PUT AND GET, PRIMITIVES FOR SYNCHRONOUS UNRELIABLE MESSAGE PASSING

J.A. Bergstra, Department of Computer Science,
University of Amsterdam.
Department of Philosophy,
University of Utrecht.

This research was sponsored in part
by ESPRIT project METEOR (nr. 432).

October 1985.

University of Utrecht,
Department of Philosophy,
Heidelberglaan 2,
3584 CS Utrecht,
The Netherlands.

PUT AND GET, PRIMITIVES FOR SYNCHRONOUS UNRELIABLE MESSAGE PASSING

J.A. Bergstra, Department of Computer Science,
University of Amsterdam.
Department of Philosophy,
University of Utrecht.

ABSTRACT

Two message passing mechanisms are described. These mechanisms are both synchronous and unreliable. Within concrete process algebra with Θ operator for priorities these mechanisms, called put and get can be defined without introducing any new syntax.

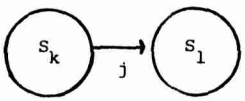
The properties of put and get are analysed in two pathological communication protocols. In a last section we indicate that the put mechanism can be generalised to a broadcasting mechanism.

1. INTRODUCTION

In process algebra [4,5,6] and CCS [10] communication is handled using the synchronous handshaking mechanism. In other formalisms like CIRCAL [9] and TCSP [8] and trace theory as in [9] communication between processes $X_1,\ldots,X_n$ takes the form that all of them simultaneously perform an action a provided a occurs in their alphabet (various modifications are conceivable here).

In this note we analyse the relationship between these mechanisms from the point of view of process algebra. This takes the form of introducing various new mechanisms in ACP which are all related to the communication mechanisms in CIRCAL, TCSP and trace theory.

Now suppose that the processes S_k and S_l are connected by port j



$sj(d)|rj(d)=cj(d)$

According to the handshaking mechanism value passing through j works as follows: S_k can perform an action sj(d) (send d along j) and S_l can perform action rj(d) (receive datum d along j). For successful communication S_k and S_l must perform these actions simultaneously (in synchronisation). A system action cj(d) that combines the effects of sj(d) and rj(d) will result. One writes $sj(d)|rj(d)=cj(d)$.

This paper focusses attention to two modifications of the above scheme.

(A) put mechanism. This mechanism allows a process S_k connected by port j to S_l to perform an action putj(d). This action can always be performed irrespective of S_l being able to receive the message. However, if S_l happens to be in a state that enables rj(d), then putj(d) and rj(d) must synchronise into cj(d). On the other hand, if putj(d) is performed when rj(d) is not enabled then no synchronisation will occur. For S_k it is not clear which of the two options have materialised, such information has to be signalled back from S_l.

(B) get mechanism. This mechanism allows S_l to be in a state of the form

$S=\sum_{d\in D} getj(d).X_d + getj(error).X_{error}$.

Then it can perform its next action getj(d) or getj(error).

If at the moment that S_l performs a getj action S_k is able to perform a send action sj(d) then the getj action will actually be getj(d) and a synchronisation $cj(d)=sj(d)|getj(d)$ results. If however S_k is not able to perform a send action sj(d) then S_l will actually perform getj(error), without having any influence on S_k.

Motivation. The put action typically models a signal of a physical output device which is issued irrespective of the fact that the intended listener actually listens. For instance, if we consider a communication protocol with sender S, receiver R and channels K and L (see fig. 1)

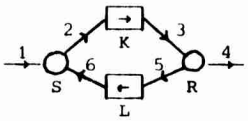


fig. 1

then the communication along port 3 will be like the put mechanism. The communication can only take place at the moment that a physical signal has passed through K and reaches R.

The get mechanism is a natural counterpart to the put mechanism, it allows an active component to inspect an input queue at regular time intervals. Emptyness of the input queue is then signalled by the error value.

In this paper we will describe in detail how the handshaking, put and get mechanism can be modeled using the system ACP_θ from [1].

Then we analyse several examples. It is found that KFAR, Koomen's fair abstraction rule plays a fundamental role in these examples. In order to improve the understanding of these matters an ad hoc real time semantics is considered. It follows that under certain natural timing constraints the predictions made by calculation in ACP_τ+KFAR are not valid!

Finally we generalise the put mechanism to a broadcasting mechanism for arbitrarily many receivers.

We assume that the reader knows process algebra up to the level of ACP_θ [1] and ACP_τ+KFAR [2,5,6].

For completeness we include some remarks about the θ operator here.

Let A be the alphabet of atomic actions including δ and let $R \subseteq A \times A$ be a partial order with δ as least element ($a \neq \delta \rightarrow R(\delta,a)$). We write $a < b$ for $R(a,b)$. Now an operator $\theta_R = \theta$ is defined on all finite processes over A. The idea is that θ replaces each subterm a or aX of its argument p by δ that occurs in a context $C[\ldots+b]$ or $C[\ldots+bX']$ where $a < b$. Thus $\theta(p)$ is like p but such that in each sum always a guard (first action) with highest priority must be chosen.

In [1] θ was defined using an auxiliary operator $\lhd$ using an equational specification. In [1] it was shown that this specification gives rise to a complete (confluent and terminating) rewrite system modulo the permutative reductions $X+Y \rightarrow Y+X$ and $(X+Y)+Z \rightarrow X+(Y+Z)$, $X+(Y+Z) \rightarrow (X+Y)+Z$.

Here we provide an equational specification not involving any auxiliary function. The specification is not very efficient and useless for rewriting

but expresses relevant properties of θ.

There are three classes of equations for θ.

(I) for all $a,b \in A$ with $b < a$:

$$\theta(a+b+Z) = \theta(a+Z) \qquad (1ab)$$

$$\theta(a+bX+Z) = \theta(a+Z) \qquad (2ab)$$

$$\theta(aX+b+Z) = \theta(aX+Z) \qquad (3ab)$$

$$\theta(aX+bY+Z) = \theta(aX+Z) \qquad (4ab)$$

(II) for all $a \in A$:

$$\theta(aX+a+Z) = \theta(aX+Z) + \theta(a+Z) \qquad (5a)$$

$$\theta(aX+aY+Z) = \theta(aX+Z) + \theta(aY+Z) \qquad (6a)$$

(III) for all K,M with $0 \leq K \leq M \leq \#(A)$ and for all $a_1 \ldots a_M \in A$ such that for all $i,j \in \{1 \ldots M\}$ $\neg(a_i < a_j)$:

$$\theta\Big(\sum_{i=1}^{K} a_i + \sum_{i=K+1}^{M} a_i X_i\Big) = \sum_{i=1}^{K} a_i + \sum_{i=K+1}^{M} a_i \theta(X_i) \qquad (7,K,M,a_1 \ldots a_M)$$

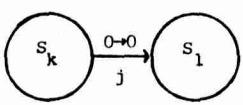
Due to the axioms of type III the total number of axioms may grow exponentially with $\#(A)$. (In this respect the specification of [1] is much better there the number of equations is bounded by $C1.\#(A)^2+C2$).

Using the above axioms each ACP_θ term can be shown equal to a term not involving θ.

2. SEND, RECEIVE, PUT and GET

In this section we outline in detail how the actions send (si(d)), receive (ri(d)), put (puti(d)) and get (get i(d) or get i(error)) are to be used in process algebra.

2.1 Handshaking



The symbol 0→0 indicates that communication and message passing along port j has direction $S_k \rightarrow S_1$ and must be implemented using synchronous handshaking. We assume that S_k and S_1 are given processes with action alphabets $\alpha(S_k)$ and $\alpha(S_1)$. Then the following properties of alphabets are assumed (with $rj(D) = \{rj(d) \mid d \in D\}$; sj(D) and cj(D) similarly defined).

$$sj(D) \subseteq \alpha(S_k)$$

$$rj(D) \subseteq \alpha(S_1)$$

$$sj(D) \cap \alpha(S_1) = \emptyset$$

$$rj(D) \cap \alpha(S_k) = \emptyset$$

Communications: $sj(d)|rj(d)=cj(d)$.

All other communications are δ.

Now the composition of S_k and S_l as parallel processes communicating through j is described by

$$\partial_{sj(D)} \circ \partial_{rj(D)} (S_k \| S_l)$$

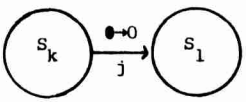
Often we write

$$\partial_{Hj} (S_k \| S_l)$$

with $Hj=sj(D) \cup rj(D)$.

2.2 Put actions

In symbolic notation:



Now we assume that:

$$putj(D) \subseteq \alpha(S_k)$$
$$rj(D) \subseteq \alpha(S_l)$$
$$putj(D) \cap \alpha(S_l) = \emptyset$$
$$rj(D) \cap \alpha(S_k) = \emptyset$$

where $putj(D)=\{putj(d) \mid d \in D\}$.

Communication is given by

$$putj(d)|rj(d)=cj(d).$$

For the communications cj(d) we must assume

$$cj(D) \cap \alpha(S_k) = \emptyset$$
$$cj(D) \cap \alpha(S_l) = \emptyset.$$

Thus putj(d) communicates just like sj(d), the difference is revealed in the composition mechanism. The point is that putj(d) can be performed in isolation, i.e. without synchronisation with rj(d), provided rj(d) is not enabled.

Let Rj be the following partial order:

$$\delta < putj(d) < cj(d) \quad \text{for all } d \in D$$
$$\delta < a \quad \text{for all atoms a not of the form putj(d) or cj(d).}$$

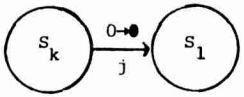
Then the composition of S_k and S_l is given by

$$\partial_{rj(D)} \, \theta_{Rj} \, (S_k \| S_l) .$$

The operator θ_{Rj} gives cj(d) priority over putj(d) thus enforcing synchronisation whenever possible. (Note that by putting an operator $\partial_{putj(D)}$ in front of this expression we are back in the handshaking situation.) The operator $\partial_{rj(D)}$ expresses the fact that read actions can only occur in synchronisation with put actions.

2.3 Get actions

For the get action mechanism we suggest the following symbolic indication



This time the underlying assumptions about $\alpha(S_k)$ and $\alpha(S_1)$ are as follows:

$$sj(D) \subseteq \alpha(S_k)$$

$$getj(D^*) \subseteq \alpha(S_1)$$

where $D^* = D \cup \{error\}$ and $getj(D^*) = \{getj(d) \mid d \in D\} \cup \{getj(error)\}$

$$sj(D) \cap \alpha(S_1) = \emptyset$$

$$getj(D^*) \cap \alpha(S_k) = \emptyset$$

$$cj(D) \cap \alpha(S_1) = \emptyset$$

$$cj(D) \cap \alpha(S_k) = \emptyset$$

Communication works as expected

$$sj(d) \mid getj(d) = cj(d)$$

Now the getj(error) action is possible whenever S_k is not able to perform a send action. Again the Θ operator is useful to model this mechanism. Let R_j^* be the following partial order on actions:

$$\delta < getj(error) < cj(d) \quad \text{for } d \in D$$

$$\delta < \text{all other actions.}$$

Then the parallel composition of S_k and S_1 communicating through port $\xrightarrow[j]{0 \to \bullet}$ is given by

$$\partial_{getj(D)} \cdot \partial_{sj(D)} \Theta_{R_j^*}(S_k \parallel S_1)$$

or, with $Hj = getj(D) \cup sj(D)$

$$\partial_{Hj} \Theta_{R_j^*}(S_k \parallel S_1)$$

(The handshaking mechanism can be obtained from this by applying $\partial_{\{getj(error)\}}$ to the expression.)

3. EXAMPLES

3.1 Consider the architecture in fig. 2.

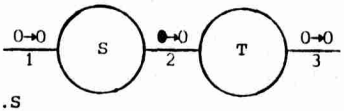


fig. 2

Let $S = \sum_{d \in D} r1(d).put2(d).S$

$T = \sum_{d \in D} r2(d).s3(d).T$

$P = \partial_{r2(D)} \Theta_{R2}(S \parallel T) = \Gamma(S \parallel T)$

We can eliminate the operator Γ in favour of more recursion equations.

$P = \sum_{d \in D} r1(d).\Gamma(put2(d).S \,||\, T)$

$\Gamma(put2(d).S \,||\, T) = c2(d).\Gamma(S \,||\, s3(d).T)$

$\Gamma(S \,||\, s3(d).T) = \sum_{b \in D} r1(b).\Gamma(put2(b).S \,||\, s3(d).T) + s3(d).P$

$\Gamma(put2(b).S \,||\, s3(d).T) = put2(b).\Gamma(S \,||\, s3(d).T) + s3(d).\Gamma(put2(b).S \,||\, T)$

This is a system of $\#(D)^2 + 2\#(D) + 1$ equations.

3.2 Consider the alternating bit protocol as described in [6]. Fig. 3 gives an architectual picture of the protocol

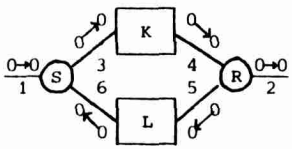


fig. 3

Now it can be verified that with the same components S, K, L and R (only substituting puti(d) for si(d) whenever indicated) the following architecture works just as well (fig. 4).

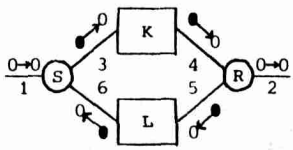


fig. 4

The verification of this follows the same lines as the calculation in [6].

4. ON REPLACING PUT AND GET BY HANDSHAKING

One might expect that a correct communication protocol using unreliable message passing works just as well when reliable message passing is used. This need not be the case however. Our proof uses an entirely pathological and unrealistic protocol but it serves to make the point.

The proof that P is correct makes essential use of Koomen's fair abstraction rule (KFAR). It is not entirely obvious that KFAR is a sound rule in pathological cases as presented in this section.

The architecture of P is depicted in fig. 5.

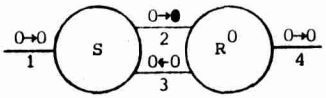


fig. 5

Recursion equations for S and R^0

$$S = \sum_{d\in D} r1(d).S^d$$

$$S^d = s2(d).i.S^d + r3(ack).S$$

$$R^0 = \sum_{d\in D} get2(d).s4(d).s3(ack).R^1 + get2(error).R^0$$

$$R^1 = \sum_{d\in D} get2(d).R^1 + get2(error).R^0$$

For the protocol P we find:

$$P = \partial_{H3}\partial_{H2}\Theta_{R_2^*}(S \| R^0)$$

$$= \Gamma(S \| R^0)$$

First we simplify the definition of P by eliminating Γ in favour of more recursive equations (1,2d,3,4d,5d,6d)

(1) $P = \sum_{d\in D} r1(d).\Gamma(S^d \| R^0) + get2(error).P$

(2d) $\Gamma(S^d \| R^0) = c2(d).(i \| s4(d)).c3(ack).\Gamma(S \| R^1)$

(3) $\Gamma(S \| R^1) = \sum_{d\in D} r1(d).\Gamma(S^d \| R^1) + get2(error).P$

(4d) $\Gamma(S^d \| R^1) = c2(d).\Gamma(i.S^d \| R^1)$

(5d) $\Gamma(i.S^d \| R^1) = i.\Gamma(S^d \| R^1) + get2(error).\Gamma(i.S^d \| R^0)$

(6d) $\Gamma(i.S^d \| R^0) = i\Gamma(S^d \| R^0) + get2(error).\Gamma(i.S^d \| R^0)$

Let $I = \{get2(error), i, c3(ack)\} \cup c2(D)$.

I contains the internal actions of P.

Using $KFAR_1$ on (1) we find

(7) $\tau_I(P) = \tau.\sum_{d\in D} r1(d).\tau_I(\Gamma(S^d \| R^0))$

Using $KFAR_2$ on the system of equations (4d) and (5d) one obtains

(8d) $\tau_I(\Gamma(S^d \| R^1)) = \tau.\tau_I(\Gamma(i.S^d \| R^0))$

Again, using $KFAR_1$ on (6d), one gets

(9) $\tau_I(\Gamma(i.S^d \| R^0)) = \tau.\tau_I(\Gamma(S^d \| R^0))$.

Now we compute

$$\tau_I(\Gamma(S \| R^1)) = \sum_{d \in D} r1(d).\tau_I(\Gamma(S^d \| R^1)) + \tau.\tau_I(P) \text{ by (3)}$$

$$= \sum_{d \in D} r1(d).\tau_I(\Gamma(S^d \| R^0)) + \tau.\tau_I(P) \text{ by (8d), (9)}$$

$$= \tau_I(P) \text{ by (7) and the second } \tau\text{-law.}$$

Then, finally

$$\tau_I(P) = \tau. \sum_{d \in D} r1(d).\tau_I(\Gamma(S^d \| R^0)) \text{ by (7)}$$

$$= \tau. \sum_{d \in D} r1(d).s4(d).\tau_I(\Gamma(S \| R^1)) \text{ by (2d)}$$

$$= \tau. \sum_{d \in D} r1(d).s4(d).\tau_I(P) \text{ by the above.}$$

But this means that P works correctly, after abstracting from internal actions it is just a one bit buffer.

Now suppose that we implement the same protocol with handshaking, as indicated in fig. 6.

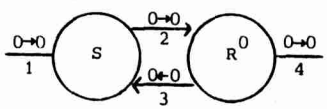


fig. 6

The equations for R^0 and R^1 are now modified; S remains the same

$$\bar{R}^0 = \sum_{d \in D} r2(d).s4(d).s3(ack).\bar{R}^1$$

$$\bar{R}^1 = \sum_{d \in D} r2(d).\bar{R}^1$$

The system $\bar{P}$ is then given by

$$\partial_{H2}\partial_{H3}(S \| \bar{R}^0) = \bar{\Gamma}(S \| \bar{R}^0)$$

With I as before we compute and find

$$\tau_I(\bar{P}) = \sum_{d \in D} r1(d).s4(d). \sum_{b \in D} r1(b).\delta$$

Thus the system deadlocks, i.e. the performance degrades with improved synchronisation. In order to verify the above statement we first replace $\bar{\Gamma}$ by recursion equations.

$$\bar{P} = \sum_{d \in D} r1(d).c2(d).(i \| s4(d)).c3(ack). \sum_{b \in D} r1(b).\bar{\Gamma}(S^b \| \bar{R}^1)$$

$\bar{\Gamma}(S^b \| R^1) = c2(b).i.\bar{\Gamma}(S^b \| \bar{R}^1)$

Using $KFAR_2$ on this last equation we get

$\tau_I(\bar{\Gamma}(S^b \| \bar{R}^1)) = \tau.\delta$

Thus $\tau_I(\bar{P}) = \sum_{d \in D} r1(d).s4(d) . \sum_{b \in D} r1(b).\delta.$

Some explanation is in order. The deadlock is in fact a livelock which transforms to deadlock due to KFAR. This livelock is avoided in the original protocol P because KFAR forces that sooner or later an action get2(error) occurs. This is precisely what R^1 needs to return to state R^0.

5. AN EXAMPLE WHERE "KFAR" IS TOO OPTIMISTIC

We consider the architecture of fig. 6.

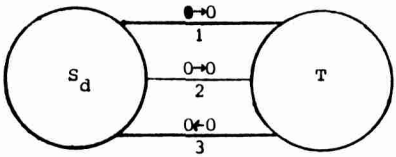


fig. 6

Recursion equations for S_d and T:

$S_d = put1(d).s2(ack).(r3(pack)+r3(nack).S_d)$

$T = i.(\sum_{e \in D} r1(e).r2(ack).s3(pack).print(e)+r2(ack).s3(nack).T)$

Let $D=\{d\}$, $H2=\{s2(ack), r2(ack)\}$, $H1=r1(D)$

$H3=\{s3(pack), s3(pack), r3(nack), r3(nack)\}$

Communication is standard. The full system P is thus given by

$P=\partial_{H3}\partial_{H2}\partial_{H1}\Theta_{R1}(S_d \| T) = \Gamma(S_d \| T)$

First we derive recursion equations for P over + and ·.

$P=i.c1(d).c2(ack).c3(pack).print(d)+put1(d).i.c2(ack).c3(nack).P$

Let $I=\{i, c1(d), c2(ack), c3(pack), c3(nack), put1(d)\}$.

Then using $KFAR_4$

$\tau_I(P) = \tau.print(d).$

Thus using KFAR one predicts that P prints d (after some delay).

We will now investigate how plausible this prediction is. From the above proof it can be extracted that the role of KFAR is to ensure that after finitely many iterations it will be the case that i precedes the put action of S_d and consequently the put action will be successful.

Now after each communication c3(nack) P restarts in its initial state. If we assume that the action i takes always 2 or more seconds whereas S_d always performs the put instruction within 1 second, then during each iteration the

put instruction will come too early and fail.

Under these entirely plausible real time conditions P will rather behave as a solution of the equation

$$\tilde{P}=put1(d).i.c2(ack).c3(nack).\tilde{P}$$

Now $\tau_I(\tilde{P})=\tau.\delta$ using $KFAR_4$. This deadlock comes from a livelock which is due to the real time constraints.

Thus the predictions about P made by KFAR may well be too optimistic. At least in the ad hoc real time semantics outlined above the prediction $\tau_I(P)=\tau.print(d)$ is plainly wrong.

This may well be a fundamental problem with KFAR and bisimulation semantics. At least the question under what conditions KFAR is generally sound deserves further attention.

6. A BROADCASTING MECHANISM

In this section a generalisation of the put mechanism to a broadcasting mechanism is discussed. Essentially the puti(d) statement broadcasts d along port i to 0 or 1 listeners. For an arbitrary number of listeners we will have to work with an infinite alphabet. As such that presents no fundamental difficulties and the algebraic style is fully preserved as long as infinite sums are avoided.

For a port name j we will use the following actions.

brcj(d)	broadcast d through j
rj(d)	receive d through j
$rj^m(d)$	d is received $m\geq 2$ times simultaneously $(rj(d)=rj^1(d))$
$cj^m(d)$	d is communicated $m\geq 1$ times along j

It is convenient to write $cj^0(d)$ for brcj(d). The communication function | works as follows: $rj^m(d)|rj^k(d)=rj^{m+k}(d)$; $cj^p(d)|rj^k(d)=cj^{p+k}(d)$ for $m,k\geq 1$, $p\geq 0$, where $rj^1(d)$ denotes rj(d). In particular $brcj(d)|rj^k(d)=cj^k(d)$. All other communications lead to δ. Let us consider the architecture depicted in fig. 7

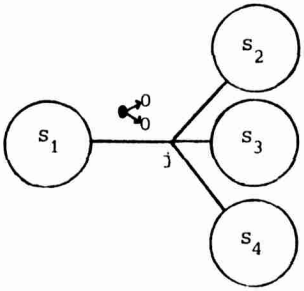


fig. 7

The intuition is here that if S_1 performs brcj(d) as many as possible of the S_2 S_3 S_4 try to receive d. Mathematically this is modeled by introducing these priorities

R'_j: $\delta < cj^0(d) < cj^1(d) < cj^2(d) < \ldots$ for $d \in D$

$\delta <$ a for all other actions.

Let $rj^N(D) = \bigcup_{m \in N} rj^m(D)$.

Then the system is given by

$P = \partial_{rj^N(D)}\, \Theta_{R'j}(S_1 \| S_2 \| S_3 \| S_4)$

A different presentation can be given as follows:

Let R''_j: $\delta < rj^1(d) < rj^2(d) < \ldots \quad d \in D$

$\delta <$ a all other a.

Then $P = \partial_{rj^N(D)} \Theta_{R'j}(S_1 \| \Theta_{R''_j}(S_2 \| S_3 \| S_4))$

We conclude this section with an example. The architecture is depicted in fig. 8.

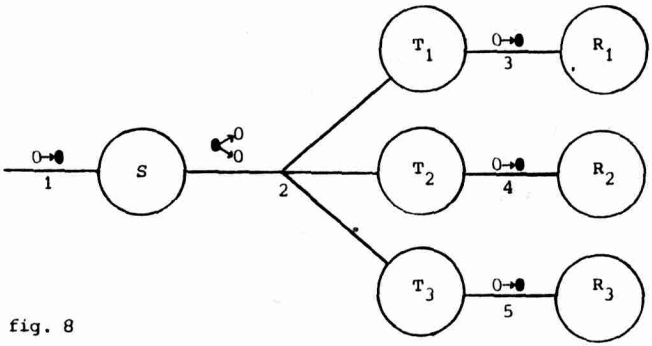


fig. 8

Along 1 and 3,4,5 data $d \in D$ are transmitted. Along 2 an additional value reset can be mailed. Let $D' = D \cup \{reset\}$.

$S = \sum_{d \in D} get1(d).brc2(d).i.brc2(reset).S + get1(error).S.$

$T_i = \sum_{d \in D} r2(d).(si{+}2(d) + r2(reset)).T_i$

$R_i = \sum_{d \in D} geti{+}2(d).print(d).R_i + geti{+}2(error).R_i$

$P = \Gamma(S \| T_1 \| T_2 \| T_3 \| R_1 \| R_2)$

with Γ as follows:

$\partial_{s3(D)} \circ \partial_{s4(D)} \circ \partial_{s5(D)} \circ \partial_{r2^N(D')} \circ$

$\Theta_{R_3^*} \circ \Theta_{R_4^*} \circ \Theta_{R_5^*} \circ \Theta_{R'_2}$

What happens is this: S reads data at its own speed. As soon as it read d this is broadcasted to all of T_1, T_2, T_3. All will be able to receive d. Then simultaneously S waits some time (i) and the T_k allow the corresponding R_k to fetch d. After S has timed out (performed i) it issues a reset. All processes T_k which have not yet handed over their datum to R_k will receive the reset instruction and thereby return to their initial state.