EINDHOVEN UNIVERSITY OF TECHNOLOGY

Compositional semantics for real-time distributed computing

echnology

Compositional Semantics for Real-Time Distributed Computing

by

R. Koymans

86.08

Compositional Semantics for Real-Time Distributed Computing

by

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

86.08

October 1986

85.42

Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB EINDHOVEN
The Netherlands
All rights reserved
editor: F.A.J. van Neerven

COMPOSITIONAL SEMANTICS FOR REAL-TIME DISTRIBUTED COMPUTING [‡]

R. Koymans[1,3,4,5]
R.K. Shyamasundar[2,6]
W.P. de Roever[1,4]
R. Gerth[1,4]
S. Arun−Kumar[2]

October 1986

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/Müldner, that allows for 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 the real-time flight control software of the NASA space shuttle ([CACM 84]).

[‡] This paper is an extension of a preliminary version presented at the 1985 Logics of Programs Conference, Brooklyn, June 17-19, 1985.
This research was done as an activity in the Dutch National Project Concurrency (Dutch acronym LPC).

[1] Department of Mathematics and Computing Science, Eindhoven University of Technology, Den Dolech 2, P.O.Box 513, 5600 MB Eindhoven, The Netherlands.

[2] NCSDCT, Tata Institute for Fundamental Research, Homi Bhaba Road, Bombay - 400 005, India.

[3] 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).

[4] The author is currently working in and partially supported by ESPRIT project 937: Debugging and Specification of Ada Real-Time Embedded Systems (DESCARTES).

[5] Electronic mail address: mcvax!eutrc3!wsinronk.UUCP or WSDCRONK@HEITHE5.BITNET.

[6] Supported by a visitors grant from the Netherlands Organization for the Advancement of Pure Research (ZWO).

1. INTRODUCTION

Although concurrency in programming has been seriously investigated for more than 25 years ([Dij 59]), the specific problems of real-time have been the object of little theoretical reflection. Currently used real-time languages represent almost no evolution with respect to assembly language ([Cam 82]). Consequently no serious analysis of complexity, no design methodology, no standard for implementation and no concept of portability exist for real-time languages.

The response to this has been the development of new real-time languages such as (1) Ada - developed for the military -, (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 rigorously defined ([Ada 83], [BO 80], [BLW 82], [Occ 84]). Yet their official standards lack any acceptable characterization of concurrency (with the exception of Occam), let alone of real-time (which is also lacking for Occam).

All these arguments emphasize the need to develop formal models for real-time concurrency, and, what is more important, to discover structuring methods which lead to hierarchical and modular development of real-time concurrent systems. Obviously, models based on interleaving, such as [BH 81], can be immediately discarded as being unrealistic, since such models allow unbounded delay to be incurred between any two actions in a concurrent component.

A model such as SCCS ([Mil 83]), although an improvement by allowing truly concurrent activity, remains unsatisfactory because it either enforces complete synchronicity in executions (so that any communication must be performed immediately to circumvent deadlock) or does not exclude interleaving (by using delay-operations). Petri-net theory remains a viable direction for discovering structuring methods, yet is still unsatisfactory because it does not incorporate (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 the sense 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 (see sections 9 and 10); it is based on Salwicki's notion of maximal concurrency ([SM 81]), discussed in section 3,

- applies to programming languages for distributed computing such as Ada and Occam which are based on *synchronized communication* (for asynchronized communication as in CHILL, see [KVR 83]),

- implies a sound and relatively complete method for verification since it is *compositional*; we base ourselves in this respect on the method developed by Misra & Chandy ([MC 81]) and Zwiers ([ZRB 85]), and joint research together with Pnueli leading to the incorporation of maximal parallelism within the temporal framework of [BKP 84],

- meets the standard of rigour as provided by denotational semantics.

Some of these aspects are also covered by work of Zijlstra ([Zij 84]) and G. Jones ([Jon 82]).

We have developed a real-time variant of CSP, called CSP-R, which allows the modelling of the essential Ada ([Ada 83]) real-time features (see Appendix A). Our study of real-time distributed computing is carried by a subset of this language, Mini CSP-R (see section 2). Extending our techniques to CSP-R introduces some notational complications, but is straightforward and is briefly discussed in Appendix A. In this paper we develop a denotational semantics for Mini CSP-R (in section 7), stressing compositionality, based on the linear history semantics for CSP of [FLP 84]:

- the basic domain consists of non-empty *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 (this makes the approach applicable to nested parallelism),

- the histories contain enough information to detect deadlock, eliminating the expectation states of [FLP 84].

The basic domain and its interpretation is given in section 6.

Histories are modelled as sequences of *bags* of communication assumption records as we allow truly concurrent actions: There is a clear operational difference between one process offering a particular communication capability and two (or more) processes, executing in parallel, each offering the same capability. It is to model this distinction that we have to use bags instead of sets (see also example 3 in section 8).

The general notations and technical preliminaries for these concepts are defined in section 5 which serves as a general reference point.

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 section 4).

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

- communication claims $<i,j,v>$, modelling the execution of an I/O command: $<i,j,v>$ claims that the value v is passed from process i (the sender) to process j (the receiver),

- no-match claims $<i,j>$, modelling the absence of a possibility for the execution of an I/O command α (this means that there is no matching I/O command $\bar{\alpha}$ such that α and $\bar{\alpha}$ can be executed simultaneously): $<i,j>$ claims that no value could be passed from process i (the sender) to process j (the receiver).

The combination of the communication assumption records $<i,j,v>$ and $<i,j>$ can be used to describe all possible behaviours when executing an I/O command concerning communication from i to j: $<i,j,v>$ claims that communication from i to j (transferring value v) is *possible* and $<i,j>$ claims that a communication from i to j is *impossible*.

Note that a no-match claim $<i,j>$ implies the *waiting* for a possibility to communicate from i to j. The constraint of no unrealistic waiting that the maximal parallelism model imposes on parallel execution, can now be formulated as: two processes may not make the same no-match claim, i.e., waiting at both sides for the same communication between each other is prohibited.

The communication claim record is the same as the communication record of [FLP 84]. Internal moves within a process (the δ-record of [FLP 84]) are modelled by empty bags.
The no-match claim record is new and allows

- the checking of the maximal parallelism constraints, i.e., no unnecessary waiting (see above),

- the detection of (established) deadlock (i.e., waiting for a communication that will never come), rendering expectation states as in [FLP 84] unnecessary.

Finally, section 11 contains conclusions and outlines some of the research going on.

2. MINI CSP-R

In this section we describe our language Mini CSP-R. Mini CSP-R consists of the programming constructs of our interest in their basic form without syntactic sugar. In appendix A we show how Mini CSP-R can easily be extended to a language CSP-R that can simulate the basic Ada real-time and communication primitives.

Mini CSP-R essentially is CSP (see [Hoa 78]) with the addition of the real-time construct **wait** d. This construct can be used both as instruction and as guard in a selection or loop. As guard it functions as a time-out, revoking the willingness of a process to communicate (through one of the I/O guards).

In the syntax we use the following conventions:

- a *process identification* is an element of $\{P_1,P_2,...\}$.
- a *duration* is an integer-valued expression.

We assume that expressions e and boolean expressions b have some unspecified syntax.

The primitive language elements are the *instructions*, notation Instr:

1. $x := e$	- assignment
2. **wait** d	- wait instruction (d is a duration)
3.1 $P_i!e$	- output (send) to process i the value of the expression e
3.2 $P_i?x$	- input (receive) from process i a value and assign this value to the variable x.

Instructions of form 3 are called I/O commands: $P_i!e$ is an output command and $P_i?x$ an input command.

The important notion of *syntactic matching* of two I/O commands in two processes is defined as follows: two pairs $<P_i,\alpha>$ and $<P_j,\beta>$ (α,β I/O commands) match syntactically iff ($\equiv$ stands for syntactical equality): ($\alpha \equiv P_j!e$ and $\beta \equiv P_i?x$) or ($\alpha \equiv P_j?x$ and $\beta \equiv P_i!e$).

Communication between processes i and j takes place when $<i,\alpha>$ *semantically matches* $<j,\beta>$:
- $<P_i,\alpha>$ and $<P_j,\beta>$ match syntactically,
- control in P_i and P_j is in front of α, respectively β.
The result of a semantic match is the simultaneous execution of the I/O commands as indicated by 3.1 and 3.2. Its effect is the assignment of the value of the expression of the sending process to the variable of the receiving process.

A *guard* is of one of the following forms:

1. b	- pure boolean guard
2.1 α	- pure I/O guard
2.2 b; α	- boolean I/O guard
3.1 **wait** d	- pure wait guard
3.2 b; **wait** d	- boolean wait guard.

In these clauses, b is a boolean expression, (e.g. $x>0$), α is an I/O command and d is a duration. For a guard g, its *boolean part* $\bar{g}$ is defined as: $\bar{b} = b$, $\bar{\alpha} = \mathbf{true}$, $\overline{b;\alpha} = b$, $\overline{\mathbf{wait}\ d} = \mathbf{true}$, $\overline{b;\mathbf{wait}\ d} = b$.
A guard g is called *open* if $\bar{g}$ evaluates to true.

To complete the definition of Mini CSP-R, we define *commands*, notation Comm, together with *parallel commands*, notation ParComm, and the set of *visible subprocesses* of a command, notation vsp, inductively as follows:

1. every instruction is a command; $\mathrm{vsp}(T) = \emptyset$ for every $T \in \mathrm{Instr}$.

2. if $T_1, T_2 \in \mathrm{Comm}$, then $T_1; T_2$ is a (sequential composition) command with $\mathrm{vsp}(T_1;T_2) = \mathrm{vsp}(T_1) \cup \mathrm{vsp}(T_2)$.

3. if $T_1,\ldots,T_n \in \mathrm{Comm}$ and $g_1,\ldots, g_n$ are guards ($n \geqslant 1$), then $[\overset{n}{\underset{j=1}{\square}}\ g_j \rightarrow T_j]$ is an (alternative) command and $*[\overset{n}{\underset{j=1}{\square}}\ g_j \rightarrow T_j]$ is a (repetitive) command with $\mathrm{vsp}([\overset{n}{\underset{j=1}{\square}}\ g_j \rightarrow T_j]) = \mathrm{vsp}(*[\overset{n}{\underset{j=1}{\square}}\ g_j \rightarrow T_j]) = \bigcup_{j=1}^{n} \mathrm{vsp}(T_j)$.

4.1 if $T \in \mathrm{Comm}$ and $i>0$, then $P_i{::}T$ is a (named) parallel command.

4.2 if $T_1, T_2 \in \mathrm{ParComm}$ and the following two restrictions are satisfied:

(r1) the variables occurring in T_1 are different from those occurring in T_2,
(r2) the visible subprocesses of T_1 are different from those of T_2

then $(T_1 \parallel T_2)$ is a (composite) parallel command.

5. a parallel command is also a command with $\mathrm{vsp}(P_i{::}T) = \{i\}$ and $\mathrm{vsp}((T_1 \parallel T_2)) = \mathrm{vsp}(T_1) \cup \mathrm{vsp}(T_2)$.

Note that in a composite parallel command $(T_1 \parallel T_2)$ all non-composite commands are of the form $P_i{::}T$. We further adopt the naming conventions of [Hoa 78, FLP84]: an I/O command within a (named) command $P_i{::}T$ may address only one of P_i's sibling processes or one of its ancestor's sibling processes. Note that such a naming convention may result in a match with a subprocess of the named sibling (see example 5 in section 8).

We can interpret Mini CSP-R informally as follows (this interpretation applies also to CSP-R):

1.1 An assignment has its usual interpretation: the value of the expression e is assigned to the variable x.

1.2 The wait instruction suspends execution of the process in which it occurs for the value of d (but at least one) time units.

1.3 The interpretation of I/O commands was already indicated above: an I/O command α in process i waits for a semantic match with an I/O command β in a process j.

2 The interpretation of sequential composition is as usual: the execution of T_1 is followed by the execution of T_2.

3.1 The interpretation of an alternative command is as follows: First check if none of the guards is open. If this is the case, execution aborts. Otherwise, check whether there is at least one open pure boolean guard. If this is the case select non-deterministically one of these guards. In the case that at least one of the guards is open but there are no open pure boolean guards, execution of an alternative command proceeds as follows. The waitvalue is defined to be infinite if there are no open wait guards and otherwise the maximum of 1 and the minimum of the values of the durations of the open wait guards. For waitvalue time units wait for a semantic match with one of the open I/O guards. As soon as a semantic match occurs within this time period, take it (if more semantic matches occur at the same moment, non-deterministically choose one of them). If no semantic match occurs within waitvalue time units, after this time period one of the open wait guards with a minimal duration is selected. A selection of a guard g_j in all these cases is followed by the execution of the corresponding command T_j.
Observe that in this interpretation of an alternative command a choice has been made; viz., commands guarded by open boolean guards have priority over commands guarded by open I/O guards for which an immediate semantic match is available. This choice is motivated by our aim to model Ada's real-time features (see Appendix A2).

3.2 The interpretation of a repetitive command is the repeated execution of the alternative command contained in it. Now, however, execution terminates normally whenever in this repetition none of the guards is open.

4.1 The interpretation of a named parallel command is as follows:
P_i::T executes its body T. Furthermore, for a semantic match of *any* I/O command α in T with an I/O command outside T, α is considered to be part of process i and process i only. Hence if α occurs in the body of some visible subprocess of T, α is not addressable by the name of that visible subprocess from outside T anymore. Even more, the visible subprocesses of T are no longer visible outside P_i::T.

4.2 The interpretation of a composite parallel command involves the parallel execution of the parts T_1 and T_2. The underlying parallel execution semantics is *not* interleaving semantics, but a semantics based on the maximal parallelism model (see sections 3 and 9). For Mini CSP-R this means that whenever there is a choice between different semantic matches for some I/O command in a process, always one of the semantic matches that occurred earliest in time is non-deterministically chosen.

3. THE MAXIMAL PARALLELISM MODEL

Under maximal parallelism, the number of instructions in concurrently executing processes that can be executed simultaneously without violating synchronization requirements, is maximalized (see [SM 81] for a formal definition). So, in the program $[P_1{::}\ x := 1 \parallel P_2{::}\ x := 3 \parallel P_3{::}\ y := 2]$ *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, we take maximal parallelism to mean "first-come first-served" (fcfs) in some global time scale (see section 4).

Consider the Mini CSP-R program $(P_1{::}\ (P_{11}{::}P_2!0 \parallel (P_{12}{::}P_{13}!1 \parallel P_{13}{::}P_{12}?x;\ P_2!x)) \parallel P_2{::}P_1?y;\ P_1?y)$.

According to *interleaving* semantics two scenarios are possible:

(1) P_{11} communicates with P_2 while P_{12} communicates with P_{13}; after that P_{13} communicates with P_2

(2) P_{12} first communicates with P_{13}; after that P_{13} communicates with P_2; finally, P_{11} communicates with P_2.

According to *maximal parallelism* semantics, only (1) is possible since P_{11} and P_2 can *immediately* become engaged in a rendezvous and hence do not wait for P_{12} and P_{13} to communicate earlier.

The model is however not intended to maximize the amount of ongoing activity in a global way. What a process does is decided locally, partially based on the process' knowledge of communications that are being offered to it but otherwise independent of what goes on elsewhere. What the model does guarantee is that whenever a process wants to communicate it will do so at the earliest opportunity and that local noncommunicating actions are executed without any delay.

As we shall reason in section 9, the maximal parallelism model has some unrealistic aspects for distributed systems in general. We shall develop a whole family of real-time models that range from interleaving to maximal parallelism semantics and that incorporate the transmission time for messages in a system.

4. OUR VIEW OF TIME

To express real-time properties such as "the system responds to 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. NOTATIONS AND TECHNICAL PRELIMINARIES

This section is intended as a reference to our notation.

5.1 Numbers, sets, cartesian product and finite sequences

$\mathbb{N} = \{0,1,2,\ldots\}$ is the set of natural numbers ordered by $0 < 1 < 2 < \ldots$

$\mathbb{N}^{\infty} = \mathbb{N} \cup \{\infty\}$, inherits the ordering on $\mathbb{N}$ and is additionally ordered by $n < \infty$ for all $n \in \mathbb{N}$.

The empty set is denoted by $\emptyset$.

The powerset of a base set E, i.e., the set of all subsets of E, is denoted by $\mathbf{P}(E)$.

If $E_1,\ldots,E_n$ are sets, then $\mathop{X}\limits_{i=1}^{n} E_i$ denotes their cartesian product.

If all E_i's are equal (to E), we write E^n for $\mathop{X}\limits_{i=1}^{n} E_i$.

π_i, for $1 \leqslant i \leqslant n$, denote the associated projection functions for elements of $\mathop{X}\limits_{i=1}^{n} E_i$: $\pi_i(< e_1,\ldots,e_n >) = e_i$.

A finite sequence over a base set E is an element of $\mathbf{S}(E) \overset{\text{def.}}{=} \bigcup_{n \in \mathbb{N}} E^n$, denoted by $<e_1,\ldots,e_n>$ or $<e_i>_{i=1}^{n}$ where $e_i \in E$, $1 \leqslant i \leqslant n$.

If all e_i's are equal (to e), we write $<e>^n$ for $<e_i>_{i=1}^{n}$.
A special case is n=0: it is called the empty sequence, notation λ.
The length of a sequence $s = <e_1,\dots,e_n>$, notation $|s|$, is n.
For a sequence $s = <e_1,\dots,e_n>$ and $1 \leqslant k \leqslant n$ we define the k-th element of s, notation s(k), as e_k.
For $e \in E$ and $s \in \mathbf{S}(E)$, we say that e is an element of s, notation $e < s$, if there exists a k, $1 \leqslant k \leqslant |s|$, such that $s(k) = e$.
Given $s_1, s_2 \in \mathbf{S}(E)$, we can concatenate them, notation $s_1\hat{}s_2$: if $s_1 = <e_1,\dots, e_n>$ and $s_2 = <e_1',\dots,e_m'>$, then $s_1\hat{}s_2 = <e_1,\dots,e_n, e_1',\dots, e_m'>$. Note that $\hat{}$ is closed, associative and has identity element λ:
$s_1, s_2 \in \mathbf{S}(E) \Rightarrow s_1\hat{}s_2 \in \mathbf{S}(E)$, $(s_1\hat{}s_2)\hat{}s_3 = s_1\hat{}(s_2\hat{}s_3)$ and $s\hat{}\lambda = \lambda\hat{}s = s$.
For $s,s' \in \mathbf{S}(E)$ we say that s' is a prefix of s, notation $s' \leqslant s$, if there exists a $s'' \in \mathbf{S}(E)$ such that $s = s'\hat{}s''$.

5.2 *Functions and partial functions*

The set of all functions from X (the domain) to Y (the range) is denoted by Y^X. The domain and range of a function f are denoted by dom(f), respectively ran(f). A partial function from X to Y is an element of $Y^{X'}$ where $X' \in \mathbf{P}(X)$, i.e., a function from a subset of X to Y.
For f a partial function from X to Y, $x \in X$ and $y \in Y$, f[y/x] is the partial function with dom(f[y/x]) = dom(f) $\cup$ {x} and ran(f[y/x]) = Y defined by

$$(f[y/x])(x') = \begin{cases} y & \text{if } x' = x, \\ f(x') & \text{if } x' \in \text{dom}(f)\setminus\{x\}. \end{cases}$$

5.3 *Bags*

A bag (or multiset) over a base set E is an element of $\mathbf{B}(E) \overset{\text{def.}}{=} \mathbb{N}^E$, i.e., a function from E to $\mathbb{N}$.
For $e \in E$ and $B \in \mathbf{B}(E)$ we say that e is an element of B, notation $e \in B$, if $B(e) > 0$.
For finite bags we often use the notation $[e_1^{i_1}, \dots, e_n^{i_n}]$ where $n \in \mathbb{N}$, $i_k \geqslant 1$, $e_k \in E$, all e_k different ($1 \leqslant k \leqslant n$) which corresponds to the bag $B \in \mathbf{B}(E)$ defined by

$$B(e) = \begin{cases} i_k & \text{if } e = e_k,\ 1 \leqslant k \leqslant n, \\ 0 & \text{otherwise.} \end{cases}$$

If $i_k = 1$, we just write e_k instead of e_k^1.
A special case is n=0, the empty bag, notation [ ].

6. THE SEMANTIC DOMAIN AND ITS INTERPRETATION

6.1 The semantic domain

Because our basic domain consists of state-history pairs, we first explain what states and histories are.
Let Id be a (fixed) set of identifiers (i.e., a set of strings over some alphabet). Since we gave no syntax for expressions in Mini CSP-R, we assume furthermore the existence of a set V of expression values.
S, the set of *proper* states, is defined to be the set of partial functions from Id to V. So a proper state $s \in S$ maps *certain* identifiers to their value.
Σ, the total set of states, can now be defined as $S \cup \{\perp, \bullet\}$ where $\perp$ denotes an incomplete computation and $\bullet$ denotes failure (both explained later).
Let $CAR = (\mathbb{N}\times\mathbb{N}) \cup (\mathbb{N}\times\mathbb{N}\times V)$ be the set of communication assumption records (for the intuition, see the last part of the introduction).
H, the set of histories, is, as was motivated in the introduction, $\mathbf{S}(\mathbf{B}(CAR))$. It would in fact suffice to take $H = \mathbf{S}(\mathbf{P}(\mathbb{N}\times\mathbb{N})\times\mathbf{B}(\mathbb{N}\times\mathbb{N}\times V))$, as bags are only needed to collect communication claims. Obviously, for claiming the *absence* of a communication possibility between process i and j, it suffices to do this only once. However, we prefer the first notationally simpler definition.
The technical reason for using bags instead of sets is illustrated in example 3 of section 8.
Our central domain is that of non-empty prefix-closed sets of state-history pairs, notation ΣH.

Definition: A set $X \in \mathbf{P}(\Sigma\times H)$ is *prefix-closed* iff for all $<\sigma, h> \in X$, if $h' \leqslant h$, then $<\perp, h'> \in X$.

The *prefix-closure* of X, PFC(X), is defined as
$X \cup \{<\perp, \lambda>\} \cup \{<\perp, h'> \mid \exists\sigma\exists h\, (<\sigma, h> \in X \wedge h' \leqslant h)\}$.

Note that $PFC(X) \in \Sigma H$, for all $X \in \mathbf{P}(\Sigma\times H)$.

ΣH can be turned into a complete lattice:

- the partial ordering is $\subseteq$, set-inclusion
- the least upper bound is obtained by $\cup$, set-union.

Its least element is $\{<\perp, \lambda>\}$.

The technical motivation for the introduction of $\perp$ lies in the simplicity of the ordering of ΣH: several proofs, in particular those for continuity of operators, become very simple.

The introduction of a separate failure state $\bullet$ is needed for the detection of non-deterministic failure (see below, in section 6.2).

We want elements of ΣH to be non-empty, because otherwise the least element of ΣH would be $\emptyset$. Since $\emptyset$ contains no history at all, and sequential composition is essentially modelled by concatenation of histories, this choice of least element would imply that the denotation of $*$[**true**$\rightarrow P_2!5$] would be empty. Although consistent with the view that a command is a transformation of initial states to final states when characterizing sequential constructs relationally, this does not capture our intuition that an unbounded set of communication possibilities may have been offered by $*$[**true**$\rightarrow P_2!5$] (cf. example 1 in section 8).

Remark: As E.-R. Olderog observed in the context of the linear history semantics for CSP (see [FLP 84]), here too, we do not need to order our domain. This is a consequence of the fact that our recursions are always guarded (see loops) and that histories, once they have been generated, can not 'shrink', i.e., they remain the same or are extended to a longer history. For details, see the Appendix of [FLP 84].

6.2 *Interpretation of* ΣH

We can interpret $X \in \Sigma H$ as the set of all possible computations of a program P (cf. [FLP84]):

- $<s,h> \in X$ with $s \in S$, models a computation of P producing history h that terminates in s,

- $<\bullet,h> \in X$ models a failure of P after producing history h,

- $<\perp,h> \in X$ models an incomplete computation of P which is *either* an approximation of a computation $<\sigma,h'>$ with $\sigma \neq \perp$ and $h \leq h'$ *or* an element in a chain of approximations $<\perp,h_0>, <\perp,h_0\hat{}h_1>, \ldots$ (all $h_i \neq \lambda$) which models an *infinite* computation of P with history $h_0\hat{}h_1\hat{}\ldots$ (this interpretation can be justified by an appeal to König's Lemma, based on an intuitive operational semantics).

If only deterministic failure can occur, there is no need for a separate failure state $\bullet$ because $\perp$ can be used for that purpose: deterministic failure of P after history h is then modelled by $<\perp,h> \in X$ such that there exists neither $<s,h'> \in X$ with $s \in S$, $h \leq h'$ nor $<\perp,h'> \in X$ with $h \leq h'$, $h \neq h'$. However, we have to include the possibility of non-deterministic failure as demonstrated by the following Mini CSP-R program fragment: [**true** → [**false** → x := 0] □ **true** → x := 1].

Using the above interpretation of ΣH, we can informally define a notion of observable behaviour. The observable entities are: a communication history, termination, failure and infinite computation. The observable behaviour of a communication history has already been given in the introduction. The other observable entities are given in the above interpretation of ΣH:

- termination: indicated by a proper state $s \in S$.

- failure: indicated by ●.

- infinite computation: indicated by an infinite chain of approximations.

Both divergence and established deadlock are viewed as infinite computations: divergence is making internal steps while time passes, established deadlock is waiting for a communication that will not come, while time passes. This means that divergence and established internalized deadlock are observed in the same way, and hence can not be distinguished. In our view this is a perfectly reasonable standpoint: the only observation that can be made from the outside is the ticking of the global clock while no communication with the environment can occur. In other words: there is no context that can distinguish a diverging process from such a deadlocked one (cf. example 2 in section 8).

7. MAXIMAL PARALLELISM SEMANTICS FOR MINI CSP-R

7.1 Introduction

The meaning of Mini CSP-R commands is defined *denotationally* by giving for all commands T, an equation which relates the meaning of T, notation $M[\![T]\!]$, to the meaning of T's constituents in a compositional way. In section 7.2 we show that it suffices to define $M[\![T]\!]$ as a function from S to ΣH.

To define the alternative command $[\overset{n}{\underset{j=1}{\square}} g_j \rightarrow T_j]$ compositionally, we use an auxiliary semantic function $G[\![g,A]\!]$ from S to ΣH which gives the meaning of guard g in the context of a set A of alternative guards (the other guards in the alternative command). We use the context A in a compositional way, i.e., A depends only on the alternative command in which g occurs. G is furthermore used in defining the meaning of guards that occur as instructions (these are the pure waitguards and pure I/O guards). The meaning of such an instruction is simply the meaning of the guard in an empty context.

Since we gave no syntax for (boolean) expressions in Mini CSP-R, we assume the existence of semantic functions V and W, such that $V[\![e]\!]$ for e an expression is a function from S to V, and $W[\![b]\!]$ for b a boolean expression is a predicate on S, i.e., for $s \in S$ $W[\![b]\!]s$ is either true or false.

To define the meaning of constructs like $P_1::P_2!5$ compositionally, we have to give a meaning for $P_2!5$ separately, i.e., in a context where it is not known that this construct belongs to the process with identification 1. In order to do so, we introduce as semantic entity the 'unknown process', with process identification 0, and use this e.g. to generate records $<0,2,5>$ in the meaning for $P_2!5$ and later, in the meaning for $P_1::P_2!5$, replace 0 by 1.
Therefore, we identify process identifications with natural numbers.

Just as for the syntax we need a notion of visible subprocesses of a command T, VS(T). The difference with the definition in section 2 is the use of $\{0\}$ instead of $\emptyset$:

$VS(T) = \{0\}$ for T an instruction,
$VS(T_1;T_2) = VS((T_1 \parallel T_2)) = VS(T_1) \cup VS(T_2)$,
$VS([\overset{n}{\underset{j=1}{\Box}} g_j \rightarrow T_j]) = VS(*[\overset{n}{\underset{j=1}{\Box}} g_j \rightarrow T_j]) = \{0\} \cup \overset{n}{\underset{j=1}{\cup}} VS(T_j)$,
$VS(P_i::T) = \{i\}$.

In the third line, the zero is needed to account for I/O guards as e.g. in $P_1::[P_2!0 \rightarrow P_3::x:=0]$.

To keep the semantics simple, we assume that the evaluation of expressions takes no time. However, this restriction can easily be relaxed by introducing time-parameters that represent evaluation times of expressions. Furthermore, we make the realistic assumption that the execution of commands takes at least one unit of time unless failure occurs (this can only occur if an alternative command which has no open guard is executed). The idea behind this decision is that we want to exclude the unrealistic possibility of an infinite loop taking zero time. Such a loop is possible in Ada, as shown in Appendix A2, and obviously this possibility must be excluded. Appendix A2 contains a discussion how to do so.

7.2 Extending the meaning function

$M[\![T]\!]$, the meaning of a construct T, only depends on a proper state $s \in S$: $M[\![T]\!]s \in \Sigma H$ represents all possible state changes and computational histories produced by T starting from s. It therefore seems sufficient to let $M[\![T]\!]$ be a function from S to ΣH. However, to define sequential composition we have to extend the meaning function to a function from ΣH to ΣH (this situation is analogous to that for a purely sequential non-deterministic language where the meaning function is generalized to sets of states). This extension shall be defined uniformly for all functions from S to ΣH, so we can still use $M[\![T]\!]$ as a function from S to ΣH keeping in mind that this extension must be used when composing meaning functions. We first extend a function ϕ from S to ΣH to a function ϕ^+ from Σ to ΣH and next to a function ϕ^* from ΣH to ΣH.

Definition: Let ϕ be a function from S to ΣH. Then ϕ^+ is the function from Σ to ΣH defined by

$$\phi^+(\sigma) = \begin{cases} \phi(\sigma) & \text{if } \sigma \in S, \\ PFC(\{<\sigma,\lambda>\}) & \text{otherwise.} \end{cases}$$

Furthermore, ϕ^* is the function from ΣH to ΣH defined by
$\phi^*(X) = \{<\sigma',h\hat{\ }h'> \mid <\sigma,h> \in X \ \wedge \ <\sigma',h'> \in \phi^+(\sigma)\}$.

ϕ^* extends ϕ in a canonical way: for $X \in \Sigma H$ it takes $<\sigma,h> \in X$ and extends h with an additional history h' formed by applying ϕ^+ to σ; ϕ^+ behaves like ϕ on S but takes care that histories of pairs $<\sigma,h> \in X$ with $\sigma \notin S$ are not extended; the new state σ' is the state after applying ϕ^+ to σ.

The histories h represent communication assumptions that have been made and can only be supplemented with additional communication assumptions. In other words: the extension of histories is independent of their contents. The meaning function should certainly have this property. A property of ϕ^* is that it is always strict and continuous, as proved below. This means that we do not have to worry about the continuity of operators in our semantics!

Proposition: For all ϕ from S to ΣH, ϕ^* is a strict and continuous function from ΣH to ΣH.

Proof: $\phi^*(\{<\perp,\lambda>\}) = \{<\sigma',\lambda\hat{\ }h'> \mid <\sigma',h'> \in \phi^+(\perp)\} = \phi^+(\perp) = \{<\perp,\lambda>\}$ and

$$\begin{aligned}\phi^*(\bigcup_{i\in I} X_i) &= \{<\sigma',h\hat{\ }h'> \mid <\sigma,h> \in \bigcup_{i\in I} X_i \ \wedge \ <\sigma',h'> \in \phi^+(\sigma)\} \\ &= \bigcup_{i\in I}\{<\sigma',h\hat{\ }h'> \mid <\sigma,h> \in X_i \ \wedge \ <\sigma',h'> \in \phi^+(\sigma)\} \\ &= \bigcup_{i\in I} \phi^*(X_i). \qquad \blacksquare\end{aligned}$$

7.3 *Definition of* **G**

In the definition of **G** we use the following two auxiliary notions for guards:

Definition 1: For a set of guards G and $s \in S$, define RTA(G,s)$\in$**B**(CAR), the bag of real-time assumptions concerning the open I/O guards of G in state s, as follows:

$$RTA(G,s)(r) = \begin{cases} 1 & \text{if } r \in \{<0,i> \mid \exists g \in G(g \equiv P_i!e \ \vee \ (g \equiv b{:}P_i!e \ \wedge \ \mathbf{W}[\![b]\!]s))\} \\ & \quad \cup \{<i,0> \mid \exists g \in G(g \equiv P_i?x \ \vee \ (g \equiv b{:}P_i?x \ \wedge \ \mathbf{W}[\![b]\!]s))\}, \\ 0 & \text{otherwise.} \end{cases}$$

Remark: If e.g. $P_2!4$ and $P_2!6$ occur in G one might expect a multiplicity 2 (instead of 1) for the record $<0,2>$ in the above definition. This is unnecessary (see the discussion of bags versus sets in section 6.1).

Definition 2: For a guard g and $s \in S$, define **waitvalue**$(g,s) \in \mathbb{N}^\infty$ as follows:

$$\text{waitvalue}(g,s) = \begin{cases} 0 \quad \text{if } g \equiv b \;\wedge\; \mathbf{W}[\![b]\!]s, \\ \max\{\mathbf{V}[\![d]\!]s,1\} \quad \text{if } g \equiv \textbf{wait } d \;\vee\; (g \equiv b;\textbf{wait } d \;\wedge\; \mathbf{W}[\![b]\!]s), \\ \infty \quad \text{otherwise.} \end{cases}$$

Furthermore, for a set of guards G and $s \in S$, define minwait$(G,s) \in \mathbb{N}^\infty$ as min $\{\text{waitvalue}(g,s) \mid g \in G\}$ (where by convention min $\emptyset = \infty$).

Note that the guard **true** has waitvalue 0 while the guards **wait** 0 and **wait** 1 have waitvalue 1. The decision to let **wait** 0 have waitvalue 1 is explained in Appendix A2.

The equations for **G** are (see section 7.1 for its use and motivation):

$$\mathbf{G}[\![b,A]\!]s = \begin{cases} \text{PFC}(\{<s,\lambda>\}) \quad \text{if } \mathbf{W}[\![b]\!]s, \\ \{<\perp,\lambda>\} \quad \text{otherwise.} \end{cases}$$

A boolean acts as a filter: s is maintained only if b evaluates to true in s.

$$\mathbf{G}[\![\textbf{wait } d,A]\!]s = \text{PFC}(\{<s,<\text{RTA}(A,s)>^T> \mid \max\{\mathbf{V}[\![d]\!]s,1\} = \text{minwait}(A \cup \{\textbf{wait } d\},s) \overset{\text{def.}}{=} T\}).$$

A pure wait guard in the context A can be selected after its waitvalue time units elapsed provided this value equals the minimal waitvalue T (note that $T \in \mathbb{N}$) *and* no semantic match for an open I/O guard in A occurred in this period. If there is at least one open boolean guard in A, then T=0 and no wait guard can be selected.

$$\mathbf{G}[\![P_j!e,A]\!]s = \text{PFC}(\{<s,<\text{RTA}(\text{GRDS},s)>^t \,\hat{}\, <[<0,j,\mathbf{V}[\![e]\!]s>]>> \mid 0 \leqslant t < \text{minwait}(A,s)\}),$$
where GRDS = A $\cup$ $\{P_j!e\}$.

A pure I/O guard in the context A can be selected (indicated by the last triple of the history above) within the minimum waitvalue of A (the bound on t above) under the condition that no semantic match for any open I/O guard in GRDS occurred earlier (indicated by the first t elements of the history above). If there is at least one open boolean guard in A, then minwait(A,s) = 0 and no output guard (in fact, no I/O guard) can be selected. The possibility that no guard at all is selected can only occur if there are no open boolean guards and no open wait guards (hence minwait(A,s) = ∞) and furthermore no semantic match for an open I/O guard ever occurs. This case is represented by the subset $\{<\perp, <RTA(GRDS,s)>^t> \mid t \in \mathbb{N}\}$ of $\mathbf{G}[\![P_j!e,A]\!]s$ (remember, this is a prefix-closed set).

$\mathbf{G}[\![P_j?x,A]\!]s = PFC(\{<s[v/x],<RTA(GRDS,s)>^t \hat{} <[<j,0,v>]>> \mid v \in V, 0 \leq t < minwait(A,s)\})$,
where GRDS = A $\cup$ $\{P_j?x\}$.

The same remarks as for $\mathbf{G}[\![P_j!e,A]\!]s$ apply here. In comparison with $\mathbf{G}[\![P_j!e,A]\!]s$ we see that in the last triple of the history sender and receiver are reversed. Furthermore, for an input command $P_j?x$ we have to 'guess' the value v that will be assigned to x. When binding the inputting process with the outputting process we check that the values correspond (see the last three examples in section 8). This 'guessing' models Bekiç's and Milner's concept of renewal (see [Mil 73]).

$\mathbf{G}[\![b;g,A]\!]s = \mathbf{G}[\![g,A]\!]^* (\mathbf{G}[\![b,A]\!]s)$, where $g \equiv P_j!e$ or $g \equiv P_j?x$ or $g \equiv$ **wait** d.

The meaning of a sequential composition of guards is the functional composition (using the extension operator '*') of the meanings of the separate guards.

7.4 Definition of M

7.4.1 $\mathbf{M}[\![T]\!]$ *for* T∈Comm \ ParComm

In this subsection we give the meaning of the non-parallel commands of Mini CSP-R.

$\mathbf{M}[\![x := e]\!]s = PFC(\{<s[\mathbf{V}[\![e]\!]s/x],<[\,]>>\})$.

To keep the semantics simple, an assignment takes exactly one time unit (indicated by the empty bag).

$\mathbf{M}[\![g]\!]s = \mathbf{G}[\![g,\emptyset]\!]s$ for $g \equiv$ **wait** d or $g \equiv P_j!e$ or $g \equiv P_j?x$.

This use of G was already discussed in section 7.1.

$\mathbf{M}[\![T_1;T_2]\!]s = \mathbf{M}[\![T_2]\!]^*(\mathbf{M}[\![T_1]\!]s)$.

$$\mathbf{M}[\![[\mathop{\Box}_{j=1}^{n} g_j \rightarrow T_j]]\!]s = \begin{cases} \bigcup_{j=1}^{n} \mathbf{M}[\![T_j]\!]^*(G[\![g_j, \{g_k \mid 1 \leq k \leq n,\ k \neq j\}]\!]s) & \text{if } \bigvee_{j=1}^{n} \mathbf{W}[\![\bar{g}_j]\!]s, \\ \mathrm{PFC}(\{<\bullet, \lambda>\}) & \text{otherwise.} \end{cases}$$

The meaning of the alternative command depends on the presence of an open guard: if no such guard is present this means failure, otherwise one guard is selected where each guard is considered in the context of the remaining guards ($\bar{g}_j$ is the boolean part of g_j, see section 2).

Let C abbreviate $[\mathop{\Box}_{j=1}^{n} g_j \rightarrow T_j]$.

$$\mathbf{M}[\![*C]\!]s = \bigcup_{i \in \mathbb{N}} \phi_i(s)$$

where the ϕ_i $(i \in \mathbb{N})$ are functions from S to ΣH defined inductively by

$\phi_0(s) = \{<\bot, \lambda>\}$ for all $s \in S$,

$$\phi_{i+1}(s) = \begin{cases} \phi_i^*(\mathbf{M}[\![C]\!]s) & \text{if } \bigvee_{j=1}^{n} \mathbf{W}[\![\bar{g}_j]\!]s, \\ \mathrm{PFC}(\{<s, <[]> >\}) & \text{otherwise.} \end{cases}$$

The ϕ_i's represent as usual the i-th iteration step of the loopbody. If at some point of iteration there are no open guards anymore, the loop terminates (this last iteration is indicated by the empty bag because the execution of commands takes at least one time unit).
For an illustration of the loop equation see the first two examples in section 8 (these give also a demonstration why $\{<\bot, \lambda>\}$ and not $\emptyset$ should be the least element of ΣH).
The loop equation can alternatively be written as a fixed-point equation over the complete partial order of functions from S to ΣH with the usual ordering on function domains:

$$\mathbf{M}[\![*C]\!] = \mu(\lambda\phi.\lambda s.\ \textbf{if } \bigvee_{j=1}^{n} \mathbf{W}[\![\bar{g}_j]\!]s \textbf{ then } \phi^*(\mathbf{M}[\![C]\!]s) \textbf{ else } \mathrm{PFC}(\{<s, <[\,]> >\}) \textbf{ fi}),$$

where μ is the least fixed-point operator.

7.4.2 *The meaning of* P_i::T

The effect caused by P_i::T is the renaming of the visible subprocesses of T by i. To this end, we need a definition for substitution of a certain process, in this case i, in place of a collection of processes I, in this case VS(T), both for bags over CAR as for elements of ΣH. Although the substitution for bags over CAR is intuitively clear, the technical definition is rather awkward and is therefore given in Appendix B. So, assuming we have defined $B[I \rightarrow i] \in \mathbf{B}(CAR)$ for $B \in \mathbf{B}(CAR)$, $I \in \mathbf{P}(\mathbb{N})$ and $i \in \mathbb{N}$, we can extend this componentwise to elements of ΣH:

$$X[I \rightarrow i] = \{ <\sigma, <h(k)[I \rightarrow i]>_{k=1}^{|h|} > \mid <\sigma, h> \in X \}.$$

Lemma: $X[I \rightarrow i] \in \Sigma H$ for all $X \in \Sigma H$, $I \in \mathbf{P}(\mathbb{N})$ and $i \in \mathbb{N}$.

Proof: $X[I \rightarrow i]$ non-empty: $X \in \Sigma H$ implies $<\perp, \lambda> \in X$ and hence $<\perp, \lambda> \in X[I \rightarrow i]$.

$X[I \rightarrow i]$ prefix-closed:
Let $<\sigma, h> \in X$ and $h' \leqslant <h(k)[I \rightarrow i]>_{k=1}^{|h|}$.
Then $|h'| \leqslant |h|$, so there exists a $h'' \leqslant h$ with $|h''| = |h'|$.
Because $X \in \Sigma H$ it follows that $<\perp, h''> \in X$ and hence $<\perp, h'> =$
$<\perp, <h(k)[I \rightarrow i]>_{k=1}^{|h'|}> \; = \; <\perp, <h(k)[I \rightarrow i]>_{k=1}^{|h''|}> = \; <\perp, <h''(k)[I \rightarrow i]>_{k=1}^{|h''|}> \in X[I \rightarrow i]$. ■

Now we can define

$$M[\![P_i::T]\!]s = (M[\![T]\!]s)[VS(T) \rightarrow i].$$

7.4.3 *The meaning of* $(T_1 \parallel T_2)$

7.4.3.1 *Intuition for parallel composition*

It remains to define the meaning of the most important construct, the parallel composition. Intuitively, when binding two processes, the information of the states is combined, the histories are checked for consistency, and then are merged. Actually this consistency check can be split into two independent parts to be applied at each instant of time:

(c1) Check that histories have matching communication claims, i.e., that histories agree on the communications that occur between the two processes (their internal communications).

(c2) Check that there is no unnecessary waiting, i.e., that histories do not indicate a situation where both processes are waiting for a communication that the other process can provide (in other words: two processes do not wait if there is a semantic match between them).

Check (c1) is the communication consistency check for CSP as in [FLP84]. We call (c2) the real-time consistency check because it enforces maximal parallelism (see the end of section 1). Since the equation for $M[\![(T_1 \| T_2)]\!]s$ is rather complex, we give the intuition behind its steps below, and postpone its formal definition till section 7.4.3.6.

To combine the meanings of $M[\![T_1]\!]s$ and $M[\![T_2]\!]s$ to $M[\![(T_1 \| T_2)]\!]s$, first the states of $M[\![T_1]\!]s$ and $M[\![T_2]\!]s$ should be combined. Although trivial at first sight, this raises problems since we can not always assume that such states have disjoint domains, as illustrated by the program $x{:=}\,0;\ (P_1{::}x{:=}\,1 \| P_2{::}y{:=}\,2)$. This is solved in section 7.4.3.2.

Next consistency checks (c1) and (c2) must be applied to the communication assumption records in $M[\![T_1]\!]s$ and $M[\![T_2]\!]s$. Note that for (c1) it is desired to have a *common* communication claim record in both histories while, on the contrary, (c2) checks that there is *no common* no-match claim record in both histories. Moreover, our semantics is such that in the records in the generated histories of a command always at least one of the processes involved is a visible subprocess of that command (see the History Property in section 7.5). Consequently, for (c2) it is sufficient to check for the absence of identical no-match claims. For (c1), however, one first must establish the visible subprocesses of T_1 and T_2 prior to checking whether a communication claim record in one history should be complemented by an identical record in the other history (since a visible subprocess of T_1 *may* address a process outside of T_2). Therefore, it would be nice if we could first merge the histories that are consistent according to (c2) and *after that* check (c1). Unfortunately this is unfeasible, as is illustrated by the programs
$(P_1{::}P_2!0 \| P_2{::}P_1?x)$ and $(P_1{::}(P_{11}{::}P_2!0 \| P_{12}{::}P_2!0) \| P_2{::}x{:=}0)$.
When following the above approach, the semantics of both these programs would contain the history $<[<1,2,0>^2]>$. Now, this history should represent both a successful communication (the first program) *and* deadlock (the second program): an impossibility. We solve this problem through first subtracting equal communication claim records from each other, and after that check whether any internal communication claims are left. Together with the definition of the real-time consistency check (c2), this is worked out in detail in section 7.4.3.3.

Thirdly, not all histories should be compared when merging. When combining state-history pairs with $\perp$ as state component(s), representing incomplete computation, special care should be taken to guarantee that indeed *all* the events occurring at a particular time are collected in the resulting history. E.g., $<\perp,\lambda> \in M[\![P_1{::}P_3!5]\!]s$ should not be merged with $<s[0/x],<[]>> \in M[\![P_2{::}x{:=}\,0]\!]s$, because the result $<\perp,<[]>>$ will not represent the attempt of P_1 to communicate with P_3 at time 1. This is treated in section 7.4.3.4.

As last step, when giving the meaning of $(T_1 \| T_2)$ in terms of its components, the real-time assumptions (represented by the no-match claim records) concerning the visible subprocesses of T_1 and T_2 should be checked and removed. This is illustrated by the program $(P_1::P_2!5 \| P_2::P_1!5)$. Some histories of P_1 contain the no-match claim $<1,2>$, and some of P_2 the no-match claim $<2,1>$. After binding P_1 and P_2, the real-time assumptions concerning the collection of processes $\{1,2\}$ should be checked; in this case, exactly $<1,2>$ and $<2,1>$. After this check they are not needed anymore and can be removed, since it has been established that no communication will occur.

These four steps correspond with those of the definition of $\mathbf{M}[\![(T_1 \| T_2)]\!]s$, in that order.

7.4.3.2 Combining states

For $\mathbf{M}[\![(T_1 \| T_2)]\!]s$, the states of $\mathbf{M}[\![T_1]\!]s$ and $\mathbf{M}[\![T_2]\!]s$ should be combined. Because of the syntactic restriction that the variables of T_1 and T_2 are disjoint (see section 2, definition of commands), it seems that one can simply form the disjoint union of such states. This is however not the case: the state s of the computation up till now can cause problems. For example, in the program $x:=0; (P_1::x:=1 \| P_2::y:=2)$, x is defined *both* in P_1 and P_2. Fortunately, this is only the case for variables that were defined earlier in the program, or in other words: variables that belong to the domain of s. Variables outside the domain of s belong either to P_1 or P_2 (because of the above mentioned syntactic restriction). The union of states of $\mathbf{M}[\![T_1]\!]s$ and $\mathbf{M}[\![T_2]\!]s$ can now be defined relative to $s \in S$:
Let for $1 \leq i \leq 2$, $s_i \in S$ belong to $\mathbf{M}[\![T_i]\!]s$ (then $dom(s) \subseteq dom(s_i)$).
Define the union of s_1 and s_2 relative to s, notation $s_1 \cup_s s_2$, as follows:

$dom(s_1 \cup_s s_2) = dom(s_1) \cup dom(s_2)$ and

$(s_1 \cup_s s_2)(x) \overset{def.}{=} s_i(x)$ if $x \in dom(s_i) \setminus dom(s)$ or $x \in dom(s), s(x) = s_{3-i}(x)$.

As remarked above, if $x \in dom(s_i) \setminus dom(s)$ then $x \notin dom(s_{3-i})$. In that case, x is a new variable of T_i and the value of that variable in the combined state is $s_i(x)$. For example, for $t:=0; (P_1::y:=1 \| P_2::z:=2)$, $dom(s) = \{t\}$, $dom(s_1) = \{t,y\}$, and $dom(s_2) = \{t,z\}$.
On the other hand, if $x \in dom(s)$, then at most one of T_1 and T_2 can use x, hence $s_i(x) = s(x)$ for $i=1$ or $i=2$. In this case, the value of x in the combined state is $s_{3-i}(x)$. For example, for $t:=0; (P_1::t:=1 \| P_2::z:=2)$, $dom(s) = \{t\}$, $dom(s_1) = \{t\}$, $dom(s_2) = \{t,z\}$ and the value of t after this program is 1.
Note that $\cup_s$ (for all $s \in S$) is commutative and associative.

It remains to extend $\cup_s$ for s_i that belong to $\mathbf{M}[\![T_i]\!]s$ but with s_1 or s_2 (or both) not in S. The idea is that whenever one of the s_i represents an incomplete computation the combination represents the same; otherwise, when one of the states represents failure, the combination represents failure:

$\perp \cup_s \sigma = \sigma \cup_s \perp = \perp$ for all $s \in S$, $\sigma \in \Sigma$ and

$\bullet \cup_s \sigma = \sigma \cup_s \bullet = \bullet$ for all $s \in S$, $\sigma \in \Sigma \setminus \{\perp\}$.

Note that this extension maintains commutativity and associativity.

7.4.3.3 The consistency check

There is a direct correspondence between the two parts of the consistency check and the two types of communication assumption records:

(c1) concerns triples $<i,j,v>$ such that i and j are internal processes, i.e., processes that belong to the collection of processes represented by the two histories whose consistency is checked; check (c1) corresponds to: each such triple in one history should also occur in the other history at the same time and vice versa

(c2) concerns pairs $<i,j>$; it corresponds to: no pair $<i,j>$ in one history may occur at the same time in the other history.

Note that for (c1) we need to know the set of internal processes while this is not necessary for (c2). The reason for this is that in all records in the histories generated by our semantics one of the processes i and j refers to the process that generated this record (this history property is proved in section 7.5). Because (c2) checks that two histories representing different processes do not contain at the same time a common record $<i,j>$, this means that i and j must be internal processes anyway.

The real-time consistency check (c2) is formulated by

$$h_1 \not\subset^{RT} h_2 \overset{def.}{=} \neg\exists\, i,j,k \in \mathbb{N}\, (1 \leqslant k \leqslant \min\{|h_1|, |h_2|\} \wedge <i,j> \in h_1(k) \wedge <i,j> \in h_2(k)).$$

Of course, the consistency check as a whole (and similarly for its part (c1)) could be applied pairwise to histories with the set of internal processes, say I, as parameter: $h_1 \not\subset_I h_2$. However, we prefer to pair histories without such a parameter. Ideally, we would like to combine state-history pairs (states are united, histories merged) for which the histories are real-time consistent and *after that* apply the check (c1). This approach is unfeasible, as is shown by the programs
$(P_1::P_2!0 \parallel P_2::P_1?x)$ and $(P_1::(P_{11}::P_2!0 \parallel P_{12}::P_2!0) \parallel P_2::x:=0)$.
If we would follow the strategy above, the meanings of these programs would both contain the history $<[<1,2,0>^2]>$. The problem is, that we somehow must remove this history from the meaning of the second program (it deadlocks), but reduce the same history to $<[\ ]>$ in the meaning of the first one (showing a successful internal communication); this is clearly an impossibility.

There is, however, an easy trick to circumvent this problem. The above example suggests that we should *subtract equal communication claim records from each other* while merging: for the first program this would result in no $<1,2,0>$-records at all while for the second program the two $<1,2,0>$-records would still be maintained. Check (c1) can then be completed by testing whether after this special merging there are any 'internal communications' left, i.e., communication claims $<i,j,v>$ with i and j internal. Formally, for $X \in \Sigma H$ and $I \in \mathbf{P}(\mathbb{N})$ we define

$¢_I^{IC}(X) = X \setminus \{<\sigma,h> \mid \exists B \leqslant h\ \exists i,j \in I\ \exists v \in V\ <i,j,v> \in B\}$.

Lemma: $¢_I^{IC}(X) \in \Sigma H$ for all $X \in \Sigma H$ and $I \in \mathbf{P}(\mathbb{N})$.

Proof: $¢_I^{IC}(X)$ non-empty: $X \in \Sigma H$ implies $<\perp,\lambda> \in X$ and because there does not exist a $B \leqslant \lambda$ it follows that $<\perp,\lambda> \in ¢_I^{IC}(X)$.

$¢_I^{IC}(X)$ prefix-closed:

$¢_I^{IC}(X)$ deletes pairs from X for which the history has a certain property. Immediately from the definition it follows that all extensions of a history with this property also have this property. Reversing this we get: if a history does not have this property, then none of its prefixes can have this property. This is used in the last step of the chain of implications

$<\sigma,h> \in ¢_I^{IC}(X) \Rightarrow <\sigma,h> \in X \Rightarrow <\perp,h'> \in X \Rightarrow <\perp,h'> \in ¢_I^{IC}(X)$ for all $h' \leqslant h$. ■

The above mentioned special merge is denoted by # and does the following. Up to the length of the shortest history, # subtracts equal records (of course taking the absolute value). It is unnecessary to check especially for communication claim records because histories with equal $<i,j>$-pairs were previously removed in the real-time consistency check. After the length of the shortest history, the longer history is just copied.

Formally:

Let $h_1,h_2 \in H$.

Then $h_1 \# h_2 = <B_k^{h_1,h_2}>_{k=1}^{\max\{|h_1|,|h_2|\}}$,

where $B_k^{h_1,h_2} \in \mathbf{B}(CAR)$ are defined as follows:

for $1 \leqslant k \leqslant \min\{|h_1|, |h_2|\}$, $B_k^{h_1,h_2}(r) = |h_1(k)(r) - h_2(k)(r)|$,

for $\min\{|h_1|, |h_2|\} < k \leqslant \max\{|h_1|, |h_2|\}$, $B_k^{h_1,h_2}(r) = \begin{cases} h_1(k)(r) & \text{if } |h_1| > |h_2| \\ h_2(k)(r) & \text{if } |h_2| > |h_1|. \end{cases}$

In general # is commutative but not associative. However, in the context of $((T_1 \parallel T_2) \parallel T_3)$ and $(T_1 \parallel (T_2 \parallel T_3))$ we may assume because of the syntactic restriction that the visible subprocesses must be disjoint in a parallel composition (in that case vsp and VS coincide): $VS(T_i) \cap VS(T_j) = \emptyset$ for $1 \leqslant i < j \leqslant 3$. In that case, for $s \in S$, $<\sigma_i, h_i> \in M[\![T_i]\!]s$ $(1 \leqslant i \leqslant 3)$, it always holds that $(h_1 \# h_2) \# h_3 = h_1 \# (h_2 \# h_3)$ (see the Corollary in section 7.5). This is used to prove the important property that $M[\![((T_1 \parallel T_2) \parallel T_3)]\!]$ equals $M[\![(T_1 \parallel (T_2 \parallel T_3))]\!]$, see the theorem in section 7.5.

7.4.3.4 An additional condition for combining state-history pairs

When combining state-history pairs $<\sigma_i, h_i>$, $1 \leqslant i \leqslant 2$, in the parallel composition of two processes, we should take care that the condition $\sigma_i = \perp \Longrightarrow |h_i| \geqslant |h_{3-i}|$, $1 \leqslant i \leqslant 2$, is satisfied, i.e., that neither history that can be extended ($\sigma_i = \perp$) is shorter than the other one. Here is why:
Consider the program fragment $(P_1::P_3!5 \parallel P_2::x:=0)$.
For $s \in S$, $<\perp, \lambda> \in \mathbf{M}[\![P_1::P_3!5]\!]s$ and $<s[0/x], <[\ ]>> \in \mathbf{M}[\![P_2::x:=0]\!]s$. If we combine these two state-history pairs without the extra condition above, we get the combined pair $<\perp, <[\ ]>>$. However, this pair should *not* belong to the parallel composition of processes 1 and 2, because only the internal step (the assignment) of P_2 is represented and *not* the attempt of P_1 to communicate with P_3 that occurs *at the same time*.

7.4.3.5 The removal of real-time assumptions

When giving the meaning of $(T_1 \parallel T_2)$ the real-time assumptions (represented by the no-match claim records) concerning the visible subprocesses of T_1 and T_2 should be checked. It is our policy to do this as soon as possible, that is in the first context in which the processes i and j of a no-match claim $<i,j>$ can be identified. The following program fragment illustrates this: $(P_1::P_2!5 \parallel P_2::P_1!5)$. In this case some histories of process 1 contain the no-match claim $<1,2>$ and some of process 2 $<2,1>$. After binding processes 1 and 2, the real-time assumptions concerning the collection of processes $\{1,2\}$ should be checked; in this case, exactly $<1,2>$ and $<2,1>$. After this check they are not needed anymore and will be removed.

In general, for $B \in \mathbf{B}(CAR)$ and a collection of processes $I \in \mathbf{P}(\mathbb{N})$, we can define $RTA_I(B) \in \mathbf{B}(CAR)$ which removes from B the no-match claims concerning I:

$$RTA_I(B)(r) = \begin{cases} 0 & \text{if } r = <i,j> \text{ with } i,j \in I, \\ B(r) & \text{otherwise.} \end{cases}$$

We have to extend this operator to elements of ΣH in the same way as we extended $B[I \rightarrow i]$ to $X[I \rightarrow i]$ (see section 7.4.2):

$$RTA_I(X) = \{<\sigma, <RTA_I(h(k))>_{k=1}^{|h|}> \mid <\sigma, h> \in X\}.$$

Lemma: $RTA_I(X) \in \Sigma H$ for all $X \in \Sigma H$ and $I \in \mathbf{P}(\mathbb{N})$.

Proof: The same as for the lemma in section 7.4.2. ■

7.4.3.6 *Putting it altogether: the meaning of* $(T_1 \parallel T_2)$

$$\mathbf{M}[\![(T_1 \parallel T_2)]\!]s = RTA_{tvs}(\not{c}^{IC}_{tvs}(\{<\sigma_1 \cup_s \sigma_2, h_1 \# h_2> \mid <\sigma_i, h_i> \in \mathbf{M}[\![T_i]\!]s \ \wedge\ h_1 \not{c}^{RT} h_2 \ \wedge\ \sigma_i = \perp \Rightarrow |h_i| \geqslant |h_{3-i}|,\ 1 \leqslant i \leqslant 2\}))$$

where tvs = $VS((T_1 \parallel T_2)) = VS(T_1) \cup VS(T_2)$, the total visible subprocesses.

Lemma: $\{<\sigma_1 \cup_s \sigma_2, h_1 \# h_2> \mid <\sigma_i, h_i> \in \mathbf{M}[\![T_i]\!]s \ \wedge\ h_1 \not{c}^{RT} h_2 \ \wedge\ \sigma_i = \perp \Rightarrow |h_i| \geqslant |h_{3-i}|,\ 1 \leqslant i \leqslant 2\} \in \Sigma H$.

Proof: Abbreviate the above set to X.

X non-empty: $\mathbf{M}[\![T_i]\!]s \in \Sigma H$ implies $<\perp, \lambda> \in \mathbf{M}[\![T_i]\!]s$ $(1 \leqslant i \leqslant 2)$. $\lambda \not{c}^{RT} \lambda$ and $|\lambda| \geqslant |\lambda|$ are satisfied, hence $<\perp \cup_s \perp, \lambda \# \lambda> = <\perp, \lambda> \in X$.

X prefix-closed:
Let $<\sigma_1 \cup_s \sigma_2, h_1 \# h_2> \in X$ and $h' \leqslant h_1 \# h_2$.
The proof splits into two cases, dependent on the length of h':

case 1: $|h'| \leqslant \min\{|h_1|, |h_2|\}$.
Take $h'_i \leqslant h_i$, $|h'_i| = |h'|$ $(1 \leqslant i \leqslant 2)$.
Then $<\perp, h'_i> \in \mathbf{M}[\![T_i]\!]s$ and $h'_1 \not{c}^{RT} h'_2$ and $|h'_i| \geqslant |h'_{3-i}|$ $(1 \leqslant i \leqslant 2)$ and $h'_1 \# h'_2 = h'$,
hence $<\perp \cup_s \perp, h'_1 \# h'_2> = <\perp, h'> \in X$.

case 2: $|h'| > \min\{|h_1|, |h_2|\}$.
From $h' \leqslant h_1 \# h_2$ it follows that $|h'| \leqslant |h_1 \# h_2| = \max\{|h_1|, |h_2|\}$.
Taking these two conditions on $|h'|$ together we see that $|h_1| \neq |h_2|$. Without loss of generality we can suppose $|h_1| > |h_2|$.
Take $h'_1 \leqslant h_1$ with $|h'_1| = |h'|$.
Then $<\perp, h'_1> \in \mathbf{M}[\![T_1]\!]s$ and $<\sigma_2, h_2> \in \mathbf{M}[\![T_2]\!]s$ and $h'_1 \not{c}^{RT} h_2$ and $|h'_1| \geqslant |h_2|$
(and $\sigma_2 \neq \perp$ because $|h_1| > |h_2|$) and $h'_1 \# h_2 = h'$,
hence $<\perp \cup_s \sigma_2, h'_1 \# h_2> = <\perp, h'> \in X$. ■

Proposition: For all $s \in S$, $\mathbf{M}[\![(T_1 \parallel T_2)]\!]s \in \Sigma H$.

Proof: Immediate by the lemma and the fact that both $\not{c}_I^{IC}$ and RTA_I map elements of ΣH to elements of ΣH (see the lemma in section 7.4.3.3, respectively 7.4.3.5). ■

7.5 Properties of the semantics

In this section we derive some general properties of the semantics and use them to prove commutativity and associativity of parallel composition.

We start with a property concerning the records in the histories generated by our semantics: in the records in the histories of the semantics of a command at least one of the processes involved is a visible subprocess of that command.

History Property: For all commands T, $s \in S$, $<\sigma, h> \in M[\![T]\!]s$ the following holds:
$\forall B \leqslant h\ \forall r \in B\ (\pi_1(r) \in VS(T)\ \vee\ \pi_2(r) \in VS(T))$.

Proof: From the definition of $M[\![T]\!]$, by an easy structural induction on T. ■

The following lemma and its corollary concern properties in the context of the parallel composition of T_1, T_2 and T_3 (cf. the end of section 7.4.3.3). The lemma states that under certain conditions (which are met in the case of a parallel composition) three histories can not contain a common communication assumption record. The corollary then says that under the same conditions the special merge # of section 7.4.3.3 is associative.

Main Lemma: Let $s \in S$, $<\sigma_i, h_i> \in M[\![T_i]\!]s$ $(1 \leqslant i \leqslant 3)$ and suppose $VS(T_i) \cap VS(T_j) = \emptyset$ for all i,j, $1 \leqslant i < j \leqslant 3$.
Then for all $r \in CAR$, all k such that $1 \leqslant k \leqslant \min\{|h_i| \mid 1 \leqslant i \leqslant 3\}$ there exists an i, $1 \leqslant i \leqslant 3$, with $h_i(k)(r) = 0$.

Proof: From the History Property and the condition $VS(T_i) \cap VS(T_j) = \emptyset$ $(1 \leqslant i < j \leqslant 3)$ it easily follows that there cannot exist $r \in CAR$ and k, $1 \leqslant k \leqslant \min\{|h_i| \mid 1 \leqslant i \leqslant 3\}$, such that $r \in h_i(k)$ for all i, $1 \leqslant i \leqslant 3$. ■

Corollary: Let $s \in S$, $<\sigma_i, h_i> \in M[\![T_i]\!]s$ $(1 \leqslant i \leqslant 3)$ and suppose $VS(T_i) \cap VS(T_j) = \emptyset$ for all i,j, $1 \leqslant i < j \leqslant 3$.
Then $(h_1 \# h_2) \# h_3 = h_1 \# (h_2 \# h_3)$.

Proof: From the Main Lemma observing that $\big| |k-m| - n \big| = \big| k - |m-n| \big|$ for all $k,m,n \in \mathbb{N}$ such that k=0 or m=0 or n=0. ■

The preceding properties enable us to prove that pairwise binding of processes is independent of the order in which the processes are bound. E.g. for three processes $M[\![((T_1 \parallel T_2) \parallel T_3)]\!]$ equals $M[\![(T_1 \parallel (T_2 \parallel T_3))]\!]$. This associativity property together with commutativity $M[\![(T_1 \parallel T_2)]\!] = M[\![(T_2 \parallel T_1)]\!]$ justifies the writing of $M[\![(T_1 \parallel T_2 \parallel T_3)]\!]$ for any order of binding T_1, T_2 and T_3. This immediately generalizes to $M[\![(T_1 \parallel \cdots \parallel T_n)]\!]$ for any order of binding $T_1, \ldots, T_n$ $(n \geqslant 2)$.

Theorem: $M[\![(T_1 \parallel T_2)]\!] = M[\![(T_2 \parallel T_1)]\!]$ and
$M[\![((T_1 \parallel T_2) \parallel T_3)]\!] = M[\![(T_1 \parallel (T_2 \parallel T_3))]\!]$.

Proof: Commutativity: immediately from the commutativity of $\cup_s$, $\#$ and $\not{c}^{RT}$.
Associativity:
We shall give a meaning to '$M[\![(T_1 \parallel T_2 \parallel T_3)]\!]s$' and show that $M[\![((T_1 \parallel T_2) \parallel T_3)]\!]s$ and $M[\![(T_1 \parallel (T_2 \parallel T_3))]\!]s$ both are equal to it.
Note that in the context of the parallel composition of T_1, T_2 and T_3 (in both orders above), we may assume (see the end of section 7.4.3.3)

(a) $VS(T_i) \cap VS(T_j) = \varnothing$ $(1 \leqslant i < j \leqslant 3)$.

Hence for $s \in S$, $<\sigma_i, h_i> \in M[\![T_i]\!]s$ $(1 \leqslant i \leqslant 3)$ we can apply both the Main Lemma and the Corollary.
Because of associativity of $\cup_s$ and the Corollary we can define $M[\![(T_1 \parallel T_2 \parallel T_3)]\!]s =$

$$RTA_{\overset{3}{\underset{i=1}{\cup}} VS(T_i)} \; (\not{c}^{IC}_{\overset{3}{\underset{i=1}{\cup}} VS(T_i)} \; (\{<\sigma_1 \cup_s \sigma_2 \cup_s \sigma_3, h_1 \# h_2 \# h_3> \mid <\sigma_i, h_i> \in M[\![T_i]\!]s \; (1 \leqslant i \leqslant 3)$$
$$\wedge \; h_i \not{c}^{RT} h_j \; (1 \leqslant i < j \leqslant 3) \; \wedge \; \sigma_i = \perp \Rightarrow |h_i| \geqslant |h_j| \; (1 \leqslant i, j \leqslant 3)\})).$$

Now, for all $s \in S$,
$M[\![((T_1 \parallel T_2) \parallel T_3)]\!]s =$

$$RTA_{\overset{3}{\underset{i=1}{\cup}} VS(T_i)} \; (\not{c}^{IC}_{\overset{3}{\underset{i=1}{\cup}} VS(T_i)} \; (\{<\sigma \cup_s \sigma_3, h \# h_3> \mid$$
$$<\sigma, h> \in RTA_{\overset{2}{\underset{i=1}{\cup}} VS(T_i)} \; (\not{c}^{IC}_{\overset{2}{\underset{i=1}{\cup}} VS(T_i)} \; (\{<\sigma_1 \cup_s \sigma_2, h_1 \# h_2> \mid$$
$$<\sigma_1, h_1> \in M[\![T_1]\!]s \; \wedge \; <\sigma_2, h_2> \in M[\![T_2]\!]s \; \wedge \; h_1 \not{c}^{RT} h_2$$
$$\wedge \; \sigma_1 = \perp \Rightarrow |h_1| \geqslant |h_2| \; \wedge \; \sigma_2 = \perp \Rightarrow |h_2| \geqslant |h_1|\}))$$
$$\wedge \; <\sigma_3, h_3> \in M[\![T_3]\!]s \; \wedge \; h \not{c}^{RT} h_3 \; \wedge \; \sigma = \perp \Rightarrow |h| \geqslant |h_3| \; \wedge \; \sigma_3 = \perp \Rightarrow |h_3| \geqslant |h|\}))$$

$$\overset{(*)}{=} RTA_{\overset{3}{\underset{i=1}{\cup}} VS(T_i)} \; (\not{c}^{IC}_{\overset{3}{\underset{i=1}{\cup}} VS(T_i)} \; (\{<(\sigma_1 \cup_s \sigma_2) \cup_s \sigma_3, (h_1 \# h_2) \# h_3> \mid$$
$$<\sigma_1, h_1> \in M[\![T_1]\!]s \; \wedge \; <\sigma_2, h_2> \in M[\![T_2]\!]s \; \wedge \; <\sigma_3, h_3> \in M[\![T_3]\!]s$$
$$\wedge \; h_1 \not{c}^{RT} h_2 \; \wedge \; (h_1 \# h_2) \not{c}^{RT} h_3 \; \wedge \; \sigma_1 = \perp \Rightarrow |h_1| \geqslant |h_2| \; \wedge \; \sigma_2 = \perp \Rightarrow |h_2| \geqslant |h_1|$$
$$\wedge \; \sigma_1 \cup_s \sigma_2 = \perp \Rightarrow |h_1 \# h_2| \geqslant |h_3| \; \wedge \; \sigma_3 = \perp \Rightarrow |h_3| \geqslant |h_1 \# h_2|\}))$$

$$\overset{(**)}{=} M[\![(T_1 \parallel T_2 \parallel T_3)]\!]s \overset{(***)}{=} M[\![(T_1 \parallel (T_2 \parallel T_3))]\!]s$$

where the three crucial steps are explained by

(*) we can leave out the operators $\mathrm{RTA}_{\bigcup_{i=1}^{2} VS(T_i)}$ and $\not c^{IC}_{\bigcup_{i=1}^{2} VS(T_i)}$ because they only concern records with $\pi_1(r) \in \bigcup_{i=1}^{2} VS(T_i)$ and $\pi_2(r) \in \bigcup_{i=1}^{2} VS(T_i)$. Because of the History Property and (a) it follows that such records r cannot occur in h_3. This implies that such records do not interfere with records of h_3, e.g., such records are maintained in the merge $h \# h_3$. The effect of the two above operators is then contained in the effect of $\mathrm{RTA}_{\bigcup_{i=1}^{3} VS(T_i)}$ and $\not c^{IC}_{\bigcup_{i=1}^{3} VS(T_i)}$ since $\bigcup_{i=1}^{2} VS(T_i) \subsetneq \bigcup_{i=1}^{3} VS(T_i)$.

(**) this holds because of

(1) associativity of $\cup_s$ and the Corollary.

(2) $(h_1 \# h_2) \not c^{RT} h_3 \Leftrightarrow (h_1 \not c^{RT} h_3 \wedge h_2 \not c^{RT} h_3)$:

$\Leftarrow$: easy because $r \in (h_1 \# h_2)(k)$ implies that $r \in h_i(k)$ for i=1 or i=2

$\Rightarrow$: according to the Main Lemma $r \in h_3(k)$ and $r \in h_i(k)$ $(1 \leqslant i \leqslant 2)$ implies that $k > |h_{3-i}|$ or that $h_{3-i}(k)(r) = 0$; in both cases $r \in (h_1 \# h_2)(k)$.

(3) $(\sigma_1 \cup_s \sigma_2) = \perp \Leftrightarrow (\sigma_1 = \perp \vee \sigma_2 = \perp)$ and $|h_1 \# h_2| = \max\{|h_1|, |h_2|\}$.

(***) the previous equations hold as well when h_1, h_2 and h_3 are interchanged. ■

7.6 *Concluding remarks*

The proposition in section 7.2 shows that we do not have to worry about continuity of the meaning function.

After all these technicalities the next section gives some examples which illustrate the basic ideas, and illustrate what is observable.

8. EXAMPLES

In the examples below E_n abbreviates the program (fragment) of example n and s is an arbitrary element of S.

Example 1: $E_1 \equiv P_1::*[\,\mathbf{true} \rightarrow P_2!5]$.

First we compute

$$M[\![\,[\mathbf{true} \rightarrow P_2!5]]\!]s = \bigcup_{j=1}^{1} M[\![P_2!5]\!]^*(G[\![\mathbf{true},\varnothing]\!]s) = M[\![P_2!5]\!]^*\,(PFC(\{<s,\lambda>\})) \overset{(*)}{=} M[\![P_2!5]\!]s$$

$$= G[\![P_2!5,\varnothing]\!]s = PFC(\{<s,<[<0,2>]>^t \,\hat{}\, <[<0,2,5>]>>\mid t\in \mathbb{N}\}).$$

(*) in general, by writing out the definitions of section 7.2, we see that $\phi^*(PFC(\{<s,\lambda>\})) = \phi(s)$.

Then

$$M[\![*[\mathbf{true}\rightarrow P_2!5]]\!]s = \bigcup_{i\in\mathbb{N}}\phi_i(s) \text{ where } \phi_0(s) = \{<\perp,\lambda>\},\ \phi_{i+1}(s) = \phi_i^*\,(M[\![[\mathbf{true}\rightarrow P_2!5]]\!]s).$$

By induction we can prove for all $n\in\mathbb{N}$

$$\phi_n(s) = PFC(\{<\perp,<[<0,2>]>^{t_1}\,\hat{}\,<[<0,2,5>]>\,\hat{}\ldots\hat{}\,<[<0,2>]>^{t_n}\,\hat{}\,<[<0,2,5>]>>\mid t_1,\ldots,t_n\in\mathbb{N}\}).$$

Hence

$$M[\![E_1]\!]s = PFC(\{<\perp,<[<1,2>]>^{t_1}\,\hat{}\,<[<1,2,5>]>\,\hat{}\ldots\hat{}\,<[<1,2>]>^{t_n}\,\hat{}\,<[<1,2,5>]>>\mid n\in\mathbb{N},\ t_1,\ldots,t_n\in\mathbb{N}\}).$$

Remark: This example shows why elements of ΣH should be non-empty. Otherwise $\varnothing$ would be the least element of ΣH and for the ϕ_i above we would then get $\phi_n(s) = \varnothing$ for all $n\in\mathbb{N}$ and hence $M[\![E_1]\!]s = \varnothing$. This is caused by the fact that we should have a starting point for the histories and $\varnothing$ contains no histories at all.

Example 2: $E_2 \equiv P_1::(P_{11}::*[P_2!5 \rightarrow \mathbf{wait}\ 1 \,\square\, \mathbf{wait}\ 1 \rightarrow \mathbf{wait}\ 1] \parallel$
$P_{12}::\mathbf{wait}\ 1;\ *[P_2!5 \rightarrow \mathbf{wait}\ 1 \,\square\, \mathbf{wait}\ 1 \rightarrow \mathbf{wait}\ 1])$.

We should have $M[\![E_1]\!] = M[\![E_2]\!]$!

E_1 and E_2 have indeed the same observable behaviour: they both continuously try to output value 5 to process 2.

Let C abbreviate $[P_2!5 \rightarrow \mathbf{wait}\ 1 \,\square\, \mathbf{wait}\ 1 \rightarrow \mathbf{wait}\ 1]$

(then $E_2 \equiv P_1::(P_{11}::*C \parallel P_{12}::\mathbf{wait}\ 1;\ *C)$).

We first compute

$$M[\![C]\!]s = M[\![\mathbf{wait}\ 1]\!]^*\,(G[\![P_2!5,\{\mathbf{wait}\ 1\}]\!]s) \cup M[\![\mathbf{wait}\ 1]\!]^*\,(G[\![\mathbf{wait}\ 1,\{P_2!5\}]\!]s)$$

$= \mathbf{M}[\![\mathbf{wait\ 1}]\!]^* (PFC(\{<s,<[<0,2>]>^{t\frown} <[<0,2,5>]>> \mid 0 \leq t < 1\}))$
$\cup\ \mathbf{M}[\![\mathbf{wait\ 1}]\!]^* (PFC(\{<s,<[<0,2>]>^{1}>\}))$
$= PFC(\{<s,<[<0,2,5>],[\,]>>\}) \cup PFC(\{<s,<[<0,2>],[\,]>>\})$.

Then

$\mathbf{M}[\![*C]\!]s = \bigcup_{i \in \mathbb{N}} \phi_i(s)$ where $\phi_0(s) = \{<\perp,\lambda>\}, \phi_{i+1}(s) = \phi_i^* (\mathbf{M}[\![C]\!]s)$.

By induction we can prove for all $n \in \mathbb{N}$

$\phi_n(s) = PFC(\{<\perp,<[r_1],[\,]>^{\frown}\ldots^{\frown}<[r_n],[\,]>> \mid \forall i,\ 1 \leq i \leq n,\ r_i = <0,2,5> \ \vee\ r_i = <0,2>\})$.

Hence $\mathbf{M}[\![P_{11}{::}*C]\!]s =$

$PFC(\{<\perp,<[r_1],[\,]>^{\frown}\ldots^{\frown} <[r_n],[\,]>> \mid n \in \mathbb{N},\ \forall i,\ 1 \leq i \leq n,\ r_i = <11,2,5> \ \vee\ r_i = <11,2>\})$

and

$\mathbf{M}[\![P_{12}{::}\mathbf{wait\ 1};*C]\!]s = (\mathbf{M}[\![*C]\!]^* (\mathbf{M}[\![\mathbf{wait\ 1}]\!]s))[\{0\} \rightarrow 12] =$

$PFC(\{<\perp,<[\,]> {}^{\frown} <[r_1],[\,]> {}^{\frown}\ldots^{\frown}<[r_n],[\,]>> \mid$
$n \in \mathbb{N},\ \forall i,\ 1 \leq i \leq n,\ r_i = <12,2,5> \ \vee\ r_i = <12,2>\})$.

Next we compute the parallel composition of $P_{11}{::}*C$ and P_{12} :: **wait 1**; $*C$:

$\mathbf{M}[\![(P_{11}{::}*C \parallel P_{12}{::}\mathbf{wait\ 1};\ *C)]\!]s = RTA_{\{11,12\}} (\phi^{IC}_{\{11,12\}} (\{<\sigma_1 \cup_s \sigma_2,\ h_1 \# h_2> \mid$
$<\sigma_1,h_1> \in \mathbf{M}[\![P_{11}{::}*C]\!]s \ \wedge\ <\sigma_2,h_2> \in \mathbf{M}[\![P_{12}{::}\mathbf{wait\ 1};*C]\!]s$
$\wedge\ h_1 \not\subset^{RT} h_2 \ \wedge\ \sigma_i = \perp \Rightarrow |h_i| \geq |h_{3-i}|,\ 1 \leq i \leq 2\}))$

$= \{<\perp,<[r_i]>_{i=1}^{n}> \mid n \in \mathbb{N}, \forall i,\ 1 \leq i \leq n,\ odd(i) \Rightarrow (r_i = <11,2,5> \ \vee\ r_i = <11,2>)$
$\wedge\ even(i) \Rightarrow (r_i = <12,2,5> \ \vee\ r_i = <12,2>)\}$.

Hence $\mathbf{M}[\![E_2]\!]s = \{<\perp,\ <[r_i]>_{i=1}^{n}> \mid n \in \mathbb{N}, \forall i,\ 1 \leq i \leq n,\ r_i = <1,2,5> \ \vee\ r_i = <1,2>\}$.

That $\mathbf{M}[\![E_1]\!] = \mathbf{M}[\![E_2]\!]$ holds, can be easily seen by an analogy with formal language theory: $Prefixes((b^*a)^*) = (a \cup b)^*$.

Remark: These two examples illustrate that established deadlock is just a special case of an infinite computation (and is not distinguishable from other infinite computations such as divergence: see the end of section 6.2): E_1 deadlocks when process 2 from some point on does not ask for a value to be input from process 1; in the same context E_2 behaves more or less as 'busy waiting' which is another form of infinite computation.

Example 3: $E_3 \equiv (P_1{::}(P_{11}{::}P_2!3 \parallel P_{12}{::}P_2!7) \parallel P_2{::}P_1?x)$.

We should get an infinite computation, in this case an established deadlock of either P_{11} or P_{12} after the succesful communication of the other with P_2.

First compute

$\mathbf{M}[\![P_{11}{::}P_2!3]\!]s = PFC(\{<s,<[<11,2>]>^{t\frown} <[<11,2,3>]>> \mid t \in \mathbb{N}\})$,

$\mathbf{M}[\![P_{12}{::}P_2!7]\!]s = PFC(\{<s,<[<12,2>]>^{t\frown} <[<12,2,7>]>> \mid t \in \mathbb{N}\})$ and

$\mathbf{M}[\![P_2::P_1?x]\!]s = PFC(\{<s[v/x], <[<1,2>]>^t \ \hat{} <[<1,2,v>]>> \mid v \in V, t \in \mathbb{N}\})$.

Next

$\mathbf{M}[\![(P_{11}::P_2!3 \parallel P_{12}::P_2!7)]\!]s = RTA_{\{11,12\}}(\cancel{c}^{IC}_{\{11,12\}}(\{<\sigma_1 \cup_s \sigma_2, h_1 \# h_2> \mid$
$<\sigma_1, h_1> \in \mathbf{M}[\![P_{11}::P_2!3]\!]s \wedge <\sigma_2, h_2> \in \mathbf{M}[\![P_{12}::P_2!7]\!]s$
$\wedge\ h_1 \cancel{c}^{RT} h_2 \wedge \sigma_i = \perp \Rightarrow |h_i| \geq |h_{3-i}|, 1 \leq i \leq 2\}))$

$= PFC(\{<s, <[<11,2>, <12,2>]>^{t_1} \ \hat{} <[<11,2,3>, <12,2>]> \ \hat{}$
$<[<12,2>]>^{t_2} \ \hat{} <[<12,2,7>]>> \mid t_1, t_2 \in \mathbb{N}\}$
$\cup \{<s, <[<11,2>, <12,2>]>^t \ \hat{} <[<11,2,3>, <12,2,7>]>> \mid t \in \mathbb{N}\}$
$\cup \{<s, <[<11,2>, <12,2>]>^{t_1} \ \hat{} <[<11,2>, <12,2,7>]> \ \hat{}$
$<[<11,2>]>^{t_2} \ \hat{} <[<11,2,3>]>> \mid t_1, t_2 \in \mathbb{N}\})$.

Hence

$\mathbf{M}[\![P_1::(P_{11}::P_2!3 \parallel P_{12}::P_2!7)]\!]s =$
$PFC(\{<s, <[<1,2>^2]>^{t_1} \ \hat{} <[<1,2,3>, <1,2>]> \ \hat{}$
$<[<1,2>]>^{t_2} \ \hat{} <[<1,2,7>]>> \mid t_1, t_2 \in \mathbb{N}\}$
$\cup \{<s, <[<1,2>^2]>^t \ \hat{} <[<1,2,3>, <1,2,7>]>> \mid t \in \mathbb{N}\}$
$\cup \{<s, <[<1,2>^2]>^{t_1} \ \hat{} <[<1,2>, <1,2,7>]> \ \hat{}$
$<[<1,2>]>^{t_2} \ \hat{} <[<1,2,3>]>> \mid t_1, t_2 \in \mathbb{N}\})$.

Note that here the use of bags instead of sets is essential, especially if we replace 3 and 7 both by the same value.

Then

$\mathbf{M}[\![E_3]\!]s = RTA_{\{1,2\}}(\cancel{c}^{IC}_{\{1,2\}}(\{<\sigma_1 \cup_s \sigma_2, h_1 \# h_2> \mid <\sigma_1, h_1> \in \mathbf{M}[\![P_1::(P_{11}::P_2!3 \parallel P_{12}::P_2!7)]\!]s$
$\wedge <\sigma_2, h_2> \in \mathbf{M}[\![P_2::P_1?x]\!]s \wedge h_1 \cancel{c}^{RT} h_2 \wedge \sigma_i = \perp \Rightarrow |h_i| \geq |h_{3-i}|, 1 \leq i \leq 2\}))$
$= RTA_{\{1,2\}}(PFC(\{<\perp, <[<1,2>]> \ \hat{} <[<1,2>]>^{t_2}> \mid t_2 \in \mathbb{N}\}))$
$= \{<\perp, <[\ ]>^t> \mid t \in \mathbb{N}\}$.

Example 4: $E_4 \equiv (P_1::(P_{11}::P_2!3 \parallel P_{12}::P_2!7) \parallel P_2::P_1?x; P_1?x)$.

In this example one of the processes 11 and 12 first communicates with P_2 and then the other. The total program terminates in two time units. For $\mathbf{M}[\![P_1::(P_{11}::P_2!3 \parallel P_{12}::P_2!7)]\!]s$ see example 3.

Furthermore

$\mathbf{M}[\![P_2::P_1?x; P_1?x]\!]s =$
$PFC(\{<s[v_2/x], <[<1,2>]>^{t_1} \ \hat{} <[<1,2,v_1>]> \ \hat{}$
$<[<1,2>]>^{t_2} \ \hat{} <[<1,2,v_2>]>> \mid v_1, v_2 \in V, t_1, t_2 \in \mathbb{N}\})$.

Then

$\mathbf{M}[\![E_4]\!]s = RTA_{\{1,2\}}(PFC(\{<s[7/x], <[<1,2>], [\ ]>>\} \cup \{<s[3/x], <[<1,2>], [\ ]>>\})) =$
$PFC(\{<s[v/x], <[\ ]>^2> \mid v \in \{3,7\}\})$.

Example 5: $E_5 \equiv (P_1::(P_{11}::P_2!3 \parallel P_{12}::P_2!7) \parallel P_2::(P_{21}::P_1?x \parallel P_{22}::P_1?y))$.

In this example processes 11 and 12 communicate *simultaneously* with processes 21 and 22. The total program terminates in one time unit.

For $\mathbf{M}[\![P_1::(P_{11}::P_2!3 \parallel P_{12}::P_2!7)]\!]s$ see example 3.

Similarly we can compute

$$\mathbf{M}[\![P_2::(P_{21}::P_1?x \parallel P_{22}::P_1?y)]\!]s =$$

$$\begin{aligned}
&PFC\,(\{<s[v_1/x][v_2/y],<[<1,2>^2]>^{t_1} \char`^ <[<1,2,v_1>,<1,2>]> \char`^ \\
&\qquad <[<1,2>]>^{t_2} \char`^ <[<1,2,v_2>]>> \mid v_1,v_2 \in V,\ t_1,t_2 \in \mathbb{N}\} \\
&\quad \cup \{<s[v_1/x][v_2/y],<[<1,2>^2]>^{t} \char`^ <[<1,2,v_1>,<1,2,v_2>]>> \mid v_1,v_2 \in V,\ t \in \mathbb{N}\} \\
&\quad \cup \{<s[v_1/x][v_2/y],<[<1,2>^2]>^{t_1} \char`^ <[<1,2>,<1,2,v_2>]> \char`^ \\
&\qquad <[<1,2>]>^{t_2} \char`^ <[<1,2,v_1>]>> \mid v_1,v_2 \in V,\ t_1,t_2 \in \mathbb{N}\}).
\end{aligned}$$

Then

$$\mathbf{M}[\![E_5]\!]s =$$

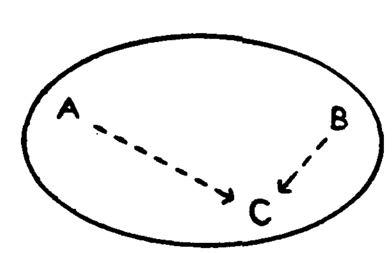
$$RTA_{\{1,2\}}\,(PFC\,(\{<s[v_1/x][v_2/y],<[\,]>> \mid (v_1 = 3 \wedge v_2 = 7) \vee (v_1 = 7 \wedge v_2 = 3)\})) =$$

$$PFC\,(\{<s[v_1/x][v_2/y],<[\,]>> \mid (v_1 = 3 \wedge v_2 = 7) \vee (v_1 = 7 \wedge v_2 = 3)\}).$$

9. REAL-TIME MODELS

9.1 Introduction

The maximal parallelism model as used here, is flawed by some conceptual problems. We illustrate these problems with an example.



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, imposing a fcfs-principle upon the order of communications induces non-trivial requirements upon an underlying communication layer; requirements that we would like not to make. Similar problems occur if processors communicate, e.g., via a common bus where assumptions about bus-arbitration have to be taken into account.

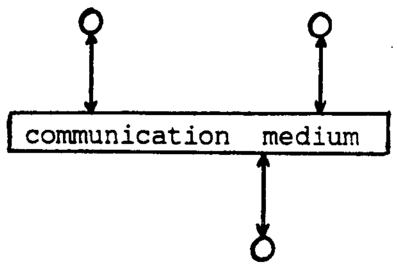
The lesson that should be drawn from this example is, that whereas our current model applies the fcfs-principle to the order of initiations of requests, the principle should rather be applied to the order in which a process becomes aware of requests. In doing so, we create the freedom to relax the stringent impositions of the original model on the behaviour of a communication layer. Specifically, in this way it becomes possible to vary the time gap (0 in the original model) between the initiation and receipt of a communication request, which reflects the uncertainties about the communication layer.

This variation of the time gap is the essential feature of the $MAX_\gamma(\delta,\epsilon)$ model of distributed concurrency. The parameters δ and ϵ function as lower and upper bound on the above time gaps which are allowed to take on any value inbetween these bounds. As a consequence, communications that are initiated too close in time (relative to a global clock) cannot be temporally ordered anymore. These time bounds may be interpreted as an abstraction of the propagation delays within some communication layer. The third parameter, γ, of the model is used to extend communications in time and denotes the number of time units it takes.

9.2 $MAX_\gamma(\delta,\epsilon)$ *model of concurrency*

The model is based on the Salwicki/Müldner maximal parallelism model: there is no unnecessary waiting between the execution of actions. Communication between processes is served on a first-come first-served basis.
Additionally, the following model pertains to process-communication:

- processes communicate via a medium,



- it takes between δ and ϵ time units (ϵ *not* included) for the medium to become aware of a process expressing its willingness to communicate or withdrawing its willingness (time-out),

- communication between two processes only occurs after the medium has become aware of both processes' willingness,

- a communication takes an additional γ time units during which period the processes remain synchronized,

- a communication that is in progress at a time when the medium receives a time-out from one of the participating processes, will be completed; a communication that might be started at such a time, will not be executed.

Remarks:

- Communication always takes at least $\delta+\gamma$ time units.

- $MAX_0(0,2) \not\models \{true\}\ (P_1::P_2?x;P_2?y \parallel P_2::(P_{21}::P_1!1 \parallel P_{22}::\mathbf{wait}\ 1;P_1!2))\ \{x=1\}$, and
$MAX_0(0,1) \models \{true\}\ (P_1::P_2?x;P_2?y \parallel P_2::(P_{21}::P_1!1 \parallel P_{22}::\mathbf{wait}\ 1;P_1!2))\ \{x=1\}$.
In other words, there is an uncertainty interval of $\epsilon-\delta$: if requests for communication are initiated $\epsilon-\delta$ or more time units apart, the first request will indeed be served first; if, on the other hand, these requests are initiated within this interval, the order in which these requests are served is undefined.

- $MAX_0(0,1)$ gives rise to pure maximal parallelism;
$MAX_0(0,\infty)$ to pure interleaving semantics (with respect to the communication actions). It is to have the latter correspondence that the medium has to become aware of requests *within* ϵ time units. Otherwise, $MAX_0(0,\infty)$ would allow infinite delays.

10. REAL-TIME SEMANTICS FOR MINI CSP-R

The $MAX_\gamma(\delta,\epsilon)$ model only influences the semantics of communication actions. So, the definition of the auxiliary function G has to change, but no additional changes are needed in the definition of M. The intention of these changes is to have G "generate" any history that is consistent with the parameters of the model. As these (additional) consistency-requirements are a purely local affair, the parallel composition of processes requires no additional effort.
Consider an I/O command. The changes in the sets of generated histories that $MAX_\gamma(\delta,\epsilon)$ induces are three-fold:

1. histories must be generated in which the first waiting-action (i.e., the first no-match claim record) occurs τ time units later than the time at which communication was requested; and this for any τ such that $\delta \leq \tau < \epsilon$,

2. in no history can communication or waiting start within δ time units of the request,

3. communication takes γ time units; this is modelled by having the associated communication claim record mark the time, in a history, at which communication starts and by appending empty bags to trace out γ time units.

The changes to **G** are complicated by the necessity of applying the above considerations to every I/O command in the environment (i.e., in the selection or repetition).

Hence, to take care of the first point above, the basic idea is to associate with an environment $\{g_1,\dots,g_n\}$ a set of times $\{t_1,\dots,t_n\}$ such that $\delta \leqslant t_i < \epsilon$. These times represent the delays of the first waiting action for the corresponding guards, i.e., the delays until the medium becomes aware of the corresponding requests. One such choice corresponds to one possible history. To generate the corresponding sequences of bags of no-match claim records, we introduce two auxiliary functions:

Definition: For sets of guards G and times (i.e., natural numbers) T, time t and state $s \in \mathbf{S}$, define

- $A(G,T,t) = \{g_i \in G \mid t_i < t,\ 1 \leqslant i \leqslant n\}$,
  where $\{g_1,\dots,g_n\} \subseteq G$ are the I/O guards in G and $T = \{t_1,\dots,t_n\}$.

- $Ext(G,T,t,s) = \langle RTA(A(G,T,k),s)\rangle_{k=1}^{t}$.

$Ext(\{\alpha_1,\dots,\alpha_n\},\{t_1,\dots,t_n\},t,s)$ yields a sequence of bags of no-match claim records for the I/O commands $\alpha_1,\dots,\alpha_n$. The time t_i represents the delay of the first waiting action (i.e., no-match claim record) for α_i; t is the time at which communication or a time-out occurs. The function A is auxiliary to Ext.

Now, we are ready to define **G** (terminology as in section 7):

$$\mathbf{G}[\![b,A]\!]s = \begin{cases} PFC(\{\langle s,\lambda\rangle\}) & \text{if } \mathbf{W}[\![b]\!]s, \\ \{\langle \bot,\lambda\rangle\} & \text{otherwise.} \end{cases}$$

$$\mathbf{G}[\![\mathbf{wait}\ d,A]\!]s = PFC(\{\langle s,Ext(A,T,t+\tau,s)\rangle \mid \max\{\mathbf{V}[\![d]\!]s,1\} = minwait(A \cup \{\mathbf{wait}\ d\},s) \overset{\text{def.}}{=} t,$$
$$T \overset{\text{def.}}{=} \{t_1,\dots,t_n\},\ \delta \leqslant t_i < \epsilon\ (1 \leqslant i \leqslant n),\ \delta \leqslant \tau < \epsilon\}),$$
where n is the number of I/O guards in A.

$$\mathbf{G}[\![P_j!e,A]\!]s = PFC(\{\langle s,Ext(GRDS,T,t,s)\hat{\ }\langle[\langle 0,j,\mathbf{V}[\![e]\!]s\rangle]\rangle\hat{\ }\langle[]\rangle^{\gamma}\rangle \mid$$
$$\delta \leqslant t < minwait(A,s)+\epsilon-1,\ T \overset{\text{def.}}{=} \{t_1,\dots,t_n\},\ \delta \leqslant t_i < \epsilon\ (1 \leqslant i \leqslant n)\}),$$
where $GRDS = A \cup \{P_j!e\}$ and n is the number of I/O guards in GRDS.

The upperbound on t takes the delay of the arrival of the time-out message in the medium into account. The '-1' factor corresponds to the fact that the medium becomes aware of requests *before* ϵ time units have elapsed.

$$\mathbf{G}[\![P_j?x,A]\!]s = PFC(\{<s[v/x],Ext(GRDS,T,t,s){}^\frown<[<j,0,v>]>{}^\frown\ <[]>^{\gamma}> \mid$$
$$v\in V,\ \delta\leqslant t<minwait(A,s)+\epsilon-1,\ T \overset{def.}{=} \{t_1,\ldots,t_n\},\ \delta\leqslant t_i<\epsilon\ (1\leqslant i\leqslant n)\}),$$

where $GRDS = A\cup\{P_j?x\}$ and n is the number of I/O guards in GRDS.

$\mathbf{G}[\![b;g,A]\!]s = \mathbf{G}[\![g,A]\!]^*\ (\mathbf{G}[\![b,A]\!]s)$, where g is either a pure I/O guard or a pure wait guard.

We illustrate these equations by the example in the second remark of section 9.2.
Let $P \equiv (P_1::P_2?x;P_2?y \parallel P_2::(P_{21}::P_1!1 \parallel P_{22}::\mathbf{wait}\ 1;P_1!2))$.
We claim that $MAX_0(0,2) \not\models \{true\}\ P\ \{x=1\}$ but $MAX_0(0,1) \models \{true\}\ P\ \{x=1\}$. In other words, we claim that $MAX_0(0,2)$ allows computations in which P_{22} communicates first, that are disallowed by $MAX_0(0,1)$.
So, assume $\gamma=0,\ \delta=0,\ \epsilon=2$:

$\mathbf{M}[\![P_1!2]\!]s = \mathbf{G}[\![P_1!2,\varnothing]\!]s = PFC(\{<s,Ext(\{P_1!2\},\{t_1\},t,s){}^\frown<[<0,1,2>]>> \mid t\in\mathbb{N},\ 0\leqslant t_1\leqslant 1\})$.
Now, $Ext(\{P_1!2\},\{0\},t,s) = <RTA(\{P_1!2\},s)>^{t} = <[<0,1>]>^{t},\ t\in\mathbb{N}$,
$Ext(\{P_1!2\},\{1\},0,s) = \lambda$, and
$Ext(\{P_1!2\},\{1\},t,s) = <RTA(\varnothing,s)>{}^\frown<RTA(\{P_1!2\},s)>^{t-1} = <[]>{}^\frown\ <[<0,1>]>^{t-1}, t>0.$
Hence
$\mathbf{M}[\![P_1!2]\!]s = PFC(\{<s,<[]>^{\tau}{}^\frown<[<0,1>]>^{t}{}^\frown<[<0,1,2>]>> \mid 0\leqslant\tau\leqslant 1,\ t\in\mathbb{N}\})$.
Analogously, we obtain the semantics of $P_1!1$ and of the input commands of P_1. Moreover, $\mathbf{M}[\![\mathbf{wait}\ 1]\!]s = PFC(\{<s,<[]>^{\tau}> \mid 1\leqslant\tau\leqslant 2\})$, hence
$\mathbf{M}[\![P_{21}::P_1!1]\!]s = PFC(\{<s,<[]>^{\tau_{21}}{}^\frown<[<21,1>]>^{t_{21}}{}^\frown<[<21,1,1>]>> \mid 0\leqslant\tau_{21}\leqslant 1,\ t_{21}\in\mathbb{N}\})$
$\mathbf{M}[\![P_{22}::\mathbf{wait}\ 1;P_1!2]\!]s = PFC(\{<s,<[]>^{\tau_{22}}{}^\frown<[<22,1>]>^{t_{22}}{}^\frown<[<22,1,2>]>> \mid 1\leqslant\tau_{22}\leqslant 3,\ t_{22}\in\mathbb{N}\})$
$$\mathbf{M}[\![P_1::P_2?x;P_2?y]\!]s = PFC(\{<s[v_1/x][v_2/y],<[]>^{\tau_{11}}{}^\frown<[<2,1>]>^{t_{11}}{}^\frown<[<2,1,v_1>]>{}^\frown<[]>^{\tau_{12}}{}^\frown$$
$$<[<2,1>]>^{t_{12}}{}^\frown<[<2,1,v_2>]>> \mid 0\leqslant\tau_{11},\tau_{12}\leqslant 1,\ t_{11},t_{12}\in\mathbb{N},\ v_1,v_2\in V\}).$$

Consider the histories for P_{21} and P_{22} in which $\tau_{21} = \tau_{22} = t_{21} = 1,\ t_{22} = 0$. In particular consider P_{21}'s history $<[],[<21,1>],[<21,1,1>]>$ and P_{22}'s history $<[],[<22,1,2>]>$. These compatible histories yield the following history for P_2: $<[],[<2,1>,<2,1,2>],[<2,1,1>]>$. This is compatible with P_1's history $<[],[<2,1,2>],[<2,1,1>]>$, obtained by taking $\tau_{11} = 1, \tau_{12} = t_{11} = t_{12} = 0, v_1 = 2, v_2 = 1$. From these two histories we can compute the following element in the denotation for P: $<s[2/x][1/y],<[]>^{3}>$. To show that this computation cannot be generated by the $MAX_0(0,1)$-model (i.e., the maximal parallelism model, as used in section 7) is straightforward: now, choosing $\tau_{11} = \tau_{21} = 1$ is illegal (cf. example 4 in section 8).

11. CONCLUSIONS

We have given a denotational semantics for real-time distributed computing stressing:

(1) compositionality, thus supplying a basis for compositional specification and verification techniques.

(2) a model of concurrency that is realistic, in contrast with interleaving, in the context of real-time: the maximal parallelism model,

(3) simplicity by basing our techniques upon the linear history semantics for CSP of Francez et al.

We feel that our way of dealing with real-time is particularly simple. Timing aspects of programs relate to the length of the histories. Maximal parallelism constraints are made explicit by recording not only the occurrence of communications but also the act of waiting for one. When binding two processes, these constraints imply that at no instant of time both processes are waiting for a mutual communication.

Exact clocking of instructions is unrealistic because then all actions can be exactly determined in time. In a shared variables context, this would imply that mutual exclusion, for example, could be programmed without any additional means such as semaphores. This is resolved in Milner's SCCS by introducing the nondeterministic but bounded wait synchronization primitive δ *which may violate* the maximal parallelism constraints. In our set-up, however, shared variables are excluded, so the mutual exclusion anomaly above does not occur. Additionally, by extending the maximal parallelism model by introducing non-deterministic intervals modelling synchronization delays, again this anomaly disappears. Joe Halpern et al. arrived independently at the same extended model, in their case to achieve coordinated actions in a real-time distributed system [HMM85]. This extension furthermore shows that our techniques can easily accomodate more detailed real-time features. Another example of this is modelling the drifting of local clocks. Since only initial and final states and histories are observable, we hope that exact clocking of instructions together with the extension of the maximal parallelism model result in a realistic simplification of the phenomena inherent in the description of real-time *distributed* computing.

We based our research on CSP-R, a language that captures the essential real-time features of Ada, as supported by the simulation of Ada by CSP-R in Appendix A2. In fact, we had to solve three problems: Firstly, how to model maximal concurrency in a compositional way. Secondly, how to deal with CSP's particular form of naming communication partners, i.e. of process-naming. The latter is a non-trivial problem and its solution definitely complicates our semantics: the use of bags instead of sets in our histories and many of the complications in parallel composition are a direct consequence of it. Thirdly, the rather peculiar semantics of Ada's delay guards, as occurring in e.g. selective waits with delay statement **delay** 0. Our ideas about modelling maximal parallelism are independent of this and, we claim, are of general applicability. This is illustrated by [Ger 85] in which a formal semantics for (recursive) Occam is given, that is surprisingly simple because of the much cleaner communication mechanism of Occam, using communication channels between pairs of processes.

There is a clear correspondence between the readiness semantics of CSP (see [HH 83]) and ours: our sets of no-match claim records - like the ready sets - record the disposition to participate in certain communications. There is also a clear difference, since unlike ready sets, a no-match claim record witnesses such a disposition at only one time instant and does not imply anything about future behaviour. Since dispositions change over time this means that we have to record such dispositions at every time instant. There is also a difference in use since apart from detecting deadlock, no-match claim records are also used to enforce maximal parallelism.

Certain aspects which cause the readiness model to be not fully abstract, thus leading to the failure set model (see [BHR 84]), are also present in our model:
Our semantics differentiates the two program fragments
[**true** → P_1!0; **wait** 1 □ **true** → P_2!0; **wait** 1] and
[**true** → P_1!0; **wait** 1 □ **true** → P_2!0; **wait** 1 □ **true** → [P_1!0 → **wait** 1 □ P_2!0 → **wait** 1]],
although their observable behaviour is the same.
In [GHR 86] the authors develop a fully abstract version of our semantics for an Occam-like language and give a proof of full abstractness. Like for the ready set semantics, full abstraction is attained by an "upward closure" operation on the no-match claim records. In [BG 86] the resulting model is investigated and developed as an extension of the failure set model. In fact, independently from us, Andy Boucher ([B 86]) developed quite similar techniques to give denotational semantics to Occam.

Having discovered on a semantic level how to reason compositionally about maximal parallelism we now have a firm basis for developing compositional specification and verification methods. In fact, the present paper laid the foundation for our participation in ESPRIT project 937: Debugging and Specification of Ada Real-Time Embedded Systems (DESCARTES). Some of the topics that will be addressed in that context are:

- developing a syntax-directed specification language and corresponding proof system based on the fully abstract semantics of [GHR 86].

- developing a fully abstract temporal logic for real-time distributed computing.

- specializing these specification languages and proof systems to a real-time fragment of Ada and to Occam (through incorporating local clocks).

A first result to use our compositional semantics to get a compositional proof system, generalizing the work of Zwiers et al. ([ZRB 85]) to real-time, is represented by [Hoo 86].

ACKNOWLEDGEMENTS

We are indebted to the second author, who, during a four-month visit to the universities of Nijmegen and Utrecht in the fall of 1983, started this research by writing [SR 83].
Our thanks goes to Amir Pnueli, who assisted at several occasions in correcting and improving previous versions of this paper, and to the two referees for their comments and improvements.
The Netherlands Organization for the Advancement of Pure Research (ZWO) is thanked for support of three of the authors.
Finally, we thank Mijem Tosendjojo and Edmé van Thiel-Niekoop for their assistance with the typing of this paper.

REFERENCES

[Ada 83] *The programming language Ada. Reference manual.* LNCS 155, Springer, 1983.

[B 86] A. Boucher. D. Phil. thesis, Department of Computer Science, University of Oxford, 1986.

[BG 86] A. Boucher, R. Gerth. *A Timed Failures Model for Communicating Sequential Processes.* Draft, 1986.

[BH 81] A. Bernstein, P.K. Harter jr. *Proving Real-time Properties of Programs with Temporal Logic.* 8th ACM SOSP, pp. 1-11, 1981.

[BHR 84] S.D. Brookes, C.A.R. Hoare, A.W. Roscoe. *A theory of Communicating Sequential Processes.* JACM 31-3, pp. 560-599, July 1984.

[BKP 84] H. Barringer, R. Kuiper, A. Pnueli. *Now You May Compose Temporal Logic Specifications.* 16th ACM STOC, pp. 51-63, 1984.

[BLW 82] P. Branquart, G. Louis, P. Wodon. *An Analytical Description of CHILL, the CCITT High Level Language VI*, LNCS 128, Springer, 1982.

[BO 80] D. Bjørner, O.N. Oest (eds.). *Towards a Formal Description of Ada.* LNCS 98, Springer, 1980.

[CACM 84] *A Case Study: The Space Shuttle Software System*. CACM 27-9, 1984.

[Cam 82] J. Camerini. *Semantique Mathematique de Primitives Temps Reel*. These de 3 eme cycle, IMA, Université de Nice, 1982.

[Dij 59] E.W. Dijkstra. *Communication with an automatic computer*. Ph.D. thesis, Mathematical Centre, Amsterdam, 1959.

[FLP 84] N. Francez, D. Lehmann, A. Pnueli. *A linear-history semantics for languages for distributed programming*. TCS 32, pp. 25-46, 1984.

[Ger 85] R. Gerth. *A Maximal Parallelism Semantics for Occam*. Notes, 1985.

[GHR 86] R. Gerth, C. Huizing, W.P. de Roever. *Full Abstraction of a Real-Time Denotational Semantics for an OCCAM-like Language*. Department of Mathematics and Computing Science, Eindhoven University of Technology, August 1986 (accepted for POPL'87).

[HH 83] E.C.R. Hehner, C.A.R. Hoare. *A more complete model of communicating processes*. TCS 26, pp. 105-120, 1983.

[HMM85] J.Y. Halpern, N. Megiddo, A.A. Munshi. *Optimal Precision in the Presence of Uncertainty*. IBM Research Lab., San Jose, 1985.

[Hoa 78] C.A.R. Hoare. *Communicating Sequential Processes*. CACM 21-8, 1978.

[Hoo 86] J. Hooman. *A compositional proof theory for real-time distributed message passing*. Draft, Department of Mathematics and Computing Science, Eindhoven University of Technology, October 1986.

[Jon 82] G. Jones. D. Phil. thesis, Oxford, unpublished, 1982.

[Koy 84] R. Koymans. *Denotational semantics for real-time programming constructs in concurrent languages*. Notes, 1984.

[KVR 83] R. Koymans, J. Vytopil, W.P. de Roever. *Real-time Programming and Asynchronous Message Passing*. 2nd ACM PODC, pp. 187-197, 1983.

[MC 81] J. Misra, K.M. Chandy. *Proofs of Networks of Processes*. IEEE TOSE, Vol. SE-7, No. 4, pp. 417-426, July 1981.

[Mil 73] R. Milner. *An approach to the semantics of parallel programs*. Proc. of the Convegno di Informatica Teorica, Pisa, 1973.

[Mil 83] R. Milner. *Calculi for Synchrony and Asynchrony*. TCS 25, pp. 267-310, 1983.

[Occ 84] *The Occam language reference manual*. Prentice Hall, 1984.

[SM 81] A. Salwicki, T. Müldner. *On the algorithmic properties of concurrent programs*. LNCS 125, pp. 169-197, Springer, 1981.

[SR 83] R.K. Shyamasundar, W.P. de Roever. *Semantics of real-time Ada*. Notes, 1983.

[ZRB 85] J. Zwiers, W.P. de Roever, P. van Emde Boas. *Compositionality and Concurrent Networks: Soundness and Completeness of a Proofsystem*. 12th ICALP, LNCS 194, pp. 509-519, Springer, 1985.

[Zij 84] E. Zijlstra. *Real-time semantics*. Master thesis, University of Amsterdam, 1984.

APPENDIX A: CSP-R AND THE SIMULATION OF ADA

A1. CSP-R

The only difference between Mini CSP-R (see section 2) and CSP-R lies in the definition of I/O commands. CSP-R extends Mini CSP-R in the following ways:

- communication takes place via (a form of) channels,
- the expressions in output commands and the variables in input commands are vectors,
- process identifiers can be communicated and can be used in subsequent communications to determine the target process,
- communication with an arbitrary process can be requested instead of only addressing a particular process.

The syntax of Mini CSP-R is changed in the following way:

Replace forms 3.1 and 3.2 of instructions by

3.1.1 $P_i.c!\vec{e}$	- 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
3.1.2 $id.c!\vec{e}$	- as 3.1.1, but now the target process is determined by the value of the identification variable id
3.1.3 $.c!\vec{e}[\#id]$	- output via channel c to *any* 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)
3.2.1 $P_i.c?\vec{x}$	- the analogon of 3.1.1, but now values are received and are assigned to the variables in the list $\vec{x}$
3.2.2 $id.c?\vec{x}$	- the analogon of 3.1.2
3.2.3 $.c?\vec{x}[\#id]$	- the analogon of 3.1.3.

An identification variable is a variable ranging over $\{P_1,P_2,...\}$. It can only be assigned to using an instruction of the form 3.1.3 or 3.2.3.

The notions of syntactic and semantic matching of I/O commands have to be reformulated.

$<P_i,\alpha>$ and $<P_j,\beta>$ match syntactically iff:

1. α and β specify the same channel,
2. the vectors have equal length,
3. if α is an input command, then β is an output command and vice versa, and
4. if $\alpha(\beta)$ is of the form 3.1.1 or 3.2.1 then the specified target process should be $P_j(P_i)$.

$<i,\alpha>$ and $<j,\beta>$ match semantically iff:

1. $<P_i,\alpha>$ and $<P_j,\beta>$ match syntactically,
2. control in P_i and P_j is in front of both α and β, and
3. if $\alpha(\beta)$ is of the form 3.1.2 or 3.2.2, 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 3.1.1 - 3.2.3 above. Its effect is the assignment of the expression values to the variables and, possibly, the assignment to identification variables. Because of form 3.1.3 and 3.2.3 it is possible that $<i,\alpha>$ has more than one semantic match $<j,\beta>$. In that case, one of these β's is non-deterministically chosen and executed simultaneously with α.

The remaining syntax and interpretation of CSP-R is the same as for Mini CSP-R.

As for the extension of our denotational semantics to CSP-R, like the assumptions we have to record about values in the denotations for input commands, we now additionally record assumptions about the communication target in the denotations for I/O commands of the form 3.1.3 and 3.2.3.
Of course, the communication assumption records have to change. The communication claim records now have to record the communication channel and the communicated *vector* of values (instead of a single value). The no-match claim records now record the communication channel and the *length* of the communicated vector of values. Additionally, because of the I/O commands of the form 3.1.3 and 3.2.3, no-match claim records have to indicate with which *set* of processes a match is impossible (a single process for the forms 3.1.1, 3.1.2, 3.2.1 and 3.2.2, and all processes for the forms 3.1.3 and 3.2.3).
The denotations and techniques such as the consistency check have to be adapted corresponding to the above changes. These adaptations are straightforward except for a slight complication in the meaning of P_i::T: Because any communication target is assumed in the denotations for I/O commands of the form 3.1.3 and 3.2.3, now constructs like P_i:: .c?e generate communication claim records in which process i communicates with itself. This is clearly impossible and such records should be removed by

an additional operator. (Notice that this problem did not occur for Mini CSP-R, because constructs like P_i::P_i!e were prohibited syntactically by the naming conventions, see section 2.) The resulting semantics can be found in [Koy 84].

A2. Simulating Ada

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

1. the timed entry call ([Ada 83, § 9.7.3]).

select $T_i.a(\vec{e},\vec{x})$; S_1 **or delay** t; S_2 **end select**;

The semantics of this statement prescribes that if a rendezvous 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:
$[T_i.a!(\vec{e},\vec{x}) \rightarrow T_i.a?\vec{x};\ \tau(S_1) \ \square\ \textbf{wait}\ t \rightarrow \tau(S_2)]$.

2. the selective wait (without terminate alternative)([Ada 83, § 9.7.1]).

select or(i=1..n)**when** $b_i \Rightarrow S_i$ **or**(j=1..m)**when** $b^j \Rightarrow$ **delay** E^j; S^j **end select**;
where $S_i \equiv$ **accept** $a_i(\vec{u}_i\ \#\ \vec{v}_i)$ **do** S_{i_1} **end**; S_{i_2} (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_{i_2} 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:

$$[\mathop{\square}_{i=1}^{n} b_i;.a_i?(\vec{u}_i,\vec{v}_i)\#\ id \rightarrow \tau(S_{i_1});\ id.a_i!\vec{v}_i;\ \tau(S_{i_2})$$

$$\square$$

$$\mathop{\square}_{j=1}^{m} b^j;\ \textbf{wait}\ E^j \rightarrow \tau(S^j)].$$

We quote [Ada 83, § 9.7.1] for the semantics of a delay alternative in a selective wait: 'an open delay alternative will be selected if no accept alternative can be selected before the specified delay has elapsed (immediately, for a negative or zero delay in the absence of queued entry calls)'. This means that a delay alternative **delay** 0 is selected *immediately* , although it should be checked whether there are no queued entry calls. Not only is this unrealistic, it also gives rise to the following anomaly:
Consider a call of the recursive procedure P declared by
procedure P = **begin select accept** A; **or delay** 0; P; **end select end**; in a context where entry A is not called immediately. According to [Ada 83] there need not pass any time between the calling of P and any inner call of P, i.e., an infinite execution sequence takes no execution time!
Note that we could incorporate recursion easily into CSP-R on account of the structure of our semantic domain. Anyway, even in CSP-R without recursion, we can expand the calling of P arbitrarily deep. Keeping the same semantics as in [Ada 83] would then mean that an arbitrarily long execution sequence would take no execution time.

We removed this anomaly in our semantics by making **wait** 0 equivalent to **wait 1** (that is, a wait guard has a waitvalue of at least 1, see sections 2 and 7), thus reflecting the fact that it takes time to check whether immediate communication is possible or not. Now we get the desired semantics by simply translating Ada's **delay** t into CSP-R's **wait** t.
It is interesting to note that our techniques are in fact not capable to model the anomaly above: In our semantics the assumptions on the impossibility of communication are incorporated *within* the history, in fact within the mechanism that describes the passage of time. If we would have formulated these assumptions as independent conditions *on* the history (which would then contain only communication claim records), the modelling of the above anomaly would have been possible. E.g., when calling procedure P above an *empty communication history* is produced *under the condition* that entry A is not called immediately. Such independent conditions, however, would disturb the simple structure of our semantic domain and for such an unrealistic possibility in the Ada semantics this is certainly not worth the trouble.

APPENDIX B: DEFINITION OF $B[I \rightarrow i]$

Definition 1: For $I, J \in \mathbf{P}(\mathbb{N})$ define $R(I,J) \in \mathbf{P}(CAR)$ as $R(I,J) = \{r' \in CAR \mid \pi_1(r') \in I \ \wedge \ \pi_2(r') \in J\}$.
R(I,J) restricts the first and second component of pairs and triples in CAR.

Definition 2: For $r \in CAR$ define $ETC(r) \in \mathbf{P}(CAR)$ as
$ETC(r) = \{r' \in CAR \mid |r'| = |r| \ \wedge \ |r| = 3 \Rightarrow \pi_3(r') = \pi_3(r)\}$.
Equal Third Component of r selects pairs r' if r is a pair (and hence contains no third component) and otherwise triples r' with the same third component as r.

Definition 3: For $B \in \mathbf{B}(CAR)$, $I \in \mathbf{P}(\mathbb{N})$ and $i \in \mathbb{N}$ we define $B[I \rightarrow i] \in \mathbf{B}(CAR)$ as follows:

$$B[I \rightarrow i](r) = \begin{cases} 0 & \text{if } \pi_1(r) \in I \setminus \{i\} \ \vee \ \pi_2(r) \in I \setminus \{i\} \\ B(r) + \displaystyle\sum_{r' \in ETC(r) \cap R(I \setminus \{i\}, \{\pi_2(r)\})} B(r') & \text{if } \pi_1(r) = i \ \wedge \ \pi_2(r) \notin I \cup \{i\} \\ B(r) + \displaystyle\sum_{r' \in ETC(r) \cap R(\{\pi_1(r)\}, I \setminus \{i\})} B(r') & \text{if } \pi_1(r) \notin I \cup \{i\} \ \wedge \ \pi_2(r) = i \\ B(r) + \displaystyle\sum_{r' \in ETC(r) \ \cap \ (R(\{i\}, I \setminus \{i\}) \cup R(I \setminus \{i\}, \{i\}) \cup R(I \setminus \{i\}, I \setminus \{i\}))} B(r') & \text{if } \pi_1(r) = \pi_2(r) = i \\ B(r) & \text{otherwise.} \end{cases}$$

When substituting i for the elements of I in B, the components in the records that get changed are the elements of $I \setminus \{i\}$: these components are replaced by i. With this in mind , the second line is concerned with records before the substitution of the form <j,k> or <j,k,v>, the third line with <k,j> or <k,j,v> and the fourth line with <i,j> or <i,j,v> or <j,i> or <j,i,v> or <j,m> or <j,m,v>, where $j, m \in I \setminus \{i\}$ and $k \notin I \cup \{i\}$.
When r, the record after substitution, has a third component only records r' before the substitution should be considered above that have a third component with the same value. This is taken care of in the equation by ETC.

86/08	R. Koymans R.K. Shyamasundar W.P. de Roever R. Gerth S. Arun Kumar	Compositional semantics for real-time distributed computing (Inf.&Control 1987)
86/09	C. Huizing R. Gerth W.P. de Roever	Full abstraction of a real-time denotational semantics for an OCCAM-like language
86/10	J. Hooman	A compositional proof theory for real-time distributed message passing
86/11	W.P. de Roever	Questions to Robin Milner - A responder's commentary (IFIP86)
86/12	A. Boucher R. Gerth	A timed failure semantics for communicating processes
86/13	R. Gerth W.P. de Roever	Proving monitors revisited: a first step towards verifying object oriented systems (Fund. Informatica IX-4)
86/14	R. Koymans	Specifying passing systems requires extending temporal logic
87/01	R. Gerth	On the existence of sound and complete axiomatizations of the monitor concept
87/02	Simon J. Klaver Chris F.M. Verberne	Federatieve Databases
87/03	G.J. Houben J.Paredaens	A formal approach distri- buted information systems
87/04	T.Verhoeff	Delay-insensitive codes - An overview

Available Reports from the Theoretical Computing Science Group

	Author(s)	Title	Classification EUT	Classification DESCARTES
TIR83.1	R. Koymans, J. Vytopil, W.P. de Roever	Real-Time Programming and Synchronous Message passing (2nd ACM PODC)		
TIR84.1	R. Gerth, W.P. de Roever	A Proof System for Concurrent Ada Programs (SCP4)		
TIR84.2	R. Gerth	Transition Logic - how to reason about temporal properties in a compositional way (16th ACM FOCS)		
TIR85.1	W.P. de Roever	The Quest for Compositionality - a survey of assertion-based proof systems for concurrent progams, Part I: Concurrency based on shared variables (IFIP85)		
TIR85.2	O. Grünberg, N. Francez, J. Makowsky, W.P. de Roever	A proof-rule for fair termination of guarded commands (Inf.& Control 1986)		
TIR85.3	F.A. Stomp, W.P. de Roever, R. Gerth	The μ-calculus as an assertion language for fairness arguments (Inf.& Control 1987)		
TIR85.4	R. Koymans, W.P. de Roever	Examples of a Real-Time Temporal Logic Specification (LNCS207)		
TIR86.1	R. Koymans	Specifying Message Passing and Real-Time Systems (extended abstract)	CSN86/01	
TIR86.2	J. Hooman, W.P. de Roever	The Quest goes on: A Survey of Proof Systems for Partial Correctness of CSP (LNCS227)	EUT-Report 86-WSK-01	

TIR86.3	R. Gerth, L. Shira	On Proving Communication Closedness of Distributed Layers (LNCS236)	CSN86/07	
TIR86.4	R. Koymans, R.K. Shyamasundar, W.P. de Roever, R. Gerth, S. Arun Kumar	Compositional Semantics for Real-Time Distributed Computing (Inf.&Control 1987)	CSN86/08	
TIR86.5	C. Huizing, R. Gerth, W.P. de Roever	Full Abstraction of a Real-Time Denotational Semantics for an OCCAM-like Language	CSN86/09	PE.01
TIR86.6	J. Hooman	A Compositional Proof Theory for Real-Time Distributed Message Passing	CSN86/10	TR.4-1-1(1)
TIR86.7	W.P. de Roever	Questions to Robin Milner - A Responder's Commentary (IFIP86)	CSN86/11	
TIR86.8	A. Boucher, R. Gerth	A Timed Failure Semantics for Communicating Processes	CSN86/12	TR.4-4(1)
TIR86.9	R. Gerth, W.P. de Roever	Proving Monitors Revisited: a first step towards verifying object oriented systems (Fund. Informatica IX-4)	CSN86/13	
TIR86.10	R.Koymans	Specifying Message Passing Systems Requires Extending Temporal Logic	CSN86/14	PE.02
TIR87.1	R. Gerth	On the existence of sound and complete axiomatizations of the monitor concept	CSN87/01	