Process Specification in a UNITY Format

Jacob Brunekreef

Programming Research Group, University of Amsterdam

Kruislaan 403, 1098 SJ Amsterdam, The Netherlands

e-mail: jacob@fwi.uva.nl

Abstract

In this paper a "UNITY Format" for process specifications is introduced. The format is based on conditions on process states and process data. Several aspects of this format are discussed: a straightforward normalisation of the parallel composition of processes, the relation between the ACP priority operator θ and conditions and the correspondence with a term rewriting system, which opens certain perspectives with respect to the validation and verification of a specification. Throughout the paper the simple and well-known PAR protocol serves as a running example.

Keywords & phrases: process algebra, formal specification, data oriented, UNITY.
1987 CR Categories: D.1.3, D.2.10, F.1.2.

1 Introduction

Different "specification styles" can be chosen for the specification of processes. In the field of ACP traditionally a *process oriented* style is used: a set of process names, possibly parameterised with data, is used to represent different process states. For each process name a defining equation is given. In this paper we present a specification style in which most of the process complexity is put in (functions on) data parameters of a single process equation. Because of its resemblance with the UNITY formalism of [CM88] we will call the proposed specification format the UNITY Format. In section 3 the format is introduced. It is based on conditions on process states and functions on data parameters. Several aspects of the format are discussed in this paper:

- In section 4 it is shown that normalisation of the parallel composition of two or more processes specified in the UNITY Format is rather simple.
- In section 5 it is shown that in this format a priority relation between actions can be translated to an extra condition.
- In section 6 a function on process states and data parameters of a process is introduced. A Term Rewriting System (TRS) on this function is considered as an abstract representation of the process. It is discussed how the TRS can be used for validation and verification purposes.

Preceding the discussion of these aspects, in section 2 some ACP axioms are given and some identities are derived from these axioms. Section 7 contains some concluding remarks. Throughout the paper the well-known PAR protocol is used as a running example. Appendix A contains a specification in the UNITY Format of the PAR protocol in the executable formalism PSF. In appendix B the TRS corresponding with the specification of the PAR protocol is specified in PSF. In these appendices it is discussed to what extend the PSF tools can be used for the validation of a process specification in the UNITY Format.

2 ACP axioms

In this section a subset of the ACP axioms is given. The axioms in Table 1 (from [BW90]) form the axiom system for "standard" ACP. The axioms in Table 2 (from [BB92]) deal with conditional

$x + y = y + x$		A1
$(x + y) + z = x + (y + z)$		A2
$x + x = x$		A3
$(x + y) \cdot z = x \cdot z + y \cdot z$		A4
$(x \cdot y) \cdot z = x \cdot (y \cdot z)$		A5
$x + \delta = x$		A6
$\delta \cdot x = \delta$		A7
$x \parallel y = x \llfloor y + y \llfloor x + x \mid y$		CM1
$a \llfloor x = a \cdot x$		CM2
$(a \cdot x) \llfloor y = a \cdot (x \parallel y)$		CM3
$(x + y) \llfloor z = (x \llfloor z) + (y \llfloor z)$		CM4
$(a \cdot x) \mid b = (a \mid b) \cdot x$		CM5
$a \mid (b \cdot x) = (a \mid b) \cdot x$		CM6
$(a \cdot x) \mid (b \cdot y) = (a \mid b) \cdot (x \parallel y)$		CM7
$(x + y) \mid z = x \mid z + y \mid z$		CM8
$x \mid (y + z) = x \mid y + x \mid z$		CM9
$a \mid b = \gamma(a, b)$	if γ defined	CF1
$a \mid b = \delta$	otherwise	CF2

Table 1: Standard ACP axioms.

process expressions. The boolean condition c always evaluates to *true* or *false*. It is supposed that $\neg true = false$ and $\neg false = true$.

In section 5 a relation between the ACP operator θ (the priority operator) and conditional process expressions is derived. The axioms in Table 3 (from [BBK86], also given in [BW90]) define this operator. In this table an auxiliary operator is used: the binary *unless* operator, in [BBK86] denoted by the symbol $\triangleleft$. In order to avoid any confusion between this operator and the left triangle of a conditional process expression, in this paper we will denote the unless operator with the symbol $\lll$.
The last three axioms from Table 3 (from [BB92]) deal with the distributivity of the priority operator and the unless operator over a conditional process expression.

From the axioms the following identities can easily be derived. These identities will be used in the coming sections.

Lemma 2.1

1. $x \triangleleft c \triangleright x = x$
2. $x \triangleleft c \triangleright y = x \triangleleft c \triangleright \delta + y \triangleleft \neg c \triangleright \delta$
3. $(x \triangleleft c_1 \triangleright \delta) \triangleleft c_2 \triangleright \delta = x \triangleleft c_1 \wedge c_2 \triangleright \delta$

$x \lhd \mathit{true} \rhd y = x$	CO1
$x \lhd \mathit{false} \rhd y = y$	CO2
$\mathit{true} :\rightarrow x = x$	GC1
$\mathit{false} :\rightarrow x = \delta$	GC2
$c :\rightarrow x = x \lhd c \rhd \delta$	CG1
$x \lhd c \rhd y = c :\rightarrow x + (\neg c) :\rightarrow y$	CG2
$c :\rightarrow \delta = \delta$	GC9
$c :\rightarrow (x + y) = (c :\rightarrow x) + (c :\rightarrow y)$	GC10
$(c_1 \vee c_2) :\rightarrow x = (c_1 :\rightarrow x) + (c_2 :\rightarrow y)$	GC11
$c_1 :\rightarrow (c_2 :\rightarrow x) = (c_1 \wedge c_2) :\rightarrow x$	GC12
$(x \lhd c \rhd y) \mathbin{\lfloor\!\lfloor} z = (x \mathbin{\lfloor\!\lfloor} z) \lhd c \rhd (y \mathbin{\lfloor\!\lfloor} z)$	CC1
$(x \lhd c \rhd y) \vert z = (x \vert z) \lhd c \rhd (y \vert z)$	CC2
$x \vert (y \lhd c \rhd z) = (x \vert y) \lhd c \rhd (x \vert z)$	CC3
$\partial_H(x \lhd c \rhd y) = \partial_H(x) \lhd c \rhd \partial_H(y)$	CC4

Table 2: Conditional ACP axioms.

Proof: (the acronyms above the identity symbol refer to the labels of the axioms)

1. $x \lhd c \rhd x \stackrel{\text{CG2}}{=} c :\rightarrow x + (\neg c) :\rightarrow x =$ (c always evaluates to *true* or *false*)

   $\mathit{true} :\rightarrow x + \mathit{false} :\rightarrow x \stackrel{\text{GC1,GC2}}{=} x + \delta \stackrel{\text{A6}}{=} x$

2. $x \lhd c \rhd y \stackrel{\text{CG2}}{=} c :\rightarrow x + (\neg c) :\rightarrow y \stackrel{\text{CG1}}{=} x \lhd c \rhd \delta + y \lhd \neg c \rhd \delta$

3. $(x \lhd c_1 \rhd \delta) \lhd c_2 \rhd \delta \stackrel{\text{CG2}}{=} c_2 :\rightarrow (c_1 :\rightarrow x + (\neg c_1) :\rightarrow \delta) + (\neg c_2) :\rightarrow \delta \stackrel{\text{GC9,A6}}{=}$

   $c_2 :\rightarrow (c_1 :\rightarrow x) \stackrel{\text{GC12}}{=} (c_1 \wedge c_2) :\rightarrow x \stackrel{\text{CG1}}{=} x \lhd c_1 \wedge c_2 \rhd \delta$ □

3 A UNITY Format for process specifications

In a lot of ACP specifications it is common practise to specify a process with a (set of) linear equations, each in the following format:

$$X = a_1 \cdot X_1 + \ldots + a_i \cdot X_i + a_{i+1} + \ldots + a_j$$

See e.g. [Bae90] for many examples of specifications in this format. In this section we present a normal form for process specification that is based on a single (recursive) equation, in which a process is specified as the alternative composition of a number of conditional process terms. The process name is parameterised with one or more data variables that contain state information about the process. Each conditional process term is built from an atomic action, followed by a condition, or the sequential composition of an atomic action and the process name (usually with some substitutions in its parameter list), followed by a condition. The condition determines in which state the atomic action is enabled. Because of the resemblance with the UNITY formalism of [CM88] we will call this format the UNITY Format for process specifications. In this format a process X is specified as follows:

$a \triangleleft\!\!\triangleleft b = a$	if $\neg(a < b)$	P1
$a \triangleleft\!\!\triangleleft b = \delta$	if $a < b$	P2
$x \triangleleft\!\!\triangleleft y \cdot z = x \triangleleft\!\!\triangleleft y$		P3
$x \triangleleft\!\!\triangleleft (y + z) = (x \triangleleft\!\!\triangleleft y) \triangleleft\!\!\triangleleft z$		P4
$x \cdot y \triangleleft\!\!\triangleleft z = (x \triangleleft\!\!\triangleleft z) \cdot y$		P5
$(x + y) \triangleleft\!\!\triangleleft z = x \triangleleft\!\!\triangleleft z + y \triangleleft\!\!\triangleleft z$		P6
$\theta(a) = a$		TH1
$\theta(x \cdot y) = \theta(x) \cdot \theta(y)$		TH2
$\theta(x + y) = \theta(x) \triangleleft\!\!\triangleleft y + \theta(y) \triangleleft\!\!\triangleleft x$		TH3
$x \triangleleft\!\!\triangleleft (y \lhd c \rhd z) = (x \triangleleft\!\!\triangleleft y) \lhd c \rhd (x \triangleleft\!\!\triangleleft z)$		PC1
$(x \lhd c \rhd y) \triangleleft\!\!\triangleleft z = (x \triangleleft\!\!\triangleleft z) \lhd c \rhd (y \triangleleft\!\!\triangleleft z)$		PC2
$\theta(x \lhd c \rhd y) = \theta(x) \lhd c \rhd \theta(y)$		THC

Table 3: Axioms for ACP_θ with conditions.

$$
\begin{aligned}
X(D) = \quad & \ a_1 \cdot X(D_1) \lhd c_1 \rhd \delta \\
& + \ \ldots \\
& + \ a_m \cdot X(D_m) \lhd c_m \rhd \delta \\
& + \ a_{m+1} \lhd c_{m+1} \rhd \delta \\
& + \ \ldots \\
& + \ a_n \lhd c_n \rhd \delta
\end{aligned}
$$

D denotes a parameter list, $D_1 \ldots D_m$ denote a parameter list with substitutions for some of the elements of D. $a_1 \ldots a_n$ denote atomic actions. $c_1 \ldots c_n$ denote boolean conditions, based on elements of D. If a condition c_i is invariantly *true*, a summand can be written as $a_i \cdot X(D_i)$ or just a_i.

In the sequel we will only look at non–terminating processes, the non–recursive summands are left out. All results obtained in the coming sections can also easily be derived for terminating processes.

We will give an example of a specification in this format by giving a specification of the components of the well-known PAR protocol. PAR stands for *Positive Acknowledgement with Retransmission*. In this protocol, after the transmission of a frame, the sender waits for an acknowledgement from the receiver before a new frame is transmitted. The channels between sender and receiver are faulty: frames may be corrupted (resulting in a *checksum error* at the receiving side) or may get lost. When the receiver receives a frame with a checksum error no acknowledgement is sent back, the receiver just waits for a new frame. If the sender does not receive an acknowledgement within a certain time interval, a timeout is generated, which leads to a retransmission of the last frame sent. In order to distinguish retransmitted frames from new ones, a frame from the sender to the receiver contains an identification bit, which is flipped each time a new frame is transmitted. The specification of the protocol contains five components: the sender, the receiver, a timer process and two channels, K and L, between the sender and the receiver. Figure 1 shows the various components with their connections.

In the specification of the components below, the atomic read and send actions are labelled with two capitals. The first capital indicates the source of the communication, the second capital indicates the destination. So the atomic action *send_SK* denotes a send action from the sender S to the channel K. The capital E refers to the environment of the PAR protocol.

The sender process reads a data element from the environment, after which a frame with this el-

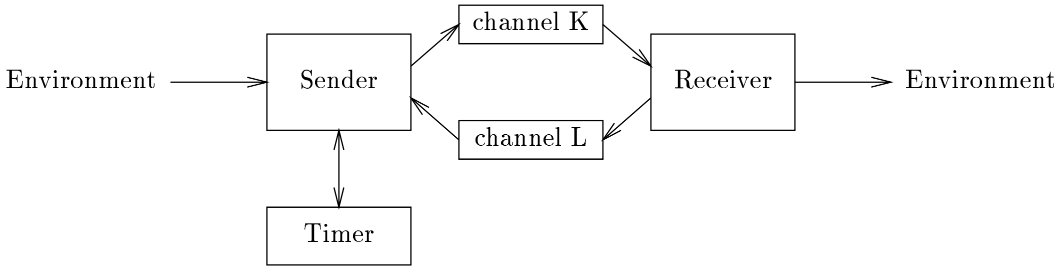

Figure 1: Components of the PAR protocol.

ement and a control bit is sent to the receiver via the channel K. The timer is started by a message from the sender to the timer. Next an acknowledgement or a checksum error from the channel L is awaited. If an acknowledgement is received, a new data element from the environment is expected. If nothing is received from the channel L, a timeout will be received from the timer process. In the case of the reception of a checksum error from channel L or a timeout, the old frame is retransmitted. So the sender process has four states: waiting for input from the environment, sending a frame to the channel K, sending a message to the timer and waiting for something to arrive from the channel L or the timer. In the specification below the first parameter of the process S denotes the sender state. The value of ss ranges from $s1$ to $s4$. The other two parameters denote the data element that has to be transmitted (sd) and the additional bit (sb). In each summand the condition on the value of ss determines whether the action is enabled or not.

$$
\begin{aligned}
S(ss, sd, sb) = \quad & \textstyle\sum_{d \in D} \mathit{read\_ES}(d) \cdot S(s2, d, sb) \lhd ss = s1 \rhd \delta \\
+ \quad & \mathit{send\_SK}(sd, sb) \cdot S(s3, sd, sb) \lhd ss = s2 \rhd \delta \\
+ \quad & \mathit{send\_ST} \cdot S(s4, sd, sb) \lhd ss = s3 \rhd \delta \\
+ \quad & \mathit{read\_LS}(ac) \cdot S(s1, sd, 1 - sb) \lhd ss = s4 \rhd \delta \\
+ \quad & \mathit{read\_LS}(ce) \cdot S(s2, sd, sb) \lhd ss = s4 \rhd \delta \\
+ \quad & \mathit{read\_TS} \cdot S(s2, sd, sb) \lhd ss = s4 \rhd \delta
\end{aligned}
$$

The receiver process awaits a frame from the channel K. If the control bit in the frame is as expected, the data element is delivered to the environment and an acknowledgement is sent to the sender via the channel L. In case the control bit is wrong only an acknowledgement is sent. If a checksum error is received, the receiver shows no reaction at all.

The receiver process R is parameterised with rs (the receiver state, ranging from $r1$ to $r3$), rd (the received data element), rb (the received additional bit) and eb (the expected bit in a frame to receive). In two summands the condition is not only based on the process state rs, but also on other process parameters: the (in)equality of two bit values.

$$
\begin{aligned}
R(rs, rd, rb, eb) = \quad & \textstyle\sum_{d \in D, b \in B} \mathit{read\_KR}(d, b) \cdot R(r2, d, b, eb) \lhd rs = r1 \rhd \delta \\
+ \quad & \mathit{read\_KR}(ce) \cdot R(rs, rd, rb, eb) \lhd rs = r1 \rhd \delta \\
+ \quad & \mathit{send\_RE}(rd) \cdot R(r3, rd, rb, 1 - eb) \lhd rs = r2 \wedge rb = eb \rhd \delta \\
+ \quad & \mathit{send\_RL}(ac) \cdot R(r1, rd, rb, eb) \lhd (rs = r2 \wedge rb = 1 - eb) \vee (rs = r3) \rhd \delta
\end{aligned}
$$

The timer process is very simple: in any state it awaits a start message from the sender process. Once started, the timer may send a timeout message to the sender process. The timer process T is only parameterised with ts, the timer state, with the value $t1$ or $t2$. As the timer may be started in both states, the condition in the process term with the action *read_ST* is omitted.

$$
\begin{aligned}
T(ts) = \quad & \mathit{read\_ST} \cdot T(t2) \\
+ \quad & \mathit{send\_TS} \cdot T(t1) \lhd ts = t2 \rhd \delta
\end{aligned}
$$

The channel process K receives a frame from the sender process. Next, three things may occur: the frame is delivered to the receiver process correctly, the frame is delivered to the receiver process with a checksum error or the frame is not delivered at all (it is lost in the channel). These three alternatives are modelled in an alternative composition. [1]

The channel process K is parameterised with ks (the channel state, with the value $k1$ or $k2$), kd (the data element in the channel) and kb (the bit in the channel). The loss of a frame is modelled by the atomic action $klost$.

$$\begin{aligned} K(ks,kd,kb) = & \quad \textstyle\sum_{d\in D,\, b\in B} read\_SK(d,b)\cdot K(k2,d,b) \lhd ks = k1 \rhd \delta \\ & + \; send\_KR(kd,kb)\cdot K(k1,kd,kb) \lhd ks = k2 \rhd \delta \\ & + \; send\_KR(ce)\cdot K(k1,kd,kb) \lhd ks = k2 \rhd \delta \\ & + \; klost \cdot K(k1,kd,kb) \lhd ks = k2 \rhd \delta \end{aligned}$$

The channel process L is a simple version of the process K. It has only one parameter: the channel state ls, with the value $l1$ or $l2$. The atomic action $llost$ denotes the loss of an acknowledgement.

$$\begin{aligned} L(ls) = & \quad read\_RL(ac)\cdot L(l2) \lhd ls = l1 \rhd \delta \\ & + \; send\_LS(ac)\cdot L(l1) \lhd ls = l2 \rhd \delta \\ & + \; send\_LS(ce)\cdot L(l1) \lhd ls = l2 \rhd \delta \\ & + \; llost \cdot L(l1) \lhd ls = l2 \rhd \delta \end{aligned}$$

In the following two sections we will derive an equation for the (encapsulated) parallel composition of the components of the PAR protocol.

4 Parallel composition and conditions

With the axioms from Table 1 the parallel composition (merge) of two process terms can be *normalised*: rewritten to a process term with only sequential composition and alternative composition. In this section we will derive a lemma about the parallel composition of two conditional processes in the UNITY Format. The lemma will be illustrated with the normalisation of the PAR protocol: the encapsulated merge of the five processes from the previous section.

We consider two processes in the UNITY Format:

$$X(D) = \textstyle\sum_{i=1}^{n}(a_i \cdot X(D_i) \lhd c_i \rhd \delta) \qquad \text{and} \qquad Y(E) = \sum_{j=1}^{m}(b_j \cdot Y(E_j) \lhd c'_j \rhd \delta).$$

The parallel composition of these processes, $X(D) \parallel Y(E)$, is denoted by $XY(D,E)$. The following lemma shows how this parallel composition can be rewritten to a process term in the required format.

Lemma 4.1

$$\begin{aligned} XY(D,E) = & \textstyle\sum_{i=1}^{n}(a_i \cdot XY(D_i,E) \lhd c_i \rhd \delta) + \sum_{j=1}^{m}(b_j \cdot XY(D,E_j) \lhd c'_j \rhd \delta) + \\ & \textstyle\sum_{i=1}^{n}\sum_{j=1}^{m}((a_i \mid b_j)\cdot XY(D_i,E_j) \lhd c_i \wedge c'_j \rhd \delta) \end{aligned}$$

Proof: (as before, the acronyms above the identity symbol refer to the labels of the axioms from section 2)

$$X(D) \parallel Y(E) \stackrel{\text{CM1}}{=} X(D) \mathbin{\|\!\_} Y(E) + Y(E) \mathbin{\|\!\_} X(D) + X(D) \mid Y(E).$$

[1] In the specification of existing protocols we consider internal channel actions like i, j or *skip* preceeding the alternatives as superfluous.

We will elaborate each summand separately.

$$X(D) \mathbin{\|\!\!\_} Y(E) \stackrel{\text{CM4}}{=} \textstyle\sum_{i=1}^{n}((a_i \cdot X(D_i) \lhd c_i \rhd \delta) \mathbin{\|\!\!\_} Y(E)) \stackrel{\text{CC1}}{=}$$

$$\textstyle\sum_{i=1}^{n}((a_i \cdot X(D_i) \mathbin{\|\!\!\_} Y(E)) \lhd c_i \rhd (\delta \mathbin{\|\!\!\_} Y(E))) \stackrel{\text{CM3,CM2,A7}}{=}$$

$$\textstyle\sum_{i=1}^{n}((a_i \cdot (X(D_i) \parallel Y(E))) \lhd c_i \rhd \delta) =$$

$$\textstyle\sum_{i=1}^{n}(a_i \cdot XY(D_i, E) \lhd c_i \rhd \delta)$$

In the same way the following identity is proved:

$$Y(E) \mathbin{\|\!\!\_} X(D) = \textstyle\sum_{j=1}^{m}(b_j \cdot XY(D, E_j) \lhd c'_j \rhd \delta).$$

Remains the third summand:

$$X(D) \mid Y(E) = \textstyle\sum_{i=1}^{n}(a_i \cdot X(D_i) \lhd c_i \rhd \delta) \mid \sum_{j=1}^{m}(b_j \cdot Y(E_j) \lhd c'_j \rhd \delta) \stackrel{\text{CM8,CM9}}{=}$$

$$\textstyle\sum_{i=1}^{n}\sum_{j=1}^{m}((a_i \cdot X(D_i) \lhd c_i \rhd \delta) \mid (b_j \cdot Y(E_j) \lhd c'_j \rhd \delta)) \stackrel{\text{CC2,CC3}}{=}$$

$$\textstyle\sum_{i=1}^{n}\sum_{j=1}^{m}(((a_i \cdot X(D_i)) \mid (b_j \cdot Y(E_j)) \lhd c'_j \rhd \delta) \lhd c_i \rhd \delta) \stackrel{\text{CM7}}{=}$$

$$\textstyle\sum_{i=1}^{n}\sum_{j=1}^{m}(((a_i \mid b_j) \cdot (X(D_i) \parallel Y(E_j)) \lhd c'_j \rhd \delta) \lhd c_i \rhd \delta) \stackrel{\text{CG1,GC12}}{=}$$

$$\textstyle\sum_{i=1}^{n}\sum_{j=1}^{m}((a_i \mid b_j) \cdot XY(D_i, E_j) \lhd c_i \wedge c'_j \rhd \delta) \qquad \square$$

So the parallel composition of two processes X and Y consists of three groups of summands: a group with the atomic actions and conditions from the process X, a group with the actions and conditions from process Y and a group of actions that is the result of communications between actions from process X and process Y. In the last group the conditions consist of the boolean and of two separate conditions from the processes X and Y. Without further proof we claim that this result is also valid for the parallel composition of more than two processes. We will use this claim in the (encapsulated) parallel composition of the five processes of the PAR protocol as specified in the previous section. We name this process P. The data parameters of P come from the constituent processes.

$$\begin{aligned} &P(ss, rs, ts, ks, ls, sd, sb, rd, rb, eb, kd, kb) = \\ &\quad \partial_H(S(ss, sd, sb) \parallel R(rs, rd, rb, eb) \parallel T(ts) \parallel K(ks, kd, kb) \parallel L(ls)) \end{aligned}$$

The set H contains the atomic actions which have to be encapsulated in order to enforce the desired communications between the five processes. Below the communications between a *read* and a *send* action are denoted by a *comm* action.

For the process P the following lemma holds. We will use a "notational shortcut" from [BG93]: in the parameter lists of P at the right side of the equation only the substitutions of data elements are listed. As the equation for P is left–linear with respect to the data parameters, this will not lead to any confusion. A substitution is denoted by x/y, which means that the parameter y is replaced by the value of the term x.

Lemma 4.2

$$\begin{aligned} &P(ss, rs, ts, ks, ls, sd, sb, rd, rb, eb, kd, kb) = \\ &\qquad \textstyle\sum_{d \in D} read\_ES(d) \cdot P(s2/ss, d/sd) \lhd ss = s1 \rhd \delta \end{aligned}$$

$$
\begin{array}{ll}
+ & comm\_SK(sd,sb)\cdot P(s3/ss,k2/ks,sd/kd,sb/kb) \lhd ss=s2 \;\wedge\; ks=k1 \rhd \delta\\
+ & comm\_ST\cdot P(s4/ss,t2/ts) \lhd ss=s3 \rhd \delta\\
+ & comm\_KR(kd,kb)\cdot P(r2/rs,k1/ks,kd/rd,kb/rb) \lhd rs=r1 \;\wedge\; ks=k2 \rhd \delta\\
+ & comm\_KR(ce)\cdot P(k1/ks) \lhd rs=r1 \;\wedge\; ks=k2 \rhd \delta\\
+ & klost\cdot P(k1/ks) \lhd ks=k2 \rhd \delta\\
+ & send\_RE(rd)\cdot P(r3/rs,1-eb/eb) \lhd rs=r2 \;\wedge\; rb=eb \rhd \delta\\
+ & comm\_RL(ac)\cdot P(r1/rs,l2/ls) \lhd (rs=r2\wedge rb=1-eb\vee rs=r3) \;\wedge\; ls=l1 \rhd \delta\\
+ & comm\_LS(ac)\cdot P(s1/ss,l1/ls,1-sb/sb) \lhd ss=s4 \;\wedge\; ls=l2 \rhd \delta\\
+ & comm\_LS(ce)\cdot P(s2/ss,l1/ls) \lhd ss=s4 \;\wedge\; ls=l2 \rhd \delta\\
+ & llost\cdot P(l1/ls) \lhd ls=l2 \rhd \delta\\
+ & comm\_TS\cdot P(s2/ss,t1/ts) \lhd ss=s4 \;\wedge\; ts=t2 \rhd \delta
\end{array}
$$

Proof: by application of a generalised version of lemma 4.1 and the encapsulation operator. Each pair of communicating actions results in a summand with a *comm* action and a condition which is the boolean "and" of the conditions under which the separate actions were enabled. Each non–communicating and non–encapsulated action from one of the processes is found as a summand of P with its original condition. □

5 From priority relation to condition

In ACP$_\theta$ the priority operator θ is used to prevent certain actions to be enabled while other actions are enabled. This operator is used, for instance, to prevent premature timeouts in communication protocols or to enforce strict sequential composition of two actions, see [Vaa90]. The priority operator is not included in ACP-based formalisms as PSF ([MV90]) and μCRL ([GP91]). This makes it difficult to "translate" an ACP$_\theta$ specification to a PSF specification or a μCRL specification. The following lemma makes such a translation possible, if the processes are specified in the UNITY Format. In this lemma the order relations on actions are translated to (extra) conditions on the summands with the actions with a lower priority.

Lemma 5.1

$$\theta(\sum_{i=1}^{n}(a_i\cdot X(D_i) \lhd c_i \rhd \delta)) = \sum_{i=1}^{n}(a_i\cdot \theta(X(D_i)) \lhd c_i \wedge \neg(\bigvee_{1\le j\le n,\ a_j>a_i} c_j) \rhd \delta)$$

Proof: We give a proof for $n=2$. A proof for $n>2$ follows the same lines.
We will prove the following identities:

1. $\theta(a_1\cdot X(D_1) \lhd c_1 \rhd \delta + a_2\cdot X(D_2) \lhd c_2 \rhd \delta) = a_1\cdot\theta(X(D_1)) \lhd c_1 \rhd \delta + a_2\cdot\theta(X(D_2)) \lhd c_2 \rhd \delta$
   if $\neg(a_1<a_2),\ \neg(a_2<a_1)$

2. $\theta(a_1\cdot X(D_1) \lhd c_1 \rhd \delta + a_2\cdot X(D_2) \lhd c_2 \rhd \delta) = a_1\cdot\theta(X(D_1)) \lhd c_1\wedge\neg c_2 \rhd \delta + a_2\cdot\theta(X(D_2)) \lhd c_2 \rhd \delta$
   if $a_1<a_2$

3. $\theta(a_1\cdot X(D_1) \lhd c_1 \rhd \delta + a_2\cdot X(D_2) \lhd c_2 \rhd \delta) = a_1\cdot\theta(X(D_1)) \lhd c_1 \rhd \delta + a_2\cdot\theta(X(D_2)) \lhd c_2\wedge\neg c_1 \rhd \delta$
   if $a_2<a_1$

Proof: We start with a general transformation of the term $\theta(\ldots)$, regardless of the specific order relation between a_1 and a_2.

$$\theta(a_1\cdot X(D_1) \lhd c_1 \rhd \delta + a_2\cdot X(D_2) \lhd c_2 \rhd \delta) \stackrel{\text{TH3}}{=}$$

$$\theta(a_1\cdot X(D_1) \lhd c_1 \rhd \delta) \lhd\!\!\lhd (a_2\cdot X(D_2) \lhd c_2 \rhd \delta) + \theta(a_2\cdot X(D_2) \lhd c_2 \rhd \delta) \lhd\!\!\lhd (a_1\cdot X(D_1) \lhd c_1 \rhd \delta) \stackrel{\text{PC1}}{=}$$

$$(\theta(a_1 \cdot X(D_1) \lhd c_1 \rhd \delta) \lll a_2 \cdot X(D_2)) \lhd c_2 \rhd (\theta(a_1 \cdot X(D_1) \lhd c_1 \rhd \delta) \lll \delta) +$$
$$(\theta(a_2 \cdot X(D_2) \lhd c_2 \rhd \delta) \lll a_1 \cdot X(D_1)) \lhd c_1 \rhd (\theta(a_2 \cdot X(D_2) \lhd c_2 \rhd \delta) \lll \delta) \stackrel{\text{P1}}{=}$$

$$(\theta(a_1 \cdot X(D_1) \lhd c_1 \rhd \delta) \lll a_2 \cdot X(D_2)) \lhd c_2 \rhd \theta(a_1 \cdot X(D_1) \lhd c_1 \rhd \delta) +$$
$$(\theta(a_2 \cdot X(D_2) \lhd c_2 \rhd \delta) \lll a_1 \cdot X(D_1)) \lhd c_1 \rhd \theta(a_2 \cdot X(D_2) \lhd c_2 \rhd \delta) \stackrel{\text{P3}}{=}$$

$$(\theta(a_1 \cdot X(D_1) \lhd c_1 \rhd \delta) \lll a_2) \lhd c_2 \rhd \theta(a_1 \cdot X(D_1) \lhd c_1 \rhd \delta) +$$
$$(\theta(a_2 \cdot X(D_2) \lhd c_2 \rhd \delta) \lll a_1) \lhd c_1 \rhd \theta(a_2 \cdot X(D_2) \lhd c_2 \rhd \delta) \stackrel{\text{THC}}{=}$$

$$((\theta(a_1 \cdot X(D_1)) \lhd c_1 \rhd \theta(\delta)) \lll a_2) \lhd c_2 \rhd (\theta(a_1 \cdot X(D_1)) \lhd c_1 \rhd \theta(\delta)) +$$
$$((\theta(a_2 \cdot X(D_2)) \lhd c_2 \rhd \theta(\delta)) \lll a_1) \lhd c_1 \rhd (\theta(a_2 \cdot X(D_2)) \lhd c_2 \rhd \theta(\delta)) \stackrel{\text{TH1}}{=}$$

$$((\theta(a_1 \cdot X(D_1)) \lhd c_1 \rhd \delta) \lll a_2) \lhd c_2 \rhd (\theta(a_1 \cdot X(D_1)) \lhd c_1 \rhd \delta) +$$
$$((\theta(a_2 \cdot X(D_2)) \lhd c_2 \rhd \delta) \lll a_1) \lhd c_1 \rhd (\theta(a_2 \cdot X(D_2)) \lhd c_2 \rhd \delta) \stackrel{\text{TH2+TH1}}{=}$$

$$((a_1 \cdot \theta(X(D_1)) \lhd c_1 \rhd \delta) \lll a_2) \lhd c_2 \rhd (a_1 \cdot \theta(X(D_1)) \lhd c_1 \rhd \delta) +$$
$$((a_2 \cdot \theta(X(D_2)) \lhd c_2 \rhd \delta) \lll a_1) \lhd c_1 \rhd (a_2 \cdot \theta(X(D_2)) \lhd c_2 \rhd \delta) \stackrel{\text{PC2}}{=}$$

$$((a_1 \cdot \theta(X(D_1)) \lll a_2) \lhd c_1 \rhd (\delta \lll a_2)) \lhd c_2 \rhd (a_1 \cdot \theta(X(D_1)) \lhd c_1 \rhd \delta) +$$
$$((a_2 \cdot \theta(X(D_2)) \lll a_1) \lhd c_2 \rhd (\delta \lll a_1)) \lhd c_1 \rhd (a_2 \cdot \theta(X(D_2)) \lhd c_2 \rhd \delta) \stackrel{\text{P1}}{=}$$

$$((a_1 \cdot \theta(X(D_1)) \lll a_2) \lhd c_1 \rhd \delta) \lhd c_2 \rhd (a_1 \cdot \theta(X(D_1)) \lhd c_1 \rhd \delta) +$$
$$((a_2 \cdot \theta(X(D_2)) \lll a_1) \lhd c_2 \rhd \delta) \lhd c_1 \rhd (a_2 \cdot \theta(X(D_2)) \lhd c_2 \rhd \delta) \stackrel{\text{P5}}{=}$$

$$((a_1 \lll a_2) \cdot \theta(X(D_1)) \lhd c_1 \rhd \delta) \lhd c_2 \rhd (a_1 \cdot \theta(X(D_1)) \lhd c_1 \rhd \delta) +$$
$$((a_2 \lll a_1) \cdot \theta(X(D_2)) \lhd c_2 \rhd \delta) \lhd c_1 \rhd (a_2 \cdot \theta(X(D_2)) \lhd c_2 \rhd \delta) \equiv \mathcal{Z}$$

From this point we distinguish three cases:

1. $\neg(a_1 < a_2), \neg(a_2 < a_1)$. We get: $\mathcal{Z} \stackrel{\text{P1}}{=}$

$$(a_1 \cdot \theta(X(D_1)) \lhd c_1 \rhd \delta) \lhd c_2 \rhd (a_1 \cdot \theta(X(D_1)) \lhd c_1 \rhd \delta) +$$
$$(a_2 \cdot \theta(X(D_2)) \lhd c_2 \rhd \delta) \lhd c_1 \rhd (a_2 \cdot \theta(X(D_2)) \lhd c_2 \rhd \delta) \stackrel{\text{lemma 2.1.1}}{=}$$

$a_1 \cdot \theta(X(D_1)) \lhd c_1 \rhd \delta + a_2 \cdot \theta(X(D_2)) \lhd c_2 \rhd \delta$ which proves lemma 5.1.1.

2. $a_1 < a_2$. We get: $\mathcal{Z} \stackrel{\text{P2}}{=}$

$$(\delta \cdot \theta(X(D_1)) \lhd c_1 \rhd \delta) \lhd c_2 \rhd (a_1 \cdot \theta(X(D_1)) \lhd c_1 \rhd \delta) +$$
$$(a_2 \cdot \theta(X(D_2)) \lhd c_2 \rhd \delta) \lhd c_1 \rhd (a_2 \cdot \theta(X(D_2)) \lhd c_2 \rhd \delta) \stackrel{\text{A7,lemma 2.1.1}}{=}$$

$$\delta \lhd c_2 \rhd (a_1 \cdot \theta(X(D_1)) \lhd c_1 \rhd \delta) + a_2 \cdot \theta(X(D_2)) \lhd c_2 \rhd \delta \stackrel{\text{lemma 2.1.2,3}}{=}$$

$a_1 \cdot \theta(X(D_1)) \lhd c_1 \wedge \neg c_2 \rhd \delta + a_2 \cdot \theta(X(D_2)) \lhd c_2 \rhd \delta$ which proves lemma 5.1.2.

3. $a_2 < a_1$. Along the same lines as the proof of lemma 5.1.2 we get:

$\mathcal{Z} = a_1 \cdot \theta(X(D_1)) \lhd c_1 \rhd \delta + a_2 \cdot \theta(X(D_2)) \lhd c_2 \wedge \neg c_1 \rhd$ which proves lemma 5.1.3. □

We will illustrate the usefulness of this lemma by a refinement of the specification of the PAR protocol. The PAR protocol is not robust with respect to premature timeouts: if the timeout interval is very short, it is impossible for the sender to determine whether a new frame or an old frame is acknowledged by the receiver. A solution to this problem is the application of the priority operator with the timeout action having a lower priority than every other action in the protocol, see [Vaa90]. A drawback of this solution is that in several formalisms related to ACP, e.g. PSF and μCRL, the priority operator is not included. This leads to other, more or less artificial, solutions to the problem of premature timeouts, see [vW93b]. Lemma 5.1 offers the possibility to "translate" priority relations on actions to extra conditions on actions with a lower priority for a process specified in the UNITY Format. Conditional process expressions are included in both PSF and μCRL.

The (refined) PAR protocol is specified by the following equation:

$$PAR(ss,rs,ts,ks,ls,sd,sb,rd,rb,eb,kd,kb) = \theta_T(\ P(ss,rs,ts,ks,ls,sd,sb,rd,rb,eb,kd,kb)\ )$$

with the process P defined in the previous section. The set T contains the order relations for the timeout: $comm\_TS < a$ where a stands for all other actions in the protocol. In line with [Vaa90] we will give the highest priority to the action $comm\_ST$, which guarantees that the timer is started immediately after the transmission of a frame. So the set T also contains the order relations $b < comm\_ST$, where b stands for all the actions in the protocol except $comm\_ST$.

Application of lemma 5.1 leads to the following equation for the process PAR:

Lemma 5.2

$$\begin{array}{ll}
\multicolumn{2}{l}{PAR(ss,rs,ts,ks,ls,sd,sb,rd,rb,eb,kd,kb) =} \\
 & \sum_{d\in D} read_ES(d)\cdot PAR(s2/ss,d/sd) \lhd ss = s1 \rhd \delta \\
+ & comm_SK(sd,sb)\cdot PAR(s3/ss,k2/ks,sd/kd,sb/kb) \lhd ss = s2\ \wedge\ ks = k1 \rhd \delta \\
+ & comm_ST\cdot PAR(s4/ss,t2/ts) \lhd ss = s3 \rhd \delta \\
+ & comm_KR(kd,kb)\cdot PAR(r2/rs,k1/ks,kd/rd,kb/rb) \\
 & \qquad \lhd rs = r1\ \wedge\ ks = k2\ \wedge\ \neg(ss = s3) \rhd \delta \\
+ & comm_KR(ce)\cdot PAR(k1/ks) \lhd rs = r1\ \wedge\ ks = k2\ \wedge\ \neg(ss = s3) \rhd \delta \\
+ & klost\cdot PAR(k1/ks) \lhd ks = k2\ \wedge\ \neg(ss = s3) \rhd \delta \\
+ & send_RE(rd)\cdot PAR(r3/rs,1-eb/eb) \lhd rs = r2\ \wedge\ rb = eb\ \wedge\ \neg(ss = s3) \rhd \delta \\
+ & comm_RL(ac)\cdot PAR(r1/rs,l2/ls) \\
 & \qquad \lhd (rs = r2 \wedge rb = 1-eb \vee rs = r3)\ \wedge\ ls = l1\ \wedge\ \neg(ss = s3) \rhd \delta \\
+ & comm_LS(ac)\cdot PAR(s1/ss,l1/ls,1-sb/sb) \lhd ss = s4\ \wedge\ ls = l2 \rhd \delta \\
+ & comm_LS(ce)\cdot PAR(s2/ss,l1/ls) \lhd ss = s4\ \wedge\ ls = l2 \rhd \delta \\
+ & llost\cdot PAR(l1/ls) \lhd ls = l2\ \wedge\ \neg(ss = s3) \rhd \delta \\
+ & comm_TS\cdot PAR(s2/ss,t1/ts) \lhd ss = s4\ \wedge\ ts = t2\ \wedge\ rs = r1\ \wedge\ ks = k1\ \wedge\ ls = l1 \rhd \delta
\end{array}$$

Proof: according to lemma 5.1 the order relations $b < comm\_ST$ for $b \neq comm\_ST$ lead to an extra conjunct $\neg(ss = s3)$ in the condition of each summand except the summand with $comm\_ST$ itself. Now we suppose that for each process the number of states is finite and the states are unique (PS is the set of process states):

$$\forall i \in PS : ((ps = i) \iff \forall j \in PS, j \neq i : (ps \neq j)).$$

So, in each condition with a conjunct $(ss = si)$, $i \neq 3$, the conjunct $\neg(ss = s3)$ is superfluous. This means that in six summands of the equation for the process PAR the conjunct $\neg(ss = s3)$ in the condition can be deleted.

The condition in the last summand is derived in the following way. As each action has a higher priority than the action $comm\_TS$, the conditions related to all these actions will appear as additional negative conditions in the last summand. So this condition now is equal to

$$(ss = s4) \wedge (ts = t2) \wedge \neg(ss = s1) \wedge \neg(ss = s2 \wedge ks = k1) \wedge \ldots$$
$$\neg(ls = l2 \wedge \neg(ss = s3)) \equiv to\_condition.$$

Some straightforward boolean calculations lead to the following identity:

$$\begin{aligned} to\_condition = {} & (ss = s4) \wedge (ts = t2) \wedge \neg(\, (ss = s1) \vee (ss = s2 \wedge ks = k1) \vee (ss = s3) \vee \\ & (ks = k2 \wedge \neg(ss = s3)) \vee (rs = r2 \wedge eb = rb \wedge \neg(ss = s3)) \vee \\ & (rs = r2 \wedge eb = 1 - rb \wedge ls = l1 \wedge \neg(ss = s3)) \vee (rs = r3 \wedge ls = l1 \wedge \neg(ss = s3)) \vee \\ & (ss = s4 \wedge ls = l2) \vee (ls = l2 \wedge \neg(ss = s3))) \end{aligned}$$

Now we use the fact that a bit variable has two possible values, so

$$(eb = rb) \Longleftrightarrow (eb \neq 1 - rb) \quad , \quad (eb = rb) \Longleftrightarrow (eb = 1 - rb).$$

Furthermore, we have already noticed that for each process the number of states is finite and the states are unique. So, $\neg(ks = k2)$ implies $ks = k1$, since only two values are possible for ks. In the same way the other parts of the timeout condition are derived. □

It should be noticed that the derived condition is intuitively clear: a timeout is needed (and permitted), only if the sender process is waiting for input from the channel L ($ss = s4$), the timer is running ($ts = t2$), but all other components are at rest ($rs = r1, ks = k1, ls = l1$). In this state a timeout is needed to re–activate several components.

The equation for the process PAR contains no priority operator. This means that it can be translated into specification formalisms like PSF and μCRL. A PSF specification of the process PAR, together with the data modules needed, is given in appendix A.

6 A corresponding Term Rewriting System

In the UNITY Format for process specifications the behaviour of a process in terms of state transitions is completely governed by conditions on data and functions on data. This raises the question to what extend something meaningful can be stated about the validation or verification of a specification by looking only at the data part. In this section we will give a tentative answer on this question.

We start with the introduction of a Term Rewriting System (TRS) for a function, derived in three steps from the specification of a process X, specified in the UNITY Format.

1. Delete from each summand in the definition of X the atomic action.
2. Define a function f of some sort S with the same arguments as the process name X, excluded the arguments that are not related to the conditions.
3. Consider each "amputated" summand as a rewrite rule from an instantiation of f to another instantiation of f in which, if possible, the conditions on the data parameters appear in the arguments of f at the left side of a rule and the data substitutions appear in the arguments of f at the right side of the rule. If the conditions cannot be reflected in data parameters at the left side, a conditional rewrite rule will be needed. The notation of a conditional rewrite rule is $R \Leftarrow c$ with R a rewrite rule and c the condition. Only if c evaluates to true, the application of R is permitted.

Such a TRS is an abstract representation of the process X. Each rewrite rule resembles a state transition of X. As an example we will derive the TRS corresponding with the PAR protocol. Therefore we define a function par with the arguments $ss, rs, ts, ks, ls, sb, rb, eb, kb$. (The "data arguments" sd, rd and kd are left out.) According to the steps described above we get the following TRS (the rules are numbered, so they can be referred to later on):

1. $par(s1, rs, ts, ks, ls, sb, rb, eb, kb) \rightarrow par(s2, rs, ts, ks, ls, sb, rb, eb, kb)$
2. $par(s2, rs, ts, k1, ls, sb, rb, eb, kb) \rightarrow par(s3, rs, ts, k2, ls, sb, rb, eb, sb)$
3. $par(s3, rs, ts, ks, ls, sb, rb, eb, kb) \rightarrow par(s4, rs, t2, ks, ls, sb, rb, eb, kb)$
4. $par(ss, r1, ts, k2, ls, sb, rb, eb, kb) \rightarrow par(ss, r2, ts, k1, ls, sb, kb, eb, kb) \;\Leftarrow\; ss \neq s3$
5. $par(ss, r1, ts, k2, ls, sb, rb, eb, kb) \rightarrow par(ss, r1, ts, k1, ls, sb, rb, eb, kb) \;\Leftarrow\; ss \neq s3$
6. $par(ss, rs, ts, k2, ls, sb, rb, eb, kb) \rightarrow par(ss, rs, ts, k1, ls, sb, rb, eb, kb) \;\Leftarrow\; ss \neq s3$
7. $par(ss, r2, ts, ks, ls, sb, eb, eb, kb) \rightarrow par(ss, r3, ts, ks, ls, sb, eb, 1-eb, kb) \;\Leftarrow\; ss \neq s3$
8. $par(ss, r2, ts, ks, l1, sb, 1-eb, eb, kb) \rightarrow par(ss, r1, ts, ks, l2, sb, 1-eb, eb, kb) \;\Leftarrow\; ss \neq s3$
9. $par(ss, r3, ts, ks, l1, sb, rb, eb, kb) \rightarrow par(ss, r1, ts, ks, l2, sb, rb, eb, kb) \;\Leftarrow\; ss \neq s3$
10. $par(s4, rs, ts, ks, l2, sb, rb, eb, kb) \rightarrow par(s1, rs, ts, ks, l1, 1-sb, rb, eb, kb)$
11. $par(s4, rs, ts, ks, l2, sb, rb, eb, kb) \rightarrow par(s2, rs, ts, ks, l1, sb, rb, eb, kb)$
12. $par(ss, rs, ts, ks, l2, sb, rb, eb, kb) \rightarrow par(ss, rs, ts, ks, l1, sb, rb, eb, kb) \;\Leftarrow\; ss \neq s3$
13. $par(s4, r1, t2, k1, l1, sb, rb, eb, kb) \rightarrow par(s2, r1, t1, k1, l1, sb, rb, eb, kb)$

What does this strange–looking TRS tell us about the PAR protocol? We will look at some properties of the protocol which should correspond with some properties of the TRS. If these properties can be derived by an (automated) analysis of the TRS, we have a kind of verification or validation of the related protocol properties. In the following we will look at three protocol properties:

1. Deadlock freedom.
2. Guaranteed delivery of data.
3. The need of the exclusion of premature timeouts.

In this section we will analyse the relation between these properties and the TRS as defined above. In appendix B we will investigate to what extend the properties can be derived with the tools from the PSF Toolkit (shortly described in [MV93]).

Before we look at the protocol properties we will derive all possible rewrite sequences starting with the term $par(s1, r1, t1, k1, l1, 0, 1, 0, 0)$. This term represents the initial state of the protocol. The numbers above the arrows refer to the numbers of the rewrite rules from the TRS. The symbol b denotes a bit value.

$$par(s1, r1, t1, k1, l1, 0, 1, 0, 0) \xrightarrow{1.} par(s2, r1, t1, k1, l1, 0, 1, 0, 0) \xrightarrow{2.}$$
$$par(s3, r1, t1, k2, l1, 0, 1, 0, 0) \xrightarrow{3.} par(s4, r1, t2, k2, l1, 0, 1, 0, 0) \equiv \mathcal{A}(0, 1, 0, 0)$$

$$\mathcal{A}(b, 1-b, b, b) \xrightarrow{4.} par(s4, r2, t2, k1, l1, b, b, b, b) \xrightarrow{7.} par(s4, r3, t2, k1, l1, b, b, 1-b, b) \xrightarrow{9.}$$
$$par(s4, r1, t2, k1, l2, b, b, 1-b, b) \equiv \mathcal{B}(b, b, 1-b, b)$$

$\mathcal{A}(b,1-b,b,b) \xrightarrow{5./6.} par(s4,r1,t2,k1,l1,b,1-b,b,b) \xrightarrow{13.} par(s2,r1,t1,k1,l1,b,1-b,b,b) \xrightarrow{2.}$
$par(s3,r1,t1,k2,l1,b,1-b,b,b) \xrightarrow{3.} par(s4,r1,t2,k2,l1,b,1-b,b,b) \equiv \mathcal{A}(b,1-b,b,b)$

$\mathcal{A}(b,b,1-b,b) \xrightarrow{4.} par(s4,r2,t2,k1,l1,b,b,1-b,b) \xrightarrow{8.} par(s4,r1,t2,k1,l2,b,b,1-b,b) \equiv$
$\mathcal{B}(b,b,1-b,b)$

$\mathcal{A}(b,b,1-b,b) \xrightarrow{5./6.} par(s4,r1,t2,k1,l1,b,b,1-b,b) \xrightarrow{13.} par(s2,r1,t1,k1,l1,b,b,1-b,b) \xrightarrow{2.}$
$par(s3,r1,t1,k2,l1,b,b,1-b,b) \xrightarrow{3.} par(s4,r1,t2,k2,l1,b,b,1-b,b) \equiv \mathcal{A}(b,b,1-b,b)$

$\mathcal{B}(b,b,1-b,b) \xrightarrow{10.} par(s1,r1,t2,k1,l1,1-b,b,1-b,b) \xrightarrow{1.} par(s2,r1,t2,k1,l1,1-b,b,1-b,b) \xrightarrow{2.}$
$par(s3,r1,t2,k2,l1,1-b,b,1-b,1-b) \xrightarrow{3.} par(s4,r1,t2,k2,l1,1-b,b,1-b,1-b) \equiv$
$\mathcal{A}(1-b,b,1-b,1-b)$

$\mathcal{B}(b,b,1-b,b) \xrightarrow{11.} par(s2,r1,t2,k1,l1,b,b,1-b,b) \xrightarrow{2.} par(s3,r1,t2,k2,l1,b,b,1-b,b) \xrightarrow{3.}$
$par(s4,r1,t2,k2,l1,b,b,1-b,b) \equiv \mathcal{A}(b,b,1-b,b)$

$\mathcal{B}(b,b,1-b,b) \xrightarrow{12.} par(s4,r1,t2,k1,l1,b,b,1-b,b) \xrightarrow{13.} par(s2,r1,t1,k1,l1,b,b,1-b,b) \xrightarrow{2.}$
$par(s3,r1,t1,k2,l1,b,b,1-b,b) \xrightarrow{3.} par(s4,r1,t2,k2,l1,b,b,1-b,b) \equiv \mathcal{A}(b,b,1-b,b)$

In Figure 2 the corresponding rewrite graph is given.

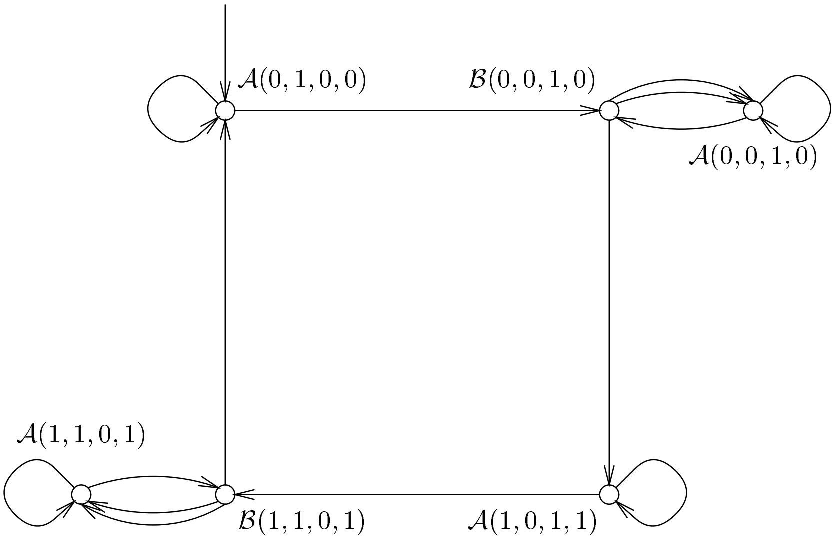

Figure 2: Rewrite graph of the TRS.

1. Deadlock freedom.

Like other distributed algorithms, the PAR protocol is supposed to be free of deadlocks. For the TRS this means that, starting from the initial term, we must get a non–terminating sequence of rewritings, the TRS is supposed to be not (weak or strong) terminating. Analysis of all possible rewrite sequences will have to show that each sequence is cyclic without any exits. From Figure 2 it is immediately clear that for the initial term there is no terminating rewrite sequence, from which we may conclude that the PAR protocol is deadlock–free.

2. Guaranteed delivery of data.
If a data element is read in by the sender process, sooner or later it has to be delivered at the other side by the receiver process. Input of data (reflected in rewrite rule 1.) takes place in the rewrite sequence $\mathcal{B}(b,b,1-b,b) \twoheadrightarrow \mathcal{A}(1-b,b,1-b,1-b)$ (the vertical arrows in the main square of Figure 2). Output of data (reflected in rewrite rule 7.) takes place in the rewrite sequence $\mathcal{A}(b,1-b,b,b) \twoheadrightarrow \mathcal{B}(b,b,1-b,b)$ (the horizontal arrows in the main square of the same Figure). From the rewrite graph it is immediately clear that no two input actions are possible without an output action in between and vice versa. If we adopt a certain notion of fairness for our TRS ("in a cyclic TRS each rewrite cycle will be chosen sooner or later"), we may conclude that each input action is followed by an output action sooner or later. Note: as we have abstracted in our TRS from the inputted data element itself, we cannot explicitly verify that a specific data element is delivered by the receiver.

3. Premature timeouts.
As mentioned before, the PAR protocol is not robust with respect to premature timeouts. In the specification of the protocol in section 5 this has lead to an extra conjunct in the condition on the summand with the timeout action: $(rs = r1) \wedge (ks = k1) \wedge (ls = l1)$. If we remove this extra conjunct, we can show that things go wrong by analysing the TRS with a modified last rewrite rule:

$$13' \;\; par(s4, rs, t2, ks, ls, sb, rb, eb, kb) \rightarrow par(s2, rs, t1, ks, ls, sb, rb, eb, kb)$$

In the modified TRS the following rewrite sequence can be derived:

$$\begin{array}{l} \mathcal{B}(b,b,1-b,b) \equiv par(s4, r1, t2, k1, l2, b, b, 1-b, b) \xrightarrow{13'} \ldots \xrightarrow{2.} \ldots \xrightarrow{3.} \ldots \xrightarrow{4.} \ldots \xrightarrow{10.} \\ \quad \ldots \xrightarrow{1.} \ldots \xrightarrow{2.} \ldots \xrightarrow{3.} \ldots \xrightarrow{6.} \ldots \xrightarrow{8.} \ldots \xrightarrow{10.} \ldots \xrightarrow{1.} \ldots \end{array}$$

This rewrite sequence corresponds with two input actions at the sender side (rewrite rule 1.) without an intermediate output action at the receiver side (rewrite rule 7.): one data element is lost somewhere in the protocol. In the original TRS this sequence is not possible: the extra conditions on the last summand of the protocol specification are necessary in order to guarantee a protocol without incorrect behaviour like the sequence described above.

7 Conclusions

The data oriented specification style as presented in this paper can be evaluated by comparing it to the more traditional process oriented specification style. In this section we present as a set of conclusions such a comparison with respect to a number of different aspects concerning specification, validation and verification of processes.

1. Writing a specification. Writing a process oriented specification in the form of a set of linear recursive equations usually is rather simple. Writing a specification in the UNITY Format is possibly slightly more difficult, because one has to be aware of the process state which has to be added to each summand, both in the process parameters and in the condition.

2. Readability of a specification. In a process oriented style meaningful identifiers can be chosen for each process name in the set of linear recursive equations. In a data oriented style only a single identifier is used for a process. Together with the overhead of conditions and extra process parameters this makes a specification in the UNITY Format harder to understand than a process oriented specification. A process oriented specification in general will be more close to human intuitions about

process behaviour.

3. Normalising a specification. The normalisation of the parallel composition of several processes specified in the UNITY Format is quite easy: communicating actions are replaced by the defined communication action, the condition for this summand is the boolean and of the separate conditions (see section 4). In a process oriented specification style normalisation usually requires a lot of non–trivial calculations. With respect to this item we conclude that the data oriented specification style is highly in favour of a process oriented style.

4. The priority operator. As we have seen in section 5, in the UNITY Format priority relations between actions can be translated to extra conditions on summands. This makes such a specification executable in a formalism in which the priority operator is not defined, e.g. PSF. This can be regarded as a minor advantage of the UNITY Format.

5. Validation and verification. The relation between a specification in the UNITY Format and a term rewriting system opens certain perspectives in the field of validation and verification. Requirement properties of a specification can be formulated in terms of required properties of the corresponding TRS. As we have seen in section 6, the analysis of the TRS by hand is only manageable for small specifications. The TRS for the PAR protocol already gets close to the limits of what is feasible without tool support. However, in appendix B it is made clear that at least the PSF tools for term rewriting are not very useful for our purposes.

In this paper we will not present a final conclusion in favour of a data oriented or a process oriented specification style. From the items above it is clear that for specification purposes only, a process oriented style might be preferable because of its greater readability. For validation and verification purposes the (often needed) normalisation of the parallel composition of several processes is much easier to perform for specifications in the UNITY Format. It is not very clear how helpful a corresponding TRS can be in this field. More case studies will be needed and more advanced tools will have to be developed, before a firm answer on this question can be given. Another case study is presented in [BKKM93] in which three leader election protocols are specified in the format that has been presented in this paper.

Acknowledgements. Thanks to Sjouke Mauw (TU Eindhoven) for some initiating ideas concerning data oriented process specifications. Thanks to Jan Bergstra for his comments on an earlier version of this paper.

References

[Bae90] J.C.M. Baeten, editor. *Applications of Process Algebra.* Cambridge Tracts in Theoretical Computer Science 17. Cambridge University Press, 1990.

[BB92] J.C.M. Baeten and J.A. Bergstra. Process algebra with signals and conditions. In M. Broy, editor, *Programming and Mathematical Methods, Proceedings Summer School Marktoberdorf 1990*, pages 273–323. Springer, 1992.

[BBK86] J.C.M. Baeten, J.A. Bergstra, and J.W. Klop. Syntax and defining equations for an interrupt mechanism in process algebra. *Fund. Inf.*, IX:127–168, 1986.

[BG93] M.A. Bezem and J.F. Groote. A correctness proof of a One-bit Sliding Window Protocol in μCRL. Technical Report Logic Group Preprint Series nr. 99, Department of Philosophy, Utrecht University, 1993.

[BKKM93] J.J. Brunekreef, J.P. Katoen, R.L.C. Koymans, and S. Mauw. Design and analysis of dynamic leader election protocols in broadcast networks. Technical Report P9324, Programming Research Group, University of Amsterdam, 1993.

[BW90] J.C.M Baeten and W.P. Weijland. *Process Algebra.* Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.

[CM88] K.M. Chandy and J. Misra. *Parallel program design: a foundation.* Addison Wesley, Reading (MA), 1988.

[GP91] J.F. Groote and A. Ponse. μCRL: A base for analyzing processes with data. In E. Best and G. Rozenberg, editors, *Proceedings of the 3^{rd} Workshop on Concurrency and Compositionality*, pages 125–130. Universität Hildesheim, 1991.

[MV90] S. Mauw and G.J. Veltink. A Process Specification Formalism. *Fund. Inf.*, XII:85–139, 1990.

[MV93] S. Mauw and G.J. Veltink, editors. *Algebraic specification of communication protocols.* Cambridge Tracts in Theoretical Computer Science 36. Cambridge University Press, 1993.

[Vaa90] F.W. Vaandrager. Two simple protocols. In Baeten [Bae90], pages 23–44.

[vW93a] J.J. van Wamel. A library for PSF. Technical Report P9301, Programming Research Group, University of Amsterdam, 1993.

[vW93b] J.J. van Wamel. Simple protocols. In Mauw and Veltink [MV93], pages 47–70.

A A PSF specification of the PAR protocol

In this appendix we give a PSF specification of the PAR protocol. The specification is a straightforward translation of the specification given in section 5. First we give the data modules. Several modules from the standard PSF library ([vW93a]) are imported. A parameterised module `States` contains a generic specification of process states. In order to simplify the specifications of the equality function on process states, a process state is specified as a function `state` with a digit string as input argument. So the state $s1$ is specified as `s(^1)`. In the data module `PARstates` the actual states for the PAR protocol are specified. In the data module `TimeOutCondition` these states are bound to the generic state specification by repeated import of the module `States` with different parameter bindings. In this module the timeout condition, $ss = s4 \wedge ts = t2 \wedge rs = r1 \wedge ks = k1 \wedge ls = l1$, is specified.

```
data module Frames
begin
  exports
  begin
    sorts
      FRAME
    functions
      frame : DATA # BIT -> FRAME
      ac    :            -> FRAME
      ce    :            -> FRAME
  end
  imports
    Sample-Data, Bits
end Frames

data module States
begin
  parameters
  Astate
  begin
    sorts
      STATE
    functions
      state : DIGIT-STRING -> STATE
  end Astate
  exports
  begin
    functions
      eq : STATE # STATE -> BOOLEAN
  end
  imports
    Digit-Strings
  variables
    x,y : -> DIGIT-STRING
  equations
  [1]  eq(state(x),state(y)) = eq(x,y)
end States

data module PARstates
begin
  exports
```

```
  begin
    sorts
      SS, RS, TS, KS, LS
    functions
      s : DIGIT-STRING -> SS
      r : DIGIT-STRING -> RS
      t : DIGIT-STRING -> TS
      k : DIGIT-STRING -> KS
      l : DIGIT-STRING -> LS
  end
  imports
    Digit-Strings
end PARstates

data module TimeOutCondition
begin
  exports
  begin
    functions
      to-condition : SS # RS # TS # KS # LS -> BOOLEAN
  end
  imports
    States {Astate bound by [STATE -> SS, state -> s] to PARstates},
    States {Astate bound by [STATE -> RS, state -> r] to PARstates},
    States {Astate bound by [STATE -> TS, state -> t] to PARstates},
    States {Astate bound by [STATE -> KS, state -> k] to PARstates},
    States {Astate bound by [STATE -> LS, state -> l] to PARstates}
  variables
    ss : -> SS
    rs : -> RS
    ts : -> TS
    ks : -> KS
    ls : -> LS
  equations
  [1]  to-condition(ss,rs,ts,ks,ls) =
         and(and(and(and(eq(ss,s(^4)),eq(ts,t(^2))),
         eq(rs,r(^1))),eq(ks,k(^1))),eq(ls,l(^1)))
end TimeOutCondition
```

The process module Par is a straightforward translation into PSF of the definition of the process PAR in section 5.

```
process module Par
begin
  exports
  begin
    atoms
      read-ES : DATA
      send-RE : DATA
      comm-SK : FRAME
      comm-KR : FRAME
      comm-RL : FRAME
      comm-LS : FRAME
      comm-ST
      comm-TS
      klost
```

```
      llost
    processes
      PAR : SS # RS # TS # KS # LS # DATA # BIT # DATA # BIT # BIT #
            DATA # BIT
      Start
  end
  imports
    Frames, TimeOutCondition
  variables
    ss : -> SS
    rs : -> RS
    ts : -> TS
    ks : -> KS
    ls : -> LS
    sd,rd,kd     : -> DATA
    sb,rb,eb,kb : -> BIT
  definitions
     PAR(ss,rs,ts,ks,ls,sd,sb,rd,rb,eb,kd,kb) =
        [eq(ss,s(^1)) = true] -> sum(d in DATA, read-ES(d).
            PAR(s(^2),rs,ts,ks,ls,d,sb,rd,rb,eb,kd,kb))
      + [and(eq(ss,s(^2)),eq(ks,k(^1))) = true] -> comm-SK(frame(sd,sb)).
            PAR(s(^3),rs,ts,k(^2),ls,sd,sb,rd,rb,eb,sd,sb)
      + [eq(ss,s(^3)) = true] -> comm-ST.
            PAR(s(^4),rs,t(^2),ks,ls,sd,sb,rd,rb,eb,kd,kb)
      + [and(and(eq(rs,r(^1)),eq(ks,k(^2))),not(eq(ss,s(^3)))) = true] ->
            comm-KR(frame(kd,kb)).
            PAR(ss,r(^2),ts,k(^1),ls,sd,sb,kd,kb,eb,kd,kb)
      + [and(and(eq(rs,r(^1)),eq(ks,k(^2))),not(eq(ss,s(^3)))) = true] -> comm-KR(ce).
            PAR(ss,rs,ts,k(^1),ls,sd,sb,rd,rb,eb,kd,kb)
      + [and(eq(ks,k(^2)),not(eq(ss,s(^3)))) = true] -> klost.
            PAR(ss,rs,ts,k(^1),ls,sd,sb,rd,rb,eb,kd,kb)
      + [and(and(eq(rs,r(^2)),eq(rb,eb)),not(eq(ss,s(^3)))) = true] -> send-RE(rd).
            PAR(ss,r(^3),ts,ks,ls,sd,sb,rd,rb,bit-inv(eb),kd,kb)
      + [and(and((or(and(eq(rs,r(^2)),eq(rb,bit-inv(eb))),eq(rs,r(^3))),
         eq(ls,l(^1))),not(eq(ss,s(^3)))) = true] -> comm-RL(ac).
            PAR(ss,r(^1),ts,ks,l(^2),sd,sb,rd,rb,eb,kd,kb)
      + [and(eq(ss,s(^4)),eq(ls,l(^2))) = true] -> comm-LS(ac).
            PAR(s(^1),rs,ts,ks,l(^1),sd,bit-inv(sb),rd,rb,eb,kd,kb)
      + [and(eq(ss,s(^4)),eq(ls,l(^2))) = true] -> comm-LS(ce).
            PAR(s(^2),rs,ts,ks,l(^1),sd,sb,rd,rb,eb,kd,kb)
      + [and(eq(ls,l(^2)),not(eq(ss,s(^3)))) = true] -> llost.
            PAR(ss,rs,ts,ks,l(^1),sd,sb,rd,rb,eb,kd,kb)
      + [to-condition(ss,rs,ts,ks,ls) = true] -> comm-TS.
            PAR(s(^2),rs,t(^1),ks,ls,sd,sb,rd,rb,eb,kd,kb)

    Start = PAR(s(^1),r(^1),t(^1),k(^1),l(^1),d1,0',d1,1',0',d1,0')
end Par
```

The specification has been tested with the PSF simulator. All traces showed a desired behaviour of the PAR protocol.

An interesting question from a practical point of view is how the PSF simulator performs with normalised data oriented specifications, compared to specifications in the usual process oriented style (consisting of an encapsulated merge of several parallel processes). In a data oriented specification for each step (action) the simulator has to evaluate a number of conditions, which means a lot of rewriting on data functions. In a process oriented specification for each step a more or less complex process term has to be evaluated in order to determine which actions are enabled in a certain state. In order to get a kind of answer on the performance question, simulation runs of two specifications of the PAR protocol have been compared: the specification given in this appendix and the specification given in [vW93b]. The simulation runs showed no significant difference with respect to the number of observed input and output actions produced in a certain time interval. Measurements on other specification pairs will have to bring more clarity on this point.

B Analysis of a corresponding TRS with the PSF Toolkit

It is common practise to validate a PSF specification by means of simulation runs, generated by the tool `sim` from the PSF Toolkit. In this appendix we will investigate to what extend data oriented PSF tools like `trs` (a term rewriter) and `initial` (calculates (part of) the initial algebra of a given data specification) can be used for validation purposes. As we have seen in section 6 it is possible to analyse by hand a TRS for a small process like the PAR protocol. It would be helpful if for the validation of bigger processes computer tools could be used.

First we will take a closer look at the tools mentioned above. The tool `trs` produces for a given term a single finite rewrite sequence, if any exists. The tool does not produce *all* possible rewrite sequences. Furthermore, it does not signal cycles in sequences. Its algorithm is simple: from above the first redex is sought that matches the term that has to be rewritten. The concerned rewrite rule is executed, after which the same process is repeated with the reduct, until no match can be found any more. Optionally it is possible to look for a matching redex from below instead of from above. The tool `initial` produces (a part of) the initial algebra of a given specification. First the constants of a sort for which no rewrite rule exists are given. Next the terms with functions on these constants are added, as far as they cannot be rewritten to a constant. In a third step terms with these new elements as input for the functions are analysed, and so on. The number of initial algebra elements that has to be calculated can be specified by the user. In general, for a process–like TRS there will be many combinations of input arguments for the function representing the process, for which no rewrite rule exists in the TRS. This means that the tool `initial` will produce many normal forms for the given function. In theory it is possible to draw conclusions from a complete set of normal forms, in practise it will be too complicated. Therefore we conclude that the tool `initial` is not appropriate for analysing a TRS corresponding with a process. In the sequel we will only look at the tool `trs`.

Before we discuss the usefulness of this tool for our validation purposes it is good to be aware of the fact that the data oriented PSF tools were not designed for such purposes. The tool `trs` is designed to analyse strong terminating and confluent term rewriting systems without cycles. The tool `initial` is designed to calculate the initial algebra of a meaningful abstract data type.

In section 6 the TRS corresponding with the PAR protocol is given. The following data module contains the PSF specification of the function *par*, with the rewrite rules as equations.

```
data module PAR-TRS
begin
  exports
    begin
      sorts
        S
      functions
        par : SS # RS # TS # KS # LS # BIT # BIT # BIT # BIT -> S
    end
  imports
    TRS-data, Bits
  variables
    ss          : -> SS
```

```
    rs           : -> RS
    ts           : -> TS
    ks           : -> KS
    ls           : -> LS
    sb,rb,eb,kb  : -> BIT
  equations
  [01] par(s(^1),rs,ts,ks,ls,sb,rb,eb,kb) =
          par(s(^2),rs,ts,ks,ls,sb,rb,eb,kb)
  [02] par(s(^2),rs,ts,k(^1),ls,sb,rb,eb,kb) =
          par(s(^3),rs,ts,k(^2),ls,sb,rb,eb,sb)
  [03] par(s(^3),rs,ts,ks,ls,sb,rb,eb,kb) =
          par(s(^4),rs,t(^2),ks,ls,sb,rb,eb,kb)
  [04] par(ss,r(^1),ts,k(^2),ls,sb,rb,eb,kb) =
          par(ss,r(^2),ts,k(^1),ls,sb,kb,eb,kb) when not(eq(ss,s(^3))) = true
  [05] par(ss,r(^1),ts,k(^2),ls,sb,rb,eb,kb) =
          par(ss,r(^1),ts,k(^1),ls,sb,rb,eb,kb) when not(eq(ss,s(^3))) = true
  [06] par(ss,rs,ts,k(^2),ls,sb,rb,eb,kb) =
          par(ss,rs,ts,k(^1),ls,sb,rb,eb,kb)    when not(eq(ss,s(^3))) = true
  [07] par(ss,r(^2),ts,ks,ls,sb,rb,eb,kb) =
          par(ss,r(^3),ts,ks,ls,sb,rb,bit-inv(eb),kb)
                               when and(eq(rb,eb),not(eq(ss,s(^3)))) = true
  [08] par(ss,r(^2),ts,ks,l(^1),sb,rb,eb,kb) =
          par(ss,r(^1),ts,ks,l(^2),sb,rb,eb,kb)
                          when and(not(eq(rb,eb)),not(eq(ss,s(^3)))) = true
  [09] par(ss,r(^3),ts,ks,l(^1),sb,rb,eb,kb) =
          par(ss,r(^1),ts,ks,l(^2),sb,rb,eb,kb) when not(eq(ss,s(^3))) = true
  [10] par(s(^4),rs,ts,ks,l(^2),sb,rb,eb,kb) =
          par(s(^1),rs,ts,ks,l(^1),bit-inv(sb),rb,eb,kb)
  [11] par(s(^4),rs,ts,ks,l(^2),sb,rb,eb,kb) =
          par(s(^2),rs,ts,ks,l(^1),sb,rb,eb,kb)
  [12] par(ss,rs,ts,ks,l(^2),sb,rb,eb,kb) =
          par(ss,rs,ts,ks,l(^1),sb,rb,eb,kb)    when not(eq(ss,s(^3))) = true
  [13] par(s(^4),r(^1),t(^2),k(^1),l(^1),sb,rb,eb,kb) =
          par(s(^2),r(^1),t(^1),k(^1),l(^1),sb,rb,eb,kb)
end PAR-TRS
```

In section 6 the TRS was analysed with respect to three protocol properties: deadlock freedom, guaranteed delivery of data and the harm of premature timeouts. How useful is the tools `trs` with respect to these properties?

1. Deadlock freedom. The tool `trs` only produces one out of possibly many rewrite sequences for a given term and a given specification. Although this sequence may be non–terminating (which can only be observed by the absence of response from `trs`), other sequences might. The only way to validate non–termination of *each* rewrite sequence is a test-run with each permutation of the set of rewrite rules. It may be clear that for a TRS with more than a few rewrite rules this is not a practical option. Therefore we conclude that the tool `trs` is not appropriate for the validation of the absence of deadlocks. For this purpose a stronger tool is required that produces *all* possible rewrite sequences for a given term and that is aware of cycles in these sequences.

2. Guaranteed delivery of data. In general it is possible to show with the tool `trs` the existence of a specific rewrite sequence from a given term to another term. Sometimes the ordering of the rewrite rules will have to be chosen carefully in order to get the desired sequence. If the final term of the sequence is not a normal form (in general this will be the case), then a sequence terminating with the desired term can be obtained by the insertion of an extra rewrite rule in which the final term

rewrites to a term that is a normal form (e.g. a special defined constant) or by a (temporal) removal of all the rewrite rules for the final term. However, the existence of a specific rewrite sequence does not guarantee that all sequences starting with a certain term will end up in the same final term (or will terminate at all). For the TRS corresponding with the PAR protocol this means that we can validate the existence of a rewrite sequence from the term (state) that corresponds with the input of data to the term (state) that corresponds with its delivery ("it is possible that inputted data is delivered"). A validation of *guaranteed* delivery of data is not possible.

3. Premature timeouts. As noticed before, it is possible to validate the existence of a specific rewrite sequence with the tool `trs`. This means that it is possible in the modified TRS for the PAR protocol (see section 6) to derive the rewrite sequence stated there. The following output has been derived from the tool `trs`. The output is edited: it was not possible to derive this rewriting sequence in a single run, several parts have been derived while using the `-r` option of `trs`. Also, the numbers of the applied rewrite rules have been added (between square brackets).

```
par(s(^_(4)),r(^_(1)),t(^_(2)),k(^_(1)),l(^_(2)),0',0',1',0') ->     [13']
par(s(^_(2)),r(^_(1)),t(^_(1)),k(^_(1)),l(^_(2)),0',0',1',0')

par(s(^_(2)),r(^_(1)),t(^_(1)),k(^_(1)),l(^_(2)),0',0',1',0') ->     [02]
par(s(^_(3)),r(^_(1)),t(^_(1)),k(^_(2)),l(^_(2)),0',0',1',0')

par(s(^_(3)),r(^_(1)),t(^_(1)),k(^_(2)),l(^_(2)),0',0',1',0') ->     [03]
par(s(^_(4)),r(^_(1)),t(^_(2)),k(^_(2)),l(^_(2)),0',0',1',0')

   [condition:  not(eq(s(^_(4)),s(^_(3)))) =  true]
       eq(s(^_(4)),s(^_(3)))
      ->  eq(^_(4),^_(3))
       eq(^_(4),^_(3))
      ->  false
       not(false)
      ->  true
   [success]
par(s(^_(4)),r(^_(1)),t(^_(2)),k(^_(2)),l(^_(2)),0',0',1',0') ->     [04]
par(s(^_(4)),r(^_(2)),t(^_(2)),k(^_(1)),l(^_(2)),0',0',1',0')

   [condition:  and(eq(0',1'),not(eq(s(^_(4)),s(^_(3))))) =  true]
       eq(s(^_(4)),s(^_(3)))
      ->  eq(^_(4),^_(3))
       eq(^_(4),^_(3))
      ->  false
       not(false)
      ->  true
       eq(0',1')
      ->  false
       and(false,true)
      ->  false
   [failure]    -- on rule [07]

par(s(^_(4)),r(^_(2)),t(^_(2)),k(^_(1)),l(^_(2)),0',0',1',0')
       bit-inv(0') ->  1'                                        ->     [10]
par(s(^_(1)),r(^_(2)),t(^_(2)),k(^_(1)),l(^_(1)),1',0',1',0')

par(s(^_(1)),r(^_(2)),t(^_(2)),k(^_(1)),l(^_(1)),1',0',1',0') ->     [01]
par(s(^_(2)),r(^_(2)),t(^_(2)),k(^_(1)),l(^_(1)),1',0',1',0')
```

```
par(s(^_(2)),r(^_(2)),t(^_(2)),k(^_(1)),l(^_(1)),1',0',1',0') ->     [02]
par(s(^_(3)),r(^_(2)),t(^_(2)),k(^_(2)),l(^_(1)),1',0',1',1')

par(s(^_(3)),r(^_(2)),t(^_(2)),k(^_(2)),l(^_(1)),1',0',1',1') ->     [03]
par(s(^_(4)),r(^_(2)),t(^_(2)),k(^_(2)),l(^_(1)),1',0',1',1')

   [condition:  not(eq(s(^_(4)),s(^_(3)))) =  true]
       eq(s(^_(4)),s(^_(3)))
      ->  eq(^_(4),^_(3))
       eq(^_(4),^_(3))
      ->  false
       not(false)
      ->  true
   [success]
par(s(^_(4)),r(^_(2)),t(^_(2)),k(^_(2)),l(^_(1)),1',0',1',1') ->     [06]
par(s(^_(4)),r(^_(2)),t(^_(2)),k(^_(1)),l(^_(1)),1',0',1',1')

   [condition:  and(eq(0',1'),not(eq(s(^_(4)),s(^_(3))))) =  true]
       eq(s(^_(4)),s(^_(3)))
      ->  eq(^_(4),^_(3))
       eq(^_(4),^_(3))
      ->  false
       not(false)
      ->  true
       eq(0',1')
      ->  false
       and(false,true)
      ->  false
   [failure]      -- on rule [07]

   [condition:  and(not(eq(0',1')),not(eq(s(^_(4)),s(^_(3))))) =  true]
       eq(s(^_(4)),s(^_(3)))
      ->  eq(^_(4),^_(3))
       eq(^_(4),^_(3))
      ->  false
       not(false)
      ->  true
       eq(0',1')
      ->  false
       not(false)
      ->  true
       and(true,true)
      ->  true
   [success]
par(s(^_(4)),r(^_(2)),t(^_(2)),k(^_(1)),l(^_(1)),1',0',1',1') ->     [08]
par(s(^_(4)),r(^_(1)),t(^_(2)),k(^_(1)),l(^_(2)),1',0',1',1')

par(s(^_(4)),r(^_(1)),t(^_(2)),k(^_(1)),l(^_(2)),1',0',1',1')
       bit-inv( 1') ->  0'                                    ->     [10]
par(s(^_(1)),r(^_(1)),t(^_(2)),k(^_(1)),l(^_(1)),0',0',1',1')

par(s(^_(1)),r(^_(1)),t(^_(2)),k(^_(1)),l(^_(1)),0',0',1',1') ->     [01]
par(s(^_(2)),r(^_(1)),t(^_(2)),k(^_(1)),l(^_(1)),0',0',1',1')
```

This rewrite sequence is not found by `trs` “spontaneously”. Only with a lot of tricks its existence could be validated with the tool. So in this case the tool only serves for the validation afterwards of erroneous behaviour. Erroneous behaviour is not detected by the tool itself.