

An Expressive Real-Time CCS*

C. J. Fidge[†] J. J. Žic[‡]

December 1994

Abstract

We describe a new ‘real-time’ process algebra with simple semantics but considerable expressive power. It exhibits the advantages of both the ‘constraint-oriented’ and ‘marker variable’ specification styles. The definition extends Milner’s CCS, firstly with a simple notion of absolute time added to actions, and then with relative timing expressions which may refer to time markers.

1 Introduction

Process algebras such as CSP, CCS and LOTOS are popular formalisms for specifying concurrent systems, but they lack a model for real time. Although many proposals for real-time process algebras have been made, they generally suffer from overly complex semantics, especially where the concurrency operators are concerned, or rely on counter-intuitive priority-based concepts in order to achieve expressibility.

We define a new real-time process algebra which combines simple semantics with good expressive power. To do this we have combined the best features of two previously separate streams of real-time research, the ‘constraint-oriented’ and ‘marker variable’ approaches. As a concrete illustration we describe an extended version of Milner’s Calculus of Communicating Systems (CCS).

This paper defines the new language and illustrates its application with a small case study. Some familiarity with CCS is assumed.

2 Motivation

Numerous real-time extensions to the process algebraic specification languages have been proposed in the last 6–7 years, yet none has emerged as dominant.

*Extended version. A shorter version of this paper appeared in the *Proceedings of the 2nd Australasian Conference on Parallel and Real-Time Systems (PART’95)*, Fremantle, September 1995, pp. 365–372.

[†]Software Verification Research Centre, Department of Computer Science, The University of Queensland, Queensland 4072, Australia. E-mail: cjf@cs.uq.edu.au.

[‡]Software Engineering Research Group, School of Computer Science and Engineering, University of New South Wales, New South Wales 2052, Australia. E-mail: John.Zic@serg.cse.unsw.edu.au.

We believe this is because their semantic definitions have been too complex, or they have been based on concepts and operators that are counter-intuitive or ill-defined at the user level. We therefore seek to develop a real-time process algebra with simple semantics, that supports a natural style of specification.

In order to do this we are combining two previously distinct streams of research. Firstly, the ‘constraint-oriented’, or ‘well-caused’, approach takes advantage of true concurrency semantics in order to define a real-time model that does not suffer from the problematic inter-dependence of processes usually found when time is added to process algebra concurrency operators. This model was originally suggested by Fidge [10] and then independently by Aceto and Murphy [1] who explored its implications in depth. Secondly, the ‘marker variable’ model was suggested by Žic [21, 22] as an intuitively appealing way of expressing real-time requirements in process algebraic specifications relative to preceding events.

To illustrate the combined approach this paper describes how it can be embedded in CCS [17]. The resulting language has the following aims and characteristics.

- No new operators or transition rules are needed. Syntactically time is introduced by adding extra annotations to the existing prefix operator. Semantically time introduces extra information to the transition labels, but no new rules are added.
- The meaning of existing operators is not changed. In particular, we reject the notion of ‘time-determinism’ in which choices can be ‘decided’ by the passage of time. The times attached to alternatives do not alter the ability to select any ‘enabled’ action. Similarly, we aim to retain the good compositional properties that have made the process algebras so popular for defining concurrent behaviours. The combined behaviours of two processes are merely a subset of their separate behaviours.
- Logically independent behaviours evolve *independently*. Non-interleaving semantics are used so that parallel behaviours are dependent on one another only at explicit interaction points. This greatly simplifies the introduction of time to the parallel composition operators, and means that parallel behaviours can be analysed separately.
- The passage of time is not forced. In particular, we reject the concept of ‘tick events’ or ‘aging’ functions to make time progress. Such models are overly constructive and troublesome in the context of parallel composition because otherwise independent behaviours are required to agree on the passage of time.
- Generalised timing expressions are allowed. Marker variables are used to express relative timing requirements by holding the time at which significant actions occurred. These variables may then appear in timing expressions. Sets of absolute times form the basis of the timing model because they are more powerful than time intervals. They can express

non-contiguous ranges of times, e.g., for when there are two distinct ways to achieve a behaviour with widely differing execution times. Also set logic is well understood so a separate ‘interval algebra’ does not need to be defined.

- Specifications are non-constructive. They say *what* happens and *when*, but not *how*. Actions are dimensionless in time and zero-time separation of actions is allowed. Since process algebras are *specification*, not implementation, languages their actions merely denote the moments at which significant events (state changes) must be observed. Such denotations have no execution-time overheads in themselves, and the number of ‘names’ we may choose to give to a particular moment in time is unbounded.

3 Language definition

This section formally defines the real-time process algebra. An example of its use can be found in Section 4.

The language definition extends both the ‘basic’ and ‘full’ CCS languages. In the former case a simple definition of time is introduced via a single *absolute* time associated with all actions. Although conceptually adequate for expressing any desired real-time behaviour, this notation is too inconvenient for general use. We therefore then introduce marker variables and set-valued timing expressions to the full language so that concise and *relative* timing requirements can be stated.

3.1 Basic language

The basic language is identical to standard CCS [17, §2.4] with extra annotations on the prefix operator.

Let \mathcal{A} be a set of action *names*; $\bar{\mathcal{A}}$ a set of *co-names*; $\mathcal{L} = \mathcal{A} \cup \bar{\mathcal{A}}$ the *labels*; $Act = \mathcal{L} \cup \{\tau\}$ the set of all possible *actions* including the *silent* action τ ; f a *relabelling* function on actions; I an *indexing* set with $\{E_i \mid i \in I\}$ a family of expressions indexed by I .

Also let \mathcal{T} be a set of *absolute time* values. If a discrete time model is desired \mathcal{T} can be the set of natural numbers. For a continuous time model it is the non-negative reals. The following rules are independent of the time model adopted.

Let \mathcal{E} be the set of *agent expressions*. It is defined as follows, where E, E_i are already in \mathcal{E} .

1. $\alpha @ t^c E$, a Prefix ($\alpha \in Act$; $t, c \in \mathcal{T}$)
2. $\sum_{i \in I} E_i$, a Summation
3. $E_1 \mid E_2$, a Composition
4. $E \setminus L$, a Restriction ($L \subseteq \mathcal{L}$)

5. $E[f]$, a Relabelling

We also assume the existence of a set of *agent constants* \mathcal{K} each of which can be defined as follows, where \mathcal{G} is the set of *agents*, i.e., agent expressions without free variables.

6. $A \stackrel{\text{def}}{=} G$, agent constant Definition ($A \in \mathcal{K}, G \in \mathcal{G}$)

As usual we allow the ‘nil’ agent $\mathbf{0}$ to be defined as a choice between no alternatives [17, p.44]:

$$\mathbf{0} \stackrel{\text{def}}{=} \sum_{i \in \{ \}} E_i .$$

The only unusual features are the absolute time t associated with each action, and the ‘context’ c associated with each prefix operator. The absolute time t is supplied by the specifier. It states that action α can occur at exactly time t only. Obviously this is very restrictive—in the full language far more powerful timing specification mechanisms are built from this simple basis, however.

The *context* c is used to support the definitions only. It defines the time at which the *causally* preceding action (if any) occurred and is used to ensure that time does not go backwards. It is not intended to be seen by the specifier. The context may therefore always be omitted in specifications, in which case a default value is assumed:

$$\alpha@t \cdot E = \alpha@t^0 E .$$

3.1.1 Transitional semantics

The transitional semantics is closely based on standard CCS [17, §2.5] except that transition labels carry the absolute time at which each action occurred and the transition rule for prefix is extended.

A *labelled transition system* $(S, T, \{ \xrightarrow{a} \mid a \in T \})$ consists of a set S of *states* (taken to be \mathcal{E}), a set T of *transition labels* and a set of *transitions* $\xrightarrow{a} \subseteq S \times S$ for each $a \in T$.

Each transition label $a \in T$ is a pair (α, t) consisting of an action name $\alpha \in \text{Act}$, and an absolute time $t \in \mathcal{T}$ recording when the transition occurred.

Transition rules for all basic CCS operators are shown in Figure 1, where $\alpha \in \text{Act}$; $t, c \in \mathcal{T}$; $a \in T$; $E, F \in \mathcal{E}$; $G \in \mathcal{G}$; $l \in \mathcal{L}$; $L \subseteq \mathcal{L}$.

The significant change is in **Act**. It states that action α , specified to occur at absolute time t , can do so only if t is greater than or equal to the context time c . Since c denotes the time at which the causally preceding action occurred, this ensures that time cannot go backwards. In Figure 2, for instance, action c is specified to occur at time 4 but it cannot because its context states that the previous action occurred at time 5.

After an action is performed the subsequent behaviour E is updated in **Act** to record the fact that the most recent action occurred at time t . This is denoted $(E)^t$ and is defined easily by induction on agent expressions as follows.

1. $(\alpha@u^c E)^t = \alpha@u^t E$
2. $(\sum_{i \in I} E_i)^t = \sum_{i \in I} (E_i)^t$

$$\begin{array}{l}
\mathbf{Act} \quad \frac{}{\alpha @ t :^c E \xrightarrow{(\alpha, t)} (E)^t} \quad (t \geq c) \\
\mathbf{Sum}_j \quad \frac{E_j \xrightarrow{a} E'_j}{\sum_{i \in I} E_i \xrightarrow{a} E'_j} \quad (j \in I) \\
\mathbf{Com}_1 \quad \frac{E \xrightarrow{a} E'}{E | F \xrightarrow{a} E' | F} \\
\mathbf{Com}_2 \quad \frac{F \xrightarrow{a} F'}{E | F \xrightarrow{a} E | F'} \\
\mathbf{Com}_3 \quad \frac{E \xrightarrow{(l, t)} E', F \xrightarrow{(\bar{l}, t)} F'}{E | F \xrightarrow{(\tau, t)} E' | F'} \\
\mathbf{Res} \quad \frac{E \xrightarrow{(\alpha, t)} E'}{E \setminus L \xrightarrow{(\alpha, t)} E' \setminus L} \quad (\alpha, \bar{\alpha} \notin L) \\
\mathbf{Rel} \quad \frac{E \xrightarrow{(\alpha, t)} E'}{E[f] \xrightarrow{(f(\alpha), t)} E'[f]} \\
\mathbf{Con} \quad \frac{P \xrightarrow{a} P'}{A \xrightarrow{a} P'} \quad (A \stackrel{\text{def}}{=} P)
\end{array}$$

Figure 1: Transition rules.

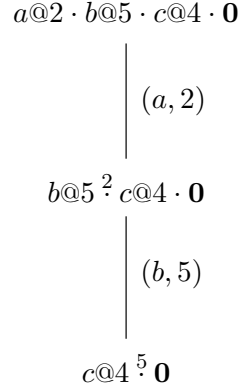


Figure 2: Derivation tree showing that time cannot go backwards in prefixes.

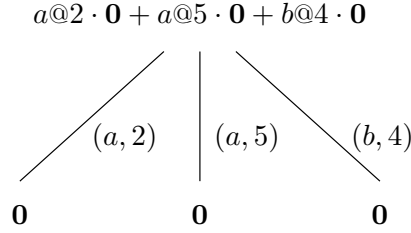


Figure 3: Derivation tree showing distinctness of timed actions and time-independent choice.

3. $(E_1 \mid E_2)^t = (E_1)^t \mid (E_2)^t$
4. $(E \setminus L)^t = (E)^t \setminus L$
5. $(E[f])^t = (E)^t[f]$
6. $(A)^t = (G)^t, \quad (A \stackrel{\text{def}}{=} G)$

$(E)^t$ thus replaces the ‘context’ of the *next* prefix(es) in E with t . (This is sufficient because action prefix is the *only* operator that can directly perform an action.)

We can think of the absolute times in the transition rules as part of the action name. Thus, in Figure 3, action a performed at time 2 is distinct from action a at time 5.

Rule **Sum** tells us that time does not influence choices. This is because we view an agent expression as a non-constructive specification, not an executable implementation. Expression

$$a@5 \cdot \mathbf{0} + b@4 \cdot \mathbf{0}$$

thus *specifies* a system that *either* performs a at time 5 *or* action b at time 4. It does *not* state that only the earlier alternative is allowed, as illustrated by the three possible behaviours in Figure 3.

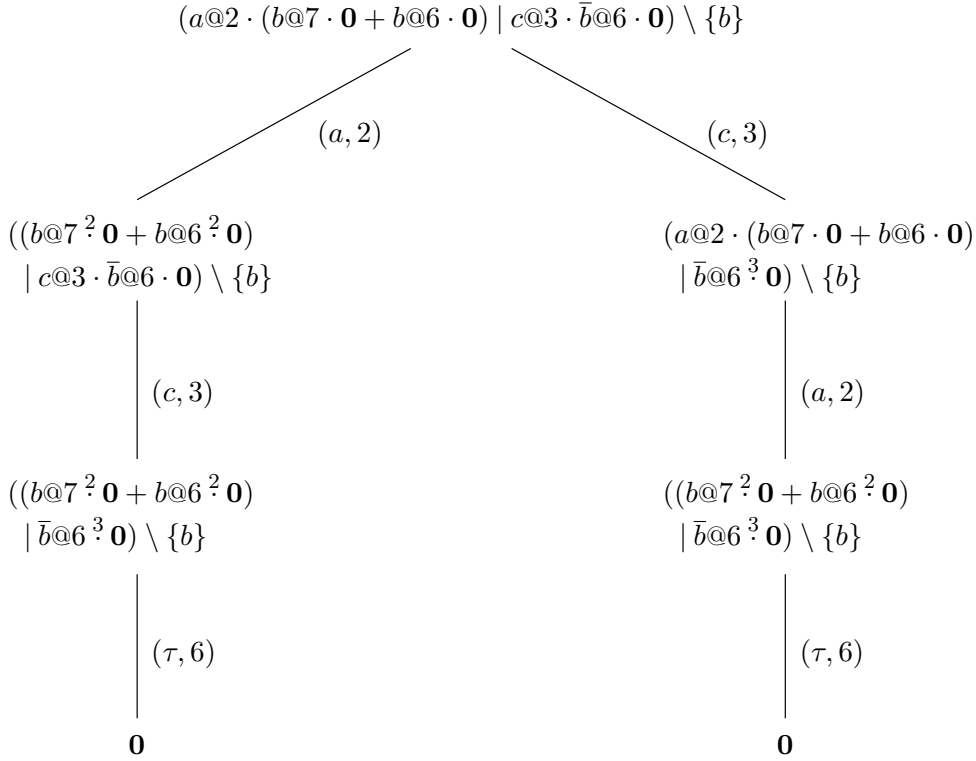


Figure 4: Derivation tree showing independence of parallel actions and the need to agree on times to interact.

Rules **Com**₁ and **Com**₂ tell us that parallel agents do not interact when performing independent actions. In Figure 4, therefore, actions a and c occur entirely separately, in either order. If viewed as an interleaving of actions it thus appears that time goes backwards on the right-hand derivation. In fact, the times on these two transitions are *incomparable* in the partial ordering defined by causality. An anomaly arises only if we attempt to impose an *arbitrary* total order on independent actions [12]. As noted by Fidge [10], the true, partial causal ordering never contradicts the temporal one or, as succinctly stated by Aceto and Murphy [1], such traces are “ill-timed but well-caused”.

Furthermore, rule **Com**₃ states that agents can interact only if they agree on the time at which the shared action occurs. Thus, in Figure 4, the shared action can occur at time 6 only, even though the first agent is also prepared to perform it at time 7. If there is *no* time at which two agents agree to interact then the action is not possible. Clearly the semantics linking timing behaviour and concurrency is very simple in this model. Furthermore, the approach accords well with the abstract ‘constraint-oriented’ specification style [8].

3.1.2 Bisimulations and equivalence

The bisimulation semantics of our language is very simple. As shown in Section 3.1.1, the ‘labels’ in our transition system consist of action name/absolute time pairs, rather than just action names. The only change that is needed to

standard CCS semantics is thus to replace the occurrences of action names with name/time pairs in the definitions.

Strong bisimulation [17, §4.2] is defined as follows.

Definition 1. A binary relation $\mathcal{S} \subseteq \mathcal{P} \times \mathcal{P}$ over agents is a *strong bisimulation* if $(P, Q) \in \mathcal{S}$ implies, for all $(\alpha, t) \in \text{Act} \times \mathcal{T}$,

1. whenever $P \xrightarrow{(\alpha, t)} P'$ then, for some $Q', Q \xrightarrow{(\alpha, t)} Q'$ and $(P', Q') \in \mathcal{S}$, and
2. whenever $Q \xrightarrow{(\alpha, t)} Q'$ then, for some $P', P \xrightarrow{(\alpha, t)} P'$ and $(P', Q') \in \mathcal{S}$.

For two agents to be strongly bisimilar, denoted $P \sim Q$, they must therefore be able to do the same actions, including τ , *at the same times*. Thus the following two agents are not bisimilar

$$a@3 \cdot b@5 \cdot \mathbf{0} \not\sim a@3 \cdot b@4 \cdot \mathbf{0},$$

even though they can both perform the same actions, because they do action b at different times.

The definition of weak equivalence [17, §5.1] is extended similarly. Let $s \in (\text{Act} \times \mathcal{T})^*$ be a sequence of ‘timed’ transition labels, and $\hat{s} \in (\mathcal{L} \times \mathcal{T})^*$ be the sequence gained by deleting all pairs in s whose first element is τ . Also define a new transition system, denoted $E \xrightarrow{s} E'$, stating that E can be transformed to E' by performing the sequence of timed actions in s , with zero or more timed τ actions before and after each action in s . Since this new transition system is defined using the old one ‘ \longrightarrow ’ [17, p.107], then in order for a ‘ \Longrightarrow ’ transition to be valid the times in the action sequence between E and E' cannot go backwards.

Definition 2. A binary relation $\mathcal{S} \subseteq \mathcal{P} \times \mathcal{P}$ over agents is a (*weak*) *bisimulation* if $(P, Q) \in \mathcal{S}$ implies, for all $(\alpha, t) \in \text{Act} \times \mathcal{T}$,

1. whenever $P \xrightarrow{(\alpha, t)} P'$ then, for some $Q', Q \xrightarrow{(\hat{\alpha}, t)} Q'$ and $(P', Q') \in \mathcal{S}$, and
2. whenever $Q \xrightarrow{(\alpha, t)} Q'$ then, for some $P', P \xrightarrow{(\hat{\alpha}, t)} P'$ and $(P', Q') \in \mathcal{S}$.

Like the usual CCS semantics for bisimilarity, denoted $P \approx Q$, the number of silent τ actions performed between observable ones is not relevant. For instance,

$$a@3 \cdot \tau@5 \cdot b@8 \cdot E \approx a@3 \cdot \tau@4 \cdot \tau@7 \cdot b@8 \cdot E.$$

However, unlike the usual semantics, τ actions can still have an effect via their influence on subsequent timing ‘contexts’. For instance,

$$a@3 \cdot \tau@5 \cdot b@8 \cdot E \not\approx a@3 \cdot \tau@9 \cdot b@8 \cdot E$$

because on the left-hand side the b action is possible, but on the right it is not, since it would require time to go backwards.

Finally, the definition of fully-substitutive equality [17, §7.2] is again a straightforward extension.

Definition 3. P and Q are *equal* or (*observation-*)*congruent*, written $P = Q$, if for all (α, t) ,

1. whenever $P \xrightarrow{(\alpha, t)} P'$ then, for some $Q', Q \xrightarrow{(\alpha, t)} Q'$ and $P' \approx Q'$, and
2. whenever $Q \xrightarrow{(\alpha, t)} Q'$ then, for some $P', P \xrightarrow{(\alpha, t)} P'$ and $P' \approx Q'$.

As usual this definition serves to cover the case where a τ action is the first one in a choice. It makes an *initial* timed τ significant, even when it takes zero time. For example, given some initial context $c \geq 0$,

$$\tau@c \cdot E \neq (E)^c,$$

even though E has the same ‘starting time’ in both cases, because the two agents are not necessarily equivalent when used as operands in a choice.

3.1.3 Equational laws

The equational laws of our revised language differ significantly from those of standard CCS [17, ch.3] in only two respects.

Most of the laws are unchanged. Propositions 1 [17, p.62], 4 [17, p.65], 8, 9, 10 [17, p.80] and Corollary 11 [17, p.81] are not altered in any way, and Corollary 7 [17, p.70] requires only a minor syntactic extension due to our new prefix operator. Thus our language continues to obey natural properties such as

- 1(1) $P + Q = Q + P$
- 7(2) $(\alpha@t \cdot Q)[f] = f(\alpha)@t \cdot Q[f]$
- 8(2) $P | (Q | R) = (P | Q) | R$
- 9(2) $P \setminus K \setminus L = P \setminus (K \cup L)$

and so on.

A significant difference occurs in the handling of τ actions in prefixes, however. Proposition 2 and Corollary 3 [17, pp.62-3] are revised as follows.

Proposition 2: τ laws

- (1) $\alpha@t \cdot \tau@u \cdot P = \alpha@t \cdot (P)^u$, if $t \leq u$
- (2) $(P)^t + \tau@t \cdot P = (P)^t$
- (3) $\alpha@t \cdot (P + \tau@u \cdot Q) + \alpha@t \cdot (Q)^u = \alpha@t \cdot (P + \tau@u \cdot Q)$, if $t \leq u$

Corollary 3 $(P)^t + \tau@t \cdot (P + Q) = \tau@t \cdot (P + Q)$

Apart from the obvious syntactic change of adding times to the actions, the main extension is the use of our auxiliary ‘context changing’ function to ensure that agent expressions that are intended to be equivalent have the same initial contexts. In Proposition 2(1), for instance, we use $(P)^u$ rather than P on the right-hand side because the occurrence of P on the left can perform actions at only time u or later. (In general some agent $(P)^{10}$ is different than $(P)^{20}$

because the former *may* be able to perform an action at time 15, whereas the latter cannot.)

The new side-conditions on Propositions 2(1) and 2(3) are an unfortunate complication. They are needed to ensure that the τ actions *can* indeed occur and are a necessary consequence of the **Act** rule. Consider the situation on the left-hand side of Proposition 2(1) if t was *greater* than u . After action α was performed the state would be $\tau@u \overset{t}{P}$ which is equivalent to the nil agent **0** because the τ action is ‘blocked’. This is different than the corresponding expression from the right-hand side, i.e., $(P)^u$, which may still perform any actions from P as long as they occur after time u . (Indeed all of the laws involving prefix become much more complicated if we allow arbitrary contexts to be supplied, rather than the default of 0, because these could block any action.) Also see Section 5.

The second major difference is that the ‘expansion law’, as defined by Proposition 5 and Corollary 6 [17, p.69], does not apply to our language (except in the degenerate case where all actions occur at the *same* time). This is a natural consequence of our decision to use true concurrency semantics—the expansion law is applicable only to interleaving models. We do not see this as a serious problem, however. The loss of this law is more than compensated for by the simplicity of our real-time semantics. It has long been recognised that this law is at odds with the requirements of modelling distributed, real-time systems [13, 9].

Nevertheless, Aceto and Murphy [1] show how a form of expansion theorem can be defined for a timed, non-interleaving model by using the transition, rather than prefix, structure of agents as the basis. To achieve this they define a new prefix-like operator which models not the ability of an action to precede an agent, but the ability of an action label to lead to an agent expression via a transition. In our case we could define such an operator, denoted ‘ \diamond ’, as follows.

$$\overline{\alpha@t \overset{c}{\diamond} E \xrightarrow{(\alpha,t)} (E)^t}$$

This transition rule is exactly the same as **Act**, except that it does not constrain time to go forwards. It can thus model the *apparent* reversal of time created by interleaving independently timed transitions. Proposition 5 and Corollary 6 can then be trivially modified as follows.

Proposition 5: the expansion law

Let $P \equiv (P_1[f_1] \mid \dots \mid P_n[f_n]) \setminus L$, with $n \geq 1$. Then

$$\begin{aligned} P &= \sum \left\{ f_i(\alpha)@t \diamond (P_1[f_1] \mid \dots \mid P'_i[f_i] \mid \dots \mid P_n[f_n]) \setminus L : \right. \\ &\quad \left. P_i \xrightarrow{(\alpha,t)} P'_i, f_i(\alpha) \notin L \cup \bar{L} \right\} \\ &+ \sum \left\{ \tau@t \diamond (P_1[f_1] \mid \dots \mid P'_i[f_i] \mid \dots \mid P'_j[f_j] \mid \dots \mid P_n[f_n]) \setminus L : \right. \\ &\quad \left. P_i \xrightarrow{(l_1,t)} P'_i, P_j \xrightarrow{(l_2,t)} P'_j, f_i(l_1) = \overline{f_j(l_2)}, i < j \right\} \end{aligned}$$

Corollary 6

Let $P \equiv (P_1 \mid \dots \mid P_n) \setminus L$, with $n \geq 1$. Then

$$P = \sum \left\{ \alpha@t \diamond (P_1 \mid \dots \mid P'_i \mid \dots \mid P_n) \setminus L : P_i \xrightarrow{(\alpha,t)} P'_i, \alpha \notin L \cup \bar{L} \right\}$$

$$+ \sum \left\{ \tau@t \diamond (P_1 \mid \dots \mid P'_i \mid \dots \mid P'_j \mid \dots \mid P_n) \setminus L : \right. \\ \left. P_i \xrightarrow{(l_1, t)} P'_i, P_j \xrightarrow{(l_2, t)} P'_j, i < j \right\}$$

3.2 The full language

Just as Milner extends basic CCS with a more expressive ‘value-passing’ notation, itself defined in terms of the basic language, we extend the primitive timing notion presented above with a much more expressive notation using marker variables and time sets.

Our full real-time language is an extension to Milner’s value-passing calculus [17, §2.8], and can thus take advantage of CCS variables in timing expressions. Let \mathcal{V} denote the set of CCS *value variable* names. We introduce a distinct set \mathcal{M} of *marker variable* names which are used to hold absolute times. We also introduce \mathcal{R} as the set of *non-empty* sets of absolute times from \mathcal{T} .

Let \mathcal{C} be the finite functions from marker variables to absolute times. Thus some function $C \in \mathcal{C}$ defines a mapping from zero or more marker variable names to absolute times. Such functions are denoted $\{m \mapsto 1, n \mapsto 4, \dots\}$ when variable m maps to 1, n maps to 4, etc. Where our basic language had ‘contexts’ consisting of a single time value c , the full language has contexts consisting of pairs $(c, C) \in \mathcal{T} \times \mathcal{C}$, i.e., an absolute time and a marker variable mapping. The absolute time defines when the causally preceding action occurred, as before. The mapping defines the values associated with previously encountered marker variables, if any.

Syntactically the major difference between the basic and full languages is a further extension to the prefix operator.

1. $\alpha@T: M \xrightarrow{(c, C)} E$, a Prefix ($\alpha \in Act$; $(c, C) \in \mathcal{T} \times \mathcal{C}$; $T \in \mathcal{R}$; $M \subseteq \mathcal{M}$)

Its context is extended as explained above. It also allows the action to be followed by a time-set-valued expression $T \in \mathcal{R}$, thus stating that this action may occur at any one of the times in T . T can be expressed using conventional set-comprehension notation, with free variables drawn from \mathcal{M} and \mathcal{V} as long as they are defined in C and the surrounding CCS variable scope, respectively. The current language limits the scope of marker variables to strictly sequential agent expressions. Furthermore the action can be followed by a set of marker variable names M . When (and if) the action occurs each of the variables in M will be assigned the value of the absolute time at which the action occurred, for use in subsequent timing expressions. (We also assume, but have not shown above, that action names in prefixes can be followed with message-passing parameters, as in value-passing CCS [17, p.55].)

A number of syntactic conveniences are used. As in the basic language, the contexts are not usually explicitly supplied by the user, so a simple default is assumed:

$$\alpha@T: M \cdot E = \alpha@T: M \xrightarrow{(0, \{\})} E .$$

We omit the set brackets for singleton sets, i.e.,

$$\alpha@t: m \xrightarrow{(c, C)} E = \alpha@\{t\}: \{m\} \xrightarrow{(c, C)} E ,$$

F	\widehat{F}
1. $a@T: M^{(c,C)} E$	$a@T\{\widehat{\vec{x}/\vec{v}}\}: M^{(c,C)} E$
2. $a@\{t_1 \dots t_n\}: M^{(c,C)} E$	$\sum_{\{t:t_1 \dots t_n t \geq c\}} a@t: \widehat{M}^{(c,C)} E$
3. $a@t: M^{(c,C)} E$	$a@t^c (E)_{(t,C \oplus \{\widehat{m}: M \cdot m \mapsto t\})}$

Figure 5: Translations from full to basic prefix operators.

where $t \in \mathcal{T}$ and $m \in \mathcal{M}$. Also empty marker variable sets may be omitted:

$$\alpha@T^{(c,C)} E = \alpha@T: \{\}^{(c,C)} E .$$

Finally, timing expressions are made optional by assuming the least-defined default:

$$\alpha: M^{(c,C)} E = \alpha@\{t : \mathcal{T} | t \geq c\}: M^{(c,C)} E$$

(the earliest time at which α can occur is c because this is the time at which the preceding action happened).

Milner lets agents F in the value-passing calculus be rewritten in basic CCS via a simple translation function, denoted \widehat{F} [17, p.56]. We extend this translation function further, as shown by the rules in Figure 5 for the new prefix operator, where $a \in Act$; $T \in \mathcal{R}$; $M \subseteq \mathcal{M}$; $(c, C) \in \mathcal{T} \times \mathcal{C}$; $t, t_1, \dots, t_n \in \mathcal{T}$; $\vec{x} \subseteq \mathcal{V} \cup \mathcal{M}$; $\vec{v} \subseteq \mathcal{T}$. It allows expressions in the full language, with marker variables and arbitrary timing expressions, to be routinely re-expressed in the basic language, with simple absolute times only, thus formally defining their meaning.

Rule 1 allows timing expressions to be eliminated, replacing them with the sets of absolute times they denote. It assumes that all elements of the set of free variables \vec{x} in timing expression T are either defined CCS value variables from \mathcal{V} , or are marker variables from \mathcal{M} that have a value defined in the current context C . Notation $T\{\vec{x}/\vec{v}\}$ then represents replacement of all free occurrences of elements of \vec{x} in T by the corresponding CCS ‘value’ or marker variable values, respectively. In the third node of Figure 6, for instance, the value of free variable m in expression $\{m + 1 \dots m + 3\}$ is bound by its appearance as a marker variable defined in context $(4, \{m \mapsto 2\})$, so it is replaced in the set expression to yield the simple set of absolute times $\{3, 4, 5\}$ (of which only the second two times are possible since the preceding action occurred at time 4).

Rule 2 then allows simple sets of absolute times to be eliminated by re-expressing them as a choice between the action occurring at each time in the set. This is a very powerful property, not enjoyed by other real-time process algebras. It is made possible only by our rejection of ‘time-deterministic’ choices [10] and is a major contributor to the simplicity of our semantics. Thus the third node

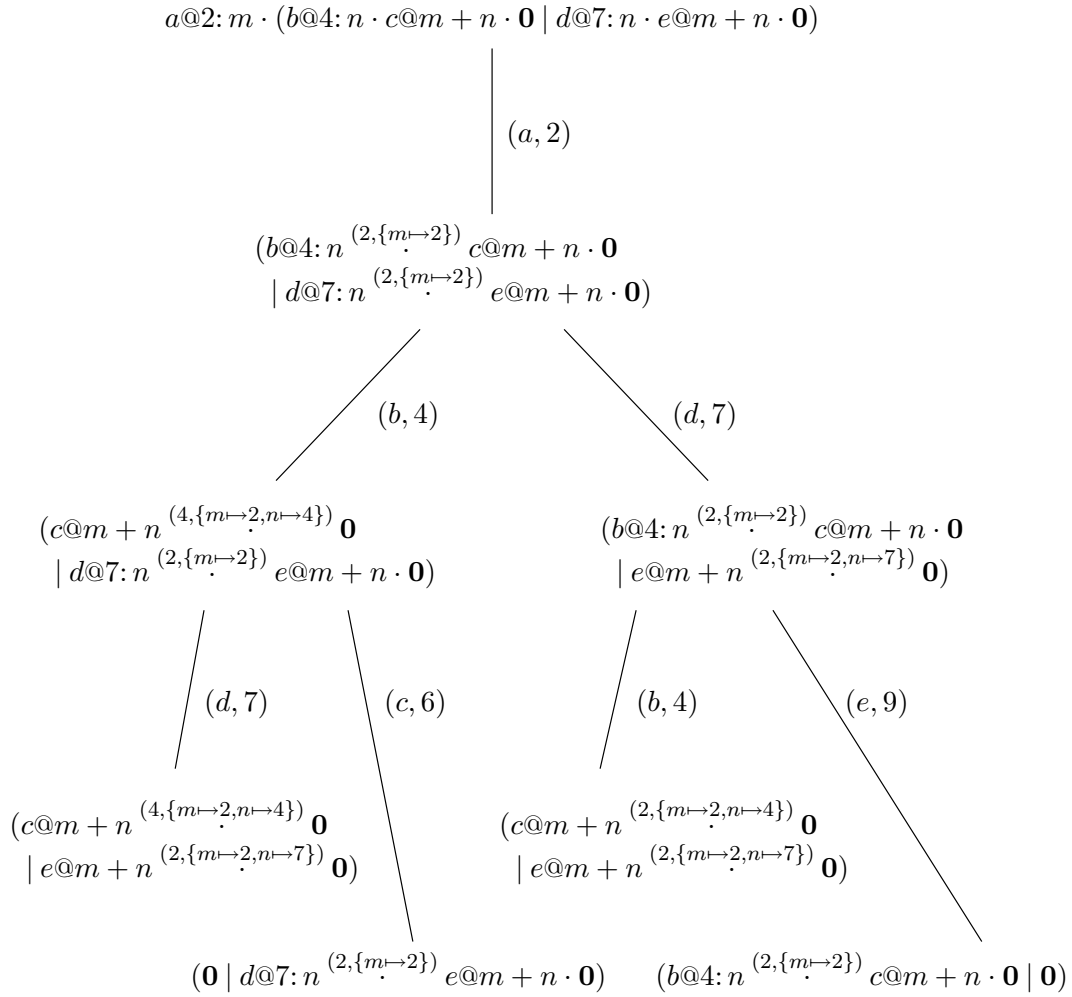


Figure 7: Partial derivation tree showing inherited and independent marker name spaces.

parallel agents to exchange marker information. If this marker exchange mechanism is allowed, the specifier must be aware that direct use of time readings is not straightforward. This is because time readings used in this manner are not, in general, comparable [12].

Simple rules for operators other than prefix allow the translation function to be recursively applied through an entire agent expression, in exactly the same way as the standard CCS model [17, p.56].

4 Case study: repeater system

To illustrate the use of the full language this section shows how a particular software requirement can be specified.

Consider a ‘repeater’ that must read numbers from a memory-mapped i/o location without missing, duplicating or mis-reading them. Numbers appear periodically, every 50 time units, but are reliably readable for only 45 time

units. There is a 5 time unit transition between each number, during which the value in the memory location is ill-defined. Thus the first value can be read reliably from time 0 to 45, inclusive, the second from 50 to 95, the third from 100 to 145, and so on.

Firstly we define a source S of such numbers as follows. It is an agent that represents the situation where an increasing series of numbers, starting at 0, appears at ‘location’ s , with timing as outlined above. It does not constrain how the numbers are ‘read’, however. It allows each number to be read several times, or not at all. Furthermore, it allows reads to occur during the transition period, in which case the error value ‘ -1 ’ is sent.

$$\begin{aligned} S &= S_0 \\ S_n &= \sum_{i \in n \dots \infty} \left(\bar{s}(i) @ \{t | i * 50 \leq t \leq i * 50 + 45\} \cdot S_i \right. \\ &\quad \left. + \bar{s}(-1) @ \{t | i * 50 + 45 < t < (i + 1) * 50\} \cdot S_i \right) \end{aligned}$$

Agent S_n represents the situation where the last value read was in the n^{th} period. The next ‘read’ may then occur in the i^{th} period, where i is at least as great as n . Thus a number of periods may go by with no read occurring, or a number of reads may occur in the same period. When the next read *does* occur there are two possibilities. Either the read occurs at the correct time, i.e., within 45 time units of the beginning of period i , in which case the number of this period is sent on channel s , or the read occurs at the wrong time, in the last 5 time units of period i , in which case the error value -1 is sent.

Our goal is to define a repeater agent R which, when placed in such an environment, i.e.,

$$(S | R) \setminus \{s\}$$

will read and repeat values from S , on some channel r , so that the sequence $0, 1, 2, \dots$ is correctly produced, with no duplicated, missing or ‘error’ values.

As a first attempt consider the following agent:

$$R1 = s(x) \cdot \bar{r}(x) \cdot R1 .$$

It reads values from s into variable x and repeats them on channel r . However it may read at *any* time, and thus may miss values entirely or read error values. Also \bar{r} may occur any time after the preceding s , since no timing requirement is specified, so even if a value is correctly read it may take so long to repeat it that subsequent values are missed.

Let us use a marker variable m to correct the second of these problems:

$$R2 = s(x) : m \cdot \bar{r}(x) @ \{t | t < m + 5\} \cdot R2 .$$

This agent guarantees that \bar{r} will occur within 5 time units of the preceding action s . Thus, if the read was successful, i.e., occurred within the first 45 time units of the period, the value will be repeated before the next number becomes available. (In fact this timing requirement is much stricter than necessary.) $R2$ may still fail to read the values correctly, however. It can still miss, duplicate or mis-read numbers.

As a further improvement we take advantage of the constraint-oriented specification style to ensure that s actions never occur ‘between’ values:

$$R3 = s(x)@t|\exists n \geq 0 \cdot n * 50 \leq t \leq n * 50 + 45\}: m \cdot \bar{r}(x)@t|t < m + 5\} \cdot R3 .$$

This ensures that s actions can occur only when a ‘good’ value is available. It thus constrains agent S so that its second alternative is never possible. But, although $R3$ will never read the error value, it can still skip numbers or read them more than once.

To complete the example we further constrain the repeater so that it will perform a read exactly once in each period:

$$\begin{aligned} R4 &= R4_0 \\ R4_n &= s(x)@t|n * 50 \leq t \leq n * 50 + 45\}: m \cdot \bar{r}(x)@t|t < m + 5\} \cdot R4_{n+1} . \end{aligned}$$

Here the CCS value-passing variable n acts like a counter for time periods. It is incremented after each (successful) read so that the next read *must* occur in the subsequent period. $R4$ thus further constrains S .

We have now fully *specified* a solution to the repeater problem. It is still far from being an implementation, however. As a final illustration of the way timed agents can constrain one another we refine our solution to a closer approximation of a particular implementation. A key feature of agent $R4$ is its ability to decide *when* to perform a s action. In practice this would be implemented via a hardware clock. Therefore, let us define a satisfactory ‘implementation’ T , which derives its timing information from a clock C :

$$\begin{aligned} R5 &= (C | T) \setminus \{a\} \\ C &= \bar{a}@0:n \cdot C1 \\ C1 &= \bar{a}@(n + 50):n \cdot C1 \\ T &= a:m \cdot s(x)@(m + 2) \cdot \bar{r}(x)@(m + 4) \cdot T . \end{aligned}$$

Agent C represents a clock whose ‘alarm’ action a occurs every 50 time units, starting at time 0. Action \bar{a} in agent $C1$ is defined to occur 50 time units since its causal predecessor, via marker n , and also defines a new value for marker n , which will be used in the *next* iteration. The repeater is then easily implemented as T by just waiting for each alarm, and reading and repeating the value in s . We have assumed that each of these actions takes exactly 2 time units in the actual implementation. (Again we have overspecified the solution, for the sake of brevity.)

At this level of abstraction we have still made a powerful assumption that C is precisely synchronised with the incoming data stream. Indeed, it is interesting to note that C can also be thought of as implementing a ‘data ready’ signal associated with the availability of each character, merely by taking advantage of the associativity of the composition operator, i.e.,

$$((S | C) | T) \setminus \{a, s\} .$$

Thus this abstract model describes the behaviour of two distinct implementations.

5 Discussion

The constraint-oriented approach advocated above cannot directly express notions of ‘priority’ or ‘urgency’, used in some other timed process algebras [3, 6, 4]. Indeed we do not feel that it is appropriate to do so. Being allowed to express a ‘priority’ between two actions carries with it subtle consequences for causality. It can make two otherwise independent actions indirectly dependent on one another [7, 11], in a manner not reflected by the communications pattern of the network. How one would be expected to *implement* a priority order between two geographically-separated actions is very unclear—in the real-time systems community it has been suggested that the concept of priority is flawed and quite meaningless across processor boundaries [18, 5, 15]. Furthermore, it appears that the notions of ‘priority’ and ‘true concurrency’ are so closely related [7] that no single language needs to support both.

Similarly, we have made no attempt to model the concept of assigning probabilities to nondeterministic choices [14, 20] because we see this as a concept entirely orthogonal to that of ‘time’.

The main disadvantage with our definitions is the clumsy algebraic laws for ‘absolutely-timed’ τ actions (see Section 3.1.3). Indeed, it is telling to note that Toetenel, when defining his own real-time CCS-based process algebra, states that “there is no syntactic τ action” but only that “an internal action with *very similar* semantics is specifiable [emphasis added]” [19]. Similarly, Aceto and Murphy [1] provide a new operator ‘WAIT’ to achieve the effect of a timed τ , where this new operator is defined using *relative*, rather than absolute, timing. The problem seems to be a universal one.

Although we could also have defined such a new operator ourselves we have not done so because adding new real-time operators increases the complexity of the semantics and can have subtle, unanticipated side-effects. For instance, when a ‘delay’ operator is added to an algebra like LOTOS [16], we note that it can interact with other operators in unfortunate ways. If such a delay is used as the second operand to the LOTOS ‘disabling’ operator then the combined behaviour no longer enjoys the simple recursive definition normally possible by ‘unwinding’ the disabling operator as a choice between the initial actions of its two operands [2], because the remaining delay period must change each time an ‘uninterrupted’ action occurs. In effect, the ‘delay’ operator changes the semantics of surrounding operators!

6 Conclusion

We have defined, and demonstrated the application of, a new real-time process algebra, based on CCS. It offers the advantages of simple semantics, but still has good expressive power. This was achieved by combining the best features of two previous streams of research, the ‘constraint-oriented’ and ‘marker variable’ approaches.

References

- [1] L. Aceto and D. Murphy. On the ill-timed but well-caused. In E. Best, editor, *Concur'93*, volume 715 of *Lecture Notes in Computer Science*, pages 97–111. Springer-Verlag, 1993.
- [2] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, 1987.
- [3] T. Bolognesi and F. Lucidi. LOTOS-like process algebras with urgent or timed interactions. In K. Parker and G. Rose, editors, *Formal Description Techniques, IV*, pages 249–264. North-Holland, 1992.
- [4] P. Brémont-Grégoire, S. Davidson, and I. Lee. CCSR 92: Calculus for communicating shared resources with dynamic priorities. In S. Purushothaman and A. Zwarico, editors, *NAPAW 92: Proc. First North American Process Algebra Workshop*, 1992.
- [5] A. Burns and A. J. Wellings. The notion of priority in real-time programming languages. *Computer Languages*, 15(3):153–162, 1990.
- [6] J. Camilleri. A conditional operator for CCS. In J. Baeten and J. Groote, editors, *Concur '91*, volume 527 of *Lecture Notes in Computer Science*, pages 142–156. Springer-Verlag, 1991.
- [7] P. Degano, R. Gorrieri, and S. Vigna. On relating some models for concurrency. In M.-C. Gaudel and J.-P. Jouannaud, editors, *TAPSOFT'93: Theory and Practice of Software Development*, volume 668 of *Lecture Notes in Computer Science*, pages 15–30. Springer-Verlag, 1993.
- [8] M. Faci, L. Logrippo, and B. Stepien. Formal specification of telephone systems in LOTOS: the constraint-oriented style approach. *Computer Networks and ISDN Systems*, 21(1):53–67, 1991.
- [9] C. J. Fidge. Process algebra traces augmented with causal relationships. In K. Parker and G. Rose, editors, *Formal Description Techniques IV (FORTE'91)*, pages 527–541. North-Holland, 1992.
- [10] C. J. Fidge. A constraint-oriented real-time process calculus. In M. Diaz and R. Groz, editors, *Formal Description Techniques V (FORTE'92)*, pages 363–378. North-Holland, 1993.
- [11] C. J. Fidge. A formal definition of priority in CSP. *ACM Transactions on Programming Languages and Systems*, 15(4):681–705, September 1993.
- [12] C. J. Fidge. Fundamentals of distributed system observation. *IEEE Software*, 13(6):77–83, November 1996.
- [13] J. Godskesen and K. Larsen. Real-time calculi and expansion theorems. In R. Shyamasundar, editor, *Foundations of Software Technology and Theoretical Computer Science*, volume 652 of *Lecture Notes in Computer Science*, pages 302–315. Springer-Verlag, 1992.
- [14] H. Hansson. Modelling timeouts and unreliable media with a timed probabilistic calculus. In K. Parker and G. Rose, editors, *Formal Description Techniques, IV*, pages 67–82. North-Holland, 1992.
- [15] H. Hansson and F. Orava. A process calculus with incomparable priorities. In S. Purushothaman and A. Zwarico, editors, *NAPAW 92: Proc. First North American Process Algebra Workshop*, 1992.

- [16] G. Leduc and L. Léonard. A timed LOTOS supporting a dense time domain and including new timed operators. In M. Diaz and R. Groz, editors, *Formal Description Techniques, V*, pages 87–102. North-Holland, 1992.
- [17] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
- [18] M. J. Pilling. Dangers of priority as a structuring principle for real-time languages. *Australian Computer Science Communications*, 13(1):18–1–18–10, February 1991.
- [19] H. Toetenel. Loose real-time communicating agents. In D. Andrews, J. Groote, and C. Middelburg, editors, *Semantics of Specification Languages*, pages 135–151. Springer-Verlag, 1993.
- [20] J. Žic. Some thoughts on communication system performance specification. In D. B. Hoang and E. Chew, editors, *Proc. Open Distributed Processing Workshop*, Sydney, January 1990.
- [21] J. Žic. *CSP+T: A Formalism for Describing Real-Time Systems*. PhD thesis, Basser Department of Computer Science, University of Sydney, July 1991.
- [22] J. Žic. Specifying a time constrained buffer in Timed CSP and CSP+T. *ACM Transactions on Programming Languages and Systems*, November 1994.