A Bounded Retransmission Protocol for Large Data Packets

A Case Study in Computer Checked Algebraic Verification

Jan Friso Groote[1] & Jaco van de Pol
Dept. of Philosophy, Utrecht University, The Netherlands
e-mail: jfg@phil.ruu.nl, jaco@phil.ruu.nl

Abstract

A protocol is described for the transmission of large data packets over unreliable channels. The protocol splits each data packet and broadcasts it in parts. In case of failure of transmission, only a limited number of retries is allowed (*bounded* retransmission), hence the protocol may give up the delivery of a part of the packet. Both the sending and the receiving client are informed adequately. This protocol is used in one of Philips' products.

We used μCRL as formal framework, a combination of process algebra and abstract data types. The protocol and its external behaviour are specified in μCRL. The correspondence between these is shown using the proof theory of μCRL. The whole proof of this correspondence has been computer checked using the proof checker Coq. This provides an example showing that proof checking of realistic protocols is feasible within the setting of process algebras.

1 Introduction

Background and motivation. During the last 15 years the state-of-the-art in the description and analysis of parallel and distributed systems has advanced enormously. Still the field has not reached a state in which the results are applied frequently and routinely in industry. This situation is improved by carrying out small scale case studies into existing industrial distributed systems. The spin-off of these experiments is generally an assessment of the theory and some indications for further developments of the encountered shortcomings. It is our belief that such hints can steer the theory towards a situation where it can effectively be used at acceptable cost. Therefore, we have started to specify and verify instances of simple distributed systems, using process algebra.

Around 1990 it was realised that process algebraic languages [1, 14] lack a sufficiently precise treatment of data. Up till that moment it seemed sufficient for verification purposes to use standard data types and the generally accepted common sense knowledge about them. This route had already been abandoned by developers of specification languages as they had experienced that commonly accepted data types do not exist (see e.g. [11, 13]). Therefore, abstract data types were added to process algebra.

Given the additional requirement that specifications in such a language should be suited for handling by computer based tools, the language μCRL (micro Common Representation Language) was born. This is a simple, semantically clear and completely formally defined language based on process algebra that incorporates data [6]. The next step was to define a proof theory that enabled to prove distributed systems correct [7]. From this point on μCRL was ready for its usability test. Several distributed systems have now been proved correct [2, 3, 5, 12]. These experiments have revealed several problems. The most important is that proofs contain very many trivial steps. For human beings it is hard to guarantee that all these steps are correct. Therefore, we think it is necessary to check the correctness proofs with automated proof checkers [15, 16, 12, 2].

Verification of the BRP. The Bounded Retransmission Protocol of Philips is an example of a distributed system which relies heavily on data. It is a simplified variant of a telecommunication protocol that is used in one of Philips' products. The protocol allows to transmit large blocks of data within a limited amount of time. After transmission it indicates whether delivery was successful.

[1]Current Affiliation: CWI, Kruislaan 413, 1089 SJ Amsterdam, The Netherlands. E-mail: jfg@cwi.nl

The key features of the protocol are that data is transferred in small chunks, and that only a limited number of retransmissions are allowed for each chunk.

The protocol and its external behaviour are specified in μCRL (Sections 2 and 3) and proved equivalent using the proof system for μCRL (Theorem 4.1). The correctness proof for the BRP is rather typical, because proof principles of process algebra, abstract data types and inductive arguments cohere in an intricate way. This was one of the motivations for designing μCRL. Instead of assuming fairness, we exclude possibly diverging internal behaviour by induction on the bounded number of retries still allowed and the length of the data packet.

The creative part of the proof is to find a suitable system of recursive equations, that has the protocol as well as the external behaviour among its solutions. This is far from trivial; in Section 4 we explain the intuition behind this system. The desired equivalence then follows from the Recursive Specification Principle (RSP), which states that a system of guarded equations has a unique solution. The by far largest part of the proof consists of a proof that the protocol is indeed a solution. This proof (Section 5) is structured by induction on the number of retries still allowed. Within this induction, a large amount of purely algebraic manipulations are necessary, using the equations of process algebra and the axioms of our abstract data types. As we will show, this part lends itself very naturally to term rewriting and hence to automated proof checking.

Finally, the whole correctness proof has been proof checked using the system Coq [4] along the lines set out in [15, 16] (see also [2]). This guarantees the highest degree of correctness that can be reached nowadays. We think that we can safely claim that all lemmas and theorems in this document are correct and that they can be proved correct using only the axioms mentioned in this document.

In Section 6 we report on this verification process. It is explained which features of Coq were used, and which missing features would have been helpful. The algebraic part of the verification has been mechanized. Apart from a rigorous discipline, the verification yields a term rewriting system to compute the expansion of parallel processes in an optimal way. Large parts of the verification can be reused for other protocols.

Discussion. The same protocol has been studied in the setting of I/O-automata [10]. Several invariants, safety, deadlock freeness and liveness results are proven. Parts of these proofs are machine checked. A more recent approach can be found in [9]. Here an abstract interpretation is given, with the help of a theorem prover. The abstract protocol, which has a finite state space, could be verified by a model checker. Our work [8] precedes these two approaches.

We feel that our approach has several merits. The description of the protocol is very compact (it fits in one page, instead of eleven pages in [9]) and completely formal. Furthermore, we give a compact, perspicuous and intuitive correctness proof. Finally, the correctness criterion is highly informative, because the protocol is proved *equivalent* to a straightforward description, representing the external behaviour of the protocol. Here equivalence (branching bisimulation) means that there is no observable difference. Hence a simple process answers all possible questions about the external behaviour of the protocol (inclusive safety, deadlock freeness). Consequently, any user only needs to understand this description. This is a real advantage in the common situation that many people work on the same project, while only a few know about the particularities of the protocol.

Of course, we leave it to the interested reader to judge which approach is mostly suited to his purposes. The common conclusion is, that a formal specification and analysis of realistic distributed systems is possible. We amplify this statement for the algebraic approach.

Acknowledgements. Thanks go to Leen Helmink, Alex Sellink, Frits Vaandrager and Thijs Winter for working on and discussing this protocol. We also thank Doeko Bosscher and Jan Springintveld for reading a preliminary version of this paper.

2 Description of External Behaviour of the BRP

As any transmission protocol, the BRP behaves like a buffer, i.e. it reads data from one client, to be delivered at another one. There are two distinguishing features that make the behaviour much more complicated than a simple buffer. Firstly, the input is a *large data packet* (modeled as a list), which is delivered in small chunks. Secondly, there is a *limited amount of time* for each chunk to be delivered, so we cannot guarantee an eventually successful delivery within the given time bound. It is assumed that either an initial part of the list or the whole list is delivered, so the chunks will not be garbled or change order. Of course, both the sender and the receiver want an *indication* whether the whole list has been delivered successfully or not.

This section ends with a formal description of the external behaviour of the Bounded Retransmission Protocol (BRP) for large data packets. This behaviour is modeled as the process X_1, defined by a system of four recursive equations, written in the syntax of μCRL. Some standard data types are specified in Appendix A. We first give an informal description of the external behaviour, illustrated by Figure 1.

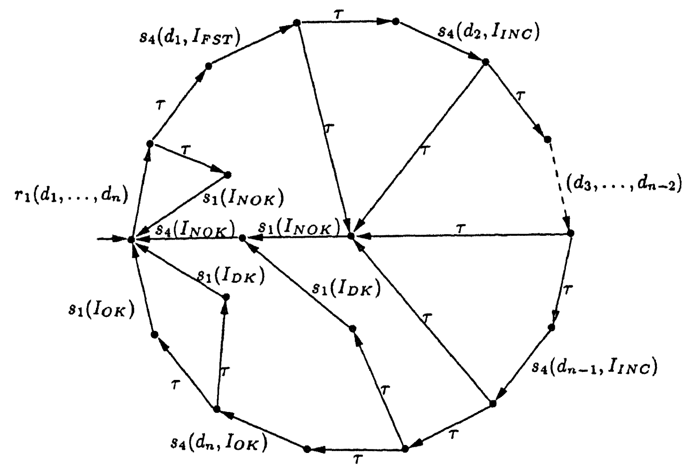

Figure 1: External behaviour of the BRP

The input is read on port 1, by the action $r_1(d_1, \cdots, d_n)$. Ideally, (the outer edge of Figure 1) each d_i is delivered on port 4. Each chunk is accompanied by an indication. This indication can be I_{FST}, I_{INC} or I_{OK}. I_{OK} is used if d_i is the last element of the list. I_{FST} is used if d_i is the first element of the list *and more will follow*. All other chunks are accompanied by I_{INC}.

However, when something goes wrong, a "not OK" indication is sent without datum, $s_4(I_{NOK})$. Note that the receiving client doesn't need a "not OK" indication before delivery of the first chunk, nor after delivery of the last one. This accounts for the irregularity before d_1 and after d_n in Figure 1. The τ-steps indicate that the choice between delivery or loss is decided by internal steps of the protocol.

The sending client is informed after transmission of the whole list, or when the protocol gives up. An indication is sent on port 1, $s_1(c)$. This indication can be I_{OK}, I_{NOK} or I_{DK}. After an I_{OK} or an I_{NOK} indication, the sender can be sure, that the receiver has the corresponding indication. A "don't know" indication I_{DK} may occur after delivery of the last-but-one chunk d_{n-1}. This situation arises, because no realistic implementation can make sure whether the last chunk got lost. The reason is that information about a successful delivery has to be transported back somehow over the same unreliable medium. In case the last acknowledgement fails to come, there is no way to know whether the last chunk d_n has been delivered or not. This explains the exception after d_{n-1} in Figure 1. After this indication, the protocol is ready to transmit a subsequent list.

The rest of this section is devoted to the formal description below. The language primitives are: basic actions ($r_i(d)$ and $s_i(d)$ stand for read and send datum d over port i, respectively), sequential composition ($x \cdot y$, or xy for short), choice over a data type ($\sum_{d:D} x(d)$), a then-if-else construction ($x \lhd b \rhd y$) and choice between two processes ($x + y$). Furthermore, τ is a silent step. These operators are enumerated in order of binding strength (strongest first). See also Table 2 in Appendix B.

In X_1 (corresponding to the leftmost point in Figure 1) some list l is read, and forwarded to X_2, in order to transmit the elements one by one. To inform X_2 whether it is sending the first element of the initial list, it is provided with an extra bit b, which equals e_1 only if the list is fresh, and e_0 if some elements of the initial list have been sent already. X_2 itself is of the form $\tau x + \tau y$, where x and y correspond to loosing or delivering the first element of l. Note that just $x + y$ would mean that the user could refuse to accept failure and force the protocol to succeed. Always chosing the second summand of X_2 corresponds with the outer edge of Figure 1. Finally, X_3 or X_4 is called, depending on whether the receiver needs an indication.

The functions C_{ind} and I_{ind} compute the indications for the sending and receiving client, respectively. See Appendix A for the auxiliary function *indl*, which yields e_1 for empty and singleton lists and e_0 otherwise.

sort $\quad Ind$

func $\quad I_{FST}, I_{OK}, I_{NOK}, I_{INC}, I_{DK} : \to Ind$

$\qquad C_{ind} : List \to Ind$

$\qquad I_{ind} : Bit \times Bit \to Ind$

$\qquad if : \mathbf{Bool} \times Ind \times Ind \to Ind$

var $\quad l : List, i_1, i_2 : Ind$

rew $\quad C_{ind}(l) = if(eq(indl(l), \mathsf{e}_0), I_{NOK}, I_{DK})$

$\qquad I_{ind}(\mathsf{e}_0, \mathsf{e}_0) = I_{INC}$

$\qquad I_{ind}(\mathsf{e}_0, \mathsf{e}_1) = I_{OK}$

$\qquad I_{ind}(\mathsf{e}_1, \mathsf{e}_0) = I_{FST}$

$\qquad I_{ind}(\mathsf{e}_1, \mathsf{e}_1) = I_{OK}$

$\qquad if(\mathsf{t}, i_1, i_2) = i_1$

$\qquad if(\mathsf{f}, i_1, i_2) = i_2$

act $\quad r_1 : List$

$\qquad s_1, s_4 : Ind$

$\qquad s_4 : D \times Ind$

proc $\quad X_1 = \sum_{l:List} r_1(l)\, X_2(l, \mathsf{e}_1)$

$$X_2(l{:}List, b{:}Bit) = \tau(X_3(C_{ind}(l)) \lhd eq(b, \mathsf{e}_1) \rhd X_4(C_{ind}(l))) + \tau\, s_4(head(l), I_{ind}(b, indl(l))) \; ((\tau\, X_3(I_{OK}) + \tau\, X_3(I_{DK})) \lhd last(l) \rhd (\tau\, X_2(tail(l), \mathsf{e}_0) + \tau\, X_4(I_{NOK})))$$

$$X_3(c{:}Ind) = s_1(c)\, X_1$$

$$X_4(c{:}Ind) = s_1(c)\, s_4(I_{NOK})\, X_1$$

3 Description of the Protocol

We now describe the protocol itself. It consists of a sender S equipped with a timer T_1, and a receiver R equipped with a timer T_2 that exchange data via two unreliable channels K and L. See Figure 2 and also the defining equations below.

The protocol has an intricate timing behaviour. The timers use a new set of signals ($TComm$). A timer can only *signal* a time out, if it is *set*; *reset*ting a timer turns it off. Because we have no explicit time in our framework, we could not deal with explicit time bounds. The timers just have the choice to expire; we only cared about order of actions. Synchronization is enforced by two extra signals. These signals are *lost* and *ready*, to be sent over the links 9 and 10. These signals are understood as "elapse of time" and not as physical signals. The dashed lines in Figure 2 indicate that 9 and 10 don't model a physical medium.

It would be interesting to describe the protocol using explicit time delays, to be able to verify that the protocol terminates transmission within the required time bound.

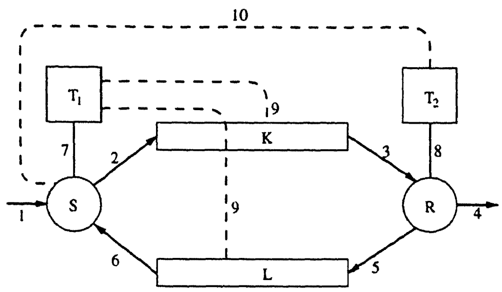

Figure 2: The structure of the BRP.

The sender reads a list at r_1 and sets the retry counter rn to 0 (equation S). Then it starts sending the elements of the list one by one in S_1. Timer T_1 is set ($s_7(set)$) and a frame is sent into channel K. This frame consists of three bits (e_0 or e_1) and a datum. The first bit indicates whether the datum is the first element of the list. The second bit indicates whether the datum is the last item of the list. The third bit is the so-called alternating bit, that is used to guarantee that data is not duplicated. After having sent the frame, the sender waits (in S_2) for an acknowledgement from the receiver, or for a time out. In case an acknowledgement arrives (r_6), the timer T_1 is reset and (depending on whether this was the last element of the list) the sending client is informed of correct transmission, or the next element of the list is sent.

If timer T_1 signals a time out, the frame is resent, (after the counter for the number of retries is incremented and the timer is set again), or the transmission of the list is broken off and timer T_2 is allowed to expire ($s_{10}(ready)$). This occurs if the retry counter exceeds its *max*imum value.

The receiver (initially equation R) waits for a first frame to arrive. This frame is delivered (in R_2) at the receiving client, timer T_2 is started and an acknowledgement is sent (s_5). Then the receiver simply waits for more frames to arrive (in R_1); the value of the alternating bit is stored.

The first bit of R_1 indicates whether the previous frame was the last element of the list; the second bit is the expected value of the alternating bit. Each frame is acknowledged, but it is handed over to the receiving client only if the alternating bit indicates that it is new. In this case timer T_2 is reset. Note that (only) if the previous frame was the last of the list, then a fresh frame will be the first of the subsequent list and a repeated frame will still be the last of the old list. This explains the double reuse of bit b.

This goes on until T_2 times out. This happens if for a long time no new frame is received, indicating that transmission of the list has been given up. The receiving client is informed, provided the last element of the list has not just been delivered. Note that if transmission of the next list starts before timer T_2 expires, the alternating bit scheme is simply continued. This scheme is only interrupted after a failure.

Timer T_1 times out if an acknowledgement does not arrive "in time" at the sender. It is set when a frame is sent and reset after this frame has been acknowledged. To avoid that a message arrives *after* the timer expires, we let the channels K and L send a signal $s_9(lost)$ to T_1, indicating that a time out may occur. This models the following meta-assumption: the total time to move a datum through K, to generate an acknowledgement in R and to transfer this via L is bounded by a fixed delay.

Timer T_2 is (re)set by the receiver ($r_8(set)$) at the arrival of each new frame. It times out if the transmission of a list has been interrupted by the sender. It is also used to model that the sender does not start reading and transmitting the next list before the receiver has properly reacted to the failure. This is necessary, because the receiver has not yet switched its alternating bit, so a new frame would be interpreted as a repetition. The time out ($s_8(signal)$) is preceded by a signal of the sender ($r_{10}(ready)$) to make sure that transmission of the current list has come to a standstill. It is followed by $r_8(ready)\ s_{10}(ready)$, to prevent the sender from sending a new list too early.

sort $TComm$
func $set, reset, signal, ready, lost : TComm$
act $r_2, s_2, c_2, s_3, r_3, c_3 : Bit \times Bit \times Bit \times D$
$r_5, s_5, c_5, r_6, s_6, c_6$
$r_7, s_7, c_7, r_8, s_8, c_8, r_9, s_9, c_9, r_{10}, s_{10}, c_{10} : TComm$
comm $r_2|s_2 = c_2 \quad r_5|s_5 = c_5 \quad r_7|s_7 = c_7 \quad r_9|s_9 = c_9$
$r_3|s_3 = c_3 \quad r_6|s_6 = c_6 \quad s_8|r_8 = c_8 \quad r_{10}|s_{10} = c_{10}$
proc $K = \sum_{b,b',b'':Bit,d:D} r_2(b,b',b'',d)\,(\tau\, s_3(b,b',b'',d) + \tau\, s_9(lost))\,K$
$L = r_5\,(\tau\, s_6 + \tau\, s_9(lost))\,L$

$$S(b'':Bit, max:\mathbb{N}) = \sum_{l:List} r_1(l)\,S_1(l, e_1, b'', 0, max)$$

$$S_1(l:List, b, b'':Bit, rn, max:\mathbb{N}) = s_7(set)\,s_2(b, indl(l), b'', head(l))\,S_2(l, b, b'', rn, max)$$

$$\begin{aligned} S_2(l:List, b, b'':Bit, rn, max:\mathbb{N}) = \\ & r_6\,s_7(reset)\,(s_1(I_{OK})\,S(inv(b''), max) \lhd last(l) \rhd S_1(tail(l), e_0, inv(b''), rn, max)) \\ & + r_7(signal)\,S_3(l, b, b'', rn, max, C_{ind}(l)) \end{aligned}$$

$$\begin{aligned} S_3(l:List, b, b'':Bit, rn, max:\mathbb{N}, c:Ind) = \\ & s_1(c)\,s_{10}(ready)\,r_{10}(ready)\,S(inv(b''), max) \lhd eq(rn, max) \rhd \delta \\ & + S_1(l, b, b'', s(rn), max) \lhd lt(rn, max) \rhd \delta \end{aligned}$$

$$T_1 = r_7(set)\,(r_9(lost)\,s_7(signal) + r_7(reset))\,T_1$$

$$R = \sum_{b',b'':Bit,d:D} r_3(e_1, b', b'', d)\,R_2(b', b'', d, I_{ind}(e_1, b'))$$

$$\begin{aligned} R_1(b, b'':Bit) = \\ & \textstyle\sum_{b':Bit,d:D} (r_3(b, b', b'', d)\,s_8(reset)\,R_2(b', b'', d, I_{ind}(b, b')) \\ & \quad + r_3(b', b, inv(b''), d)\,s_5\,R_1(b, b'')) \\ & + r_8(signal)\,(s_4(I_{NOK})\,s_8(ready)\,R \lhd eq(b, e_0) \rhd s_8(ready)\,R) \end{aligned}$$

$$R_2(b', b'':Bit, d:D, i:Ind) = s_4(d, i)\,s_8(set)\,s_5\,R_1(b', inv(b''))$$

$$\begin{aligned} T_2 = & (r_8(set)\,(r_{10}(ready)\,s_8(signal)\,r_8(ready)\,s_{10}(ready) + r_8(reset)) \\ & + r_{10}(ready)\,s_{10}(ready))\,T_2 \end{aligned}$$

The Bounded Retransmission Protocol for large data packets is obtained by putting these components together. To *enforce* communication, we encapsulate single occurrences of actions in the set $H := \{r_2, s_2, r_3, s_3, r_5, s_5, r_6, s_6, r_7, s_7, r_8, s_8, r_9, s_9, r_{10}, s_{10}\}$. The effect is that e.g. r_2 can only occur simultaneously with s_2; the resulting action is called c_2 by the **comm**-part above. Then the actions in the set $I := \{c_2, c_3, c_5, c_6, c_7, c_8, c_9, c_{10}\}$ are hidden, to indicate that the communications are internal. (See [1] for a good explanation of these concepts and cf. Table 3 in Appendix B). BRP is then specified as the parallel composition of its components, with the actions in H forbidden, and the actions in I hidden.

proc $BRP(max:\mathbb{N}) = \tau_I \partial_H(T_1 \parallel S(e_0, max) \parallel K \parallel L \parallel R \parallel T_2)$

4 The Correctness Proof

The main result to be established is that the internal and external descriptions of the BRP coincide. The rest of this section is devoted to a proof of the following theorem, where equality refers to the theory of μCRL (see Appendix B). Branching bisimulation (a stronger variant of weak bisimulation) is a model of this theory. The importance of this theorem is that a user of the BRP only needs to understand the definition of X_1. This answers all questions about the BRP, like "What happens when an empty list is sent?"[2] Note that the equation even holds when $max = 0$ and also when all signals in $TComm$ are equal, because it is not specified that they are pairwise different.

Theorem 4.1. *For all* $max : \mathbb{N}$ *we find* $\boxed{X_1 = BRP(max).}$

The heart of the correctness proof is as usually an application of RSP, which states that every guarded equation has a unique solution. So a suitable system of recursive equations is needed, having both the protocol and its external behaviour as a solution. Finding it is *the* creative step in the proof. For simple protocols, this system of equations is just the definition of the external behaviour, or a small variation of it.

In our case, the intermediate system of equations is less straightforward. The intuitive reason is that the protocol can start transmission of a list in two distinct modes. In the first mode, the receiver doesn't know what the toggle bit of the list will be; the receiver is in state R. This mode occurs after start up of the protocol and after termination due to a failure. The second mode arises after the successful transmission of a list. In this case, the receiver assumes that the alternating bit sequence is simply continued; the receiver is in state $R_1(\mathrm{e}_1, b'')$, where b'' is the expected bit.

First, we define auxiliary processes, denoting the components of the BRP in some non-initial state:

$$
\begin{aligned}
&K'(b,b',b'',d) := (\tau\, s_3(b,b',b'',d) + \tau\, s_9(lost))\, K\\
&L' := (\tau\, s_6 + \tau\, s_9(lost))\, L\\
&T_2' := (r_{10}(ready)\, s_8(signal)\, r_8(ready)\, s_{10}(ready) + r_8(reset))\, T_2\\
&T_1' := (r_9(lost)\, s_7(signal) + r_7(reset))\, T_1
\end{aligned}
$$

In the set of equations (I) below, Z_1 and Z_1' denote the two major modes described above. $Z_2, \ldots, Z_4''$ are inspired by $X_2, \ldots, X_4$ of the external behaviour, but the two modes are kept distinct. These equations can be read as the definition of processes $Z_1, \ldots, Z_4''$.

$$
\begin{aligned}
&Z_1(b'',max) = \textstyle\sum_{l:List} r_1(l)\, \tau_I \partial_H(T_1 \parallel S_1(l,\mathrm{e}_1,b'',0,max) \parallel K \parallel L \parallel R \parallel T_2)\\
&Z_1'(b'',max) = \textstyle\sum_{l:List} r_1(l)\, \tau_I \partial_H(T_1 \parallel S_1(l,\mathrm{e}_1,b'',0,max) \parallel K \parallel L \parallel R_1(\mathrm{e}_1,b'') \parallel T_2')
\end{aligned}
$$

$$
\begin{aligned}
&Z_2(l,b'',max) =\\
&\qquad (\tau\, Z_4(I_{DK},b'',max) + \tau\, Z_3(l,b'',max))\\
&\qquad \lhd last(l) \rhd\\
&\qquad (\tau\, Z_4(I_{NOK},b'',max)\\
&\qquad\quad + \tau\, s_4(head(l),I_{FST})\,(\tau\, Z_2'(tail(l),\mathrm{e}_0,inv(b''),max) + \tau\, Z_4''(I_{NOK},b'',max)))
\end{aligned}
$$

$$
\text{(I)}\qquad
\begin{aligned}
&Z_2'(l,b,b'',max) =\\
&\qquad (\tau\,(Z_4(I_{DK},b'',max) \lhd eq(b,\mathrm{e}_1) \rhd Z_4''(I_{DK},b'',max)) + \tau\, Z_3(l,b'',max))\\
&\qquad \lhd last(l) \rhd\\
&\qquad (\tau\,(Z_4(I_{NOK},b'',max) \lhd eq(b,\mathrm{e}_1) \rhd Z_4''(I_{NOK},b'',max))\\
&\qquad\quad + \tau\, s_4(head(l),I_{ind}(b,\mathrm{e}_0))\,(\tau\, Z_2'(tail(l),\mathrm{e}_0,inv(b''),max) + \tau\, Z_4''(I_{NOK},b'',max)))
\end{aligned}
$$

$$
Z_3(l,b'',max) = s_4(head(l),I_{OK})\,(\tau\, Z_4'(I_{OK},b'',max) + \tau\, Z_4(I_{DK},b'',max))
$$

[2] In this case one element is delivered, namely *head*(*empty*).

$Z_4(c, b'', max) = s_1(c)\, Z_1(inv(b''), max)$
$Z_4'(c, b'', max) = s_1(c)\, Z_1'(inv(b''), max)$
$Z_4''(c, b'', max) = s_1(c)\, s_4(I_{NOK})\, Z_1(inv(b''), max)$

At this point we need a large number of straightforward calculations, which are postponed to the next section for expository reasons. Using Lemma 5.1.17 and 5.1.18 (and axioms A5 and B1 from Appendix B), we get the following equalities:

$$Z_1(b'', max) = \sum_{l:List} r_1(l)\, Z_2(l, b'', max)$$
$$Z_1'(b'', max) = \sum_{l:List} r_1(l)\, Z_2'(l, \mathsf{e}_1, b'', max)$$

The set of equations (II) is obtained, by replacing the first two equations in (I) by these two equations. (II) is clearly guarded, so by RSP it has a unique solution. Clearly, $Z_1, \ldots, Z_4''$ is a solution of (II). "Another" solution is found by applying the following substitution of (II). This can easily be verified from the equations defining $X_1, \ldots, X_4$ (use a case distinction on $last(l)$ for the equations with X_2).

X_1 for $Z_1(b'', max)$ and $Z_1'(b'', max)$
$X_2(l, \mathsf{e}_1)$ for $Z_2(l, b'', max)$
$X_2(l, b)$ for $Z_2'(l, b, b'', max)$
$s_4(head(l), I_{OK})\,(\tau\, X_3(I_{OK}) + \tau\, X_3(I_{DK}))$ for $Z_3(l, b'', max)$
$X_3(c)$ for $Z_4(c, b'', max)$ and $Z_4'(c, b'', max)$
$X_4(c)$ for $Z_4''(c, b'', max)$

All solutions are equal, so $Z_1(b'', max) = X_1$. By Lemma 5.1.1 and the defining equation of $BRP(max)$ we find that $BRP(max) = Z_1(\mathsf{e}_0, max)$. These two results imply the theorem. ⊠

5 Algebraic Calculations

In this section we give the calculations that were needed in the correctness proof. All calculations fall within the proof theory developed in [7] including the branching τ-laws mentioned in [1].

Lemma 5.1 establishes the link between the Z_i, defined in the process equations (I) and the BRP protocol. The goals are item 17 and 18; all other items are needed in the proof. The proof is in fact by unfolding the equations. Although usually recursive equations specify infinite processes, we use the fact that for given max, rn and l, the equations in (I) contain no infinite loop. Eventually, we end up in Z_1 or Z_1'. The proof proceeds by induction on the number of retries still allowed ($minus(max, rn)$) and on list l.

Lemma 5.1. *For all $b, \bar{b}, b'', b'''$: Bit, max, rn : N, i, c : Ind with* $lt(rn, max)$ or $eq(rn, max)$, *we find*

1. $Z_1(b'', max) = \tau_I \partial_H(T_1 \parallel S(b'', max) \parallel K \parallel L \parallel R \parallel T_2)$
2. $Z_1'(b'', max) = \tau_I \partial_H(T_1 \parallel S(b'', max) \parallel K \parallel L \parallel R_1(\mathsf{e}_1, b'') \parallel T_2')$
3. $Z_4(c, b'', max) = \tau_I \partial_H(T_1 \parallel S_3(l, b, b'', max, max, c) \parallel K \parallel L \parallel R \parallel T_2)$
4. $Z_4''(c, b'', max) = \tau_I \partial_H(T_1 \parallel S_3(l, \bar{b}, b'', max, max, c) \parallel K \parallel L \parallel R_1(\mathsf{e}_0, b''') \parallel T_2')$
5. $Z_4(c, b'', max) = \tau_I \partial_H(T_1 \parallel S_3(l, \bar{b}, b'', max, max, c) \parallel K \parallel L \parallel R_1(\mathsf{e}_1, b''') \parallel T_2')$
6. $Z_4'(c, b'', max) = \tau_I \partial_H(T_1 \parallel s_1(c)\, S(inv(b''), max) \parallel K \parallel L \parallel R_1(\mathsf{e}_1, inv(b'')) \parallel T_2')$
7. $last(l) = \mathsf{f} \rightarrow$
   $\tau\,(\tau\, Z_2'(tail(l), \mathsf{e}_0, inv(b''), max) + \tau\, Z_4''(I_{NOK}, b'', max)) =$
   $\tau\, \tau_I \partial_H(T_1' \parallel S_2(l, b, b'', rn, max) \parallel K \parallel L' \parallel R_1(\mathsf{e}_0, inv(b'')) \parallel T_2')$

8. $last(l) = \mathsf{t} \rightarrow$
$\tau\,(\tau\,Z_4'(I_{OK}, b'', max) + \tau\,Z_4(I_{DK}, b'', max)) =$
$\tau\tau_I\partial_H(T_1' \parallel S_2(l, b, b'', rn, max) \parallel K \parallel L' \parallel R_1(\mathsf{e}_1, inv(b'')) \parallel T_2')$

9. $last(l) = \mathsf{t} \rightarrow$
$Z_3(l, b'', max) = \tau_I\partial_H(T_1' \parallel S_2(l, b, b'', rn, max) \parallel K \parallel L \parallel R_2(indl(l), b'', head(l), I_{OK}) \parallel T_2)$

10. $last(l) = \mathsf{t} \rightarrow$
$\tau\,(\tau\,Z_4'(I_{OK}, b'', max) + \tau\,Z_4(I_{DK}, b'', max)) =$
$\tau\,\tau_I\partial_H(T_1' \parallel S_2(l, b, b'', rn, max) \parallel K'(b, indl(l), b'', head(l)) \parallel L \parallel R_1(\mathsf{e}_1, inv(b'')) \parallel T_2')$

11. $last(l) = \mathsf{t} \rightarrow$
$\tau\,(\tau\,Z_4'(I_{OK}, b'', max) + \tau\,Z_4(I_{DK}, b'', max)) =$
$\tau_I\partial_H(T_1 \parallel S_1(l, b, b'', rn, max) \parallel K \parallel L \parallel R_1(\mathsf{e}_1, inv(b'')) \parallel T_2')$

12. $last(l) = \mathsf{f} \rightarrow$
$s_4(d, i)\,(\tau\,Z_2'(tail(l), \mathsf{e}_0, inv(b''), max) + \tau\,Z_4''(I_{NOK}, b'', max)) =$
$\tau_I\partial_H(T_1' \parallel S_2(l, b, b'', rn, max) \parallel K \parallel L \parallel R_2(\mathsf{e}_0, b'', d, i) \parallel T_2)$

13. $last(l) = \mathsf{f} \rightarrow$
$\tau(\tau\,Z_2'(tail(l), \mathsf{e}_0, inv(b''), max) + \tau\,Z_4''(I_{NOK}, b'', max)) =$
$\tau\,\tau_I\partial_H(T_1' \parallel S_2(l, b, b'', rn, max) \parallel K'(b, indl(l), b'', head(l)) \parallel L \parallel R_1(\mathsf{e}_0, inv(b'')) \parallel T_2')$

14. $last(l) = \mathsf{f} \rightarrow$
$\tau(\tau\,Z_2'(tail(l), \mathsf{e}_0, inv(b''), max) + \tau\,Z_4''(I_{NOK}, b'', max)) =$
$\tau_I\partial_H(T_1 \parallel S_1(l, b, b'', rn, max) \parallel K \parallel L \parallel R_1(\mathsf{e}_0, inv(b'')) \parallel T_2')$

15. $\tau\,Z_2(l, b'', max) = \tau\,\tau_I\partial_H(T_1' \parallel S_2(l, \mathsf{e}_1, b'', rn, max) \parallel K'(\mathsf{e}_1, indl(l), b'', head(l)) \parallel L \parallel R \parallel T_2)$

16. $\tau\,Z_2'(l, b, b'', max) =$
$\tau\,\tau_I\partial_H(T_1' \parallel S_2(l, b, b'', rn, max) \parallel K'(b, indl(l), b'', head(l)) \parallel L \parallel R_1(b, b'') \parallel T_2')$

17. $\tau\,Z_2(l, b'', max) = \tau_I\partial_H(T_1 \parallel S_1(l, \mathsf{e}_1, b'', rn, max) \parallel K \parallel L \parallel R \parallel T_2)$

18. $\tau\,Z_2'(l, b, b'', max) = \tau\,\tau_I\partial_H(T_1 \parallel S_1(l, b, b'', rn, max) \parallel K \parallel L \parallel R_1(b, b'') \parallel T_2')$

Proof. In each step, the parallel processes at the right hand side have to be expanded, to see what steps they can perform. These expansions have been omitted, as they are completely standard. We only show how the inductive proof is structured, and which facts are used.

1,2 Straightforward expansion.

3,4,5 After an expansion, use 1.

6 After an expansion, use 2.

8,10,11 Simultaneous induction on $minus(max, rn)$, the number of retransmissions still allowed. For fixed max and rn, 11 is a consequence of 10. For the base case, first prove 8, using 5 and 6; this together with 5 is used for 10. For the step case, 8 uses 6 and the induction hypothesis for 11; 10 uses the just obtained result for 8, and the induction hypothesis for 11.

9 Use 8.

7,12,13,14,16,18 First, for fixed max, rn and l, 7 $\Rightarrow$ 12, 13 $\Rightarrow$ 14 and 16 $\Rightarrow$ 18 can be seen by expansions only. The proof proceeds by simultaneous induction on $minus(rn, max)$ and within that to the list l. If $l = empty$, then 7 and 13 are vacuously true, because then $last(empty) = \mathsf{t}$. For $rn = max$, 7 implies 16. To see this, we need 4 or 5 (depending on bit b) and 9 or 12 (depending on $last(l)$).

Case 0, *add*. For 7, use 4 and the innermost induction hypothesis of 18. This, together with 4 is used for 13.

Case s**,** $empty$**.** In this case, 16 can be proved with the help of 9, and the outer induction hypothesis for 18.

Case s**,** add**.** 7 uses the outer induction hypothesis of 14 and the inner induction hypothesis of 18. This instance of 7, together with the outer induction hypothesis of 14 is used to establish the induction step for 13. For 16, the outer induction hypothesis for 18 is used, and either 9 or 12 (which holds, because 7 has just been established), depending on $last(l)$.

15,17 By simultaneous induction on $minus(max, rn)$. First note that for fixed max and rn, 15 implies 17. The base case of 15 uses 3; the step case uses the induction hypothesis of 17. Furthermore, both cases use 9 or 12, depending on whether $last(l)$ holds or not. ⊠

6 Mechanical Proof Checking using `Coq V5.8.2`

About Coq. The verification of the correctness proof has been carried out in the theorem prover `Coq V5.8.2` [4]. This system is designed as a proof checker and is not an automated theorem prover. The user can enter tactics (called *vernacular* code), which enable Coq to reproduce the proof. These tactics allow to introduce and unfold definitions, to apply previously proved theorems, to use (directed) equations and to perform induction. Moreover, such tactics can be combined by tacticals, like `Repeat` and `Orelse`. With the help of these tacticals, simple predicates (e.g. membership of lists) can be mechanized, and also a term rewriting system can be implemented within Coq.

Coq's logic is based on a powerful type theory, known as the Calculus of Inductive Constructions (i.e. higher order arithmetic). The main advantage of this strong theory is that all concepts can be defined without encoding. Although the largest part of the proof uses only first order equational logic, we benefited from Coq's expressive power. We used polymorphism to formulate schematic rules (like induction rules and RSP) as single rules. Note also that RSP quantifies over process operators (modeling the right hand of a recursive equation), which take a parametrized process $X(d)$ as argument, which in turn takes a first order datum as argument. So the RSP axiom is a fourth order object in Coq.

Reusable part of the verification. We refer to [15] for a detailed explanation how the syntax, axioms and rules of μCRL can be incorporated in Coq. We reused vernacular code from [2] for a lot of standard facts of process algebra. The files with vernacular commands are available and can be obtained by contacting the second author.

Recursive processes are defined by adding a constant for the process and putting the defining equation as an axiom. This is in fact a hidden appeal to the Recursive Definition Principle, which says that each equation has at least one solution.

Because Lemma 5.1 requires a large number of elementary calculations, we have automatized the computation of the expansion of parallel processes. Lemma 5.1.1 for instance needs the following equality:

$$\sum_{l:List} r_1(l)\,\tau_I\partial_H(T_1 \parallel S_1(l, e_1, b'', 0, max) \parallel K \parallel L \parallel R \parallel T_2) = \tau_I\partial_H(T_1 \parallel S(b'', max) \parallel K \parallel L \parallel R \parallel T_2),$$

which can be proved by expanding the right hand side once (i.e. by looking which steps this term can perform). By CM1, each pair x, y of the 6 processes put in parallel, gives rise to three scenarios: either x or y performs a first step, or they synchronize into a communication. The Handshaking axiom tells that at most two processes can synchronize. Hence, there are 36 scenarios to be considered (6 single steps, and $5 \cdot 6 = 30$ combinations), indicating a potential blow up.

However, the left hand side shows that only one scenario really occurs, namely S may perform a $r_1(l)$ step for some l. All other single steps are encapsulated by the ∂_H. Furthermore, immediate synchronizations are forbidden by the **comm** part in Section 3. In this way, the equality above can be proved by applying equations only.

We succeeded to mechanize these parts of the proof almost completely. To this end a term rewriting system was identified that computes the expansions. With outermost rewriting, the potential blow up is avoided, because the 36 scenarios are not generated at once. Before a new scenario is generated, it is tried to eliminate the old one by encapsulation and excluding synchronization. In vernacular code, the proof of the equation above has the following form: (certain details are replaced by ...)

```
Goal (b'':Bit)(max:Nat)<proc>
  (sum List [l:List] (seq (ia List r1 l) (hide IL (enc HL
    (mer T1 (mer (S1 l e1 b'' o max) (mer K (mer L (mer R T2)))))))))
 =(hide IL (enc HL
    (mer T1 (mer (S b'' max) (mer K (mer L (mer R T2))))))).
Intros.
Pattern 2 T1; Rewrite <- hnf_T1. Rewrite <- Proc_S. ...
Load exp_tac.
Rewrite -> hnf_T1; Rewrite -> Proc_S; ...
Rewrite <- S_Lmer.
Load hide_strip.
Load equal_tac.
Save Exp_1.
```

First the equation is stated as a goal. Then we unfold the definitions of the processes in the right hand side. Now the term rewriting system is called (it is stored in the file `exp_tac`). The process definitions are folded back. At this point we need an auxiliary lemma, called `S_Lmer`. Finally, we compute the result of hiding, and call the tactic `equal_tac`, which compares left and right hand side modulo associativity and commutativity. The proof is stored as `Exp_1`.

Lacking features of `Coq`**.** Unfortunately, we could not mechanize all parts of the algebraic computation. In the example above, we used a lemma (`S_Lmer`) and we had to specify which abbreviations to unfold. A full mechanization of the algebraic part would be desirable. First of all, this saves a lot of time. More importantly, unmechanized parts of the proof are very sensitive to small changes. After the first verification, we made some changes in the protocol. This only invalidated parts of the proof, where mechanization had not been successful (e.g. the proof of lemma `S_Lmer`).

Below we identify some features that would enable us to complete the mechanization. These features are currently lacking in the `Coq` system.

- *Metavariables.* We had to compute the left hand side of the `Goal` above ourselves, before entering it. Ideally, one would type a metavariable for the left hand side. With a metavariable we mean a temporary unknown part of the proof. After the computation of the expansion, the theorem prover knows the value of the metavariable and can instantiate it accordingly. A metavariable could also be generated during the application of a theorem, whose premises contain variables that are not present in the conclusion (e.g. transitivity). This variable could get its value in the next proof step. `Coq` refuses such applications.

- *Full second order matching* is needed for instance in the application of SUM3 (Table 2) from right to left. In the current version, the user has to specify p and e, preventing the use of a general tactic.

- *Definition unfolding mechanism.* In the example above, definitions have to be unfolded (and refolded) by hand. Especially the `Pattern` construct is very sensitive to small changes in the formulae.

- *Extensible vernacular language.* This has been added to Coq 5.10. It allows to add for instance a term rewriter as a common vernacular command.

Specific part of the verification. We will not fully discuss the part of the verification that is specific to this protocol. We only mention two deviations of the proof in the previous sections. The first deviation is, that we didn't use RSP in the way indicated in Section 4. Instead of this, we encoded the system of recursive equations as one single recursive equation. This simplifies the formulation of RSP considerably.

The second deviation is that we implemented the sorts $\mathbb{N}$, Bit, $List$ and $\mathbf{Bool}$ as inductive sets, instead of as abstract data types. Functions on these sets have been defined with primitive recursion. The division between *free constructors* and *functions* was done by hand. In order to check that these definitions coincide with the algebraic specification, the equalities in the specification were proved with induction. Note that the proof theory of μCRL already incorporates induction. See also [16].

This approach is advantageous, as Coq highly supports induction and primitive recursion over inductively defined sets. Furthermore, equality between terms of these sorts coincides with Coq's meta-equality, and can be checked by the system immediately. For the other sorts ($D, TComm, Ind$) this approach was not possible, because the free constructors of them are not given.

We give some statistics, in order to show which fraction of the vernacular code can be reused. The total amount of vernacular code is about 123 Kb, divided over 4367 lines. 1172 lines comprise the definitions of μCRL prove the standard facts and set up the term rewriting system; these are reusable for other protocols. The definition of the protocol requires 468 lines (228 to specify actions, 240 for protocol and behaviour). Specification and proofs regarding data took 497 lines. For Lemma 5.1 we used 1706 lines (311 for auxiliary lemmas, 548 lines for the expansions and 847 lines for the inductive argumentation). The main proof needed 524 lines of code (mainly for the encoding of the system of equations into one equation). A Sparc Station 10-514 needed 11 hours to interpret these vernacular commands and to generate a concrete proof term; this proof term is about 15 Mb large.

Appendix A Standard Data Types

Standard data types used in the BRP are described in Table 1. The functions are fully self explaining. No other facts about data types have been used in the correctness proof of the BRP than those that are mentioned in the main text and in this appendix. Some of the functions, such as $\wedge, pred$ and $minus$ have neither been used in the description of the external behaviour nor in the description of the BRP itself, but were instrumental in the correctness proof. We also used the usual rules for first order predicate logic with equality. Finally, the following induction schemata were incorporated.

$$\frac{P(\mathsf{e}_0) \qquad P(\mathsf{e}_1)}{\forall e{:}Bit\, P(e)} \qquad \frac{P(\mathsf{t}) \qquad P(\mathsf{f})}{\forall b{:}\mathbf{Bool}\, P(b)} \qquad \frac{P(0) \qquad \forall x{:}\mathbb{N}\, P(x) \rightarrow P(s(x))}{\forall x{:}\mathbb{N}\, P(x)}$$

$$\frac{P(empty) \qquad \forall d{:}D\, \forall l{:}List\, P(l) \rightarrow P(add(d,l))}{\forall l{:}List\, P(l)}$$

Appendix B Axioms of μCRL

All the process algebra axioms used to prove the BRP can be found in Table 2–5. These axioms form the basic theory that has been provided to the theorem prover Coq. We do not explain the axioms (see [1, 7]) but only include them to give an exact and complete overview of the axioms that we used. The axiom SC4 is a direct consequence of SC3 and Handshaking. Axiom CD2 is implied by CD1 and SC3. Furthermore, the γ-function is defined as follows: $\gamma(a,b) = c$ if and only if $a|b = c$ or $b|a = c$ occur in the **comm** part of the specification in Section 3.

Besides the axioms we have used the RSP principle, saying that guarded recursive equations have at most one solution. In the following x, y denote parametrized processes that can be applied to a data parameter d of arbitrary sort D, and deliver a process. The symbol Ψ is a process operator, i.e. a process which is parametrized with a process $x(d)$ and a datum element d. It models the right hand side of a recursive equation.

$$\text{RSP} \qquad \frac{\Psi \text{ is guarded} \quad \forall d{:}D.\, x(d) = \Psi(x,d) \quad \forall d{:}D.\, y(d) = \Psi(y,d)}{\forall d : D.\, x(d) = y(d)}.$$

In the correctness proof in this paper we have used a strong notion of guardedness, namely: for guarded processes p' and p'', an arbitrary process q, boolean term b and action $a(d)$, the following processes are guarded: $a(d)$, $\delta, \tau, p' + p''$, $p' \triangleleft b \triangleright p''$, $\sum_{d:D} p'(d)$, $a(d)\, q$ and $\tau\, p'$.

sort	**Bool**	**sort**	Bit
func	$\mathsf{f}, \mathsf{t} :\to$ **Bool**	**func**	$\mathsf{e}_0, \mathsf{e}_1 :\to Bit$
	$\wedge :$ **Bool** $\times$ **Bool** $\to$ **Bool**		$inv : Bit \to Bit$
var	$b :$ **Bool**		$if :$ **Bool** $\times Bit \times Bit \to Bit$
rew	$\mathsf{t} \wedge b = b$		$eq : Bit \times Bit \to Bool$
	$\mathsf{f} \wedge b = \mathsf{f}$	**var**	$b, b_1, b_2 : Bit$
		rew	$inv(\mathsf{e}_0) = \mathsf{e}_1$
sort	$D, List$		$inv(\mathsf{e}_1) = \mathsf{e}_0$
func	$d_0 :\to D$		$if(\mathsf{t}, b_1, b_2) = b_1$
	$if :$ **Bool** $\times D \times D \to D$		$if(\mathsf{f}, b_1, b_2) = b_2$
	$eq : D \times D \to$ **Bool**		$if(eq(b_1, b_2), b_1, b_2) = b_2$
	$empty :\to List$		$eq(b, inv(b)) = \mathsf{f}$
	$add : D \times List \to List$		$eq(b, b) = \mathsf{t}$
	$head : List \to D$		
	$tail : List \to List$	**sort**	$\mathbb{N}$
	$last : List \to$ **Bool**	**func**	$0 :\to \mathbb{N}$
	$indl : List \to Bit$		$s, pred : \mathbb{N} \to \mathbb{N}$
var	$d, d_1, d_2 :\to D$		$eq : \mathbb{N} \times \mathbb{N} \to$ **Bool**
	$l :\to List$		$lt : \mathbb{N} \times \mathbb{N} \to$ **Bool**
rew	$head(empty) = d_0$		$minus : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$
	$head(add(d, l)) = d$	**var**	$n, n_1, n_2 :\to \mathbb{N}$
	$tail(empty) = empty$	**rew**	$eq(0, 0) = \mathsf{t}$
	$tail(add(d, l)) = l$		$eq(0, s(n)) = \mathsf{f}$
	$last(empty) = \mathsf{t}$		$eq(s(n), 0) = \mathsf{f}$
	$last(add(d, empty)) = \mathsf{t}$		$eq(s(n_1), s(n_2)) = eq(n_1, n_2)$
	$last(add(d_1, add(d_2, l))) = \mathsf{f}$		$lt(0, s(n)) = \mathsf{t}$
	$indl(empty) = \mathsf{e}_1$		$lt(n, 0) = \mathsf{f}$
	$indl(add(d, empty)) = \mathsf{e}_1$		$lt(s(n_1), s(n_2)) = lt(n_1, n_2)$
	$indl(add(d_1, add(d_2, l))) = \mathsf{e}_0$		$pred(0) = 0$
	$if(\mathsf{t}, d_1, d_2) = d_1$		$pred(s(n)) = n$
	$if(\mathsf{f}, d_1, d_2) = d_2$		$minus(n, 0) = n$
	$eq(d, d) = \mathsf{t}$		$minus(n_1, s(n_2)) = pred(minus(n_1, n_2))$
	$if(eq(d_1, d_2), d_1, d_2) = d_2$		

Table 1: Specification of standard data types used in the BRP

A1	$x + y = y + x$	SUM1	$\Sigma_{d:D} x = x$
A2	$x + (y + z) = (x + y) + z$	SUM3	$\Sigma_{d:D}\, p(d) = \Sigma_{d:D}\, p(d) + p(e)$
A3	$x + x = x$	SUM4	$\Sigma_{d:D}(p(d) + q(d)) = \Sigma_{d:D}\, p(d) + \Sigma_{d:D}\, q(d)$
A4	$(x + y) \cdot z = x \cdot z + y \cdot z$	SUM5	$\Sigma_{d:D}(p(d) \cdot x) = (\Sigma_{d:D}\, p(d)) \cdot x$
A5	$(x \cdot y) \cdot z = x \cdot (y \cdot z)$	SUM11	$(\forall d\, p(d) = q(d)) \rightarrow \Sigma_{d:D}\, p(d) = \Sigma_{d:D}\, q(d)$
A6	$x + \delta = x$		
		Bool1	$\neg(\mathsf{t} = \mathsf{f})$
		Bool2	$\neg(b = \mathsf{t}) \rightarrow b = \mathsf{f}$
B1	$x \cdot \tau = x$	C1	$x \lhd \mathsf{t} \rhd y = x$
B2	$z \cdot (\tau \cdot (x + y) + x) = z \cdot (x + y)$	C2	$x \lhd \mathsf{f} \rhd y = y$

Table 2: pCRL axioms

SUM6	$\Sigma_{d:D}(p(d) \,⌊\!⌊\, z) = (\Sigma_{d:D}\, p(d)) \,⌊\!⌊\, z$	CF	$a(d)\|b(e) = \begin{cases} \gamma(a,b)(d) & \text{if } d = e \text{ and } \gamma(a,b) \text{ defined} \\ \delta & \text{otherwise} \end{cases}$	
SUM7	$\Sigma_{d:D}(p(d)\|z) = (\Sigma_{d:D}\, p(d))\|z$			
SUM8	$\Sigma_{d:D}(\partial_H(p(d))) = \partial_H(\Sigma_{d:D}\, p(d))$			
SUM9	$\Sigma_{d:D}(\tau_I(p(d))) = \tau_I(\Sigma_{d:D}\, p(d))$			
SUM10	$\Sigma_{d:D}(\rho_R(p(d))) = \rho_R(\Sigma_{d:D}\, p(d))$	CD1	$\delta\|x = \delta$	
		CD2	$x\|\delta = \delta$	
CM1	$x \parallel y = x \,⌊\!⌊\, y + y \,⌊\!⌊\, x + x\|y$	CT1	$\tau\|x = \delta$	
CM2	$c \,⌊\!⌊\, x = c \cdot x$	CT2	$x\|\tau = \delta$	
CM3	$c \cdot x \,⌊\!⌊\, y = c \cdot (x \parallel y)$			
CM4	$(x + y) \,⌊\!⌊\, z = x \,⌊\!⌊\, z + y \,⌊\!⌊\, z$	DD	$\partial_H(\delta) = \delta$	
CM5	$c \cdot x\|c' = (c\|c') \cdot x$	DT	$\partial_H(\tau) = \tau$	
CM6	$c\|c' \cdot x = (c\|c') \cdot x$	D1	$\partial_H(a(d)) = a(d)$	if $a \notin H$
CM7	$c \cdot x\|c' \cdot y = (c\|c') \cdot (x \parallel y)$	D2	$\partial_H(a(d)) = \delta$	if $a \in H$
CM8	$(x + y)\|z = x\|z + y\|z$	D3	$\partial_H(x + y) = \partial_H(x) + \partial_H(y)$	
CM9	$x\|(y + z) = x\|y + x\|z$	D4	$\partial_H(x \cdot y) = \partial_H(x) \cdot \partial_H(y)$	

Table 3: Primary μCRL axioms

TID	$\tau_I(\delta) = \delta$	
TIT	$\tau_I(\tau) = \tau$	
TI1	$\tau_I(a(d)) = a(d)$	if $a \notin I$
TI2	$\tau_I(a(d)) = \tau$	if $a \in I$
TI3	$\tau_I(x + y) = \tau_I(x) + \tau_I(y)$	
TI4	$\tau_I(x \cdot y) = \tau_I(x) \cdot \tau_I(y)$	

Table 4: Secondary μCRL axioms

SC1	$x \,⌊\!⌊\, (y \parallel z) = (x \,⌊\!⌊\, y) \,⌊\!⌊\, z$
SC2	$x \,⌊\!⌊\, \delta = x\delta$
SC3	$x \mid y = y \mid x$
SC4	$(x \mid y) \mid z = x \mid (y \mid z)$
SC5	$(x \mid y) \,⌊\!⌊\, z = x \mid (y \,⌊\!⌊\, z)$
Handshaking	$(x \mid y) \mid z = \delta$

Table 5: Standard Concurrency and Handshaking

References

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

[2] M. Bezem, R. Bol, and J.F. Groote. Formalizing process algebraic verifications in the calculus of constructions. Technical Report 95-02, Eindhoven University of Technology, January 1995. To appear in Formal Aspects of Computing.

[3] M.A. Bezem and J.F. Groote. A correctness proof of a one-bit sliding window protocol in μCRL. *The Computer Journal*, 37(4):289–307, 1994.

[4] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Murthy, C. Parent, C. Paulin-Mohring, and B. Werner. The Coq proof assistant user's guide. Version 5.8. Technical report, INRIA – Rocquencourt, May 1993.

[5] J.F. Groote and H.P. Korver. A correctness proof of the bakery protocol in μCRL. In A. Ponse, C. Verhoef, and S.F.M. van Vlijmen, editors, *Proc. 1st Workshop on the Algebra of Communicating Processes (ACP'94)*, Utrecht, Workshops in Computing, pages 63–86. Springer-Verlag, 1994.

[6] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. Technical Report CS-R9076, CWI, Amsterdam, December 1990.

[7] J.F. Groote and A. Ponse. Proof theory for μCRL: a language for processes with data. In D.J. Andrews, J.F. Groote, and C.A. Middelburg, editors, *Proc. of the Int. Workshop on Semantics of Specification Languages*, pages 232–251. Workshops in Computing, Springer Verlag, 1994.

[8] J.F. Groote and J.C. van de Pol. A bounded retransmission protocol for large data packets. Technical Report Logic Group Preprint Series No. 100, Utrecht University, Oct 1993.

[9] K. Havelund and N. Shankar. Experiments in theorem proving and model checking for protocol verification. Obtainable via `http://www.csl.sri.com/~shankar/shankar.html`, 1995.

[10] L. Helmink, M.P.A. Sellink, and F.W. Vaandrager. Proof-checking a data link protocol. In H.P. Barendregt and T. Nipkow, editors, *Proc. of the 1st International Workshop "Types for Proofs and Programs", May 1993*, volume 806 of *LNCS*, pages 127–165, Nijmegen, 1994.

[11] ISO. *Information processing systems – open systems interconnection – LOTOS – a formal description technique based on the temporal ordering of observational behaviour* ISO IS 8807, 1989.

[12] H. Korver and J. Springintveld. A computer-checked verification of Milner's scheduler. In M. Hagiya and J.C. Mitchell, editors, *Proc. of the Int. Symp. on Theoretical Aspects of Computer Software*, volume 789 of *LNCS*, pages 161–178, Sendai, Japan, April 1994. Springer Verlag.

[13] S. Mauw. *PSF – A Process Specification Formalism*. PhD thesis, University of Amsterdam, December 1991.

[14] R. Milner. *Communication and Concurrency*. Prentice-Hall International, Englewood Cliffs, 1989.

[15] M.P.A. Sellink. Verifying process algebra proofs in type theory. In D.J. Andrews, J.F. Groote, and C.A. Middelburg, editors, *Proc. of the Int. Workshop on Semantics of Specification Languages, Utrecht 1993*, Workshops in Computing, pages 315–339. Springer-Verlag, 1994.

[16] M.P.A. Sellink. *Computer-Aided Verification of Protocols, The Type Theoretic Approach*. Phd thesis, Utrecht University, February 1996.