# COMPOSITIONAL SEMANTICS FOR REAL-TIME DISTRIBUTED COMPUTING

R.K. Shyamasundar
W.P. de Roever
R. Gerth
R. Koymans
S. Arun-Kumar

RUU-CS-84-6
August 1984

# COMPOSITIONAL SEMANTICS FOR REAL-TIME DISTRIBUTED COMPUTING

R.K. Shyamasundar
W.P. de Roever
R. Gerth
R. Koymans
S. Arun-Kumar

Technical Report RUU-CS-84-6
August 1984

Department of Computer Science
University of Utrecht
P.O. Box 80.012, 3508 TA Utrecht
the Netherlands

# COMPOSITIONAL SEMANTICS FOR REAL-TIME DISTRIBUTED COMPUTING

R.K. Shyamasundar[1,4]
W.P. de Roever[2,3]
R. Gerth[2,5]
R. Koymans[3,5]
S. Arun-Kumar[1]

december 22, 1984

ABSTRACT.
We give a compositional denotational semantics for a real-time distributed language, based on the linear history semantics for CSP of Francez et al. Concurrent execution is not modelled by interleaving but by an extension of the maximal parallelism model of Salwicki, that allows the modelling of transmission time for communications. The importance of constructing a semantics (and in general a proof theory) for real-time is stressed by such different sources as the problem of formalizing the real-time aspects of Ada and the elimination of errors in real-time flight control software ( [Sunday Times 7-22-84]).

## 1. INTRODUCTION.

Although concurrency in programming has been seriously investigated for more than 25 years ([Dij59]), the specific problems of real-time have been the object of little theoretical reflection. Currently used real-time languages represent almost no evolution w.r.t. assembly language[Cam82]. Consequently no serious analysis of complexity, no design methodology, no standard for implementation and no concept

---

[1]NCSDCT, Tata Institute for Fundamental Research, Homi Bhaba Road, Bombay - 400 005, India.
[2]Department of Computer Science, University of Utrecht, P.O. Box 80.012, 3508 TA Utrecht, the Netherlands.
[3]Department of Computer Science, University of Nijmegen, Toernooiveld, 6525 ED Nijmegen, the Netherlands.
[4]supported by a visitors grant from the Netherlands Organization for the Advancement of Pure Research (ZWO).
[5]supported by the Foundation for Computer Science Research in the Netherlands (SION) with financial aid from the Netherlands Organization for the Advancement of Pure Research (ZWO).

of portability exist for real-time languages. This state of affairs is astonishing, for the confrontation with real-time lends a nervous twitch of actuality for arguments in favour of formalization; don't the dangers of malfunctioning real-time systems affect all of us?

Errors occurring in Boing 747's real-time flight control are a closely guarded secret [ST84], yet most of us fly in such planes. Software for space vehicles and nuclear power stations belong to the world's most prestiguous projects, yet remain notoreously unreliable ( [ACM84], The Three-Miles Island Disaster), with no prospect of improvement in the immediate future. Every industrialized country has computer controlled chemical plants in densely populated areas.

But even commercial interest points to a need for real-time systems developed as a hierarchy of modules. Recently, one of the world's leading manufacturers (Philip's Telecommunicating Industries) has had to cancel the developments of a digitized telephone-switching network in a late stage of development because its design lacked the transparancy required for adaptation to local circumstances as occurring in, e.g., Asia. And what to think of the advent of industrial robots?

The responce to this need, has been the development of new real-time languages such as (1) Ada - developed for the militatry, (2) CHILL - within the context of telecommunication industries and (3) Occam - which is even chip-implemented, for those interested in experimenting with structure. All of these are claimed to have been rigourously defined ([A 83],[Bjø80],[CHI82],[Occ84]). Yet their official standards lack any acceptable characterization of concurrency (with the exception of Occam), let alone of real-time (, which is lacking for Occam, too).

All these arguments emphasize the need to develop formal models for real-time concurrency, and, more importantly, to discover structuring methods which lead to hierarchical and modular development of real-time concurrent systems. Obviously, models based on interleaving, such as [BH82] can be immediately discarded as being unrealistic. Models based on SCCS [Mil81] such as [Ber84] are an improvement in that truly concurrent activity has been modelled, yet are unsatisfactory in that interleaving has not been excluded. Petri-net theory remains a viable direction for discovering structuring methods, yet is still unsatisfactory because it has not incorporated (1) satisfactory verification methods for liveness properties, such as temporal logic has, or (2) (machine checkable) formalisms for representing (concurrently implemented) data structures. And certainly none of these models apply to real-time features of realistic programming languages such as Ada.

The present paper aims at providing a model of real-time concurrency

- which is realistic in that concurrent actions can and will overlap in time unless prohibited by synchronization constraints, no unrealistic waiting of processors is modelled, and yet the many parameters involved in real-time behaviour are reflected by a corresponding parametrization of our models; it is based on Salwicki's notion of maximal concurrency [SM81];
- which applies to programming languages for distributed computing such as Ada and Occam which are based on synchronized communication (, for asynchronized communication as in CHILL, we developed [Koy83]);
- which implies a sound and relatively complete method for verification since it is compositional; we base ourselves in this respect on the method developed by Soundararajan [Sou83, Sou84], and joint research together with Pnueli leading to the incorporation of maximal parallellism within the temporal framework of [BKP82] ;
- which meets the standard of rigour as provided by the denotational semantics of concurrency.

Some of these aspects are also covered by work of Zijlstra [Z84]and G. Jones [J82].

Our study of real-time distributed computing is carried by a real-time variant of CSP, called CSP-R, which allows modelling of the essential Ada ([A83]) real-time features. A denotational semantics for CSP-R is given, stressing compositionality and extending the linear history semantics for CSP of [FLP80]:

- the basic domain consists of *prefix-closed sets* of pairs of states and (finite) histories of communication assumptions leading to that state,
- the ordering on this domain is simply set-inclusion,
- the denotation for the parallel execution of two processes yields a denotation *in the same domain* for a new combined process replacing the original two,
- the histories contain enough information to detect deadlock, eliminating the expectation states of [FLP80],
- process-identification parameters must be used throughout the semantics for the compositional modelling of *nested* concurrency.

Histories are modelled as sequences of *bags* of communication assumption records as we allow truly concurrent actions (; why bags are needed instead of sets is explained in §6). Real-time is modelled in the histories by relating the i-th element of a history with the i-th tick of a *conceptual global clock* (see §4).

There are two kinds of records for expressing communication assumptions in the histories:

- communication record $C(i,j,c,\vec{v})$,
  modelling the execution of an I/O command: the values $\vec{v}$ are passed from process i (the sender) to process j (the receiver) via channel c,
- no-match record $NM(i,\alpha)$,
  modelling the absence of possibility for the execution of the I/O command $\alpha$ in process i. This means that there is no matching I/O command $\bar{\alpha}$ such that $\alpha$ and $\bar{\alpha}$ can be executed simultaneously. ( used to enforce no unrealistic waiting )

Internal moves within a process are modelled by empty bags.

The no-match record is new and allows

- the checking of the maximal parallelism constraints; i.e., no unnecessary waiting,
- the detection of deadlock (§5), rendering expectation states as in [FLP80] unnecessary.

Related ideas can be found in [Z84] and the unpublished [J82].

## 2. CSP-R.

CSP-R is based on CSP [H78] (with which the reader is assumed to be familiar), which is adapted to model the real-time and communication features of Ada [A83]. The two most important differences between CSP and CSP-R are:

- in CSP-R communication takes place along channels, and
- real-time is incorporated in CSP-R by adding a '<u>wait</u> d' instruction (d is an integer expression) which may also be used as guard in a conditional or loop.

A full description can be found in the appendix. Here we limit ourselves to some examples of the novel features.

1. $[\underline{\text{wait}}\ 8 \rightarrow S_1 \ \square\ P_3.c!5 \rightarrow S_2]$, a guarded selection with intended meaning:
   if communication with process $P_3$ via channel c is possible *within* 8 time units, transmit 5 and execute $S_2$, otherwise, after 8 time units, execute $S_1$.
2. $[x>0 \rightarrow S_1 \ \square\ .c_1?y \rightarrow S_2 \ \square\ P_7.c_2?z \rightarrow S_3]$, a guarded selection with intended meaning:
   if x>0 evaluates to true, execute $S_1$; otherwise wait for communication with *any* process via $c_1$ or with process $P_7$ via $c_2$.

3. $[P{::}\ [P_1{::}\ P_2.c_1?x;Q.c_2!0\ ||\ P_2{::}\ P_1.c_1!0]\ ||\ Q{::}\ [Q_1{::}\ P.c_2?z\ ||\ Q_2{::}\ \underline{\text{wait}}\ 0]]$, a parallel command displaying nested parallelism. Notice that the *scope* of a process-name extends to the *smallest* embracing parallel command in which it is declared; observe the similarity of this with the scope-rules for blocks.

## 3. <u>THE MAXIMAL PARALLELISM MODEL.</u>

Under maximal parallelism, the number of instructions in concurrently executing processes that can be executed simultaneously without violating synchronization requirements, is maximalized (see [SM81] for a formal definition). So, in the program $[P_1{::}\ x{:=}1\ ||\ P_2{::}\ x{:=}1\ ||\ P_3{::}\ y{:=}2]$ either $P_1$ and $P_3$ or $P_2$ and $P_3$ will execute their first move simultaneously, but not $P_1$ and $P_2$; all this, under the assumption that multiple accesses to a single (shared) variable are mutually exclusive.

Implementing maximal parallelism requires separate processors for the various processes. The connection with real-time behaviour is, that when execution speed is a critical factor, separate processors should be available to all processes.

For distributed computing, maximal parallelism means "first-come first-served" (fcfs) in some global time scale (see §4). Consider the program

$[P_1{::}\ \ell_1{:}P_4.c!0\ ||\ P_2{::}\ \ell_2{:}P_3.c!1\ ||\ P_3{::}\ \ell_3^1{:}.c?x;\ell_3^2{:}P_4.c!x\ ||\ P_4{::}\ \ell_4^1{:}.c?y;\ell_4^2{:}.c?y]$

($\ell_1,\ldots,\ell_4^2$ denote labels). According to *interleaving* semantics, there are two sets of rendezvous' possible: (1) $\{\ell_2\text{-}\ell_3^1,\ \ell_1\text{-}\ell_4^1,\ \ell_3^2\text{-}\ell_4^2\}$, and (2) $\{\ell_2\text{-}\ell_3^1,\ \ell_3^2\text{-}\ell_4^1,\ \ell_1\text{-}\ell_4^2\}$. According to *maximal parallelism* semantics, only (1) is possible since $P_1$ and $P_4$ can *immediately* become engaged in a rendezvous.

As we will reason in §7, the maximal parallelism model is unrealistic for distributed systems in general. We will develop a whole family of real-time models ranging from interleaving to maximal parallelism semantics and incorporating the transmission time for messages in a system.

## 4. OUR VIEW OF TIME.

To express real-time properties such as "the system responds on a certain request within a fixed number of seconds" there must be some measure of time to relate these properties to. When we talk about abstract, i.e., implementation independent, properties of a system *as a whole*, this measure must be relative to some *global* time scale. For distributed systems this means that all events in the various processes are related to each other by means of one conceptual global clock, introduced at a metalevel of reasoning.

Clearly, no physical realization of such a global clock is possible; processors always drift from one time mutual synchronization as exemplified by the existence of clock synchronization algorithms. In our model, drifting can always be modelled by allowing (small) unpredictable variations in the execution time of basic actions.

## 5. THE SEMANTIC DOMAIN AND ITS INTERPRETATION.

Our basic domain is that of *prefix-closed* sets of pairs, $<s,h>$, of states s and histories h of bags of communication assumption records.

Definition. A set X of state-history pairs is *prefix-closed* iff for all $<s,h>\in X$, if $h' \lhd h$ (i.e., h' is a prefix of h) then $<\perp,h'>\in X$, where $\perp$ signals an incomplete computation.

The collection of non-empty prefix-closed sets becomes a complete lattice if:

- the partial ordering is *set inclusion* $\subseteq$, and

- the least upper bound is obtained by *set-theoretic union* $\cup$.

The least element of this lattice is $\{<\perp,\lambda>\}$, where $\lambda$ is the empty history.

Sets of state-history pairs, X, describe as usual (see [FLP80]) the computations of a program P. So, $<s,h>\in X$ models a computation of P with history h that terminates in s if $s\neq\perp$ , and a *partial* (i.e., not yet completed) computation otherwise. If X contains pairs $<\perp,h_0>$, $<\perp,h_0\hat{}h_1>$, ... such that each $h_i\neq\lambda$ (^ is the concatenation operator) then P has an infinite computation with history $h_0\hat{}h_1\hat{}\ldots$ .

Histories are interpreted elementwise as follows: Let h(k) denote the k-th element of a history h. Then

- h(k)=[ ], i.e., the empty bag, models an internal move at time k,
- $C(i,j,c,\vec{v})\in h(k)$ models the passing of the values $\vec{v}$ from process i to j via channel c at time k, and
- $NM(i,\alpha)\in h(k)$ models the absence of a matching I/O command $\bar{\alpha}$ for the I/O command $\alpha$ in process i at time k.

Deadlock is detected as follows: Let $B_0$, $B_1$, ... be bags of communication assumption records, each $B_k$ containing a no-match record $NM(i,\alpha)$ for some I/O command $\alpha$. If there exists a history h such that $<\perp,h\hat{}B_0\hat{}\ldots\hat{}B_k>\in X$ for every $k\geq 0$, then X contains an infinite computation in which process i is blocked on $\alpha$; i.e., X contains a deadlock situation for P.

## 6. MAXIMAL PARALLELISM SEMANTICS FOR CSP-R.

The meaning of CSP-R constructs is defined *denotationally* by giving, for all processes i and constructs T within i, an equation for $M[T]_i$ which relates the meaning of T to the meaning of T's constituents (; the identity of i is required to model nested parallelism).

Since loops occur in CSP-R, $M[T]_i$ should be a *continuous* functional so that it makes sense to apply the least fixed point operator, $\mu$. In the full paper we show that $M[T]_i$ belongs to the cpo CSSM of <u>c</u>ontinuous <u>s</u>trict <u>s</u>equential <u>m</u>appings. CSSM is a certain subset (with induced ordering) of the cpo of strict and continuous automorphisms of the lattice of non-empty prefix closed sets of state-history pairs. We note that the least element of CSSM is the functional $\perp_{CSSM} = \lambda X.\ PFC(\{<\perp,h> \mid <s,h>\in X\})$; the operator PFC maps sets of state-history pairs to their prefix closure.

We proceed with the definition of $\mathcal{M}[\cdot]_i$. The three auxiliary functions, $\mathcal{G}$, $||$ and $\text{Hide}_i$ that are used, will be defined afterwards.

<u>Definition</u> of $\mathcal{M}[T]_i$.

Induction to syntactic complexity.

$$\mathcal{M}[x{:=}e]_i = \lambda X.\ \bot_{CSSM}(X) \cup \bigcup_{\substack{<s,h>\in X\\ s\neq\bot}} \text{PFC}(\{<s[\mathcal{V}[e]s/x],\ h\hat{}[\,]>\}),$$

where $\mathcal{V}[e]s$ denotes the value of e in state s.

$\mathcal{M}[g]_i = \mathcal{G}[g,\emptyset]_i$ for guards g acting as statements (see the appendix).

$\mathcal{M}[T_1;T_2]_i = \mathcal{M}[T_2]_i \circ \mathcal{M}[T_1]_i$, where o denotes functional composition.

$$\mathcal{M}[\mathop{\square}_{j=1}^{n} g_j \rightarrow T_j]_i = \lambda X.\ \bigcup_{j=1..n} \mathcal{M}[T_j]_i(\mathcal{G}[g_j,\{g_k \mid 1\leq k\leq n,\ k\neq j\}]_i X)$$

$$\mathcal{M}[*\mathop{\square}_{j=1}^{n} g_j \rightarrow T_j]_i = \mu(\ \lambda\phi.\ \lambda X.\ \bigcup_{<s,h>\in X} \underline{\text{if}}\ \bigvee_{j=1..n} \mathcal{B}[\bar{g}_j]s\ \underline{\text{then}}\ \phi(\mathcal{M}[\mathop{\square}_{j=1}^{n} g_j \rightarrow T_j]_i \text{PFC}(\{<s,h>\}))$$
$$\underline{\text{else}}\ \{<s,h>\}\ \underline{\text{fi}}\ ),$$

where $\mu$ is the least fixed point operator and

$\mathcal{B}[\bar{g}_j]s$ denotes the value of the boolean part of guard $g_j$ in state s.

$$\mathcal{M}[[P_{i1}::T_1||\ldots||P_{in}::T_n]]_i = \lambda X.\ \text{Hide}_i(<\{i1\},\mathcal{M}[T_1]_{i1}X>||\ldots||<\{in\},\mathcal{M}[T_n]_{in}X>),$$

where $\text{Hide}_i$ hides internal communication assumptions and $||$ is the (associative) parallel binding operator.

Next we define the auxiliary functions. For the complete definitions, the reader is referred to the full paper.

<u>Definition</u> of $\mathcal{G}[g,A]_i$, defining the meaning of a single guard g in an environment A of alternative guards, in process i. There is only room for a representative sample:

$\mathcal{G}[b,A]_i X = \bot_{CSSM}(X) \cup \bigcup_{\substack{<s,h>\in X\\ s\neq\bot}} \{<s,h> \mid \mathcal{B}[b]s\}$. A boolean guard acts as a filter.

$$\mathcal{G}[P_j.c!\vec{e},A]_i X = \bot_{CSSM}(X) \cup \bigcup_{\substack{<s,h>\in X\\ s\neq\bot}} \text{PFC}(\{<s,h\hat{}\overbrace{[NM(i,GRDS,s)]\hat{}\ldots\hat{}[NM(i,GRDS,s)]}^{t\ \text{times}}\hat{}$$
$$[C(i,P_j,c,\mathcal{V}[\vec{e}]s)]> \mid i\neq P_j,\ 0\leq t<\text{minwait}(GRDS,s)\}),$$

where $GRDS = A\cup\{P_j.c!\vec{e}\}$ is the set of guards,

NM(i,GRDS,s) contains the no-match records of all I/O guards in GRDS of process i which are open in s (see the appendix),

minwait(GRDS,s) is the minimum of all wait values of the boolean and wait guards in GRDS (see the appendix) that are open in s.

A pure I/O guard is *either* taken - indicated by the communication record - within "waitvalue" time units and this guard is the first to be taken of the open

I/O guards - indicated by the no-match records - *or* it is never taken; this can happen if waitvalue is $\infty$, and models a possible deadlock. Note that no process communicates with itself ($i \neq P_j$).

$$\mathcal{G}[\underline{\text{wait}}\ d,A]_i X = \bot_{CSSM}(X) \cup \bigcup_{\substack{<s,h>\in X \\ s\neq\bot}} PFC(\{<s,h\,\hat{}\,\overbrace{[NM(i,A,s)]\,\hat{}\ldots\hat{}\,[NM(i,A,s)]}^{T\text{ times}}\,\hat{}\,[\,]> \mid \max(\{\mathcal{V}[d]s,0\})=minwait(A\cup\{\underline{\text{wait}}\ d\},s) \overset{D}{=} T\}).$$

A pure wait guard is taken - as indicated by the empty bag - after T time units provided its duration value equals this latter value and no open I/O guard could have been taken at an earlier time - indicated by the no-match records -.

<u>Definition</u> of ||, the parallel binding operator.

$$<I_1,X_1>||<I_2,X_2> = <I_1\cup I_2,\ PFC(\{<s_1+s_2,h_1\#_{I_1,I_2}h_2> \mid <s_i,h_i>\in X_i \text{ and } <I_1,h_1>\text{¢}<I_2,h_2>\})>,$$

where $I_1$ and $I_2$ are disjunct sets of processes,

$s_1+s_2$ is the combined state of $s_1$ and $s_2$ (note that variables in different processes are distinct),

$h_1\#_{I_1,I_2}h_2$ is the pointwise merge of $h_1$ and $h_2$; i.e., $h_1(1)$ is merged with $h_2(1)$ etc. Internal communications between processes in $I_1$ and $I_2$ are replaced by internal moves.

$<I_1,h_1>\text{¢}<I_2,h_2>$ is the consistency check, determining whether for all k (1) the subbags consisting of the communication records between processes in $I_1$ and $I_2$ of $h_1(k)$ and $h_2(k)$ are equal and (2) $h_1(k)$ and $h_2(k)$ do not contain no-match records for matching I/O commands.

A simple example illustrates the basic ideas:

Consider a program $P$, $[P_1::P_2.c?y \mid\mid P_2::x:=1;P_1.c!x]$. Starting with the empty history, the histories of $P_1$ are $[C(P_2,P_1,c,v)]$, $[NM(P_1,P_2.c?y)]\hat{}[C(P_2,P_1,c,v)]$, $[NM(P_1,P_2.c?y)]\hat{}[NM(P_1,P_2.c?y)]\hat{}[C(P_2,P_1,c,v)]$, ... for any value v, together with their prefixes. Similarly, the histories of $P_2$ are $[\,]\hat{}[C(P_2,P_1,c,1)]$, $[\,]\hat{}[NM(P_2,P_1.c!x)]\hat{}[C(P_2,P_1,c,1)]$, $[\,]\hat{}[NM(P_2,P_1.c!x)]\hat{}[NM(P_2,P_1.c!x)]\hat{}[C(P_2,P_1,c,1)]$, ... again, together with all their prefixes.

For histories $h_1$ of $P_1$ and $h_2$ of $P_2$ to pass the consistency check, we should have

- $C(P_2,P_1,c,v)\in h_1(k)$ iff $C(P_2,P_1,c,1)\in h_2(k)$ and $v=1$, and
- not $NM(P_1,P_2.c?y)\in h_1(k)$ if $NM(P_2,P_1.c!x)\in h_2(k)$ and vice versa.

Consequently the only pair of consistent histories is $[NM(P_1,P_2.c?y)]\hat{}[C(P_2,P_1,c,1)]$ of $P_1$ and $[\,]\hat{}[C(P_2,P_1,c,1)]$ of $P_2$. The resulting set of histories of the combined process $P$ is $[NM(P_1,P_2.c?y)]\hat{}[\,]$ , which corresponds with what we should have expected, considering that $P$ executes under maximal parallelism.

Definition of $\text{Hide}_i$.

$\text{Hide}_i(I,Y) = Y'$, where $Y'$ is obtained from $Y$ by changing it(s histories) as follows:

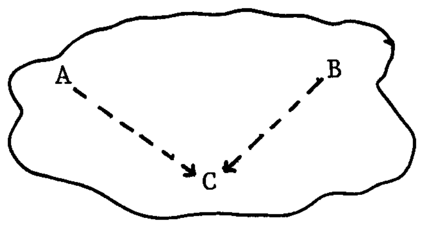
- no-match records for I/O commands that concern processes in I only, are removed,
- all process identifiers $p \in I$ in the remaining records are replaced by i.

Observe that possibilities for deadlock remain visible as infinite computations, now represented by $<\perp, h\hat{}[\,]\hat{}[\,]\hat{}\ldots>$. Consequently, *we do not distinguish between deadlock and divergence (as internal events)*.

At this point, the use of bags instead of sets becomes essential: Consider the program $P$, $[P::[P_1::Q.c!0 \;||\; P_2::Q.c!0] \;||\; Q::[Q_1::P.c?x \;||\; Q_2::y:=0]]$. $P$ will block when executed, since both $P_1$ and $P_2$ want to communicate with Q, but Q can offer only *one* such communication. After hiding, we cannot distinguish the two output commands in P anymore. So, had we used sets instead of bags, hiding would have resulted in histories with only *one* communication record. Thus, blocking would not have been detected in such cases.

## 7. REAL-TIME MODELS.

The following example illustrates the conceptual problems associated with applying (pure) maximal parallelism to communication based systems, motivating the introduction of a whole family of real-time models in which the maximal parallelism model is incorporated.

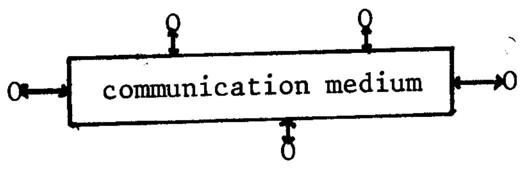


Consider a network with distributed control, and two processes A and B in different nodes that want to communicate with a process C in a third node. If A wants to communicate at an earlier time than B, relative to some global time scale, then according to the fcfs-principle, indeed, A should communicate first. Whether A's message *arrives* in C before B's message or not, depends on the topology of the network. So, it makes no sense to impose fcfs unless many more details are known about the network. Similar problems occur if processors communicate, e.g., via a common bus where assumptions about bus-arbitration have to be taken into account.

This analysis shows that the fcfs-principle is applied to the order of

initiation of communication requests while the above example shows that the time at which communication requests are noticed is relevant. This time gap between initiation and receipt is the essential feature of the $\mathrm{MAX}_{\gamma}(\delta,\varepsilon)$ modelsoof distributed concurrency. As we want to abstract from the properties of communication media, we introduce an uncertainty, bounded by $\delta$ and $\varepsilon$, in the length of such time gaps. As a consequence, communications that are initiated too close in time (relative to a global clock) cannot be temporally ordered anymore.

The $\mathrm{Max}_{\gamma}(\delta,\varepsilon)$ model is based on the following conceptual model of communication:



- processes communicate via a medium,
- it takes between $\delta$ and $\varepsilon$ time units ($\varepsilon$ not included) for the medium to become aware of a process expressing or withdrawing its willingness to communicate,
- communication between two processes only occurs after the medium has become aware of both processes' willingness,
- communication takes an additional $\gamma$ time units during which the participating processes remain synchronized,
- any communication that is in progress when the medium becomes aware of a withdraw message of one of the participating processes, will be completed normally. No new communication can be started at such a time.

Consequently, there is an uncertainty interval of $\varepsilon-\delta-1$ time units. Communications that result from requests initiated within this interval may occur in any order, i.e., are interleaved. Communications resulting from requests initiated more than $\varepsilon-\delta-1$ time units apart occur on a fcfs basis. Note that $\mathrm{MAX}_0(0,1)$ corresponds to pure maximal parallelism while $\mathrm{MAX}_0(0,\infty)$ results in pure interleaving of the communication actions.

The semantics for CSP-R using the $\mathrm{MAX}_{\gamma}(\delta,\varepsilon)$ model for arbitrary $\gamma,\delta,\varepsilon$ differs from the one in §6 only in the semantics of guards. In particular, binding and the consistency check remain unaltered. This is so because we can describe the time gap between initiation and receipt of a communication request in the semantics of the guards by adding parameters $\tau$, $\delta \leq \tau < \varepsilon$, modelling the delay of the start of a possible communication. This will be done in the full paper.

## ACKNOWLEDGEMENTS.

Our thanks goes to Amir Pnueli, who, during a joint visit of Amir and Willem to the Department of Computer Science of the University of Manchester upon invitation by Cliff Jones and supported by ICL funds, assisted in a last overhaul of the paper. We also thank Job Zwiers for many stimulating discussions.

## REFERENCES.

[A83] The programming language Ada. Reference manual. LNCS 155, Springer, 1983

[FLP80] Francez N.,D. Lehmann, A. Pnueli. A linear history semantics for distributed languages. Proc. IEEE FOCS, 1980.

[H78] Hoare C.A.R. Communicating sequential processes. CACM 21-8,1978.

[J82] Jones, G. D. Phil. thesis, Oxford, unpublished, 1982.

[SM81] Salwicki A., T. Müldner. On the algorithmic properties of concurrent programs, LNCS 125, Springer, 1981.

[Z84] Zijlstra E. Real time semantics. submitted, 1984.

[Dij59 Dijkstra E.W. Communication with an automatic computer. Ph.D. thesis, Mathematical Centre, Amsterdam, 1959.

[ACM84] A Case Study: The Space Shuttle Software System. CACM27-9, 1984.

[Ber84] Berry G., L. Cosserat. The ESTEREL Synchroneous Programming Language and its Mathematical Semantics. RdeR N$^{o}$327, INRIA, Centre Sophia Antipolis, 1984.

[BH81] Bernstein A., P.K. Harter jr. Proving Real-time Properties of Programs with Temporal Logic. 8th ACM SOSP, p1-11, 1981.

[Bjø80] Bjørner D., O.N. Oest (eds.). Towards a Formal Description of Ada. LNCS 98, Springer, 1980.

[Cam82] Camerini J. Semantique Mathematique de Primitives Temps Reel. These de 3eme cycle, IMA, Université de Nice, 1982.

[CHI82] Branquart P., G. Louis, P. Wodon. An Analytical Description of CHILL, the CCITT High Level Language VI, LNCS128, Springer, 1982.

[Koy83] Koymans R., J. Vytopil, W.P. de Roever. Real-Time Programming and Asynchroneous Message Passing. Proc. 2nd ACM PODC, 1983.

[Mil83] Milner R. Calculi for Synchrony and Asynchrony, TCS25, p267-310, 1983.

[Occ84] The Occam language reference manual. Prentice Hall, 1984.

[Sou83] Soundararajan N. Correctness Proofs for CSP Porograms, TCS23-4, 1984.

[Sou84 Soundararajan N. Axiomatic Semantics of Communicating Sequential Processes. TOPLAS6-4, p647-662, 1984.

ST84 Sunday Times 22/7.

**APPENDIX:** THE DEFINITION OF CSP-R.

CSP-R is based on CSP [H78] with the following extensions:

- communication takes place via channels,
- there is the additional possibility of requesting communication with an arbitrary process instead of only addressing a particular process,
- process identifiers can be communicated and can be used in subsequent communications to determine the target process, and
- real-time is incorporated by adding the instruction <u>wait</u> d, which can also be used as guard.

Before giving the syntax of CSP-R, we introduce some notational conventions.
A *process identification* is an element from the identification set $P_0, P_1, P_2, \ldots$ .
An *identification variable* is a variable ranging over the identification set.
A *duration* is an integer-valued expression.

The primitive language elements are the *instructions*:

1. x:=e — assignment; x should <u>not</u> be an identification variable.
2. <u>wait</u> d — wait instruction; d is duration.

3.1. $P_i.c!\vec{e}$ — output to process i via channel c the values of the expressions in the list $\vec{e}$, together with the identification of the sending process.

3.2. $id.c!\vec{e}$ — as 3.1., but now the target process is determined by the value of the identification variable id.

3.3. $.c!\vec{e}$[#id] — output via channel c to <u>any</u> process the values of the expressions in the list $\vec{e}$, together with the identification of the sender; record the identity of the receiving process in the identification variable id (the brackets [ and ] indicate that the identification variable is optional; i.e., $.c!\vec{e}$ is allowed, too).

4.1. $P_i.c?\vec{x}$ — the analogon of 3.1. but now values are received and are assigned to the variables in the list $\vec{x}$.

4.2. $id.c?\vec{x}$ — the analogon of 3.2.

4.3. $.c?\vec{x}$[#id] — the analogon of 3.3.

An identification variable can only be assigned to using an instruction of the form 3.3 or 4.3. Initially, such a variable has as value the identity of the process in which it occurs. In general, the variables of any two processes must be distinct. In other words, variables are local to a process and the only interaction between processes is by explicit communication.

We call instructions of the form 3.1-3.3, 4.1-4.3 *I/O commands*: the ones of the form 3.1-3.3 are *output* commands; the ones of the form 4.1-4.3, *input* commands.

The important notion of *syntactic matching* of 2 I/O commands is defined as follows:

Definition: Two I/O commands $\alpha$ in process $P_i$ and $\beta$ in process $P_j$ syntactically match iff:

1. $\alpha$ and $\beta$ specify the same channel,
2. the lists of expressions and variables of the two commands, have equal length,
3. if $\alpha$ is an input command, then $\beta$ should be an output command and vice versa, and
4. if $\alpha(\beta)$ is of the form 3.1 or 4.1, then the specified target process should be $P_j$ $(P_i)$.

Communication between processes $P_i$ and $P_j$ takes place when an I/O command $\alpha$ in $P_i$ *semantically matches* with an I/O command $\beta$ in $P_j$:

Definition: Two I/O commands $\alpha$ and $\beta$ in processes $P_i$ and $P_j$ semantically match iff:

1. $\alpha$ and $\beta$ match syntactically,
2. control in $P_i$ and $P_j$ is in front of both $\alpha$ and $\beta$, and
3. if $\alpha$ $(\beta)$ is of the form 3.2 or 4.2, then the identification variable must have the value $P_j$ $(P_i)$.

The result of two semantically matching I/O commands is the simultaneous execution of those commands as indicated by the clauses 3.1-4.3. Its effect is the assignment of the expression values to the variables in the commands and, possibly, the assignment to identification variables.

As *guards* we allow the following types:

| | |
|---|---|
| 1. b | - pure boolean guard, |
| 2. $\alpha$ | - pure I/O guard, |
| 3. b;$\alpha$ | - boolean I/O guard, |
| 4. <u>wait</u> d | - pure wait guard, and |
| 5. b;<u>wait</u> d | - boolean wait guard. |

Here, b is a boolean expression (such as x<0), $\alpha$ is an I/O command and d is a duration.

For a guard g, we define its boolean part $\bar{g}$ as follows: $\bar{b} \overset{D}{=} b$, $\bar{\alpha} \overset{D}{=} \text{true}$, $\overline{b;\alpha} \overset{D}{=} b$, $\overline{\underline{\text{wait}}\ d} \overset{D}{=} \text{true}$ and $\overline{b;\underline{\text{wait}}\ d} \overset{D}{=} b$. We say that a guard g is *open* if $\bar{g}$ evaluates to true.

To complete the definition of CSP-R, we define *commands* as the smallest set s.t.:

1. every instruction is a command,
2. if $T_1$ and $T_2$ are commands then $T_1;T_2$ is a command (sequential composition),
3. if $T_1, \dots, T_n$ are commands and $g_1, \dots, g_n$ are guards ($n \geq 1$), then $[\square_{j=1}^{n} g_j \to T_j]$ is an (alternative) command and $*[\square_{j=1}^{n} g_j \to T_j]$ is a (repetitive) command,
4. if $T_1, \dots, T_n$ are commands and i1, ..., in are different integers ($n \geq 1$), then $[P_{i1}::T_1 \parallel \dots \parallel P_{in}::T_n]$ is a (parallel) command. $P_{ik}$ ($1 \leq k \leq n$) is called a *process* with *body* $T_k$.

The informal interpretation of the language is as follows:

- Assignments have their usual interpretation. Recall the condition that variables are local to a process.
- The wait-instruction, wait d, suspends the execution of the process in which it occurs for the value of d (zero if this value is negative) time units.
- The interpretation of I/O commands has already been given above.
- The execution of $T_1;T_2$ is the execution of $T_1$ followed by the execution of $T_2$.
- Execution of $[\square_{j=1}^{n} g_j \to T_j]$ proceeds as follows:
  - first calculate the minimum of the duration values of the open wait guards; if there are no open wait guards, this minimum is $\infty$ (infinite). Now the waitvalue is defined as 0 if an open pure boolean guard is present, and as the minimum computed above otherwise.
  - if there are no open guards, execution is aborted.
  - if there is at least one open guard, one of these is chosen and the corresponding $T_j$ is executed. Choice of an open guard is done as follows: If within waitvalue time units a communication is offered that matches with one in an open I/O guard, this guard is taken. If no matching communication occurs for any open I/O guard then after waitvalue time units one of the open wait guards with duration equal to the waitvalue or one of the pure boolean guards is chosen. Note that if waitvalue equals $\infty$, we have a deadlock in the last case.
- Execution of $*[\square_{j=1}^{n} g_j \to T_j]$ consists of the repeated excution of $[\square_{j=1}^{n} g_j \to T_j]$; this repetition terminates when none of the guards are open.
- Execution of $[P_{i1}::T_1 \parallel \dots \parallel P_{in}::T_n]$ involves the parallel execution of the bodies $T_k$ of the processes $P_{ik}$.

## Translating Ada into CSP-R.

To illustrate the power of CSP-R we translate the basic Ada communication primitives into CSP-R. This translation is denoted by $\tau$. The Ada rendezvous is assumed to be understood.

1. the timed entry call ([A83,§9.7.3]).

   $\underline{\text{select}}\ \underline{\text{call}}\ T_i.a(\vec{e},\vec{x});\ S_1\ \underline{\text{or}}\ \underline{\text{delay}}\ t;\ S_2\ \underline{\text{endselect}}.$

   The semantics of this statement prescribes that if a rensezvous can be started within the specified duration t (or immediately), then it is performed and $S_1$ is executed afterwards. Otherwise, when the duration has expired, $S_2$ is executed.

   We offer as translation:

   $[T_i.a!(\vec{e},\vec{x}) \to T_i.a?\vec{x};\ \tau(S_1)\ \square\ \underline{\text{wait}}(t+1) \to \tau(S_2)].$

2. the selective wait (without terminate alternative) ([A83,§9.7.1]).

   $\underline{\text{select}}\ \underline{\text{or}}\ (i=1..n)\ \underline{\text{when}}\ b_i \Rightarrow S_i\ \underline{\text{or}}\ (j=1..m)\ \underline{\text{when}}\ b^j \Rightarrow \underline{\text{delay}}\ E^j;\ S^j\ \underline{\text{endselect}},$

   where $S_i \equiv \underline{\text{accept}}\ a_i(\vec{u}_i\#\vec{v}_i)\ \underline{\text{do}}\ S_{i1}\ \underline{\text{endaccept}};\ S_{i2}\ (i=1..n)$

   The semantics is, that first the minimum value, MIN, of those $E^j$ whose guard, $b^j$, is open is evaluated. If a rendezvous with one of the $a_i$'s whose guard, $b_i$, is open, can be started either immediately or within duration MIN, then it is performed and $S_{i2}$ is executed afterwards. Otherwise, when MIN time units have elapsed, one of the delay alternatives $S^j$ for which $E^j$=MIN (and whose associated guard is open) is executed.

   Our translation:

   $[\mathop{\square}_{i=1}^{n} b_i;.a_i?(\vec{u}_i,\vec{v}_i)\#id \to \tau(S_{i1});id.a_i!\vec{v}_i;\tau(S_{i2})\ \square\ \mathop{\square}_{j=1}^{m} b^j;\underline{\text{wait}}(E^j+1) \to \tau(S^j)].$