engage in only finitely many copies of any timed event e. Since its environment may always offer more copies of an event than the process is able to perform, the process must refuse another copy of event e if it has already performed as many as possible at the current time. Hence there is always some trace during which the event e may be refused.

As an example of a specification, consider a timed buffer. A t-buffer is a process which, when empty is always prepared to input a message by t, and which when non-empty is always prepared to output the next message within that time. It may be specified as follows:

$$\begin{aligned} BUFF_t \quad == \quad & tstrip(s \downharpoonleft OUT) \leqslant tstrip(s \downharpoonleft IN) \\ & \wedge \\ & tstrip(s \downharpoonleft OUT) = tstrip(s \downharpoonleft IN) \Rightarrow IN \cap \sigma(\aleph \upharpoonright end(s) + t) = \{\} \\ & \wedge \\ & tstrip(s \downharpoonleft OUT) < tstrip(s \downharpoonleft IN) \Rightarrow OUT \not\subseteq \sigma(\aleph \upharpoonright end(s) + t) = \{\} \end{aligned}$$

where M is the set of messages that may be transmitted through the buffer, and

$$\begin{aligned} IN \quad == \quad & \{in.m \mid m \in M\} \\ OUT \quad == \quad & \{out.m \mid m \in M\} \end{aligned}$$

define the event sets corresponding to input messages and output messages, respectively.

A process is t-deadlock-free if it must perform, or at least offer some event over an interval of length t. No execution may exhibit the refusal of the entire set of events over such an interval after the last visible action.

$$DF_t \quad == \quad \forall T \bullet T \geqslant end(s) \Rightarrow [T, T + t] \times \Sigma \not\subseteq \aleph$$

Observe that the use of the closed interval $[T, T+t]$ yields the specification that Σ cannot be continuously refused (after the end of the trace) for any length of time strictly greater than t. In order to disallow intervals of length exactly t as well, the half open interval $[T, T + t)$ should be used instead.

Proof System

The semantic function $\mathcal{F}_T$ is directly compositional, in the sense that every behaviour $(s, \aleph)$ of a composite program arises from a combination of a single behaviour from each component program (possibly in more than one way). It is this feature of the semantic function that makes it possible to define a sound and complete proof rule for each TCSP operator. Each rule is complete in the sense that if a specification holds of a composite process built using the corresponding operator, then the rule may be used to deduce this from specifications of the component processes. Since there is one rule for each operator in the syntax, the proof system is complete for all TCSP processes. For example,

STOP

$$\frac{}{STOP \text{ sat } s = \langle\rangle}$$

Prefix

$$\frac{P \text{ sat } S}{a \xrightarrow{t} P \text{ sat } \begin{array}{l} s = \langle\rangle \wedge a \notin \sigma(\aleph) \\ \vee \\ s = \langle (t', a) \rangle^\frown s' \wedge a \notin \sigma(\aleph \restriction t') \wedge \mathit{begin}(s') \geqslant (t' + t) \\ \qquad\qquad \wedge S[(s', \aleph) - (t' + t)/(s, \aleph)] \end{array}}$$

Nondeterministic Choice

$$\frac{\begin{array}{l} P \text{ sat } S \\ Q \text{ sat } T \end{array}}{P \sqcap Q \text{ sat } S \vee T}$$

Parallel

$$\frac{\begin{array}{l} P \text{ sat } S \\ Q \text{ sat } T \end{array}}{P \parallel Q \text{ sat } \exists \aleph_P, \aleph_Q \bullet \aleph = \aleph_P \cup \aleph_Q \wedge S[\aleph_P/\aleph] \wedge T[\aleph_Q/\aleph]}$$

Recursion

$$\frac{X \text{ sat } S \Rightarrow F(X) \text{ sat } S}{\mu X \circ F(X) \text{ sat } S} \quad [\, F \text{ contracting}, S \text{ satisfiable} \,]$$

The above recursion rule requires first a proof of satisfaction, which can sometimes be nontrivial. In [DaS89], it was shown that this requirement can be relaxed if the natural extension of F to the complete metric space of all subsets of TF preserves satisfaction.

As an illustration of the application of these rules, we prove that the process

$$\mu X \circ in \xrightarrow{1} out \xrightarrow{1} X$$

is 1-deadlock-free. To do this, we must apply the extended recursion rule. To establish the antecedent to this rule, we use the proof system to prove that the body of the recursion maintains a specification that is at least as strong as the one we wish to establish. In fact we must strengthen the specification, to include the fact that in the initial state, some event is immediately and persistently on offer; thus some event does not appear in the refusal set at all before the first event occurs.

We begin by assuming that X meets the stronger specification:

$$\begin{array}{rrll}
 & X & \textbf{sat} & DF_1 \wedge (s = \langle\rangle \Rightarrow \sigma(\aleph) \neq \Sigma) \\
\Rightarrow & out \xrightarrow{1} X & \textbf{sat} & s = \langle\rangle \wedge out \notin \sigma(\aleph) \\
 & & & \vee \\
 & & & s = \langle (t, out)\rangle^\frown s' \wedge out \notin \sigma(\aleph \restriction t) \wedge begin(s') \geqslant (t+1) \\
 & & & \qquad \wedge DF_1[(s', \aleph) - (t+1)/(s, \aleph)] \\
 & & & \qquad \wedge s' - (t+1) = \langle\rangle \Rightarrow \sigma(\aleph - (t+1)) \neq \Sigma \\
 & & & \Rightarrow DF_1 \wedge s = \langle\rangle \Rightarrow \sigma(\aleph) \neq \Sigma \\
\Rightarrow & in \xrightarrow{1} out \xrightarrow{1} X & \textbf{sat} & s = \langle\rangle \wedge in \notin \sigma(\aleph) \\
 & & & \vee \\
 & & & s = \langle (t, in)\rangle^\frown s' \wedge in \notin \sigma(\aleph \restriction t) \wedge begin(s') \geqslant (t+1) \\
 & & & \qquad \wedge DF_1[(s', \aleph) - (t+1)/(s, \aleph)] \\
 & & & \qquad \wedge s' - (t+1) = \langle\rangle \Rightarrow \sigma(\aleph - (t+1)) \neq \Sigma \\
 & & & \Rightarrow DF_1 \wedge s = \langle\rangle \Rightarrow \sigma(\aleph) \neq \Sigma
\end{array}$$

Hence the antecedent to the recursion rule holds, so we obtain that

$$\mu X \circ in \xrightarrow{1} out \xrightarrow{1} X \textbf{ sat } DF_1 \wedge s = \langle\rangle \Rightarrow \sigma(\aleph) \neq \Sigma$$

We may weaken the specification to obtain the desired result, that

$$\mu X \circ in \xrightarrow{1} out \xrightarrow{1} X \textbf{ sat } DF_1$$

3.2 Temporal Logic Specifications

The use of a tightly defined logical language for describing specifications has the advantage that verifications may more easily be expressed in terms of a reduced set of manipulations, yielding them more amenable to mechanisation. The restricted syntax also means that common properties of specifications such as safety and liveness may be easily characterised by the form they take.

We use the following language as our temporal logic specification language.

$$p \ ::= \ true \mid A \mid \neg P \mid p \wedge p \mid p \,\mathcal{U}_R\, q$$

The atoms A contain propositions of the form $\mathbb{P}_X$ (some event from the set X is performed) and $\mathbb{O}_X$ (the set X is offered) for each set $X \subseteq \Sigma$. Such atoms may also be labelled to

yield other atomic propositions, such as $\mathbb{P}'_X$ and $\mathbb{O}'_X$. In the above syntax, predicate R may take any of the following forms

$$\_ \leqslant T \ , \ \_ < T \ , \ \_ = T \ , \ \_ \geqslant T \ , \ \_ > T$$

where $T \in \mathbb{R}^+$.

Specifications written in this language are concerned with complete executions of the system, rather than the finite time behaviours used by the denotational models. We will assume the infinite behaviours of the system to be the limits of the finite ones; this assumption is unjustified for processes which have infinite branching nondeterminism (see [Sch91]); such a process may have fewer infinite executions. In the proof system described below, completeness is lost for such processes, though the system remains sound.

We define the infinite failures $\mathcal{I}_T[\![Q]\!]$ of a program Q in terms of its finite failures:

Definition 3.3 The set of infinite failures $\mathcal{I}_T[\![Q]\!]$ of a program Q is given by

$$\mathcal{I}_T[\![Q]\!] \ = \ \{(s, \aleph) \mid s \in T\Sigma^*_{\leqslant} \wedge \aleph \subseteq T\Sigma \wedge \forall t < \infty \bullet (s, \aleph) \restriction t \in \mathcal{F}_T[\![Q]\!]\}$$

$\diamond$

Infinite failures of the form $(s, \aleph)$ model propositions at times t according to the following definitions:

$$\begin{array}{rcl}
(s,\aleph), t \models \mathbb{P}_X & \Leftrightarrow & (s \uparrow t) \downharpoonright X \neq \langle\rangle \\
(s,\aleph), t \models \mathbb{O}_X & \Leftrightarrow & X \cap \sigma(\aleph \uparrow t) = \{\} \\
(s,\aleph), t \models \neg P & \Leftrightarrow & (s,\aleph), t \not\models P \\
(s,\aleph), t \models P \wedge Q & \Leftrightarrow & (s,\aleph), t \models P \text{ and } (s,\aleph), t \models Q \\
(s,\aleph), t \models P \mathcal{U}_R \, Q & == & \exists t_1 \bullet R(t_1 - t) \text{ and } \forall t_2 \bullet t < t_2 < t_1 \Rightarrow (s,\aleph), t_2 \models P \\
 & & \qquad \text{and } (s,\aleph), t_1 \models P
\end{array}$$

A number of useful abbreviations and derived specification constructs may be defined in terms of these:

$$\begin{array}{rcll}
P \mathcal{U} \, Q & == & P \mathcal{U}_{<\infty} \, Q & \text{until} \\
P \mathcal{W} \, Q & == & (P \mathcal{U} \, Q) \vee \mathbf{G}(P) & \text{unless} \\
P \overline{\mathcal{U}} \, Q & == & (P \wedge P \mathcal{U} \, Q) \vee Q & \text{reflexive until} \\
P \overline{\mathcal{W}} \, Q & == & (P \overline{\mathcal{U}} \, Q) \vee \mathbf{G}(P) & \text{reflexive unless} \\
\overline{\mathbf{F}} \, P & == & T \overline{\mathcal{U}} \, P & \text{reflexive eventually} \\
\overline{\mathbf{G}} \, P & == & \neg \overline{\mathbf{F}} \neg P & \text{reflexive always}
\end{array}$$

We then say that P **sat** S if all of P's behaviours at time 0 model the proposition S:

$$P \text{ sat } S \ == \ \forall (s, \aleph) \in \mathcal{I}_T[\![P]\!] \bullet (s, \aleph), 0 \models S$$

As an example of a TL specification, consider the requirement that the system should alternate on its performance of *in* and *out* events, starting with *in*. This may be captured in the specification language as follows:

$$\begin{array}{rcll}
\Phi & = & \overline{\mathbf{G}}(\mathbb{P}_{in} \Rightarrow (\neg \mathbb{P}_{in}) \mathcal{W} \, \mathbb{P}_{out}) & (1) \\
 & & \wedge \ \overline{\mathbf{G}}(\mathbb{P}_{out} \Rightarrow (\neg \mathbb{P}_{out}) \mathcal{W} \, \mathbb{P}_{in}) & (2) \\
 & & \wedge \ (\neg \mathbb{P}_{out}) \overline{\mathcal{W}} \, \mathbb{P}_{in} & (3)
\end{array}$$

This states (1) that whenever an *in* event is performed, then no further *in* event is performed unless an *out* event occurs; (2) that whenever an *out* event occurs, then no further *out* may be performed unless an *in* event occurs; (3) that *out* may not be performed unless *in* is performed. Since a process satisfies this specification if its behaviours model it at time 0, condition (3) is the requirement that the first *out* event may not occur before the first *in* event.

Temporal Logic Proof System

The temporal logic proof system consists of a number of rules, one for each TCSP operator. The rules are relatively complete, in the sense that if a TL specification is satisfied by a program, then the rules may be used to reduce the verification to checking a temporal logic tautology. We present few rules here for illustrative purposes; the complete set is presented in [Jac91].

STOP

$$\frac{}{STOP \textbf{ sat } \forall X \subseteq \Sigma \bullet \overline{\mathbf{G}}(\neg\, \mathbb{P}_X)}$$

Prefixing

$$\frac{\left.\begin{array}{l} P \textbf{ sat } p \\ (\mathbb{O}_a \wedge \neg\, \mathbb{P}_\Sigma) \\ \overline{\mathcal{W}} \\ (\mathbb{P}_a \wedge \neg\, \mathbb{P}_{\Sigma - \{a\}} \wedge (\neg\, \mathbb{P}_\Sigma \mathcal{U}_{=t}\, p)) \end{array}\right\} \Rightarrow r}{a \xrightarrow{t} P \textbf{ sat } r}$$

Nondeterministic Choice

$$\frac{\begin{array}{l} P \textbf{ sat } p \\ Q \textbf{ sat } q \\ (p \vee q) \Rightarrow r \end{array}}{P \sqcap Q \textbf{ sat } r}$$

Parallel Composition

$$\frac{\left.\begin{array}{l} P \textbf{ sat } p \\ Q \textbf{ sat } q \\ p[\mathbb{O}'/\mathbb{O}_\Sigma] \wedge \\ q[\mathbb{O}''/\mathbb{O}_\Sigma] \wedge \\ \overline{\mathbf{G}}(\mathbb{O} \Leftrightarrow_\Sigma \mathbb{O}' \wedge \mathbb{O}'') \end{array}\right\} \Rightarrow r}{P \parallel Q \textbf{ sat } r}$$

Recursion

$$\frac{X \textbf{ sat } r \Rightarrow F(X) \textbf{ sat } r}{\mu X \circ F(X) \textbf{ sat } r} \quad [\, r \text{ satisfiable, admissible, } F \text{ contracting} \,]$$

A satisfiable predicate r is one for which $\exists P \in TM_F \bullet P \textbf{ sat } r$. An admissible predicate is one which holds for an infinite behaviour $(s, \aleph)$ whenever it holds for all the finite approximations $(s \restriction t, \aleph \restriction t)$. This side condition is needed in the temporal logic proof system because of the use of infinite behaviours in specification. A behaviour upsetting a specification may be absent from every element of a convergent sequence, but present in the limit.

For example, the inadmissible specification $\mathbf{F}\,\mathbb{O}_\Sigma$ is preserved by the function corresponding to $WAIT\ 1\,;X$, but the recursive process $\mu X \circ WAIT\ 1\,;X$ does not satisfy it. Specifications that are built using bounded until operators as the only temporal connective will always be admissible. The issue of admissibility does not arise when only finite behaviours are considered, as in the proof system presented earlier; hence unbounded eventuality cannot even be expressed as a finite behavioural specification.

To illustrate the rules presented above, we perform a simple verification for the specification Φ given above; we prove that $\mu X \circ in \xrightarrow{1} out \xrightarrow{1} X \textbf{ sat } \Phi$. Observe that the specification Φ is admissible: if an infinite behaviour does not model it, then neither does some finite prefix.

$$\begin{array}{lll}
 & X \textbf{ sat } \Phi & \\
\Rightarrow & out \xrightarrow{1} X \textbf{ sat } (\neg \mathbb{P}_{out})\,\overline{\mathcal{W}}\,((\mathbb{P}_{out} \wedge \neg \mathbb{P}_\Sigma)\,\mathcal{U}_{t=1}\,\Phi) & \\
 & \quad \Rightarrow \begin{array}{l} (\neg \mathbb{P}_{in})\,\overline{\mathcal{W}}\,\mathbb{P}_{out} \\ \wedge\ \overline{\mathbf{G}}\,(\mathbb{P}_{in} \Rightarrow (\neg \mathbb{P}_{in}\,\mathcal{W}\,\mathbb{P}_{out})) \\ \wedge\ \overline{\mathbf{G}}\,(\mathbb{P}_{out} \Rightarrow (\neg \mathbb{P}_{out}\,\mathcal{W}\,\mathbb{P}_{in})) \end{array} \Big\} = \Phi_1 & \\
\Rightarrow & in \xrightarrow{1} out \xrightarrow{1} X \textbf{ sat } (\neg \mathbb{P}_{in})\,\overline{\mathcal{W}}\,((\mathbb{P}_{in} \wedge \neg \mathbb{P}_\Sigma)\,\mathcal{U}_{t=1}\,\Phi_1) & \\
 & \quad \Rightarrow \begin{array}{l} (\neg \mathbb{P}_{out})\,\overline{\mathcal{W}}\,\mathbb{P}_{in} \\ \wedge\ \overline{\mathbf{G}}\,(\mathbb{P}_{out} \Rightarrow (\neg \mathbb{P}_{out}\,\mathcal{W}\,\mathbb{P}_{in})) \\ \wedge\ \overline{\mathbf{G}}\,(\mathbb{P}_{in} \Rightarrow (\neg \mathbb{P}_{in}\,\mathcal{W}\,\mathbb{P}_{out})) \end{array} \Big\} = \Phi &
\end{array}$$

Hence the antecedent to the recursion rule holds, so we obtain that

$$\mu X \circ in \xrightarrow{1} out \xrightarrow{1} X \quad \textbf{sat} \quad \Phi$$

3.3 Timewise Refinement

System requirements may often be decomposed into functional and timing aspects. Correct functional behaviour of a composite program may depend upon the logical interaction of its constituents and yet be independent of timing considerations. In such cases, the more abstract nature of the untimed models makes verification of such a program easier than in the timed models, where the increased descriptive power generated by the additional complexity of timed observations has become redundant. Yet the need to consider timing behaviour forces us to complete the analysis of the program in a timed model.

Furthermore, if correctness of timing behaviour rests on functional correctness, then the latter must be established in a timed framework.

Timewise refinement permits correctness-preserving migration between models in the hierarchy. A process in an untimed model is a description of purely functional behaviour; in a timed context, this may be considered as a characterisation of the functional aspects of a process behaviour, with the timing aspects completely unspecified. A correct timed implementation of such a process must be functionally correct with respect to this characterisation, and will in addition provide details of timing behaviour. Such a timed process is said to be a timed refinement of the untimed process.

An ability to link processes throughout the hierarchy permits the use of different models to establish different aspects of a system's correctness. Requirements may be decomposed, and each resulting part may be verified in a different model; timewise refinement allows these separate verifications to be used as a complete verification of the whole system.

The refinements presented here are defined with respect to the timed failures semantics of timed processes. These definitions extend immediately to the timed failures-stability model, since the failures components of the semantics of any TCSP program in the larger model are precisely its semantics in the timed failures model. Hence the results presented also hold for the timed-failures stability model.

Traces refinement

The Traces model M_T for CSP is concerned purely with the order in which events may be performed. Within the context of the hierarchy, it means that a description of a process is concerned purely with permissible sequences of events. This model has abstracted away from all considerations of refusal information, divergence, and timing behaviour. A process in this model may therefore be considered as a specification describing which sequences of events are acceptable; it will be refined by any process in a higher model whose sequences of events do not violate this specification.

We will consider the refinement relation between programs modelled in M_T, and timed programs modelled in the Timed Failures model TM_F. We write $P \sqsubseteq_T Q$ to mean that Q is a (trace) timewise refinement of P. This states that any sequence of events performed by Q must be a possible trace of P. It is defined as follows:

Definition 3.4 The relation $\sqsubseteq_T$ is defined as follows:

$$P \sqsubseteq_T Q \;=\; \forall s \in T\Sigma^*_{\leqslant} \bullet s \in \mathit{traces}(\mathcal{F}_T[\![Q]\!]) \Rightarrow \mathit{tstrip}(s) \in \mathcal{T}[\![P]\!]$$

$\Diamond$

It follows from this definition that if an untimed process meets a specification (with free variable tr ranging over behaviours in the traces model), then a timed process meets a corresponding specification. We obtain the following proof rule:

$$\frac{P \sqsubseteq_T Q \qquad P \textbf{ sat } S}{Q \textbf{ sat } S[\mathit{tstrip}(s)/tr]}$$

The rule is complete: if S is a predicate on tr such that the conclusion holds, then there is a program P which satisfies S and which is refined by Q. If we have a specification on a timed program which is concerned purely with the order in which events can occur, then we may establish correctness by using the Traces model, and translating the correctness result into the timed model.

Refinement of programs

As we would hope, the addition of timing information to a program does not introduce unacceptable functional behaviour, though some acceptable behaviours may no longer be possible because of the additional timing constraints. We may transform an untimed program into a timed one by the addition of times to the event prefix operations, and by the insertion of delays at points throughout the program. Choices may be transformed into timeouts.

We define $\Theta : TCSP \to CSP$ to be the function that removes timing information, so

$$\begin{aligned} \Theta(a \xrightarrow{t} P) &= a \to \Theta(P) \\ \Theta(WAIT\ t) &= SKIP \\ \Theta(P \overset{t}{\rhd} Q) &= \Theta(Q) \sqcap (\Theta(P) \,\Box\, \Theta(Q)) \end{aligned}$$

and Θ distributes over the operators of TCSP that also appear in CSP. Then we obtain the welcome result that $\Theta(P) \sqsubseteq_T P$ for all TCSP programs P. Furthermore, the combination of two untimed programs by an operator is refined by the combination of any refinements by the corresponding timed operator; hence each operator preserves refinement.

For example, it is easy to see that

$$\mu X \circ in \to out \to X \quad \textbf{sat} \quad \#(tr \upharpoonright out) \leqslant \#(tr \upharpoonright in)$$

in the traces model M_T. Hence we may deduce as an immediate consequence that

$$\mu X \circ in \xrightarrow{4} out \xrightarrow{7} X \quad \textbf{sat} \quad \#(tstrip(s) \upharpoonright out) \leqslant \#(tstrip(s) \upharpoonright in$$

in the timed failures model TM_F.

Failures refinement

The untimed information that the behaviour (tr, X) is acceptable corresponds in the timed world to the information that a timed version of tr followed by the eventual continuous refusal of the set X is permitted. We need to consider the infinite failures of a process in order to define the $\sqsubseteq_F$ refinement relation, which is concerned with relating untimed and timed processes with respect to the failures of the untimed process.

The relation $\sqsubseteq_F$ is defined by requiring that whenever the timed process may perform a timed trace eventually followed by a continuous refusal of a set X, the untimed process must allow the same sequence of events, followed by the refusal of set X.

Definition 3.5

$$\begin{aligned} P \sqsubseteq_F Q \;=\; \forall (s, \aleph) \in \mathcal{I}_T[\![Q]\!], t \geqslant 0, X \subseteq \Sigma \bullet \\ [t, \infty) \times X \subseteq \aleph \Rightarrow (tstrip(s), X) \in \mathcal{F}[\![P]\!] \end{aligned}$$

$\Diamond$

Observe that any possible trace s of Q must have a corresponding trace of P, since such a trace will always accompany the empty refusal set; thus $(tstrip(s), \{\})$ should be a possible behaviour of P.

The definition of this refinement relation gives rise to the following proof rule:

$$\frac{P \sqsubseteq_F Q \quad P \textbf{ sat } S}{Q \textbf{ sat } \forall t, Y \bullet [t, \infty) \times Y \subseteq \aleph \Rightarrow S[(tstrip(s), Y)/(tr, X)]}$$

This rule may be used for example to establish that a timed process is deadlock-free by showing that it refines an untimed deadlock-free process. The untimed specification for deadlock-freedom is $X \neq \Sigma$, that the process may never refuse everything. Hence a refinement of a process satisfying this will have that whenever it refuses $[t, \infty) \times Y$, it cannot be that $Y = \Sigma$; this is the timed characterisation of deadlock-freedom. This will often be useful, since there already exist many techniques for establishing deadlock-freedom in untimed processes. In a similar way, the rule may be used to establish that a refinement of an untimed buffer must be a timed buffer.

We again obtain the result that $\Theta(P) \sqsubseteq_F P$ for all *TCSP* programs P; the introduction of timing constraints into a program maintains correctness with respect to failures specifications. It also follows that timewise refinement is preserved by all of the operators, as long as stability (in TM_{FS}) of processes is maintained.

However, the parallel operator need not preserve refinement for unstable processes. Consider a process that offers the event a during alternate seconds, beginning at time *0*, and another which offers a during alternate seconds, beginning at time *1*. They both refine an untimed process which deterministically offers a, since neither timed process is able to refuse it continuously. The parallel combination of the untimed processes cannot refuse the event a, but the combination of the timed processes that refine them is able to refuse a continuously, and therefore does not refine the untimed parallel combination.

4 Case Studies

This section describes a number of case studies to which Timed CSP has been applied since 1989. They include examples from two main fields: communications, where TCSP has been applied to the real time analysis of protocols and switching systems, and embedded systems such as real-time controllers.

Many of these studies were carried out in collaboration with other groups or companies with experience in the application area. The studies thus served two purposes: they allowed the application community to see how formal techniques in general, and Timed CSP in particular, could be applied to their area of interest, and they allowed the Timed CSP research group to gain knowledge of user requirements, providing a focus for further research and development.

The basic approach adopted in each of these studies was to examine the natural way in which the application is discussed and decomposed by workers within the formalism. Our intention was not to force applications to use a 'Timed CSP development method', but

rather to use Timed CSP as a support to the methods which seem natural for a particular application.

One consequence of attempting to relate the formal treatment of each application to the particular structure of that application is that it allows customised formal structures to be built to match the needs of a particular area. These range from simply defining appropriate notations as macros—as for the specifications of the aircraft system, which used predicates defining properties on traces, such as 'alternates' and a 'disables/enables' relation—to the development of application specific proof rules. Some of these may be restricted and specific, such as the rule for adding additional components to a flexible manufacturing system, while others are less so. Examples of general rules which have resulted from these case studies are: the rules for composing constraint-based systems used by the aircraft engine, the rules for establishing correctness of layered network protocols [Dav91], and the rules for a 'rely and guarantee method' applied to a telephone switching network.

In the following sections we describe six case studies:

- a telephone switch—a 'rely and guarantee' method is used for a specification, together with design and proofs of correctness;
- an aircraft engine starter system—timed and untimed models are constructed, and projection of processes between models is used to construct proofs of correctness;
- autonomous guided vehicles—timed failure specifications are given together with a design expressed in the process algebra and outline correctness proofs are discussed;
- an ethernet-like protocol—a Timed CSP analysis of two layers of a local area network protocol, in which each layer provides services to the layer above;
- a laboratory robot—a hierarchical layered approach is demonstrated in a control application;
- a flexible manufacturing system—timed and untimed descriptions of a simple system are analysed using the temporal logic notation.

4.1 A Rely and Guarantee Method Applied to a Telephony Example

In [KR91a, KR91b, KaR90], Kay and Reed present a timed failures specification and a distributed design, also described using the timed failures model for Timed CSP, together with proofs of correctness for a realistically-sized telephone exchange. They use a *rely and guarantee* method for Timed CSP, which can be a key factor in keeping the formalisms to a manageable, meaningful collection. The style of this method is to describe what properties a component guarantees to its environment, *provided* that its environment supplies certain other properties to it. The method has some features in common with earlier work on Timed CSP environmental conditions [Dav91, DJS90] and the rely and guarantee version of VDM [Jon83, Jon91] for shared-variable concurrency.

Standard mechanisms for reasoning about a composition of components P_i are inference rules characterised by the following form, whereby if each P_i guarantees to meet its specification $Guarantee_i$ then one can infer the conjunction of all the $Guarantee_i$:

$$\frac{P_i \textbf{ sat } Guarantee_i}{\|_i P_i \textbf{ sat } \bigwedge_i Guarantee_i}$$

The implementor is obliged to ensure that each P_i unconditionally satisfies $Guarantee_i$ in order for the above rule to be applicable.

Another approach is perhaps more appropriate for highly interdependent systems - that is, for systems for which the desired behaviour of any single component is very much dependent on the desired behaviour of its co-components. A component must guarantee that it will behave in a certain way in order to satisfy its specification. However, the implementor can rely on certain facts about the environment (of cooperating components) in which the component is to be used. Inference rules characterised by the following form are useful for reasoning about such specifications:

$$\frac{P_i \textbf{ sat } Rely_i \Rightarrow Guarantee_i}{\|_i P_i \textbf{ sat } \bigwedge_i Guarantee_i} \quad \left[\begin{array}{l}\text{conditions ensuring } Guarantees \\ \text{establish } Relys\end{array}\right]$$

For example, a proof rule for safety properties has the following form (for simplicity we assume only two components):

$$\frac{\begin{array}{l}P \textbf{ sat } S_q \Rightarrow S_p \\ Q \textbf{ sat } S_p \Rightarrow S_q\end{array}}{P \parallel Q \textbf{ sat } S_p \wedge S_q} \quad \left[\begin{array}{l} S_p(\langle\rangle) \wedge S_q(\langle\rangle) \\ \wedge \ \textbf{prevents}(S_q) \cap \textbf{prevents}(S_p) = \{\}\end{array}\right]$$

where **prevents** of a safety predicate φ is defined to be those events which can cause φ to become false, that is, thoses events which φ (viewed as a constraint on traces) prevents.

$$a \in \textbf{prevents}(\varphi) \Leftrightarrow \exists\, s : TRACES, t \geqslant end(s) \bullet \varphi(s) \wedge \neg\, \varphi(s^\frown \text{seq}\,(t, a))$$

We can use the above rule when we want to design a system to meet safety properties S_p and S_q but we do not want (for practicality) to implement S_p and S_q unconditionally for P and Q respectively. Rather the implementor of P can assume S_q, and the implementor of Q can assume S_p, that is implement the pair of weaker constraints $S_p \Rightarrow S_q$ and $S_q \Rightarrow S_p$ with processes P and Q. The apparent circularity (that both S_p and S_q be false) is prevented by the side condition which intuitively ensures that P and Q are initially correct and cannot go wrong simultaneously.

The mode of use of this rule is adopt conventions on input and output whereby the side conditions ensuring that the *guarantees* establish the *relys* are automatically satisfied. For example, if we never place safety conditions on inputs (rather, only on outputs) then it can be shown that the above "**prevents**" side condition is always satisfied for any S_p and S_q and need not be explicitly checked. The additional side condition requiring the predicates to be initially true is usually immediately obvious.

To illustrate, let us assume that a master module sends a client module *request.id* and *abortrequest.id*, and the client responds within time t with an *answer.id* and an

abortdone.id, respectively. The implementor of the client does not want to guarantee that the client will respond correctly to all *abortrequests*, rather only to those which follow the corresponding *request.id*.

The predicate S_M (for suitably defined **onlyif** to stop errant *abortrequests*):

$$S_M == \textit{abortrequest.id} \textbf{ onlyif } \textit{request.id}$$

is relied upon by the implementor of the client C in order to guarantee the predicate S_C (for suitably defined $\textbf{stops}_t$ to stop answers going to aborted requests):

$$S_C == \textit{abortreq.id} \textbf{ stops}_t \textit{ answer.id}$$

The implementor of the client need only guarantee S_C whilst relying on S_M, and the implementor of the master need only guarantee S_M whilst relying on S_C in order for the parallel system $M \parallel C$ to satisfy $S_M \wedge S_C$. The side conditions of the above inference rule need not be explicitly checked, since each component only makes (safety) guarantees about its own outputs—not inputs. Thus we are able to derive $S_M \wedge S_C$ from $S_M \Rightarrow S_C$ and $S_C \Rightarrow S_M$.

There are analogous rules treating liveness which are inevitably more complex. However, for point-to-point communication systems these rules are applicable whenever components guarantee to poll their inputs in a timely fashion. The proofs of these safety and liveness rules are given in [Kay91].

These rules are the basis for a method which enables one to use properties of a component's environment whilst reasoning about the component in isolation. For the top-level specification of the telephone exchange the concern is with just one component (the system), together with its environment. For the distributed design, there are five interacting components, together with external environment.

The specification for each of these components is given in two parts. Firstly, each component is described in terms of the messages it inputs (from one component) and outputs (to another). These constraints are guarantees which are not specifically relied upon by other components. Secondly, each interface between two components is described in terms of the messages shared by the two components. Each constraint here corresponds to (i) a condition guaranteed by one component, and (ii) the same condition relied upon by the other component. (The design of the telephone exchange has been selectively proved correct with respect to the top-level specification, and implementation of the design into executable code is planned.)

The scope of the specification and design for the telephone exchange encompasses substantial functionality including feature interactions such as caller replacing just as callee receives the ring, and other such race conditions. The method has also been effectively applied to another telephony example characterised by centralised controllers [Su91]. Both case studies indicate that this rely and guarantee approach is very straightforward and intuitive.

There are two major reasons that this method is effective for these applications: (i) the formalisms are manageable, and (ii) the verifier can use strong predicates to prove correctness whilst the implementor can use weaker predicates to produce executable code. In an interface description an individual predicate is only given once, implicitly representing two

constraints - one a guarantee and the other a rely. Thus the occurrences of required formal expressions are significantly reduced. Perhaps the most valuable asset of the method is that it bridges a gap which occurs at a design level. Here there is a conflict between strong specifications which facilitate proofs of correctness that the design meets its higher-level specification, and weak specifications which allow efficient and practical implementations but make proofs more difficult. For the design of the telephone exchange the guarantees combine together to imply the top-level specification for the exchange, whilst the relys serve to make more practical implementations for the components.

4.2 Aircraft Engine Starter System

The controllers of gas turbine aircraft engines are complex systems. High reliability operation in real time is obviously essential One function which such systems perform is controlling start sequences. This latter function is included because to start such an engine, it is necessary to control the fuel shut-off valve, starter motor air valve and igniters to a strict sequence, while checking the engine for various failure conditions, a task which would place considerable strain on the pilot. This case study considered the specification of a greatly simplified starter system.

The approach taken was to produce a set of functional requirements which defined the (simplified) behaviour of the system from the actual specifications of a specific engine project. These requirements were formalised as behavioural specifications, which were divided into two groups: the first of which regulated when certain events could occur (safety properties), and a second set which specify when events must be possible (liveness properties). An outline design was then produced which used a *constraint based* structure. A component process was defined for each safety property, which prevented that property being violated.

The control system is then defined to be the concurrent combination of these processes. The verification that such a system meets its safety requirements is simple, requiring only that the enforcing process is correct, but the liveness properties must be verified for every component. In fact, two models of the system were in fact produced. One was a model using Timed Failures specifications and Timed CSP, while a simpler untimed model was created and proved correct using the Failures model of CSP. This allowed many properties of the timed model to be proved by using the timewise refinement techniques described in the previous section.

We will now present some functional requirements typical of the simplified autostart system. The system takes inputs *FuelOn* and *FuelOff*, *AirOn* and *AirOff* from the pilot, and a *run* signal from the engine, and controls two valves: the Starter Air Valve (SAV) which controls the engine starter motor, and the Shut Off Valve (SOV) which can stop any fuel flow to the engine.

1. The SAV should only open when commanded on by the pilot.

2. When the starter is commanded off, but the SAV is open, the system should not refuse to close the valve.

3. If the SAV is open for 180s, the system must not refuse to close it. This ensures that the starter motor does not overheat.

The first item in this list is a safety property. The remaining two are liveness conditions. Global timing requirements define the maximum time allowed for a system to recognise the conditions mentioned in the liveness constraints and act accordingly. We will insist that any required action becomes available in a time T.

We will now consider these properties formalised in the Timed Failures model. The state of devices such as switches and valves depends on which of a pair of events (opening and closing, for example) occurred last. We define abbreviations to describe which states hold at the end of a given trace. The following predicate holds for a trace when it either contains at least one event a, with no subsequent b:

$$s\ ENDWITHIN(a,b) \quad == \quad \begin{array}{l} \lfloor \{a,b\} \neq \langle\rangle \\ \wedge\ last(s \lfloor \{a,b\}) = a \end{array}$$

We can define specific predicates on traces which describe the state of various elements of the system at the end of the trace. *AIRREQD* holds of a trace when the starter system is selected by the pilot at the time corresponding to the end of the current trace:

$$AIRREQD(s) \quad == \quad s\ ENDWITHIN(AirOn, AirOff)$$

OPENSAV, *FUELREQD*, *OPENSOV* and *RUNNING* (which holds when the engine is in normal operation at the end of the trace) can be defined similarly. We can also define abbreviations for common specifications. One common form of specification is that the occurrence of one event, c is enabled by another, a, and disabled by a third, b. The following specification describes the case where c is initially disabled:

$$c\ BETWEEN(a,b) \quad == \quad s = u^\frown\langle(t,c)\rangle^\frown v \Rightarrow (u\ ENDWITHIN(a,b))$$

$c\ OUTSIDE(b,a)$ can be defined similarly. The use of parameterised predicates can be considered as a shorthand for substitution.

Using the predicates defined above, our requirements can be described as follows:

- The SAV only opens when it is requested:

$$R_a \quad == \quad SAVopen\ \ BETWEEN\ (AirOn, AirOff)$$

- When starter air is turned off by the pilot, the valve should close within T:

$$R_b \quad == \quad \left.\begin{array}{l} \neg\ AIRREQD(s) \wedge \\ OPENSAV(s) \end{array}\right\} \Rightarrow SAVclose \notin \sigma\ (\aleph \upharpoonright (end(s)+T))$$

- The SAV must also be able to close when the engine reaches idle conditions, or after 3 minutes anyway:

$$R_c \quad == \quad OPENSAV(s) \Rightarrow SAVclose \notin \sigma\ (\aleph \upharpoonright (end(s)+180+T))$$

The implementation strategy used was to produce simple processes corresponding to safety requirements which govern partial correctness, while keeping their actions general enough

to satisfy the refusal requirements. The parallel combination of these processes should then meet that overall specification.

The following processes are used to reflect the repeated structure of the specification noted above: The process *DISABLED* prevents c from occurring until an a event has occurred more recently than a b, while *ENABLED* does the reverse (and allows c to occur initially). *DISABLED* corresponds to the *BETWEEN* specification forms, and in fact *DISABLED* (c, a, b) **sat** c *BETWEEN*(a, b):

$$\begin{aligned} DISABLED\ (c,a,b) \ == \quad & (a \rightarrow ENABLED\ (c,a,b)) \\ \Box\ & (b \rightarrow DISABLED\ (c,a,b)) \\ ENABLED\ (c,a,b) \ == \quad & (a \rightarrow ENABLED\ (c,a,b)) \\ \Box\ & (b \rightarrow DISABLED\ (c,a,b)) \\ \Box\ & (c \rightarrow ENABLED\ (c,a,b)) \end{aligned}$$

The parameterisation used here is a shorthand for alphabet transformation.

A single process will enforce a single constraint on *SOVopen*,

$$P_a \ == \ DISABLED\ (SAVopen, AirOn, AirOff)$$

Similar processes can be defined for other safety specifications.

To allow *SAVclose* to occur when the relevant conditions are met, two processes can be interleaved, rather than synchronised:

$$\begin{aligned} P_c \ &== \ P_{c1} \ ||| \ P_{c2} \\ P_{c1} \ &== \ ENABLED\ (SAVclose, AirOff, AirOn) \\ P_{c2} \ &== \quad (SAVopen \rightarrow P_{c3}) \\ &\quad \Box\ (WAIT\ 180\ ; (SAVclose \rightarrow P_{c3})) \end{aligned}$$

The controller is then modelled as the interlocked parallel combination of processes enforcing the required safety conditions.

$$CONTROL \ == \ (\ldots \underset{\{SAVopen, AirOn, AirOff\}}{\|} P_a) \underset{C}{\|} P_c$$

The correctness of this system can be proved simply by applying the proof system of the previous section, but a simpler method is to project the process algebra into the untimed *CSP* language, and verify the untimed properties relating to ordering of events in a simpler model such as the traces model of *CSP*. This allowed effort to be concentrated on the more complex timing issues of the design. Eventually we hope that mechanical verification of simple untimed properties will be practical in many cases, reducing the burden still further.

To summarise, this study produced timed and untimed versions of real functional requirements and captured them in the language of CSP and Timed CSP behaviour specifications. The use of a constraint based approach to their implementation allowed safety properties to be proven very simply, especially when used in conjunction with projection of processes from the timed to the untimed models. The resulting processes had a form which allowed a simple implementation to be constructed, and in fact a simulation

in the occam programming language was produced. The greatest difficulty encountered was the volume and complexity of the correctness proofs, many of which were based largely on simple manipulation of sets and sequences. This observation motivated research into simpler frameworks for verifying properties of *TCSP* processes, and indeed correlates well with a number of the other case studies to be discussed here.

4.3 Autonomous Guided Vehicles

Autonomous Guided Vehicles (AGVs) are mobile robots provided with a variety of sensing equipment, and intended to be able to find their way around an environment, usually in order to locate a target. In addition they may be expected to achieve a task such as acquiring an object, if they are provided with appropriate manipulators.

To cope with these problems many different AGV control systems have been proposed and developed, usually involving a significant degree of parallelism. One common approach is the hierarchical system where higher level control functions pass commands to lower levels which are concerned with more local control. (See section 4.4.2.) An alternative is the "subsumption" architecture which consists of modules each responsible for a specific task-achieving behaviour which may take control by overriding the outputs and inputs of other modules.

This study [Sta90] examined parts of a composite architecture being developed at Oxford University's Robotics Research Group. The AGV is controlled by a land-based computer, which passes sparse points along the intended path to a path-smoothing module in the vehicle. This in turn generates much closer points which are sent to the motor controller at regular intervals. Current work is related to adding an obstacle-avoid mechanism which overrides the normal path-following behaviour if an obstacle is detected by a ring of sonar sensors positioned around the vehicle.

The approach taken in this model was to decompose the system into modules corresponding to the elements of the architecture already proposed by the robotics specialists. The hardware environment is such that these processes could then be run concurrently, allowing the AGV to respond quickly to external stimuli.

As an example, consider the structure shown in Figure 1. The main components have the following functions: The sensors provide data about the vehicle's position and environment. In practice this will include a wide variety of hardware and software. The higher level control system represents the land-based computer system and the on-board path smoothing. It is responsible for accepting high level plans and generating points along the path which the vehicle is required to follow under normal circumstances. These points are produced more or less continually at regular intervals.

The emergency avoid system, however, would normally be in an idle state in which it monitors the vehicle sensors. If a collision appeared imminent, the module switches to an active state and provides points along some alternative same path until the danger of collision has been avoided, then returning to its idle state.

A multiplexor provides a link between the higher level control and the emergency avoid process and directs their output to a buffer. It relays data from the higher level control system to the buffer until a toggle signal is received from the emergency system. It then

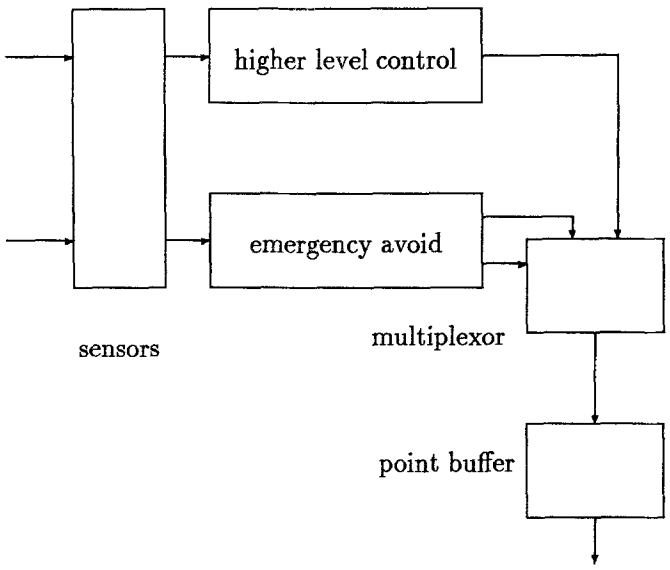

Figure 1: Structure of the AGV model

relays points from the emergency system until another toggle is signalled. In addition, it sends a signal to clear the buffer whenever it changes state.

The buffer itself is a model of the interface to the low level AGV hardware. It is fed with points from the control system, and has points removed from it at regular intervals by the low level control. The receipt of a clear signal resets it to an empty state. This design combines both hierarchical and subsumption styles: in normal operation commands are passed down a hierarchy of controllers, while the emergency system takes effect by overriding the normal hierarchical flow.

The technique adopted was to describe requirements placed on the above components in terms of Timed Failure specifications, to model the behaviour of the systems in the *TCSP* algebra, and to prove that these modules did meet their local requirements. The proof rules relating to parallel composition were then used to demonstrate some simple high level properties of the system. A number of these global properties, however, are best described in terms of the internal state of the system: in Timed CSP, a process and its environment must co-operate on all observable events, so purely internal events cannot be observed by the environment.

To handle this situation, an 'environmental' approach to specification was developed, as described in [DJS90]. Rather than conceal all the communication between the components of the system, these events are left visible, but the system behaviour is only guaranteed when the environment commits to allow such events to happen whenever the system does. Behavioural specifications are written in the form

$$E(s, \aleph) \Rightarrow F(s, \aleph)$$

where $E(s, \aleph)$ gives the assumptions which the system makes about the environment and $F(s, \aleph)$ describes what is required of the process. The 'environmental assumption'

required here is that the environment is always willing to observe events in a set A, expressed as

$$\mathrm{act}_A(s, \aleph) \quad == \quad [0, end(s, \aleph)) \times A \subset \aleph$$

Consider the multiplexor. The purpose of this device is to feed points to the point buffer. The source of the points can be either of the two channels in_1, from the emergency avoid system, or in_2 from the higher level control modules. Which of these is used at any one instant is determined by a signal *toggle* activated by the emergency avoid mechanism. The most important requirement is that the multiplexor should behave as a one-place buffer while it is propagating points from one of its input lines. This is a standard and well understood problem. In addition, it must satisfy some conditions on its real-time behaviour. For example following a toggle, a clear signal must be offered to the buffer within M_{clear}.

$$\langle (t, toggle) \rangle \textbf{ in } s \Rightarrow \quad clear \notin \sigma(\aleph \uparrow [t + M_{clear}, begin(s \upharpoonright t \downarrow clear)))$$

Also, following a clear, the multiplexor must be prepared to accept an input or a toggle event within M_{inp}, and the preceding event must have been a toggle

$$\langle (t, clear) \rangle \textbf{ in } s \Rightarrow \begin{array}{ll} & toggle \notin \sigma(\aleph \uparrow [t + M_{inp}, begin(s \upharpoonright t \downarrow toggle))) \\ \wedge & \quad in_1.PT \cap \sigma(\aleph \uparrow [t + M_{inp}, begin(s \upharpoonright t \downarrow in_1.PT))) = \{\} \\ & \vee \; in_2.PT \cap \sigma(\aleph \uparrow [t + M_{inp}, begin(s \upharpoonright t \downarrow in_2.PT))) = \{\} \\ \wedge & last(s \uparrow [0, t)) = toggle \end{array}$$

The following Timed CSP process is designed to meet the preceding specification. Using a derived interrupt operator $\underset{toggle}{\triangledown}$ to transfer control when an event *toggle* occurs, we can define

$$M1 \quad == \quad (\mu X \circ in_1?x \xrightarrow{M_{prop}} out!x \xrightarrow{M_{req}} X) \underset{toggle}{\triangledown} WAIT\ M_{clear}\ ;\ clear \xrightarrow{M_{inp}} M2$$

$$M2 \quad == \quad (\mu X \circ in_2?x \xrightarrow{M_{prop}} out!x \xrightarrow{M_{req}} X) \underset{toggle}{\triangledown} WAIT\ M_{clear}\ ;\ clear \xrightarrow{M_{inp}} M1$$

$$MUX \quad == \quad M1$$

The full multiplexor is modelled by MUX. In state M1 the multiplexor accepts input from the channel in1, and in state M2 it accepts input from channel in2. We can prove that MUX satisfies the required properties using the proof system described earlier. Other properties, and the other components can be treated in a similar manner.

Having specified the components of the AGV as separate processes, we can investigate the behaviour of them running concurrently and communicating on the appropriate channels. Consider the requirement that the system responds to an emergency input by clearing the point buffer with some time limit M_{resp}. This reduces to local properties on the sensors, the emergency avoid system, and the multiplexor. If the sensor relays its input within some time, say, M_{sens}, and the emergency controller responds within M_{avoid}, the using the property of the multiplexer above, we require that $M_{resp} \geqslant M_{avoid} + M_{sens} + M_{clear}$. The proof can be carried out using the rules for parallelism described in the previous section.

Similar properties can be formalised and proved in a similar manner. These include showing that following a trigger, a point produced by the emergency avoid mechanism will be available from the point buffer within a certain maximum time and that after this event, if the point buffer is not full, then it accepts points at a frequency determined by the rate at which the emergency avoid process calculates path points. These can be extended to make observations about the overall behaviour of the system, for example, to calculate how out-of-date the points coming from the buffer are in normal operation.

One of the major conclusions drawn from this project had also been noticed in the aircraft engine project: the specification notation is, if anything, too flexible, and allows too many ways of asserting the same things. Again some steps were taken to providing some standard predicates relating to offers and the occurrence of internal events. One difference in approach was that the approach taken to finding requirements was to have a specification for each possible event, describing what must occur before, and what may occur and must be offered after, any occurrence of it. It was found that with this method it was easier to be confident that all cases had been covered, than it was if the specifications were divided more conventionally into safety and liveness properties. In the case of the sensors the fact that the behaviour varied on a regular cycle meant that categorising by time was more useful than categorising by event.

The environmental technique proved convenient in talking about the internal behaviour of a system, as well as when dealing with the behaviour of the system as a whole. The proofs turn out to be long-winded if done with anything like formal rigour. A large part of the problem, however, is that specification of timing properties of systems involving concurrency is necessarily complex. Even when proofs of combination properties are not done, process algebra provides a concise but accurate method of representing processes.

4.4 Other Case Studies

4.4.1 A Local Area Network Protocol

A common form of local area network protocol is the *carrier-sense collision detect* protocol, of which the Xerox Ethernet [Xer80] is the best known example. In such a protocol, a number of nodes communicate over a broadcast medium—each is able to receive data transmitted by any other. Each node must wait until local activity on the broadcast medium has ceased before it attempts to transmit a message. If two nodes attempt to transmit within the same time interval, interference will be observed at both nodes. In this case, the nodes will *back off* or delay for some random period before retransmitting.

The protocol specifies interactions on two levels: one relating the physical details of the medium, and one relating the transmission of data packets from one node to another. These correspond to the lowest two levels of the seven layer ISO-OSI protocol model, which defines the function of a physical layer, a data link layer, and a further five layers above, beginning with the network layer. These layers are clients of the layers described by the Ethernet specification: the data link layer provides a virtual communication service for the layers above.

An analysis of the correctness of such a protocol is described in [Dav91]—a far more elegant description that we have space for here. Because the important functions of

protocols are specified in terms of layers and the services they provide each other, Davies develops a theory of structured specifications in Timed CSP as a basis for this case study. A typical layer L provides a service to the layer above, described in terms of shared interface A_L. The implementation of L may involve internal communications, and it may rely on the service provided by the layer K below.

If layer K provides a service S_K, and L satisfies a total specification T_L, then the combination of T_L and S_K must be enough to provide the service S_L to the layer above, under the assumption that events outside the interface A_L are to be concealed than L: we use H_L to denote this set of hidden events.

$$T_L \wedge S_K \;\Rightarrow\; S_L \setminus H_L$$

The symbol $\setminus$ corresponds to the assumption that events from H_L are to be concealed. This is the simplest form of environmental assumption.

The data link layer provides a simple service at each node i—messages received on a channel $i.in$ are encapsulated as data frames and transmitted across the network; a status report is returned on channel $i.rep$. Packets received at node i are stripped of framing information, and—if this is their destination—output on channel $i.out$. Given a set $NODE$ of nodes, PKT of packets, and a set REP of report values, the interface between the data link and its client layer is given by

$$\{i.in.p, i.rep.r, i.out.p \mid i \in NODE \wedge p \in PKT \wedge r \in REP\}$$

Some properties relate the service provided at a single node. For example, we may require that the transmitting part of a node alternates between accepting inputs and returning status reports. If IN_i and REP_i are the sets of input and report events at node i, we require

$$\forall i : NODE \bullet 1 \geqslant (\#(s \restriction IN_i) - \#(s \restriction REP_i) + 1) \geqslant 0$$

Other properties relate to the whole network: if $dest(p)$ is the destination node of packet p, a successful transmission at node i should be possibly only if the message is made available for output at the destination node

$$\left.\begin{array}{l} \langle (t, i.rep.success) \rangle \text{ in } s \\ dest(p) = j \\ last(s \restriction IN_i \uparrow [0, t) = (t', i.in.p) \end{array}\right\} \;\Rightarrow\; j.out.p \notin \sigma(\aleph \uparrow [t' + t_{max}, begin(s \restriction j.out.p)))$$

where t_{max} is the largest acceptable transmission delay.

Similar specifications can be written for the interface between the physical layer and the data link layer: the layers communicate along channels $i.put$ (transmitted data), $i.cs$ (carrier sense signal), $i.cd$ (collision detect signal) and $i.get$ (received data). The specifications of the components of the data link layer can also be formulated in the same way. In [Dav91], these specifications are presented in a high-level macro language, closer to natural descriptions.

If S_{PL} and S_{DL} denote the service provided by the physical layer and the data link respectively, and T_{DL} denotes the total specification of the data link, then we may use

the timed failures proof system to establish that

$$\left.\begin{array}{l} S_{PL}(s \restriction \Sigma_{PL}, \aleph_P) \\ T_{DL}(s \restriction \Sigma_{DL}, \aleph_D) \\ \aleph \restriction A_{PL} = (\aleph_D \cup \aleph_P) \restriction A_{PL} \\ \aleph \setminus A_{PL} = (\aleph_D \cap \aleph_P) \setminus A_{PL} \end{array}\right\} \Rightarrow S_{DL}(s, \aleph) \setminus\setminus H_{DL}$$

The total specification of the data link, together the service provided by the physical layer, is enough to guarantee the service required of the datalink.

We can extend the analysis to model an abstract implementation. For example, the transmission of a packet can be performed by a transmission link manager, *TLM*, defined as follows:

$$\begin{aligned} TLM &== down?f \xrightarrow{t_3} HOLD_{f,1} \\ HOLD_{f,n} &== cs \xrightarrow{t_{int}} (SEND_f \underset{cd}{\triangledown} HANDLE_{f,n}) \,;\, TLM \\ SEND_{\langle\rangle} &== rep!succ \xrightarrow{t_4} SKIP \\ SEND_{(x)^\frown s} &== send! \xrightarrow{t_{bit}} SEND_s \\ HANDLE_{f,n} &== BACKOFF_n \,;\, HOLD_{f,n+1} \text{if } n < 16 \\ HANDLE_{f,16} &== rep!fail \xrightarrow{t_5} SKIP \end{aligned}$$

The process *HOLD* corresponds to the state in which the node is waiting to transmit. *SEND* corresponds to bit transmission, and *HANDLE* describes the behaviour following a collision. The random delay is implemented by the *BACKOFF* process.

This case study provides a good illustration of the application of Timed CSP to the analysis of real-time protocols. The abstraction notation for CSP, with a few simple customisations, supports a clear, structured description of a complex protocol.

4.4.2 Laboratory Robot

Another control example is the study of the control of a laboratory robot [Sca90]. This work, carried out in collaboration with BP plc, considered the problem of controlling a cartesian robot used to prepare microscope slides. The control system is responsible for driving the robot to collect samples, mix solutions, place samples on clean slides, transfer slides to dry of a specified period and arrange resulting batches for collection. The parameters of each job, such as number of slides and processing time, are quite varied and the resulting system needs to be flexible.

A hierarchical structure was adopted, based on task structure. This form of architecture is common in robotics applications, as noted above. The higher levels control batch level operations, passing commands down to modules which perform operations on single slides which in turn drive processes which operate single movements or axes of motion. Commands are passed down the hierarchy and status information is passed back when the commands are completed.

The starting point for the study was to build a model of the behaviour of a typical level in terms of Timed CSP. Derived program constructs for building levels were built up,

and rules for their proving their correctness were derived. As with the communications protocol example, the correct behaviour of a layer in the system obviously depends on the service provided by lower level layers.

The properties required of the specific system being considered were captured as Timed Failures specifications. The overall form of interface specification was obviously quite simple: after a command *comm* is issued, some status message from the set *STAT* is returned within some time T:

$$\langle (t, comm) \rangle \text{ in } s \Rightarrow STAT \not\subseteq \sigma(\aleph \uparrow [t + T, begin(s \restriction t \lfloor STAT)))$$

Overall, this application demonstrated a very simple hierarchal structure which, while being quite common in communication protocols seems less common embedded system applications. This layered approach allowed program constructs to be re-used significantly in the development of this system, and closely relates to a typical systems in which we may well have one microprocessor controlling each axis of a robot, in turn controlled by a local system which guides the whole robot, which itself receives commands from some form of laboratory of plant level scheduling system. Although this higher level scheduling was not studied as part of either this example or the AGV control system case study, it is a task which involves significant communication between parallel modules and many real-time constraints, features which are amenable to analysis in the Timed CSP framework. Our next example study describes an attempt to analyse such a system in this framework, using a temporal logic specification notation to abstract away from some timing detail.

4.4.3 Flexible Manufacturing Systems

A flexible manufacturing system (FMS) is a combination of computer controlled tools, transport devices and controllers which can be used to process mixed batches of jobs of different types. Typical examples are automated machine tool shops in the mechanical engineering industry, where components are moved between stations by AGVs.

In a typical FMS raw parts of various types enter the system at discrete points of time and are processed concurrently, sharing a limited number of resources, such as machines, robots, fixtures and buffers. One major issue which must be addressed is deadlock, but there are also other properties of a manufacturing system which are also of interest, such as the times at which individual jobs start and stop processing, delays during processing, and the length of time that machines stand idle. The study described in [Wal91] considers two models of an FMS: an untimed model used for deadlock analysis, and a timed model which allows calculation of, for example, job completion times.

The FMS model is divided into four types of process: jobs, which represent the sequence of tasks to be performed on some material; machines, which represent the resources to be used; transport devices which represent the constraints on job movement, and a centralised controller.

The first model analysed a simple controller which allocates each job all the machines it needs to complete. Allocation is controlled by semaphores, one for ecah machine. The job processes must satisfy various conditions: it's initial action must be to request a set of resourcesfrom the controller, it must then request an AGV to transfer raw material to the first machine for processing, request transfers to subsequent machines and finally

request transfer to the final station and release its resources by sending a message to the controller.

The job communicates with the controller via p and v events which represent the usual semaphore operations. The AGV and jobs are synchronised by commands to the AGV of the form "get job i from machine a, transfer it to machine b". Events of the form *begin* and *end* synchronise the jobs with the machines. The properties of jobs can then be formalised in untimed CSP as predicates on trace refusal pairs. The FMS is represented as the interlocking parallel combination of the jobs, the machines, the AGV and the controller. Deadlock freedom is proved by the usual proof system for behavioural specifications.

The second model considered added timing information to the FMS model: it included the transportation times, the machines processing and resetting times, and the controllers response time. The communications outlined above are not sufficient to encode all the timing information for the model, so the events were changed to pass processing times from the job process to the machines and transport devices.

The process algebra descriptions used in the untimed model can be extended to the timed case:

$$\begin{aligned} AGV &= AGV_s \\ AGV_p &= go.(i,a,b,t) \rightarrow MOVE_{p,a}\,;\, done.a \\ &\quad \rightarrow MOVE_{a,b}\,;\, begin.b!t \rightarrow AGV_b \\ MOVE_{x,y} &== \prod_{min \leqslant t \leqslant max} WAIT(t-\delta)\,;\, \text{if } x \neq y \\ MOVE_{x,y} &== SKIP \text{if } x = y \end{aligned}$$

For simplicity, a simpler timed model was adopted for example analysis. The controller and AGV are replaced by a single scheduler: each job is decomposed into a operations which are placed in order by the scheduler. As an example of the analysis which can be performed, we may then prove a simple inductive property of the system: if an operation is added to the end of a job, and scheduled last, then the completion time is increased by the length of the operation.

The machines will be defined as they are above, except that they will each have a constant processing time p_m, and recovery time r_m. This can easily be extended to the more general variable time in the above model. So we have

$$M_m == begin.m?i \xrightarrow{p_m} done.m!i \xrightarrow{r_m} M_m$$

we assume that the time taken for the recursion is zero, i.e. *begin* will be offered at time r_m after the *done* was performed. We shall need to specify properties of the machines, for example if ever a *begin* and *end* events occur alternately, with a delay corresponding to the processing time between *begin* and *end*. These properties can be captured in temporal logic. First we define a predicate *DONE*. This simply captures the idea that when we offer the event $end.m.j$, no other events are performed until the event is performed, if it is ever performed.

$$DONE_m == ((\mathbb{O}_{end.m!j} \wedge \neg\, \mathbb{P}_{\sigma M_m})\, \overline{\mathcal{W}}\, \mathbb{P}_{end.m!j})$$

This gives the following expression of the property above:

$$P_m \;==\; \overline{\mathbf{G}}\,(\mathbb{P}_{begin.m?j} \Rightarrow \neg\, \mathbb{P}_{\sigma M_m}\, \mathcal{U}_{pm}\; DONE_m)$$

We can prove that the machine meets this specification (M_m **sat** P_m) by applying the rules discussed previously. Properties of the scheduler and operations can be captured similarly.

Representing the properties of well defined jobs by predicates $ALLDONE$, ONE_A, and writing $AF[s,t]$ for the specification which asserts that all jobs are complete at some time in the interval $[s,t]$, the proof system allows us to prove the required result: Given system FMS we may add an operation to give a system FMS' which takes between p_a and $p_a + r_a$ longer to complete.

$$\frac{\begin{array}{rcl} FMS & \textbf{sat} & (\mathrm{act}_\Sigma \Rightarrow AF_{[s,t]}) \wedge ALLDONE \\ \forall j : J & \bullet & J_j \textbf{ sat } ONE_{\sigma J_j} \\ S_J & \textbf{sat} & T_J \wedge ONE_{\surd} \end{array}}{FMS' \textbf{ sat } \mathrm{act}_\Sigma \Rightarrow AF[s + p_a, t + p_a + r_a]} \quad [\, i \in J \,]$$

where

$$FMS \;==\; \left(S_J \;{}_{A_S}\|_{A_J}\; J_J\right) \;{}_{A_S}\|_{A_M} \;|||_{i \in MACH}\; M_i$$

$$FMS' \;==\; \left(\left(S_J \,;\, OP_{i,a}\right) \;{}_{A_S}\|_{A_J}\; J_{J-\{i\}}\right) \;{}_{A_S}\|_{A_{J_i}}\; \left(J_i \,;\, OP_{i,a}\right)\Big) \;{}_{A_S}\|_{A_M} \;|||_{i \in MACH}\; M_i$$

The following observations were made: CSP can be used to model FMSs in a modular way, and in a manner which should be easily extended to include other aspects not covered here. However, even for simple systems the proofs can become long and involved. Timed CSP is able to model the timing constraints of the simple FMS, although the resulting model is even more complex. Temporal logic provides an a methods of abstracting from some of this detail, but a modular approach to the system design is still highly desirable, using higher level proof rules.

5 Conclusions

We believe that the theory of Timed CSP has now reached maturity; this is demonstrated by the case studies presented in this paper. The strength of the theory lies in its flexibility: a hierarchy of semantic models are available for reasoning at different levels of abstraction, and the methods of reasoning can be tailored to suit application-specific proof theories.

We are now actively engaged in producing two volumes to serve as definitive guides to the theory and application of Timed CSP. If Timed CSP (or any other formal method) is to be used outside the academic community, it is essential that the development process is supported by reliable software tools. The design and development of such tools is currently a major area of research at Oxford.

Acknowledgements

The theory of Timed CSP was inspired by Tony Hoare. Its development has been greatly aided by discussions with Michael Goldsmith and Geraint Jones. Jim Woodcock served as the director of the D.Phil thesis of Jim Davies, and Penny Probert co-directed the M.Sc theses of Scattergood, Stamper, and Wallace.

The foundational work on Timed CSP would not have been possible without the support of R. Grafton and particularly R.F. Wachter at the U.S. Office of Naval Research, and the support of ESPRIT BRA-3096 (SPEC). This research has benefitted considerably from discussions with other participants in the SPEC project.

Finally, we note that the first timed model for Communicating Sequential Processes was constructed in [Jon82], and that a model similar to our Timed Failures Model was constructed independently in [BoG87].

References

[BoG87] A. Boucher and R. Gerth, *A timed model for extended communicating sequential processes*, in *Proceedings of ICALP 87*, LNCS **267**, pp 95–114, Springer 1987.

[Dav91] J. Davies, *Specification and proof in real-time systems*, D.Phil thesis, Programming Research Group Technical Monograph PRG–93, Oxford University 1991.

[DJS90] J. Davies, D.M. Jackson, and S.A. Schneider, *Making things happen in Timed CSP*, Programming Research Group Technical Report TR–2–90, Oxford University 1990.

[DJS92] J. Davies, D.M. Jackson, and S.A. Schneider, *Broadcast communication for real-time processes*, in *Proceedings of the symposium on real-time and fault-tolerant systems, Nijmegen 1992,* to appear in Springer LNCS.

[DaS89] J. Davies and S.A. Schneider, *Factorising proofs in Timed CSP*, in *Proceedings of the fifth workshop on the mathematical foundations of programming language semantics*, LNCS **442**, pp 129–159, Springer 1990.

[DaS90] J. Davies and S.A. Schneider, *An extended syntax for Timed CSP*, Programming Research Group Technical Report TR–4–90, Oxford University 1990.

[DaS91] J. Davies and S.A. Schneider, *Recursion induction for real-time processes*, submitted for publication 1991.

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

[Jac89] D.M. Jackson, *The specification of aircraft engine control software in Timed CSP*, M.Sc thesis, Oxford University 1989.

[Jac90] D.M. Jackson, *Specifying timed communicating sequential processes using temporal logic*, Programming Research Group Technical Report TR–5–90, Oxford University 1990.

[Jac91] D.M. Jackson, *A temporal logic proof system for Timed CSP*, Programming Research Group Technical Report TR–2–91, Oxford University 1991.

[Ja+90] D.M. Jackson, J. Davies, G.M. Reed, A.W. Roscoe, and S.A. Schneider, *Specifying timed communicating sequential processes in temporal logic*, SPEC report, ESPRIT BRA 3096, 1990.

[Jon82] G. Jones, *A timed model of communicating processes*, D.Phil thesis, Oxford University 1982.

[Jon83] C. Jones, *Tentative steps towards a development method for interfering programs*, ACM Trans Prog Lang **5** 4, pp 596–619, 1982.

[Jon91] C. Jones, *Interference resumed*, Technical Report UMCS–91–5–1, Manchester University 1991; Australian Software Engineering Research 1991, P. Bailes (ed.), Springer 1991.

[Kay91] A. Kay, *A theory of rely and guarantee in Timed CSP*, in preparation 1991.

[KaR90] A. Kay and J.N. Reed, *A Specification of a Telephone Exchange in Timed CSP*, Programming Research Group Technical Report TR–19–90, Oxford University 1990.

[KR91a] A. Kay and J.N. Reed, *Using rely and guarantee in Timed CSP*, Programming Research Group Technical Report TR–11–91, Oxford University 1991.

[KR91b] A. Kay and J.N. Reed, *A rely and guarantee method for Timed CSP*, submitted for publication 1991.

[Low91] G. Lowe, *A probabilistic model of Timed CSP*, D.Phil status transfer thesis, Oxford University, 1991.

[MRS91] M.W. Mislove, A.W. Roscoe, and S.A. Schneider, *Fixed points without completeness*, in preparation 1991.

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

[Ree90] G.M. Reed, *A hierarchy of models for real-time distributed computing*, in *Proceedings of the Fifth Workshop on the Mathematical Foundations of Programming Language Semantics*, LNCS **442**, pp 80–128, Springer 1990.

[ReR86] G.M. Reed and A.W. Roscoe, *A timed model for communicating sequential processes*, in *Proceedings of ICALP 86*, LNCS **226**, Springer 1987.

[ReR87] G.M. Reed and A.W. Roscoe, *Metric spaces as models for real-time concurrency*, in *Proceedings of the Third Workshop on the Mathematical Foundations of Programming Language Semantics*, LNCS **298**, pp 331–343, Springer 1987.

[ReR91] G.M. Reed and A.W. Roscoe, *A study of nondeterminism in real-time concurrency*, in *Proceedings of the Second UK–Japan CS Workshop*, LNCS **491**, pp 36–63, Springer 1991.

[Sca90] B. Scattergood, *The description of a laboratory robot in Timed CSP*, M.Sc thesis, Oxford University 1990.

[Sch90] S.A. Schneider, *Correctness and communication of real-time systems*, D.Phil thesis, Oxford University 1990.

[Sch91] S.A. Schneider, *Unbounded non-determinism in Timed CSP*, SPEC report, ESPRIT BRA 3096, 1991.

[Sch92] S.A. Schneider, *An operational semantics for Timed CSP*, in *Proceedings of the Chalmers Workshop on Concurrency*, to appear 1992.

[Sc+90] S.A. Schneider, J. Davies, D.M. Jackson, G.M. Reed, and A.W. Roscoe, *Communication and correctness in Timed CSP*, SPEC report, ESPRIT BRA 3096, 1990.

[Sta90] R. Stamper, *The specification of AGV control software in Timed CSP*, M.Sc thesis, Oxford University 1990.

[Su91] S. Superville, *Specifying complex systems with Timed CSP: a decomposition and specification of a telephone exchange system which has a central controller*, M.Sc thesis, Oxford University 1991.

[Wal91] A.R. Wallace, *A TCSP case study of a flexible manufacturing system*, M.Sc thesis, Oxford University 1990.

[Xer80] The Ethernet Specification, available from the Xerox Corporation, reprinted in ACM *Computer Communication Review*, July 1981.