A Bounded Retransmission Protocol for Large Data Packets

A Case Study in Computer Checked Algebraic Verification

Jan Friso Groote & Jaco van de Pol

Department of Philosophy, Utrecht University

Heidelberglaan 8, 3584 CS Utrecht, The Netherlands

email: JanFriso.Groote@phil.ruu.nl, Jaco.vandePol@phil.ruu.nl

Abstract

This note describes a protocol for the transmission of data packets that are too large to be transferred in their entirety. Therefore, the protocol splits the data packets and broadcasts it in parts. It is assumed that in case of failure of transmission through data channels, only a limited number of retries are allowed (*bounded* retransmission). If repeated failure occurs, the protocol stops trying and the sending and receiving protocol users are informed accordingly. The protocol and its external behaviour are specified in μCRL. The correspondence between these is shown using the axioms 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.

The first author is partly supported by the Netherlands Computer Science Research Foundation (SION) with financial support of the Netherlands Organisation for Scientific Research (NWO).

1 Introduction

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 not more than a (generally negative) assessment of the theory and some indications for further developments of the encountered shortcomings. But, it is our belief that only 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.

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 [7, 8]. The next step was to define a proof theory that enabled to prove distributed systems correct [9]. From this point on μCRL was ready for its usability test. Several distributed systems have now been proved correct [2, 6, 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 necessary to check the correctness proofs with proof checkers [3, 12, 15].

The Bounded Retransmission Protocol of Philips [16] 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 transmission has been successful. The protocol can find itself in a situation where it does not know whether transmission was a success. In that case it reports accordingly.

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 block of data to be transferred.

The protocol and its external behaviour are proved equivalent using the proof system for μCRL. All axioms that have been used have been recorded in this document. In Appendix A relevant axioms about standard data types have been recorded. In Appendix B the axioms of process algebra are briefly mentioned. In the text all specially designed data types for this protocol are axiomatised. The proof that the protocol and the service are equivalent uses a form of induction that we have not encountered before in protocol verification. Furthermore, the proof turns out to be very compact, given the complexity of the protocol.

Furthermore, the whole equivalence proof has been proof checked using the system `Coq` [5] along the lines set out in [15] (see also [3]). This guarantees the highest degree of correctness of the proof that can be attained nowadays. We think that we can savely 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. Given the fact that we believe the current document provides the highest degree of correctness currently available, we are very interested in being informed of any mistake in any formal statement in this document, be it as small as a typing error. This will help us in assessing current technology.

Acknowledgements. Thanks go to Leen Helmink, Alex Sellink, Frits Vaandrager and Thijs Winter for working on and discussing this protocol. The first three mentioned persons have studied the same protocol in the setting of I/O-automata [10]. Thanks also go to Doeko Bosscher, Jan Springintveld and Jan-Joris Vereijken for their comments on this paper.

2 Definition of External Behaviour of the Protocol

Below a description in μCRL [7] of the external behaviour of the Bounded Retransmission Protocol (BRP) for large data packets is given. The purpose is to transfer data from a sending protocol user to a receiving protocol user within limited amount of time and to report whether the data is successfully delivered. The external behaviour of the protocol is described by the equations defining X_1, X_2, X_3 and X_4, below.

First we describe the external behaviour of the protocol in case nothing exceptional happens, i.e. no channel looses data. The protocol reads a large data packet represented by a list of elementary data packets l ($r_1(l)$). Then it tries to deliver the elements of l one by one at the receivers side ($s_4(head(l), I_{ind}(b, indl(l)))$ in X_2). The bit b in X_2 indicates whether elements of l have already been delivered. If so, $b = e_0$, otherwise $b = e_1$. When a data element is delivered, it is accompanied with a status flag. This flags can be I_{OK}, I_{FST} and I_{INC}. The flag I_{OK} expresses that the data element is the last one of a list of data elements. In case the list has length one, it is of course also the first. The flag I_{FST} indicates that this is the first element of a list, and more will follow. If the data element is an intermediate element of the list, the flag shows I_{INC}. The function I_{ind} yields the status, based on the bit b and the list l. See Appendix A for the function $indl$. If the list l has been delivered completely, then a status flag I_{OK} is delivered at the sender's side, and the protocol is ready to transmit the next list of data. It is noteworthy that the protocol has a strange behaviour if it reads an empty list l. In this case it still delivers one data element at the receivers side, which is $head(\emptyset)$.

In case this is considered undesirable, the external behaviour of the protocol can easily be adapted, but this of course also entails changes in the protocol itself.

Several exceptions can occur in the protocol. It may be that the channels that carry the data packets loose so many messages that the protocol decides to stop transmitting the list of data. In the external behaviour of the protocol τ-actions describe whether the next message will be delivered or whether failure will be indicated. In our particular implementation of the protocol this is realised by restricting the number of retransmissions to a total of max for each list. The value of max is irrelevant to the external behaviour, as timing details and probabilities are abstracted from. In case the protocol fails and some, but not all data has been delivered, both the sending and receiving protocol users must be notified that the protocol stops transmitting the list. At both the receiving side and the sending side this is done using the 'Not OK' flag, I_{NOK} (described by $X_4(I_{NOK})$).

In case the protocol fails and no part of the list or the whole list has been delivered, only the sending protocol user needs to be informed as the receiving protocol user either does not know that anything has been transferred, or it perceives transmission as being successful ($X_3(c)$). At the sending side either a 'Not OK' flag, I_{NOK}, or a 'Don't Know' flag, I_{DK} is shown. This last flag is only used when transmission fails after an attempt has been made to transmit the last element of a list. For the sender it is unclear whether this has arrived at the receiver. So, in this case it is not known whether transmission of the list was successful.

sort Ind

func $I_{FST}, I_{OK}, I_{NOK}, I_{INC}, I_{DK} :\rightarrow Ind$
$C_{ind} : List \rightarrow Ind$
$I_{ind} : Bit \times Bit \rightarrow Ind$
$if : \mathbf{Bool} \times Ind \times Ind \rightarrow Ind$

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

rew $C_{ind}(l) = if(eq(indl(l), e_0), I_{NOK}, I_{DK})$
$I_{ind}(e_0, e_0) = I_{INC}$
$I_{ind}(e_0, e_1) = I_{OK}$
$I_{ind}(e_1, e_0) = I_{FST}$
$I_{ind}(e_1, e_1) = I_{OK}$
$if(\mathsf{t}, i_1, i_2) = i_1$
$if(\mathsf{f}, i_1, i_2) = i_2$

act $r_1 : List$
$s_1, s_4 : Ind$
$s_4 : D \times Ind$

proc $X_1 = \sum_{l:List} r_1(l)\, X_2(l, e_1)$

$$X_2(l{:}List, b{:}Bit) = \tau(X_3(C_{ind}(l)) \triangleleft eq(b, e_1) \triangleright X_4(C_{ind}(l))) + \tau\, s_4(head(l), I_{ind}(b, indl(l)))\big((\tau\, X_3(I_{OK}) + \tau\, X_3(I_{DK})) \triangleleft last(l) \triangleright (\tau\, X_2(tail(l), e_0) + \tau\, X_4(I_{NOK}))\big)$$

$$X_3(c{:}Ind) = s_1(c)\, X_1$$
$$X_4(c{:}Ind) = s_1(c)\, s_4(I_{NOK})\, X_1$$

3 Definition of the Protocol

We now describe the protocol itself. It consist of a sender S equipped with a timer T_1, and a receiver R equipped with a timer T_2 that exchange data packets via two unreliable channels K and L. See Figure 1 and the defining equations below. The behaviour of the protocol has an intricate timing behaviour. Synchronisation is established using a new set of signals ($TComm$). Remarkably, from

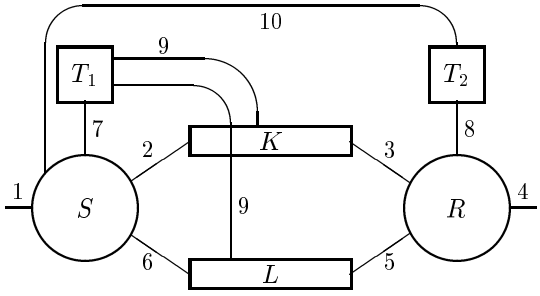

Figure 1: The structure of the bounded retransmission protocol

the proof verification on the computer it appeared that only two different signals are necessary. We first explain the behaviour of the protocol ignoring time, and then we describe how timing has been incorporated in the description.

When the channels K and L deliver their data successfully, the protocol behaves as follows. 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 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 answer to arrive from the receiver, or for a timeout. In case an answer arrives (r_6), the datum has arrived. After arrival the timer T_1 is reset, and depending on whether this was the last element of the list, the sending protocol user is informed of correct transmission, or the next element of the list is sent. It may also be the case that the timer T_1 signals a time out, in which case either the data packet 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 the receiving side in the protocol is informed accordingly by $s_{10}(ready)$. This occurs if the retry counter exceeds its maximum value.

The behaviour of the receiver is simpler. Initially (equation R), the receiver waits for a first frame to arrive. The first data packet it receives is delivered (in R_2) at the receiving protocol user and the receiver starts timer T_2. This timer times out if for a long time nothing is received at the receiver, indicating that transmission of the list has failed. The receiver also acknowledges the data packet (s_5). Then it simply waits for more packets to arrive (in R_1), which are always acknowledged, but handed over to the receiving protocol user only if the alternating bit indicates that this has not already been done. This goes on until T_2 times out, which indicates failure of transmission, or a last packet of the list arrives in which case the second bit is e_1.

Time and timing constraints are encoded as forms of synchronisation in the protocol. The timers can only *signal* a time out, if they are *set*. *Reset*ting a timer turns it off.

Timer T_1 times out if an acknowledgement does not arrive 'in time' at the sender, after it has sent a message. Timer T_1 is set when a frame is sent and reset after this frame has been acknowledged. Assuming that the channels K and L deliver data frames in time (if these do not get lost) and assuming that R is always capable of acknowledging an incoming message in bounded time, a late delivery of an acknowledgement can only occur when either channel K or channel L loses a message. In the protocol this is described by K or L sending a message $s_9(lost)$ to timer T_1, indicating that it may send a timeout.

Timer T_2 times out if the transmission of a list has been interrupted and it is clear that the sender must have stopped trying to retransmit it. It is also used to model that the sender does not start reading and transmitting a new list before an indication of failure of transmission of the current list

has appeared at the receiver. Although convenient, it is actually not fully appropriate to encode these synchronisations in timer T_2. More specifically, timer T_2 starts if it is set by the receiver ($r_8(set)$). Then it waits until it is certain that transmission of the current list has come to a standstill. In this protocol this is indicated by the sender via $s_{10}(ready)$. If $s_{10}(ready)$ occurs before the timer is reset, it signals the receiver ($s_8(signal)$), which then knows that transmission of the list has stopped, and then, via $r_8(ready)$ and $s_{10}(signal)$ guarantees that the sender does not proceed with a new list before the receiver has notified the receiving protocol user.

sort $TComm$

func $set, reset, signal, ready, lost : TComm$
$eq : TComm \times TComm \to \mathbf{Bool}$

var $k : TComm$

rew $eq(k,k) = \mathsf{t}$
$eq(set, reset) = \mathsf{f}$

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 $K = \sum_{b,b',b'':Bit,d:D}(r_2(b,b',b'',d)\,(i\,s_3(b,b',b'',d) + i\,s_9(lost))\,K)$
$L = r_5\,(i\,s_6 + i\,s_9(lost))\,L$

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

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

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

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

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

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

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

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

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

The Bounded Retransmission Protocol for large data packets can straightforwardly be described as follows. Define $I = \{i, c_2, c_3, c_5, c_6, c_7, c_8, c_9, c_{10}\}$ and
$H = \{r_2, s_2, r_3, s_3, r_5, s_5, r_6, s_6, r_7, s_7, r_8, s_8, r_9, s_9, r_{10}, s_{10}\}$

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

The main result to be established is that the BRP meets its external behaviour. In the next sections we prove the following theorem.

Theorem 3.1. *For all* $max : \mathbb{N}$ *we find*

$$\boxed{X_1 = BRP(max)}$$

Note that the protocol even behaves 'correct' if $max = 0$.

4 The Main Lemma

We linearise the behaviour of the protocol, using the set of equations below, refered to by (I).

$$Z_1(b'', max) = \textstyle\sum_{l:List}(r_1(l)\,\tau_I\partial_H(T_1 \parallel S_1(l, e_1, b'', 0, max) \parallel K \parallel L \parallel R \parallel T_2))$$
$$Z'_1(b'', max) = \textstyle\sum_{l:List}(r_1(l)\,\tau_I\partial_H(T_1 \parallel S_1(l, e_1, b'', 0, max) \parallel K \parallel L \parallel R_1(e_1, b'') \parallel T'_2))$$

$$\begin{aligned}
&Z_2(l, b'', max) = \\
&\qquad \big(\tau\, Z_4(I_{DK}, b'', max) + \tau\, Z_3(l, b'', max)\big) \\
&\qquad \lhd last(l) \rhd \\
&\qquad \big(\tau\, Z_4(I_{NOK}, b'', max) + \tau\, s_4(head(l), I_{FST}) \\
&\qquad\qquad (\tau\, Z'_2(tail(l), e_0, inv(b''), max) + \tau\, Z''_4(I_{NOK}, b'', max))\big)
\end{aligned}$$

$$\begin{aligned}
&Z'_2(l, b, b'', max) = \\
&\qquad \big(\tau\,(Z_4(I_{DK}, b'', max) \lhd eq(b, e_1) \rhd Z''_4(I_{DK}, b'', max)) + \tau\, Z_3(l, b'', max)\big) \\
&\qquad \lhd last(l) \rhd \\
&\qquad \big(\tau\,(Z_4(I_{NOK}, b'', max) \lhd eq(b, e_1) \rhd Z''_4(I_{NOK}, b'', max)) + \\
&\qquad\ \tau\, s_4(head(l), I_{ind}(b, e_0))\,(\tau\, Z'_2(tail(l), e_0, inv(b''), max) + \tau\, Z''_4(I_{NOK}, b'', max))\big)
\end{aligned}$$

$$Z_3(l, b'', max) = s_4(head(l), I_{OK})\,(\tau\, Z'_4(I_{OK}, b'', max) + \tau\, Z_4(I_{DK}, b'', max))$$

$$\begin{aligned}
Z_4(c, b'', max) &= s_1(c)\, Z_1(inv(b''), max) \\
Z'_4(c, b'', max) &= s_1(c)\, Z'_1(inv(b''), max) \\
Z''_4(c, b'', max) &= s_1(c)\, s_4(I_{NOK})\, Z_1(inv(b''), max)
\end{aligned}$$

Below we give the main lemma of this paper. All calculations of intermediate and main results below follow from the axioms for μCRL [9] and the branching τ-laws mentioned in [6] (see also Appendix B). Define

$$\begin{aligned}
&K'(b, b', b'', d) = (i\, s_3(b, b', b'', d) + i\, s_9(lost))\, K, \\
&L' = (i\, s_6 + i\, s_9(lost))\, L, \\
&T'_2 = (r_{10}(ready)(s_8(signal)\, r_8(ready) + r_8(reset))\, s_{10}(signal) + r_8(reset))\, T_2 \text{ and} \\
&T'_1 = (r_9(lost)(s_7(signal) + r_7(reset)) + r_7(reset))\, T_1\,.
\end{aligned}$$

Lemma 4.1. *For all* $b, \overline{b}, b'', b''' : Bit$, $max, rn : \mathbb{N}$, $i, c : Ind$ *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. $Z'_1(b'', max) = \tau_I \partial_H(T_1 \parallel S(b'', max) \parallel K \parallel L \parallel R_1(e_1, b'') \parallel T'_2)$
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. $Z''_4(c, b'', max) = \tau_I \partial_H(T_1 \parallel S_3(l, \overline{b}, b'', max, max, c) \parallel K \parallel L \parallel R_1(e_0, b''') \parallel T'_2)$
5. $Z_4(c, b'', max) = \tau_I \partial_H(T_1 \parallel S_3(l, \overline{b}, b'', max, max, c) \parallel K \parallel L \parallel R_1(e_1, b''') \parallel T'_2)$
6. $Z'_4(c, b'', max) = \tau_I \partial_H(T_1 \parallel s_1(c)\, S(inv(b''), max) \parallel K \parallel L \parallel R_1(e_1, inv(b'')) \parallel T'_2)$
7. $last(l) = \mathsf{f} \rightarrow$
   $\tau\,(\tau\, Z'_2(tail(l), e_0, inv(b''), max) + \tau\, Z''_4(I_{NOK}, b'', max)) =$
   $\quad \tau\, \tau_I \partial_H(T'_1 \parallel S_2(l, b, b'', rn, max) \parallel K \parallel L' \parallel R_1(e_0, inv(b'')) \parallel T'_2)$
8. $last(l) = \mathsf{t} \rightarrow$
   $\tau\,(\tau\, Z'_4(I_{OK}, b'', max) + \tau\, Z_4(I_{DK}, b'', max) =$
   $\quad \tau\, \tau_I \partial_H(T'_1 \parallel S_2(l, b, b'', rn, max) \parallel K \parallel L' \parallel R_1(e_1, inv(b'')) \parallel T'_2)$
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. $last(l) = \mathsf{t} \rightarrow$
    $\tau\,(\tau\, Z'_4(I_{OK}, b'', max) + \tau\, Z_4(I_{DK}, b'', max)) =$
    $\quad \tau\, \tau_I \partial_H(T'_1 \parallel S_2(l, b, b'', rn, max) \parallel K'(b, indl(l), b'', head(l)) \parallel L \parallel R_1(e_1, inv(b'')) \parallel T'_2)$
11. $last(l) = \mathsf{t} \rightarrow$
    $\tau\,(\tau\, Z'_4(I_{OK}, b'', max) + \tau\, Z_4(I_{DK}, b'', max)) =$
    $\quad \tau_I \partial_H(T_1 \parallel S_1(l, b, b'', rn, max) \parallel K \parallel L \parallel R_1(e_1, inv(b'')) \parallel T'_2)$
12. $last(l) = \mathsf{f} \rightarrow$
    $s_4(d, i)\,(\tau\, Z'_2(tail(l), e_0, inv(b''), max) + \tau\, Z''_4(I_{NOK}, b'', max)) =$
    $\quad \tau_I \partial_H(T'_1 \parallel S_2(l, b, b'', rn, max) \parallel K \parallel L \parallel R_2(e_0, b'', d, i) \parallel T_2)$
13. $last(l) = \mathsf{f} \rightarrow$
    $\tau(\tau\, Z'_2(tail(l), e_0, inv(b''), max) + \tau\, Z''_4(I_{NOK}, b'', max)) =$
    $\quad \tau\, \tau_I \partial_H(T'_1 \parallel S_2(l, b, b'', rn, max) \parallel K'(b, indl(l), b'', head(l)) \parallel L \parallel R_1(e_0, inv(b'')) \parallel T'_2)$
14. $last(l) = \mathsf{f} \rightarrow$
    $\tau(\tau\, Z'_2(tail(l), e_0, inv(b''), max) + \tau\, Z''_4(I_{NOK}, b'', max)) =$
    $\quad \tau_I \partial_H(T_1 \parallel S_1(l, b, b'', rn, max) \parallel K \parallel L \parallel R_1(e_0, inv(b'')) \parallel T'_2)$
15. $\tau\, Z_2(l, b'', max) = \tau\, \tau_I \partial_H(T'_1 \parallel S_2(l, e_1, b'', rn, max) \parallel K'(e_1, indl(l), b'', head(l)) \parallel L \parallel R \parallel T_2)$
16. $\tau\, Z'_2(l, b, b'', max) =$
    $\quad \tau\, \tau_I \partial_H(T'_1 \parallel S_2(l, b, b'', rn, max) \parallel K'(b, indl(l), b'', head(l)) \parallel L \parallel R_1(b, b'') \parallel T'_2)$
17. $\tau\, Z_2(l, b'', max) = \tau_I \partial_H(T_1 \parallel S_1(l, e_1, b'', rn, max) \parallel K \parallel L \parallel R \parallel T_2)$
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. The proof is given with induction on the length of l and within that with induction on $minus(max, rn)$. □

5 Final Steps of the Proof

Using the results in lemma 4.1.17 and 4.1.18 we may replace the first two equations in (I) by

$$Z_1(b'', max) = \sum_{l:List}(r_1(l)\, Z_2(l, b'', max))$$
$$Z'_1(b'', max) = \sum_{l:List}(r_1(l)\, Z'_2(l, e_1, b'', max))$$

We call the new set of equations (II). We can now show that

$$Z_1(b'', max) = X_1.$$

This follows from a straightforward application of RSP. We consider the equations (II). It is trivial to see that (II) is guarded. Now we substitute in (II) $\lambda b'', max.X_1$ for Z_1 and Z'_1. $\lambda l, b'', max.X_2(l, e_1)$ for Z_2. $\lambda l, b, b'', max.X_2(l, b)$ for Z'_2, $\lambda l, b'', max.s_4(head(l), I_{OK})(\tau\, X_3(I_{OK}) + \tau\, X_3(I_{DK}))$ for Z_3, $\lambda c, b'', max.X_3(c)$ for Z_4 and Z'_4, and $\lambda c, b'', max.X_4(c)$ for Z''_4. It is easy to see that the obtained equations are indeed derivable from the equations defining X_1, X_2, X_3 and X_4. For the equations with X_2 at the left hand side it is convenient to distinguish between $last(l) = \mathsf{t}$ (and hence $indl(l) = e_1$) and $last(l) = \mathsf{f}$ (with $indl(l) = e_0$). As $Z_1, \ldots, Z''_4$ are trivially a solution of (II) the result follows from RSP.

By lemma 4.1.1 and the definition of $BRP(max)$ we find that

$$BRP(max) = Z_1(e_0, max).$$

So, the two results imply that

$$BRP(max) = X_1$$

which had to be shown.

6 Computer Checking the Proof

The verification of the proof of the protocol has been carried out in the theorem prover `Coq V5.8.2` [5]. This system is based on type theory. Theorems are encoded as types, and inhabitants of the types (i.e. λ-terms) serve as proofs. Roughly spoken the logical strength of the system is the calculus of constructions, extended with inductive types. The details about the translation of axioms and rules of μCRL, and of process definitions and propositions into `Coq`, can be found in [15]. An overview of the axioms that we used is given in Appendix B.

We wanted to use the full power of `Coq`, and therefore the translation of the data sorts is less straightforward. For the sorts $\mathbb{N}, Bit, List$ and **Bool**, we divided the function symbols into *constructors* and *defined functions*. These sorts were then translated into inductive sets, generated by the *constructors*. This provides the following logical principles: An induction principle over the constructors (as in μCRL), inequality of different terms that consist of constructors only, and the possibility of recursive definitions on constructors. The mechanism of recursive definitions was used to define the *defined functions*. The recursive definitions follow the rewrite rules as given in the specification. In this way we benefit from the possibilities of `Coq`. For the other sorts $(D, TComm, Ind)$ this approach is not possible, because we do not know the constructors for sort D and we do not know that the elements in $TComm$ and Ind are pairwise different. We only know that *set* and *reset* are different signals.

The proof that is checked by the machine consists of a very large λ-term. The construction of this term by the machine is directed by vernacular commands, which reflect the proof. The files with vernacular commands are available and can be obtained by mailing one of the authors. The total amount of these vernacular commands is about 164 Kb, divided over 6000 lines. The vernacular code to build up the proof consists of several parts. The μCRL definitions and some standard theory

consists of 783 lines. The specification and implementation of the BRP protocol takes 604 lines. First some lemmas on data types are inferred, among which the induction principle used in Lemma 4.1 (370 lines). After these, 179 elementary communication lemmas are derived (2183 lines). These lemmas compute the communication between the different parts of the protocol. This could be done almost automatically, except in cases where full second order pattern matching was required. Then the straightforward expansions were computed (using the general expansion lemmas and the communication lemmas). After that, Lemma 4.1 could be proved smoothly, using the derived induction principle. The expansions and Lemma 4.1 fit in 1541 lines of vernacular code. This led to the final steps of the proof, good for 524 lines. The system of equations (II) was restated as one equation, using a conditional process, to be able to use the RSP-principle in one equation.

Appendix A Basic Data Types

Below basic data types that are used in the BRP are described. 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 verification.

sort **Bool**
func $\mathsf{f}, \mathsf{t} :\rightarrow \mathbf{Bool}$
$\wedge : \mathbf{Bool} \times \mathbf{Bool} \rightarrow \mathbf{Bool}$
var $b : \mathbf{Bool}$
rew $\mathsf{t} \wedge b = b$
$\mathsf{f} \wedge b = \mathsf{f}$
sort Bit
func $e_0, e_1 :\rightarrow Bit$
$inv : Bit \rightarrow Bit$
$if : \mathbf{Bool} \times Bit \times Bit \rightarrow Bit$
$eq : Bit \times Bit \rightarrow \mathbf{Bool}$
var $b, b_1, b_2 : Bit$
rew $inv(e_0) = e_1$
$inv(e_1) = e_0$
$if(\mathsf{t}, b_1, b_2) = b_1$
$if(\mathsf{f}, b_1, b_2) = b_2$
$if(eq(b_1, b_2), b_1, b_2) = b_2$
$eq(b, inv(b)) = \mathsf{f}$
$eq(b, b) = \mathsf{t}$
sort $D, List$
func $d_0 :\rightarrow D$
$if : \mathbf{Bool} \times D \times D \rightarrow D$
$eq : D \times D \rightarrow \mathbf{Bool}$
$empty :\rightarrow List$
$add : D \times List \rightarrow List$
$head : List \rightarrow D$
$tail : List \rightarrow List$
$last : List \rightarrow \mathbf{Bool}$
$indl : List \rightarrow Bit$
var $d, d_1, d_2 :\rightarrow D$
$l :\rightarrow List$
rew $head(empty) = d_0$
$head(add(d, l)) = d$
$tail(empty) = empty$
$tail(add(d, l)) = l$
$last(empty) = \mathsf{t}$
$last(add(d, empty)) = \mathsf{t}$
$last(add(d_1, add(d_2, l))) = \mathsf{f}$
$indl(empty) = e_1$
$indl(add(d, empty)) = e_1$
$indl(add(d_1, add(d_2, l))) = e_0$
$if(\mathsf{t}, d_1, d_2) = d_1$
$if(\mathsf{f}, d_1, d_2) = d_2$
$eq(d, d) = \mathsf{t}$
$if(eq(d_1, d_2), d_1, d_2) = d_2$
sort $\mathbb{N}$
func $0 :\rightarrow \mathbb{N}$
$s, pred : \mathbb{N} \rightarrow \mathbb{N}$
$eq : \mathbb{N} \times \mathbb{N} \rightarrow \mathbf{Bool}$
$lt : \mathbb{N} \times \mathbb{N} \rightarrow \mathbf{Bool}$
$minus : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$
var $n, n_1, n_2 :\rightarrow \mathbb{N}$
rew $eq(0, 0) = \mathsf{t}$
$eq(0, s(n)) = \mathsf{f}$
$eq(s(n), 0) = \mathsf{f}$
$eq(s(n_1), s(n_2)) = eq(n_1, n_2)$
$lt(0, s(n)) = \mathsf{t}$
$lt(n, 0) = \mathsf{f}$
$lt(s(n_1), s(n_2)) = lt(n_1, n_2)$
$pred(0) = 0$
$pred(s(n)) = n$
$minus(n, 0) = n$
$minus(n_1, s(n_2)) = pred(minus(n_1, n_2))$

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	$x+\delta = x$		
A7	$\delta\cdot x = \delta$	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 1: pCRL axioms

Appendix B Axioms of μCRL

All the process algebra axioms used to prove the BRP can be found in Table 1–3. These axioms form the basic theory that has been provided to the theorem prover `Coq`. We do not explain the axioms (see [1, 4, 9]) 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.

Besides the axioms we have used the principle RSP that says that guarded recursive specifications have at most one solution. In the following x, y denote parameterised processes that can be applied to a data parameter d of sort D, and deliver a process. The symbol Ψ is a process operator [4], i.e. an operator that reads a parameterised process and a datum element, and that delivers a process, and z must be a variable representing a parameterised process.

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

In the correctness proof in this paper we have used a weak notion of guardedness. A process p is *guarded* if

p is an action or δ or τ,
$p = p' + p''$ and both p' and p'' are guarded,
$p = p' \lhd B \rhd p''$ and both p' and p'' are guarded,
$p = \sum_{d:D} p'(d)$ and $p'(d)$ is guarded,
$p = a(d)\cdot p'$ with a an action,
$p = \tau\, p'$ and p' is guarded.

Furthermore we have postulated the following extensionality axiom that turned out useful in the proof. The letters x and y represent again parameterised processes.

$$\text{EXT} \qquad \frac{\forall d\ \ x(d)=y(d)}{x=y}.$$

Note that SUM11 is a consequence of the congruence of Σ and EXT.

References

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

SUM6	$\Sigma_{d:D}(p(d) \between z) = (\Sigma_{d:D}\ p(d)) \between z$		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	$x \parallel y = x \between y + y \between x + x\|y$		CT1	$\tau\|x = \delta$	
CM2	$c \between x = c{\cdot}x$		CT2	$x\|\tau = \delta$	
CM3	$c{\cdot}x \between y = c{\cdot}(x \parallel y)$				
CM4	$(x+y) \between z = x \between z + y \between z$		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 2: Primary μCRL axioms

TID	$\tau_I(\delta) = \delta$		SC1	$x \between (y \parallel z) = (x \between y) \between z$
TIT	$\tau_I(\tau) = \tau$		SC2	$x \between \delta = x\,\delta$
TI1	$\tau_I(a(d)) = a(d)$	if $a \notin I$	SC3	$x \mid y = y \mid x$
TI2	$\tau_I(a(d)) = \tau$	if $a \in I$	SC4	$(x \mid y) \mid z = x \mid (y \mid z)$
TI3	$\tau_I(x+y) = \tau_I(x) + \tau_I(y)$		SC5	$(x \mid y) \between z = x \mid (y \between z)$
TI4	$\tau_I(x{\cdot}y) = \tau_I(x){\cdot}\tau_I(y)$		Handshaking	$(x \mid y) \mid z = \delta$

Table 3: Axioms for τ, standard concurrency and handshaking

[2] 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 No. 99, Utrecht University, 1993.

[3] M.A. Bezem and J.F. Groote. A formal verification of the alternating bit protocol in the calculus of constructions. Technical Report Logic Group Preprint Series No. 88, Utrecht University, 1993.

[4] M.A. Bezem and J.F. Groote. Invariants in process algebra with data. Technical Report Logic Group Preprint Series No. 98, Utrecht University, 1993.

[5] 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.

[6] J.F. Groote and H. Korver. A correctness proof of the bakery protocol in μCRL. Technical Report Logic Group Preprint Series No. 80, Utrecht University, 1992.

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

[8] J.F. Groote and A. Ponse. μCRL: A base for analysing processes with data. In E. Best and G. Rozenberg, editors, *Proceedings* 3^{rd} *Workshop on Concurrency and Compositionality, Goslar, GMD-Studien Nr. 191*, pages 125–130. Universität Hildesheim, May 1991.

[9] J.F. Groote and A. Ponse. Proof theory for μCRL. Technical Report CS-R9138, CWI, Amsterdam, 1991.

[10] L. Helmink, M.P.A. Sellink, and F. Vaandrager. Proof-checking a data link protocol. 1994. To appear.

[11] ISO. *Information processing systems – open systems interconnection – LOTOS – a formal description technique based on the temporal ordering of observational behaviour* ISO/TC97/SC21/N DIS8807, 1987.

[12] H. Korver and J. Springintveld. A computer checked formal verification of Milner's scheduler. Technical Report Logic Group Preprint Series No. 101, Utrecht University, 1993.

[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. Technical Report Logic Group Preprint Series No. 87, Utrecht University, 1993.

[16] T. Winter. Personal Communications, 1992.