A Formal Verification of the Alternating Bit Protocol in the Calculus of Constructions

Marc Bezem
Jan Friso Groote
Department of Philosophy, Utrecht University
Heidelberglaan 8, 3584 CS Utrecht, The Netherlands
email: Marc.Bezem@phil.ruu.nl, JanFriso.Groote@phil.ruu.nl

Abstract

We report on a formal verification of the Alternating Bit Protocol (ABP) in the Calculus of Constructions. We outline a semi-formal correctness proof of the ABP with sufficient detail to be formalised. Thereafter we show by examples how the formalised proof has been verified by the automated proof checker Coq. This is part of an ongoing project aiming at the mechanisation of reasoning in (extensions of) process algebra, which we think important for the fruitful application of process algebra to concurrent systems.

Key Words & Phrases: protocol verification, process algebra, typed lambda calculi.
1985 Mathematics Subject Classification: 68B10.
1987 CR Categories: D.2.4, D.4.5, F.3.1.

1 Introduction

We report on a formal verification of the Alternating Bit Protocol [4] in the Calculus of Constructions, as part of an ongoing project aiming at the mechanisation of reasoning in (extensions of) process algebra. Formal verification distinguishes itself from verification in the usual sense in that all proof steps must follow precisely defined reasoning patterns, and every single detail must be taken into account. Correctness proofs for concurrent programs and protocols are of a combinatorial nature, and therefore error prone. We believe that, provided an automated proof checker is used, formal verification improves their level of correctness. This is important, especially in certain applications where it is expensive or sometimes even impossible to correct a problem after it manifests itself. Besides, we see formal verification also as a step towards the mechanisation of parts of the sometimes tedious and boring reasoning in this field.

We adopt the algebraic approach of Bergstra and Klop [5]. This approach is known as Algebra of Communicating Processes (ACP). More precisely, we use an extension of ACP with abstract data types called μCRL (Groote and Ponse [15]). A detailed proof system for μCRL is given in [16]. We build upon this proof system for our verification purposes.

For the proof verification we resort to type theory. Type theory (or typed lambda calculus, see for example Barendregt [3]) provides a very expressive formalism in which μCRL can conveniently be embedded. Also, it has a long standing tradition of automated verification of proofs (AUTOMATH [7], LCF [13], Nuprl [9], LEGO [17]). Without specific reason we chose the Calculus of Constructions of Coquand and Huet [10], on which the interactive proof construction and verification system Coq [11] has been based.

There are numerous examples of verifications of proofs of the correctness of programs and hardware. See for instance [6, 21]. For examples related to process theory, we only know of Cleaveland and Panangaden [8], who gave an implementation of Milner's Calculus of Communicating Systems for recursion-free processes [18] in the NuPrl system [9], and from Engberg, Grønning and Lamport [12], who verified proofs in the Temporal Logic of Actions, which is a logic for specifying and reasoning about concurrent systems.

We follow the lines as set out by Sellink in [20], who describes how μCRL (including recursive processes) with its accompanying proof theory can be embedded in Coq.

From the current experiment we draw a number of conclusions. Modeling process algebra in a type theory based logic is not too difficult (although there remain a number of fundamental questions concerning the adequacy of our approach; these are mentioned below). A major technological problem lies in dealing adequately with the large number of elementary proof steps that must all be exhibited to be verifiable by machine. Progress in this field can be defined as reducing the number of proof steps that have to be specified by the user. The proof checker should have enough intelligence to infer the 'obvious' steps by itself. One may think of relying more on general techniques, such as unification and matching, but also on specialised techniques such as conditional higher order term rewriting. Some initial experiments show that reductions of 90% of the length of the specified proof are possible. Further improvement may be expected by devising process-algebra-tailored tactics that subsume numerous elementary proof steps.

One may justifiably question whether our approach is adequate. Our formalisation uses all kinds of sophisticated primitives of Coq, such as inductive definitions and higher order logic. The reason for this is pragmatic; in doing so we benefit from the tactics that Coq provides to handle these primitives. However, formalising notions of process algebra using these primitives does not necessarily exactly preserve meaning and the consequences are not yet fully understood. We do not know whether this formalisation is conservative over μCRL — although non-conservativity would not be a problem as long as all provable equations between processes are true under some preferred semantics.

To give an example, implication is encoded by the type constructor $\rightarrow$, which corresponds to *constructive* implication, whereas in the proof theory of μCRL the *classical* $\rightarrow$ was intended. Another example is our choice for inductive equality to represent equality between processes. Logically speaking this gives equality the Leibniz property, i.e. two processes are equal if and only if they share the same properties. In a very expressive formalism such as Coq this is rather strong since in Coq one can express much more properties than, for example, in first order algebra.

One may wonder why we have not attempted to understand our formalisation to its full extent before applying it to the alternating bit protocol. The reason for this is again pragmatic. Our goal is the automated verification of correctness proofs for protocols and the present question is whether type theory provides a feasible approach to this problem. Despite the problems sketched above, we have answered this question affirmatively.

The paper is organised as follows. In Section 2 we present the Alternating Bit Protocol (ABP). Section 3 gives an overview μCRL with its proof theory and outlines the correctness proof of the ABP that has been verified. In Section 4 we illustrate our formalisation of this correctness proof in the system Coq by a number of representative examples.

A file called 'ABP.v', containing the full version of the verified proof in the vernacular of Coq, can be obtained by contacting the authors.

2 The Alternating Bit Protocol

The Alternating Bit Protocol (ABP) is a communication protocol providing reliable transmission of data through an unreliable (two-way) channel. It consists of four components: a sender S, a receiver R, a channel K from S to R and a channel L from R to S. These components are connected according to Figure 1. The numbered connection lines in Figure 1 represent gates, through which the components can communicate. The sender S reads data from the input at gate 1, sends frames consisting of a bit and a datum into the channel K at gate 2 and receives acknowledgement bits from channel L at gate 6. These actions are represented by, respectively, $r_1(d)$, $s_2(n,d)$ and $r_6(n)$. The receiver R receives

Figure 1: Alternating Bit Protocol.

frames from channel K at gate 3, writes data to the output at gate 4 and acknowledges receipts by sending bits into the channel L at gate 5. These actions are represented by $r_3(n, d)$, $s_4(d)$ and $s_5(n)$, respectively. All these r/s actions have their s/r counterpart in the component with which the gate in question is shared. Communication is synchronous, i.e. only occurs when complementary r/s actions are executed simultaneously at the same gate. The resulting action is denoted by c, i.e. $\gamma(s_j, r_j) = c_j$ for $j = 2, 3, 5, 6$. The channels may corrupt data, but if they do so they are assumed to do this explicitly by sending an error message: $s_3(\perp)$ for K and $s_6(\perp)$ for L. Moreover, the channels are assumed not to corrupt data *ad infinitum* (in that case it is obviously impossible to ensure reliable transmission).

The ABP roughly works as follows: S reads a datum d from the input and starts sending frames (e_0, d) via K to R. Once R receives a frame (e_0, d) it writes d to the output and starts acknowledging the receipt of frame (e_0, d) by sending bits e_0 via L to S. During this period occasional incoming frames $(e_0, \ldots)$ are ignored by R. Process S only stops sending frames (e_0, d) once an acknowledging bit e_0 is received, and then reads a new datum d' from the input and starts sending frames (e_1, d') to R. During this period occasional incoming acknowledgements e_0 are ignored by S. Process R only stops acknowledging with bit e_0 after a frame (e_1, d') is received, then writes d' to the output and starts acknowledging the receipt of frame (e_1, d') by sending bits e_1 to S, and so on. It should be clear that the alternating bit is essential to distinguish new frames from old ones (note that it is not excluded that $d' = d$) and to distinguish the acknowledgement of a new frame from that of an old one.

The question arises: is the ABP correct? This question can only be answered after having specified a correctness criterion: the ABP should behave externally like a buffer. This raises several other questions: what is 'the ABP', what is 'a buffer' and what is 'behave externally'? These questions should be answered by giving formal specifications, instead of e.g. the rough description of the ABP above. In the next section we provide the means to do this in a mathematical, semi-formal way.

3 Semi-formal correctness proof

3.1 Process algebra

An *algebra* is usually a set together with a number of operations on that set, in principle axiomatised by an equational theory. Process algebra complies with this tradition. The set is a set of processes and the operations are $+$ (alternative composition), $\cdot$ (sequential composition), $\|$, $\llfloor$ and $|$ (parallel merge, left merge and communication merge). Furthermore there is a constant δ (deadlock) and an encapsulation operator $\partial_H(x)$, a constant τ (silent step) and an abstraction (or hiding) operator $\tau_I(x)$, as well as a sum operator $\sum_{d:D}(x)$ for possibly infinite sums. We refer to [2] for an explanation of these operators.

Processes may use data. We model this by exhibiting the data as arguments of the processes. In order to be able to reason about processes and data one needs a little more expressivity than just process algebra. The formalism μCRL from [15], combining process algebra with abstract data types, provides already more than enough expressivity for our purpose here.

In [16] a proof system for μCRL has been given which allows to prove identities about processes. Table 2 lists the axioms of ACP in μCRL, followed by the axioms of standard concurrency (Table 3), the hiding operator (Table 4) and the sum operator (Table 5). An important principle is RSP (Recursive Specification Principle) that says that each guarded recursive equation has at most one solution (Table 6). Furthermore, we use one standard law for abstraction (T1) and a rule enforcing fairness (KFAR2, a variant of the rule in [2]). These are given in Table 7.

The data types in μCRL are rather standard, and straightforward to read. Therefore we refrain from describing this part of μCRL in the present paper and refer to [15]. In Appendix A we give the data types and some elementary lemmas that we use in this paper. In the main text we write $\perp$ for checksum errors *lce* and *sce*, and omit the functions *Tuple* and *tuple* embedding a pair of datum and bit, and a bit, respectively, into frames.

$$\begin{array}{lcl}
K & = & \sum_{x:Frame}(r_2(x)(i \cdot s_3(x) + i \cdot s_3(\perp))) \cdot K \\
L & = & \sum_{n:frame}(r_5(n)(i \cdot s_6(n) + i \cdot s_6(\perp))) \cdot L \\
S & = & S(e_0) \cdot S(e_1) \cdot S \\
S(n{:}bit) & = & \sum_{d:D}(r_1(d) \cdot S(n,d)) \\
S(n{:}bit, d{:}D) & = & s_2(n,d) \cdot T(n,d) \\
T(n{:}bit, d{:}D) & = & (r_6(toggle(n)) + r_6(\perp)) \cdot S(n,d) + r_6(n) \\
R & = & R(e_1) \cdot R(e_0) \cdot R \\
R(n{:}bit) & = & (\sum_{d:D} r_3(n,d) + r_3(\perp)) \cdot s_5(n) \cdot R(n) + \\
 & & \qquad \sum_{d:D}(r_3(toggle(n),d) \cdot s_4(d) \cdot s_5(toggle(n))) \\
ABP & = & \partial_H(S \parallel K \parallel L \parallel R)
\end{array}$$

Table 1: Specification of ABP ($H = \{r_2, s_2, r_3, s_3, r_5, s_5, r_6, s_6\}$).

3.2 Specification and correctness criterion for ABP

The ABP can be described in many ways in μCRL. We have taken the description given in [2] as starting point (see Table 1). The external behaviour of the ABP, where only activities at gates 1 and 4 are visible, is supposed to be a one-element buffer B, defined by the following equation.

$$B = \sum_{d:D}(r_1(d) \cdot s_4(d)) \cdot B.$$

So, we say that the ABP is correct if we can prove that

$$\tau_I(ABP) = B$$

where $I = \{c_2, c_3, c_5, c_6, i\}$. It is this equation of which we verify the proof.

3.3 Correctness proof for ABP

In this subsection we outline a semi-formal correctness proof for ABP. This proof consists of three stages. We start with the linearisation of the specification of ABP, which takes up 65% of ABP.v. First we define the following auxiliary processes:

A1	$x + y = y + x$	CF1	$n_1 \mid n_2 = n_3$	if $\gamma(n_1, n_2) = n_3$
A2	$x + (y + z) = (x + y) + z$			
A3	$x + x = x$	CF1′	$n_1(t_1, \ldots, t_m) \mid n_2(t_1, \ldots, t_m) = n_3(t_1, \ldots, t_m)$	if $\gamma(n_1, n_2) = n_3$
A4	$(x + y) \cdot z = x \cdot z + y \cdot z$			
A5	$(x \cdot y) \cdot z = x \cdot (y \cdot z)$			
A6	$x + \delta = x$	CF2	$a \mid b = \delta$	if $\gamma(\mathit{label}(a), \mathit{label}(b)) = \delta$
A7	$\delta \cdot x = \delta$			
		CF2′	$\neg(t_i = t_i') \rightarrow n_1(t_1, \ldots, t_m) \mid n_2(t_1', \ldots, t_m') = \delta$	for some $1 \leq i \leq m$
CM1	$x \parallel y = x \mathbin{\|\!\!\lfloor} y + y \mathbin{\|\!\!\lfloor} x + x \mid y$			
CM2	$a \mathbin{\|\!\!\lfloor} x = a \cdot x$			
CM3	$a \cdot x \mathbin{\|\!\!\lfloor} y = a \cdot (x \parallel y)$	CF2″	$n_1(t_1, \ldots, t_m) \mid n_2(t_1', \ldots, t_{m'}') = \delta$	if $m \neq m'$
CM4	$(x + y) \mathbin{\|\!\!\lfloor} z = x \mathbin{\|\!\!\lfloor} z + y \mathbin{\|\!\!\lfloor} z$			
CM5	$a \cdot x \mid b = (a \mid b) \cdot x$			
CM6	$a \mid b \cdot x = (a \mid b) \cdot x$	D1	$\partial_H(a) = a$	if $\mathit{label}(a) \notin H$
CM7	$a \cdot x \mid b \cdot y = (a \mid b) \cdot (x \parallel y)$	D2	$\partial_H(a) = \delta$	if $\mathit{label}(a) \in H$
CM8	$(x + y) \mid z = x \mid z + y \mid z$	D3	$\partial_H(x + y) = \partial_H(x) + \partial_H(y)$	
CM9	$x \mid (y + z) = x \mid y + x \mid z$	D4	$\partial_H(x \cdot y) = \partial_H(x) \cdot \partial_H(y)$	

Table 2: The ACP axioms in μCRL.

SC1	$(x \mathbin{\|\!\!\lfloor} y) \mathbin{\|\!\!\lfloor} z = x \mathbin{\|\!\!\lfloor} (y \parallel z)$	SC4	$(x \mid y) \mid z = x \mid (y \mid z)$
SC2	$x \mathbin{\|\!\!\lfloor} \delta = x\delta$	SC5	$x \mid (y \mathbin{\|\!\!\lfloor} z) = (x \mid y) \mathbin{\|\!\!\lfloor} z$
SC3	$x \mid y = y \mid x$	Handshaking	$x \mid (y \mid z) = \delta$

Table 3: Axioms of Standard Concurrency (SC).

TI1	$\tau_I(a) = a$	if $\mathit{label}(a) \notin I$
TI2	$\tau_I(a) = \tau$	if $\mathit{label}(a) \in I$
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: Axioms for abstraction.

SUM1	$\sum_{d:D} p = p$	if d not free in p
SUM2	$\sum_{d:D} p = \sum_{d:D}(p[e/d])$	if e not free in p
SUM3	$\sum_{d:D} p = (\sum_{d:D} p) + p$	
SUM4	$\sum_{d:D}(p_1 + p_2) = \sum_{d:D} p_1 + \sum_{d:D} p_2$	
SUM5	$\sum_{d:D}(p_1 \cdot p_2) = \sum_{d:D} p_1 \cdot p_2$	if d not free in p_2
SUM6	$\sum_{d:D}(p_1 \mathbin{\|\!\lfloor} p_2) = \sum_{d:D} p_1 \mathbin{\|\!\lfloor} p_2$	if d not free in p_2
SUM7	$\sum_{d:D}(p_1 \mid p_2) = \sum_{d:D} p_1 \mid p_2$	if d not free in p_2
SUM8	$\sum_{d:D} \partial_H(p) = \partial_H(\sum_{d:D} p)$	
SUM9	$\sum_{d:D} \tau_I(p) = \tau_I(\sum_{d:D} p)$	
SUM11	$\dfrac{\begin{array}{c}\mathcal{D}\\ p_1 = p_2\end{array}}{\sum_{d:D} p_1 = \sum_{d:D} p_2}$	provided d not free in the assumptions of $\mathcal{D}$

Table 5: Axioms for the sum operator.

RSP

$$\frac{(\bigwedge_{i=1}^{m}(G_i[\lambda\bar{x}_j \,.\, p_j(\bar{x}_j)/n_j]_{j=1}^{m} \wedge G_i[\lambda\bar{x}_j \,.\, q_j(\bar{x}_j)/n_j]_{j=1}^{m}))}{p_k(\bar{x}_k) = q_k(\bar{x}_k)}$$

where

- $G_1, \ldots, G_m$ is a *guarded* system of process equations,
- for $1 \leq i \leq m$ the $p_i(\bar{x}_i)$ and $q_i(\bar{x}_i)$ are process terms,
- the notation $[\ldots]_{j=1}^{m}$ abbreviates the m given, simultaneous substitutions.

Table 6: The rule RSP.

T1	$a \cdot \tau = a$	
KFAR2	$\dfrac{x = i(d) \cdot i(d) \cdot x + y}{\tau \cdot \tau_I(x) = \tau \cdot \tau_I(y)}$	if $i \in I$

Table 7: The rules for internal actions; KFAR2 and T1

Definition 3.1. As in Table 1, $H = \{r_2, s_2, r_3, s_3, r_5, s_5, r_6, s_6\}$ is a set of labels.

- $X = \partial_H(S \parallel K \parallel L \parallel R)$.
- $X1(d) = \partial_H(S(e_0, d) \cdot S(e_1) \cdot S \parallel K \parallel L \parallel R)$.
- $X2(d) = \partial_H(T(e_0, d) \cdot S(e_1) \cdot S \parallel K \parallel L \parallel s_5(e_0) \cdot R(e_0) \cdot R)$.
- $Y = \partial_H(S(e_1) \cdot S \parallel K \parallel L \parallel R(e_0) \cdot R)$.
- $Y1(d) = \partial_H(S(e_1, d) \cdot S \parallel K \parallel L \parallel R(e_0) \cdot R)$.
- $Y2(d) = \partial_H(T(e_1, d) \cdot S \parallel K \parallel L \parallel s_5(e_1) \cdot R)$.

We show that the auxiliary processes satisfy the following system of linear equations. The names at the right are those under which the results are saved in the file ABP.v.

Lemma 3.2.

- $X = \sum_{d:D}(r_1(d) \cdot X1(d))$. *Lem1*
- $X1(d) = c_2(e_0, d) \cdot (i \cdot c_3(\perp) \cdot c_5(e_1) \cdot (i \cdot c_6(\perp) + i \cdot c_6(e_1)) \cdot X1(d) +$
  $i \cdot c_3(e_0, d) \cdot s_4(d) \cdot X2(d))$. *Lem12*
- $X2(d) = c_5(e_0) \cdot (i \cdot c_6(\perp) \cdot c_2(e_0, d) \cdot (i \cdot c_3(\perp) + i \cdot c_3(e_0, d)) \cdot X2(d) + i \cdot c_6(e_0) \cdot Y)$. *Lem31*
- $Y = \sum_{d:D}(r_1(d) \cdot Y1(d))$. *Lem2*
- $Y1(d) = c_2(e_1, d) \cdot (i \cdot c_3(\perp) \cdot c_5(e_0) \cdot (i \cdot c_6(\perp) + i \cdot c_6(e_0)) \cdot Y1(d) +$
  $i \cdot c_3(e_1, d) \cdot s_4(d) \cdot Y2(d))$. *Lem22*
- $Y2(d) = c_5(e_1) \cdot (i \cdot c_6(\perp) \cdot c_2(e_1, d) \cdot (i \cdot c_3(\perp) + i \cdot c_3(e_1, d)) \cdot Y2(d) + i \cdot c_6(e_1) \cdot X)$. *Lem41*

Proof. We strictly follow the expansions as given in [5, 2]. As an example we show how the equation *Lem12* is derived.

$$
\begin{aligned}
& X1(d) = \\
& \partial_H(S(e_0,d) \cdot S(e_1) \cdot S \parallel K \parallel L \parallel R) = \\
& c_2(e_0,d) \cdot \partial_H(T(e_0,d) \cdot S(e_1) \cdot S \parallel (i \cdot s_3(e_0,d) + i \cdot s_3(\perp)) \cdot K \parallel L \parallel R) = \\
& c_2(e_0,d) \cdot (i \cdot \partial_H(T(e_0,d) \cdot S(e_1) \cdot S \parallel s_3(e_0,d) \cdot K \parallel L \parallel R) + \\
& \quad i \cdot \partial_H(T(e_0,d) \cdot S(e_1) \cdot S \parallel s_3(\perp) \cdot K \parallel L \parallel R) = \\
& c_2(e_0,d) \cdot (i \cdot c_3(e_0,d) \cdot \partial_H(T(e_0,d) \cdot S(e_1) \cdot S \parallel K \parallel L \parallel s_4(d) \cdot s_5(e_0) \cdot R(e_0) \cdot R) + \\
& \quad i \cdot c_3(\perp) \cdot \partial_H(T(e_0,d) \cdot S(e_1) \cdot S \parallel K \parallel L \parallel s_5(e_1) \cdot R(e_1) \cdot R(e_0) \cdot R)) = \\
& c_2(e_0,d) \cdot (i \cdot c_3(e_0,d) \cdot s_4(d) \cdot \partial_H(T(e_0,d) \cdot S(e_1) \cdot S \parallel K \parallel L \parallel s_5(e_0) \cdot R(e_0) \cdot R) + \\
& \quad i \cdot c_3(\perp) \cdot c_5(e_1) \cdot \partial_H(T(e_0,d) \cdot S(e_1) \cdot S \parallel K \parallel (i \cdot s_6(e_1) + i \cdot s_6(\perp)) \cdot L) \parallel R)) = \\
& c_2(e_0,d) \cdot (i \cdot c_3(e_0,d) \cdot s_4(d) \cdot X2(d) + i \cdot c_3(\perp) \cdot c_5(e_1) \cdot \\
& \quad \{i \cdot \partial_H(T(e_0,d) \cdot S(e_1) \cdot S \parallel K \parallel s_6(e_1) \cdot L \parallel R) + \\
& \quad\quad i \cdot \partial_H(T(e_0,d) \cdot S(e_1) \cdot S \parallel K \parallel s_6(\perp)) \cdot L \parallel R)\}) = \\
& c_2(e_0,d) \cdot (i \cdot c_3(e_0,d) \cdot s_4(d) \cdot X2(d) + i \cdot c_3(\perp) \cdot c_5(e_1) \cdot \\
& \quad \{i \cdot c_6(e_1) \cdot \partial_H(S(e_0,d) \cdot S(e_1) \cdot S \parallel K \parallel L \parallel R) + \\
& \quad\quad i \cdot c_6(\perp) \cdot \partial_H(S(e_0,d) \cdot S(e_1) \cdot S \parallel K \parallel L \parallel R)\}) = \\
& c_2(e_0,d) \cdot (i \cdot c_3(e_0,d) \cdot s_4(d) \cdot X2(d) + i \cdot c_3(\perp) \cdot c_5(e_1) \cdot \\
& \quad \{i \cdot c_6(e_1) \cdot X1(d) + i \cdot c_6(\perp) \cdot X1(d)\}) = \\
& c_2(e_0,d) \cdot (i \cdot c_3(e_0,d) \cdot s_4(d) \cdot X2(d) + i \cdot c_3(\perp) \cdot c_5(e_1) \cdot \{i \cdot c_6(e_1) + i \cdot c_6(\perp)\} \cdot X1(d)).
\end{aligned}
$$

□

Using only axioms in justifying merge expansions such as above would lead to a hardly attainable proof. So the proof is split up in a number of basic lemmas, thus avoiding duplication of identical reasoning patterns. Lemma 3.3 is a general lemma that can be applied to expand every merge of four processes. There are no three and four party communications as we have assumed the Handshaking axiom (Table 3).

Lemma 3.3. *Let x, y, z and u be variables that range over process terms.*

$$\begin{aligned}&\partial_H(x \parallel y \parallel z \parallel u) = && \textit{EXPH4}\\ &\quad \partial_H(x \mathbin{|\!\lfloor} (y \parallel z \parallel u)) + \partial_H(y \mathbin{|\!\lfloor} (x \parallel z \parallel u)) + \partial_H(z \mathbin{|\!\lfloor} (x \parallel y \parallel u)) + \partial_H(u \mathbin{|\!\lfloor} (x \parallel y \parallel z)) +\\ &\quad \partial_H((z \mid u) \mathbin{|\!\lfloor} (x \parallel y)) + \partial_H((y \mid z) \mathbin{|\!\lfloor} (x \parallel u)) + \partial_H((y \mid u) \mathbin{|\!\lfloor} (x \parallel z)) +\\ &\quad \partial_H((x \mid y) \mathbin{|\!\lfloor} (z \parallel u)) + \partial_H((x \mid z) \mathbin{|\!\lfloor} (y \parallel u)) + \partial_H((x \mid u) \mathbin{|\!\lfloor} (y \parallel z))\end{aligned}$$

Whenever we apply *EXPH4* in our proof of the ABP we encounter terms such as

$$\begin{aligned}&\partial_H(K \mathbin{|\!\lfloor} (S(e_0, d) \cdot S(e_1) \cdot S \parallel L \parallel R))\\ &\partial_H((K \mid L) \mathbin{|\!\lfloor} (S(e_0, d) \cdot S(e_1) \cdot S \parallel R))\\ &\partial_H((S(e_0, d) \cdot S(e_1) \cdot S) \mid K) \mathbin{|\!\lfloor} (L \parallel R))\end{aligned}$$

that need to be expanded further. In order to do this there are about 50 lemmas such as:

Lemma 3.4. *Let x range over processes.*

- $\delta = \partial_H(K \mathbin{|\!\lfloor} x)$. *LmerK*
- $\delta = \partial_H((K \mid L) \mathbin{|\!\lfloor} x)$. *CommKL*
- $\partial_H((S(d, b) \cdot y \mid K) \mathbin{|\!\lfloor} x) =$ *CommSn_dK*
  $c_2(b, d)) \cdot \partial_H(((T(b, d) \cdot y)i \parallel i \cdot s_3(b, d) \cdot K + i \cdot s_3(\perp) \cdot K \parallel x))))$.

In *CommSn_dK* a datum d and a bit b are transferred from the sender to the channel K. Our calculations follow those in [14] where the *sum-elimination* lemma is used (recall that the specification of K starts with a sum). In the verifications of the ABP in [2] these complications remain invisible as data is handled fully informally.

The next stage of the correctness proof of the ABP (covering the last 5% of ABP.v) consists of applying Koomen's Fair Abstraction Rule (KFAR). This rule expresses that a process does not always make the same choice of actions if it has several options. In other words: choices are fair. For the ABP to function properly, we must assume that frames will not always get lost, but will at least sometimes be delivered. Our calculations diverge from those in [5, 2] where different fairness rules have been used.

We define auxiliary processes $X1'(d)$, $X2'(d)$, $Y1'(d)$ and $Y2'(d)$ as follows (with $I' = \{c_2, c_3, c_5, c_6\}$):

$$\begin{aligned}&X1'(d) = i \cdot s_4(d) \cdot \tau_{I'}(X2(d)) + i \cdot i \cdot X1'(d)\\ &X2'(d) = i \cdot i \cdot X2'(d) + i \cdot \tau_{I'}(Y)\\ &Y1'(d) = i \cdot s_4(d) \cdot \tau_{I'}(Y2(d)) + i \cdot i \cdot Y1'(d)\\ &Y2'(d) = i \cdot i \cdot Y2'(d) + i \cdot \tau_{I'}(X)\end{aligned}$$

For these processes we can straightforwardly prove the following lemma:

Lemma 3.5.

- $\tau_{I'}(X1(d)) = \tau \cdot X1'(d)$. *LemLin2*
- $\tau_{I'}(X2(d)) = \tau \cdot X2'(d)$. *LemLin3*
- $\tau_{I'}(Y1(d)) = \tau \cdot Y1'(d)$. *LemLin5*
- $\tau_{I'}(Y2(d)) = \tau \cdot Y2'(d)$. *LemLin6*

Proof. It is obvious that $\tau_{I'}(X1(d))$ satisfies the defining equation for $X1'(d)$ with an additional τ prefixed at the right hand side of the '='. The other identities are proved likewise. □

The following step consists of showing the identities (where $I'' = \{i\}$) by using Koomen's Fair Abstraction Rule.

Lemma 3.6.

- $\tau \cdot \tau_{I''}(X1'(d)) = \tau \cdot s_4(d) \cdot \tau_{I''}(\tau_{I'}(X2(d)))$. *KFLin2*
- $\tau \cdot \tau_{I''}(X2'(d)) = \tau \cdot \tau_{I''}(\tau_{I'}(Y))$. *KFLin3*
- $\tau \cdot \tau_{I''}(Y1'(d)) = \tau \cdot s_4(d) \cdot \tau_{I''}(\tau_{I'}(Y2(d)))$. *KFLin5*
- $\tau \cdot \tau_{I''}(Y2'(d)) = \tau \cdot \tau_{I''}(\tau_{I'}(X))$. *KFLin6*

Proof. Straightforward using KFAR2. □

By the previous two lemmas it is straightforward to prove the correctness of the ABP:

Theorem 3.7.

$$\tau_{I''}(\tau_{I'}(ABP)) = B \qquad \textit{Hurrah}$$

Proof. Show both sides of the equation a solution of:

$$Z = \sum_{d:D}(r_1(d) \cdot s_4(d) \cdot \sum_{d:D}(r_1(d) \cdot s_4(d) \cdot Z)).$$

□

Since $I = I' \cup I''$ it follows that $\tau_I(ABP) = B$, i.e. the correctness of the ABP. This last step actually requires $\tau_I(x) = \tau_{I''}(\tau_{I'}(x))$. This is obvious and therefore we do not introduce new axioms which allow us to prove this (see [1]).

4 Formal correctness proof

In this section we take some examples from the file ABP.v to give an overview of different aspects which occur in the verification. As the size of the file is about 200Kbyte and since large parts of the proof consist of repeating similar proof steps again and again, it is not useful to explain the whole file.

4.1 $r_1 \neq r_2$

In this subsection we prove that distinct actions are not equal, which takes up almost 20% of ABP.v. We need this to show that actions do not occur in sets (see for instance rule D1 in Table 2). Following [20] we model actions using an inductive type (see [19]).

The actions are defined using the following definition.

```
Inductive Set act = r1:act | r2:act | r3:act | r5:act | r6:act
                  | s2:act | s3:act | s4:act | s5:act | s6:act
                  | c2:act | c3:act | c5:act | c6:act
                  | int:act| delta:act | tau:act.
```

This expresses that the set `act` consists of the actions `r1, r2,...,delta,tau` and nothing more (`r1, r2,...` are called the *constructors* of the inductive set `act`). A proof principle saying so (actually the induction principle for `act`) is automatically generated by the system Coq. Furthermore, a mechanism called `Match` is generated which allows to build functions that can distinguish between different elements of the set `act` (actually the schema for primitive recursion of `act`). Also in a more general situation of an inductive definition facilities for induction and recursion with respect to the so-called constructors of the inductive type are provided by Coq.

We want to show the following statement, which we announce to Coq as the goal to be proven by the command `Goal`:

```
Goal ~(<act>r1=r2).
```

Here `~` stands for 'not', which is in higher order type theory usually defined by $\lambda p{:}Prop.p{\rightarrow}False$ where $Prop$ is the sort of propositions. In Coq notation, lambda abstraction is denoted by square brackets, so that `~` abbreviates `[p:Prop]p->False`. The denotation `<...>...=...` is shorthand for an inductively defined equality relation. Logically speaking, `<act>r1=r2` amounts to Leibniz equality of actions `r1` and `r2`.

The Coq system responds (automatically, or after the command `Show`) with

```
1 subgoal
   ~(<act>r1=r2)
```

meaning that we must prove `~(<act>r1=r2)` from the empty set of premisses. In case the set of premisses is not empty, they are listed below a line =============.

The command `Red` replaces `~` by its definition and (after implicit β-reduction) the situation becomes as follows.

```
Red.

1 subgoal
  (<act>r1=r2)->False
```

The command `Intro` introduces the premiss of the current goal as hypothesis and changes the goal into the conclusion. Variants of this command are `Intros` (repeats `Intro` as many times as possible) and `Intro` *name* (introduces the hypothesis under the name *name*, instead of under a default name).

```
Intro.

1 subgoal
  False
  ============================
    H : <act>r1=r2
```

Completing this proof is not completely trivial, since it involves employing the mechanism to distinguish between the different elements of an inductive type. We use this device in the following definition:

```
Definition r1f = [a:act](<Prop>Match a with True  False False False False
                                             False False False False False
                                             False False False False
                                             False False False).
```

In the definition above, `a` is of type `act`. If `a` matches the n^{th} constructor in the inductive definition of `act`, then the `Match`-expression yields the n^{th} argument after `with`.

The idea of the proof is that `(r1f r2)` reduces to `False` and `(r1f r1)` to `True` and that these must be equal if `<act>r1=r2`. By proving `True`, which is trivial, we can then prove `False`. The following sequence of commands first changes the goal `False` into `(r1f r2)`, then substitutes `r1` for `r2` and finally completes the proof by using the canonical inhabitant `I` of `True`.

```
Change (r1f r2).
Elim H.
Exact I.
```

The substitution of `r1` for `r2` is done through the command `Elim H`. The type of `H` is `<act>r1=r2`, which is an inductive type due to the inductive definition of equality. The command `Elim H` refers to the elimination rule that comes with every inductive definition (actually the typing rule for `Match`-expressions). In the case of inductive equality, the result is the desired substitution. For reasons of space we have to refrain from explaining the command `Elim H` in a more general situation. The command `Exact` allows one to prove a goal by giving a proof term explicitly.

Finally the lemma is saved under the name `neqr1r2` (not equal r_1 and r_2) by the following command, after which it can be used in any proof to follow.

```
Save neqr1r2.
```

All other pairs of distinct actions are treated in the same way. We finish this subsection by observing that equality of identical actions (e.g. `<act>r1=r1`) comes automatically with the (inductive) definition of Leibniz equality.

4.2 $r_1 \notin H$

In this subsection we show how to prove that action r_1 is not an element of the set H. The representation we chose for the set H is the list `H` defined below. The context is assumed to contain the set of lists of actions, `ehlist:Set`, inductively defined with constructors `ehnil` and `ehcons`, as well as a predicate `In_ehlist:act -> ehlist -> Prop` defining the membership relation in the usual recursive way.

```
Definition H=(ehcons r2
               (ehcons r3
                (ehcons r5
                 (ehcons r6
                  (ehcons s2
                   (ehcons s3
                    (ehcons s5
                     (ehcons s6 ehnil)))))))).
```

The desired result $r_1 \notin H$ is obtained from a more general lemma which we prove first.

```
Goal (a:act)(~<act>a=r2)->(~<act>a=r3)->(~<act>a=r5)->(~<act>a=r6)->
  (~<act>a=s2)-> (~<act>a=s3)->(~<act>a=s5)->(~<act>a=s6)->~(In_ehlist a H).
```

Here `(a:act)` stands for: for all `a` in `act`. Actually, the round brackets stand for Π-abstraction in the same way as square brackets stand for λ-abstraction. A Π-abstraction denotes a product type. The type $\sigma \rightarrow \tau$ is actually also a product type, namely $\Pi x{:}\sigma.\tau$ with x not occurring in τ. The command `Intro` and all its variants also work for product types in general. We enter the command `Intros` and print the resulting situation.

```
Intros.
```

```
1 subgoal
  ~(In_ehlist a H)
  ============================
    H7 : ~<act>a=s6
    H6 : ~<act>a=s5
    H5 : ~<act>a=s3
    H4 : ~<act>a=s2
    H3 : ~<act>a=r6
    H2 : ~<act>a=r5
    H1 : ~<act>a=r3
    H0 : ~<act>a=r2
    a : act
```

We enter the following two commands for, respectively, expanding `~` in the goal and unfolding `(In_ehlist r1 H)` according to the recursive definition of `In_ehlist`:

```
Red.
Unfold In_ehlist.
```

```
1 subgoal
  ((<act>a=r2)
   \/(<act>a=r3)
     \/(<act>a=r5)
       \/(<act>a=r6)
         \/(<act>a=s2)
           \/(<act>a=s3)
             \/(<act>a=s5)
               \/(<act>a=s6)
                 \/False)
  ->False
  ============================
    H7 : ~<act>a=s6
    H6 : ~<act>a=s5
    H5 : ~<act>a=s3
    H4 : ~<act>a=s2
    H3 : ~<act>a=r6
    H2 : ~<act>a=r5
    H1 : ~<act>a=r3
    H0 : ~<act>a=r2
    a : act
```

Here `\/` represents disjunction. The obvious way to continue is by introducing the premiss of this implication as an assumption. This is done by an `Intro`. To continue we `Elim`inate the disjunction and obtain two subgoals, since both disjuncts must imply the goal. By convention, disjunction associates to the right.

```
Intro I1.
Elim I1.
```

```
2 subgoals
  (<act>a=r2)->False
  ============================
    I1 : (<act>a=r2)
```

```
         \/(<act>a=r3)
           \/(<act>a=r5)
             \/(<act>a=r6)
               \/(<act>a=s2)
                 \/(<act>a=s3)\/(<act>a=s5)\/(<act>a=s6)\/False
    H7 : ~<act>a=s6
    H6 : ~<act>a=s5
    H5 : ~<act>a=s3
    H4 : ~<act>a=s2
    H3 : ~<act>a=r6
    H2 : ~<act>a=r5
    H1 : ~<act>a=r3
    H0 : ~<act>a=r2
    a : act

subgoal 2 is:
  ((<act>a=r3)
   \/(<act>a=r5)
     \/(<act>a=r6)
       \/(<act>a=s2)
         \/(<act>a=s3)
           \/(<act>a=s5)
             \/(<act>a=s6)
               \/False)
  ->False
```

We continue with the first subgoal. Subgoal 2 is treated later. Due to the hypothesis `H0:~<act>a=r2`, the first subgoal is proved by a simple application of the command

```
Assumption.
```

Subgoal 2 is similar but shorter than the goal obtained just before the command `Intro I1` above. Although not explicitly shown, the same hypotheses as for Subgoal 1 may be used also for Subgoal 2. Therefore the proof is completed in a similar way by the following list of commands that nicely reflects the perfectly regular structure of this part of the proof.

```
Intro I2. Elim I2. Assumption.
Intro I3. Elim I3. Assumption.
Intro I4. Elim I4. Assumption.
Intro I5. Elim I5. Assumption.
Intro I6. Elim I6. Assumption.
Intro I7. Elim I7. Assumption.
Intro I8. Elim I8. Assumption.
Intro . Assumption.
Save HLemma.
```

The commands in the last but one line solve the goal `False -> False`. This goal could be solved equally well by the command `Exact [p:False] p.`.

The lemma just proved is used a number of times in the proof of the correctness of the ABP. It would be very convenient if this and other basic lemmas, such as `neqr1r2` from the previous subsection, could be applied automatically. This can be achieved with the following command, which adds these lemmas to the so-called hint list:

```
Hint neqr1r2 HLemma.
```

After this command, commands like `Auto` and `Trivial` are able to apply such lemmas and can sometimes finish a proof automatically, according to some fixed strategy.

In order to give the proof of $r_1 \notin H$ we assume that all true negated equations such as `neqr1r2` are added to the hint list. Now the proof can be given in the following satisfying way:

```
Goal ~(In_ehlist r1 H).
Auto.
Save Inr1H.
```

4.3 EXP3

Having done the most important parts of the ground work for the actions, we can proceed by proving useful results on processes.

In this subsection we show how axioms like CM1 from Table 2 are generalised to merges of three and more processes. First we must declare the types of the operators listed in Subsection 3.1, using the command `Parameter`. Thereafter we list the axioms A1-A7 in their Coq formulation. Note that we have reversed some equations, as this is more convenient when rewriting with the `Elim` command.

```
Parameter proc :Set.

Parameter alt  :proc->proc->proc.
Parameter seq  :proc->proc->proc.
Parameter mer  :proc->proc->proc.
Parameter Lmer :proc->proc->proc.
Parameter comm :proc->proc->proc.
Parameter sum  :(D:Set)(D->proc)->proc.
Parameter enc  :ehlist ->proc->proc.
Parameter hide :ehlist->proc->proc.

Section BPA.
Variable x,y,z:proc.
Axiom A1. Assumes <proc>(alt x y)=(alt y x).
Axiom A2. Assumes <proc>(alt x (alt y z))=(alt (alt x y) z).
Axiom A3. Assumes <proc>x=(alt x x).
Axiom A4. Assumes <proc>(alt (seq x z) (seq y z))=(seq (alt x y) z).
Axiom A5. Assumes <proc>(seq x (seq y z))=(seq (seq x y) z).
Axiom A6. Assumes <proc>x=(alt x Delta).
Axiom A7. Assumes <proc>Delta=(seq Delta x).
End BPA.
```

Entering A6 and A7 without having declared `Delta` results in an error. Before declaring `Delta` we must explain how we represent parameterised actions and processes.

There exist processes with and without data parameters. As this is unsystematic and prevents using polymorphism in an elegant way, we decided to give all processes one data argument. If the process originally had no arguments, then now it has the canonical element `i` of a fixed one-element type `one` as argument. If the process originally had more than one argument, then now it has a tuple from a cartesian product as argument.

A second point is that it is desirable to distinguish between the *name* of a process and its *argument*. For example, in $r_1(d)$ the name of the process is r_1 and its argument is d. (Process names are called *labels* in μCRL. In semi-formal process theory, processes and their names are identified (or confused), so e.g. a stands both for the name of the atomic action a and for the action itself.) This is modelled by using a (polymorphic) operator `ia`, mapping a data type, a process name and a argument of the above data type into a process. Now one can understand the following encodings.

```
Parameter ia : (E:Set)act -> E -> proc.

Inductive Set one = i:one.

Definition Delta = (ia one delta i).
```

We list the axioms CM1-CM9, SC1-SC5 and Handshaking in their Coq formulation.

```
Section PARALLEL_OPERATORS.
Variable x,y,z:proc.
Variable E,F:Set.
Variable e:E.
Variable f:F.
Variable a,b  :act.
Axiom CM1. Assumes <proc>(alt (alt (Lmer x y) (Lmer y x)) (comm x y))=(mer x y).
Axiom CM2. Assumes <proc>(seq (ia E a e) x)=(Lmer (ia E a e) x).
Axiom CM3. Assumes <proc>(seq (ia E a e) (mer x y))=(Lmer (seq (ia E a e) x) y).
Axiom CM4. Assumes <proc>(alt (Lmer x z) (Lmer y z))=(Lmer (alt x y) z).
Axiom CM5. Assumes <proc>(seq (comm (ia E a e) (ia F b f)) x)=
                         (comm (seq (ia E a e) x) (ia F b f)).
Axiom CM6. Assumes <proc>(seq (comm (ia E a e) (ia F b f)) x)=
                         (comm (ia E a e) (seq (ia F b f) x)).
Axiom CM7. Assumes <proc>(seq (comm (ia E a e) (ia F b f)) (mer x y))=
                         (comm (seq (ia E a e) x) (seq (ia F b f) y)).
Axiom CM8. Assumes <proc>(alt (comm x z) (comm y z))=(comm (alt x y) z).
Axiom CM9. Assumes <proc>(alt (comm x y) (comm x z))=(comm x (alt y z)).
End PARALLEL_OPERATORS.

Section STANDARD_CONCURRENCY.
Variable x,y,z:proc.
Axiom SC1. Assumes <proc>(Lmer x (mer y z))=(Lmer (Lmer x y) z).
Axiom SC3. Assumes <proc>(comm y x)=(comm x y).
Axiom SC4. Assumes <proc>(comm x (comm y z))=(comm (comm x y) z).
Axiom SC5. Assumes <proc>(Lmer (comm x y) z)=(comm x (Lmer y z)).
Axiom Handshaking. Assumes <proc>Delta=(comm x (comm y z)).
End  STANDARD_CONCURRENCY.
```

The lemma we want to prove is the following.

```
Goal (x,y,z:proc)
<proc>(alt (Lmer x (mer y z))
       (alt (Lmer y (mer x z))
        (alt (Lmer z (mer x y))
         (alt (Lmer (comm y z) x)
          (alt (Lmer (comm x y) z)
              (Lmer (comm x z) y))))))
      =(mer x (mer y z)).
```

We give the first two commands and the remaining goal.

```
Intros; Elim CM1.

1 subgoal
```

```
  <proc
  >(alt (Lmer x (mer y z))
      (alt (Lmer y (mer x z))
         (alt (Lmer z (mer x y))
            (alt (Lmer (comm y z) x)
               (alt (Lmer (comm x y) z) (Lmer (comm x z) y))))))
  =(alt (alt (Lmer x (mer y z)) (Lmer (mer y z) x)) (comm x (mer y z)))
  ============================
    z : proc
    y : proc
    x : proc
```

Note that the first summand at the right hand side is already correct. To proceed we want to apply CM1 to the third and fourth occurrence of `(mer y z)` in the equation. This is done by the first line of commands from the following batch. By two applications of CM9 and of CM4 we distribute $|$ and $⌊⌊$ over $+$. Thereafter the new situation is printed.

```
Pattern 3 4 (mer y z); Elim CM1.
Elim CM9; Elim CM9.
Elim CM4; Elim CM4.

1 subgoal
  <proc
  >(alt (Lmer x (mer y z))
      (alt (Lmer y (mer x z))
         (alt (Lmer z (mer x y))
            (alt (Lmer (comm y z) x)
               (alt (Lmer (comm x y) z) (Lmer (comm x z) y))))))
  =(alt
      (alt (Lmer x (mer y z))
         (alt (alt (Lmer (Lmer y z) x) (Lmer (Lmer z y) x))
            (Lmer (comm y z) x)))
      (alt (alt (comm x (Lmer y z)) (comm x (Lmer z y)))
         (comm x (comm y z))))

  ============================
    z : proc
    y : proc
    x : proc
```

The proof continues with applications of SC1 and SC5. Thereafter the proof is finished by application of the axioms $x\,|\,(y\,|\,z) = \delta$ (Handshaking), $x + \delta = x$ (A6) and the associativity of $+$ (A2).

```
Elim SC1; Elim SC1.
Elim SC5; Elim SC5.
Elim Handshaking.
Elim A6.
Elim A2; Elim A2; Elim A2.
```

The situation is now as given below, the only thing left to prove being `<proc>(mer x z) = (mer z x)` and `<proc>(mer x y) = (mer y x)`.

```
1 subgoal
```

```
  <proc
  >(alt (Lmer x (mer y z))
      (alt (Lmer y (mer x z))
         (alt (Lmer z (mer x y))
            (alt (Lmer (comm y z) x)
               (alt (Lmer (comm x y) z) (Lmer (comm x z) y))))))
  =(alt (Lmer x (mer y z))
      (alt (Lmer y (mer z x))
         (alt (Lmer z (mer y x))
            (alt (Lmer (comm y z) x)
               (alt (Lmer (comm x y) z) (Lmer (comm x z) y))))))

  ============================
    z : proc
    y : proc
    x : proc
```

This obviously requires a lemma: `(x,y:proc)<proc>(mer x y) = (mer y x)`. In a bottom-up style proof this lemma should be available. We continue the proof in top-down style, as an example of the Coq command `Cut`. Thereafter the new situation is printed by the command `Show`.

```
Cut (x,y:proc)<proc>(mer x y) = (mer y x).
Show.
2 subgoals
  ((x:proc)(y:proc)(<proc>(mer x y)=(mer y x)))->
   (<proc
    >(alt (Lmer x (mer y z))
        (alt (Lmer y (mer x z))
           (alt (Lmer z (mer x y))
              (alt (Lmer (comm y z) x)
                 (alt (Lmer (comm x y) z) (Lmer (comm x z) y))))))
    =(alt (Lmer x (mer y z))
        (alt (Lmer y (mer z x))
           (alt (Lmer z (mer y x))
              (alt (Lmer (comm y z) x)
                 (alt (Lmer (comm x y) z) (Lmer (comm x z) y))))))
    )
  ============================
    z : proc
    y : proc
    x : proc
subgoal 2 is:
  (x0:proc)(y0:proc)(<proc>(mer x0 y0)=(mer y0 x0))
```

What has happened is that the first subgoal is our previous goal weakened with the lemma as a premiss, and the second subgoal is the lemma itself. The first subgoal is solved by `Intro`ducing the lemma and using it two times. Thereafter the second subgoal is easily solved using CM1, SC3 and A1.

```
Intro H.
Elim (H x z).
Elim (H x y).
Trivial.
```

```
Intros.
Elim CM1; Elim CM1.
Elim SC3.
Elim (A1 (Lmer x0 y0) (Lmer y0 x0)).
Trivial.

Save EXP3.
```

4.4 $K \mid L = \delta$

In this subsection we prove that the channels K and L cannot communicate. This is part of the proof of *CommKL* from Lemma 3.4. First we translate the specifications of the channels from Table 1 to Coq.

```
Parameter K:one->proc.
Parameter L:one->proc.
Parameter frame : Set.
Parameter Frame : Set.
Parameter sce:frame.
Parameter lce:Frame.

Axiom ChanK. Assumes
   <proc>
    (sum Frame ([x:Frame]
     (seq (ia Frame r2 x)
      (seq
       (alt
        (seq (ia one int i) (ia Frame s3 x))
        (seq (ia one int i) (ia Frame s3 lce)))
       (K i)))))=(K i).

Axiom ChanL. Assumes
   <proc>
    (sum frame ([n:frame]
     (seq (ia frame r5 n)
      (seq
       (alt
        (seq (ia one int i) (ia frame s6 n))
        (seq (ia one int i) (ia frame s6 sce)))
       (L i)))))=(L i).
```

For reasons of space, not all axioms that are used in the proof are included in their Coq version in this paper. For example, the SUM axioms used below are translations of the corresponding axioms from Table 5. The axiom EXT given below does not correspond to any axiom of μCRL, but is natural given our modeling of processes with data parameters. Thereafter the goal is given, as well as the beginning of the proof.

```
Axiom EXT. Assumes (D:Set)(x,y:D->proc)((d:D)<proc>(x d)=(y d))-><D->proc>x=y.

Goal <proc>Delta=(comm (K i) (L i)).
```

```
Elim ChanK.
Elim SUM7.
```

The situation is now as follows.

```
1 subgoal
  <proc
  >Delta
  =sum Frame [d:Frame]
     (comm
        (seq (ia Frame r2 d)
           (seq
              (alt (seq (ia one int i) (ia Frame s3 d))
                 (seq (ia one int i) (ia Frame s3 lce)))
              (K i)))
        (L i))
```

We proceed by using a new command, `ElimType`. The effect of a command `ElimType <A>a1=a2` is that all `a2`'s in the goal are replaced by `a1`'s and that `<A>a1=a2` is added as a new subgoal. The command `ElimType` can be regarded as a special case of the sequence of commands `Cut, Intro` and `Elim`. We give the command and thereafter the new situation.

```
ElimType <Frame->proc>[d:Frame]Delta=
       [d:Frame]
     (comm
        (seq (ia Frame r2 d)
           (seq
              (alt (seq (ia one int i) (ia Frame s3 d))
                 (seq (ia one int i) (ia Frame s3 lce)))
              (K i)))
        (L i)).

2 subgoals
  <proc>Delta=sum Frame [d:Frame] Delta
subgoal 2 is:
  <Frame->proc
  >[d:Frame]Delta
  =[d:Frame]
    (comm
       (seq (ia Frame r2 d)
          (seq
             (alt (seq (ia one int i) (ia Frame s3 d))
                (seq (ia one int i) (ia Frame s3 lce)))
             (K i)))
       (L i))
```

The first subgoal is settled easily by the next two commands, after which we continue with the proof of the second subgoal. We use the command `Apply`, which applies the lemma or axiom whose name is mentioned as argument of the command. To some extent, `Apply` is able to instantiate the lemma or axiom so that its conclusion matches with the current goal. The fragment of the proof after `Apply EXT; Intro` until the next `Apply EXT; Intro` is very similar to the beginning of the proof.

```
Elim SUM1; Trivial.
```

```
Apply EXT; Intro.
Elim ChanL.
Elim SC3; Elim SUM7.
ElimType <frame->proc>[d:frame]Delta=
         [d0:frame]
     (comm
        (seq (ia frame r5 d0)
           (seq
              (alt (seq (ia one int i) (ia frame s6 d0))
                 (seq (ia one int i) (ia frame s6 sce)))
              (L i)))
        (seq (ia Frame r2 d)
           (seq
              (alt (seq (ia one int i) (ia Frame s3 d))
                 (seq (ia one int i) (ia Frame s3 lce)))
              (K i)))).
Elim SUM1; Trivial.
Apply EXT; Intro.
```

The situation is now as given below. It is clear that r_2 and r_5 cannot communicate. However, proving the goal involves `CM7`, the Coq translations of A7 and CF2$''$ from Table 2, as well as `notEQfF` postulating that `frame` and `Frame` are distinct data types. For the sake of completeness we finish this subsection with the last part of the proof.

```
1 subgoal
  <proc
  >Delta
  =(comm
      (seq (ia frame r5 d0)
         (seq
            (alt (seq (ia one int i) (ia frame s6 d0))
               (seq (ia one int i) (ia frame s6 sce)))
            (L i)))
      (seq (ia Frame r2 d)
         (seq
            (alt (seq (ia one int i) (ia Frame s3 d))
               (seq (ia one int i) (ia Frame s3 lce)))
            (K i))))

  ============================
    d0 : frame
    d : Frame

Elim CM7.
Elim CF2''.
Elim A7; Trivial.
Exact notEQfF.

Save commKL.
```

4.5 A communication which is not δ

In this subsection we prove $\partial((s_6(b)\cdot y\,|\,T(b,d)\cdot y')\,⌊\!⌊\, x) = c_6(b)\cdot\partial(y \parallel y' \parallel x)$. For reasons of space we are somewhat less detailed than in our previous proofs. We start with formulating $T(b,d)$ and the goal.

```
Axiom ProcTn_d. Assumes
   (b:bit)(d:D)
    <proc>
       (alt
        (seq
          (alt
            (ia frame r6 (tuple(toggle b)))
            (ia frame r6 sce))
          (Sn_d d b))
        (ia frame r6 (tuple b)))=(Tn_d d b).

Goal
   (x,y,y':proc)(b:bit)(d:D)
     <proc>
     (seq (ia frame c6 (tuple b))
        (enc H (mer y (mer y' x))))=
     (enc H
        (Lmer
           (comm (seq (ia frame s6 (tuple b)) y) (seq (Tn_d d b) y')) x)).
```

The proof starts in the obvious way and continues by pressing the communications as deep as possible in the terms, until they are on the level of the atomic actions. Thereafter we give the new situation.

```
Intros.
Elim (ProcTn_d b d).
Elim A4; Elim A4; Elim A4.
Elim CM9; Elim CM9.
Elim A5; Elim A5.
Elim CM7; Elim CM7; Elim CM7.

1 subgoal
  <proc
  >(seq (ia frame c6 (tuple b)) (enc H (mer y (mer y' x))))
  =(enc H
      (Lmer
         (alt
            (alt
               (seq
                  (comm (ia frame s6 (tuple b))
                     (ia frame r6 (tuple (toggle b))))
                  (mer y (seq (Sn_d d b) y')))
               (seq (comm (ia frame s6 (tuple b)) (ia frame r6 sce))
                  (mer y (seq (Sn_d d b) y'))))
            (seq (comm (ia frame s6 (tuple b)) (ia frame r6 (tuple b)))
               (mer y y')))
         x))
  ============================
    d : D
    b : bit
    y' : proc
    y : proc
    x : proc
```

We observe three communications. The first two fail (i.e. are δ) since the arguments of r_6, s_6 do not match. The third succeeds yielding the atom $c_6(b)$ according to the communication function, usually called γ (`gamma` in Coq). Thereafter the goal is simplified by eliminating the δ's using A7, A6 and A1. Finally CM3 is applied to eliminate the $\mathbin{\|\!\_}$.

```
Elim CF2'.
Elim CF2'.
Elim CF1.
Elim A7.
Elim A6; Elim A1; Elim A6.
Elim CM3.

3 subgoals
  <proc
  >(seq (ia frame c6 (tuple b)) (enc H (mer y (mer y' x))))
  =(enc H (seq (ia frame (gamma s6 r6) (tuple b)) (mer (mer y y') x)))
  ============================
    d : D
    b : bit
    y' : proc
    y : proc
    x : proc
subgoal 2 is:
  ~<frame>(tuple b)=sce
subgoal 3 is:
  ~<frame>(tuple b)=(tuple (toggle b))
```

The new situation is given above. With the first subgoal, one proceeds by reducing `(gamma s6 r6)` to `c6`. Then one uses the fact that `H` does not contain `c6` in order to press the encapsulation operator beyond `(ia frame c6 (tuple b))` by D5 and D1. Thereafter this subgoal can be easily settled. The second and third subgoal can also easily be done using the axioms on data types.

Acknowledgements

We gratefully acknowledge Christine Paulin-Mohring and Randy Pollack for their critical comments on our modeling of process algebra and for their help in experimenting with alternatives. Thanks also go to Jan Bergstra, Leen Helmink, Gertjan Kamsteeg, Jaco van de Pol, Alex Sellink and Jan Springintveld for their attention to this project and for their comments.

A Definition of data types for the ABP

In this appendix we provide the data types that have been used in the specification of the ABP: **Bool**, *bit*, *D*, *frame* and *Frame*. The boldface denotation of **Bool** stems from that fact that **Bool** is a standard data type of μCRL. In the file ABP.v the declarations of the data types can be found in the sections BOOL, BIT, DATA, frame and Frame, respectively.

Each μCRL specification has a basic type **Bool** containing at least the elements *true* and *false*. This data type is used in the conditional construct in μCRL. We have added some auxiliary functions for convenience. The differences with the μCRL definition are mostly notational. To mention one point: as the `Elim` command rewrites from right to left, we have directed our equations accordingly.

sort	**Bool**
func	$true, false : \mathbf{Bool}$
	$andb, orb : \mathbf{Bool} \to \mathbf{Bool} \to \mathbf{Bool}$
	$notb : \mathbf{Bool} \to \mathbf{Bool}$
var	$b : \mathbf{Bool}$
rew	$andb(true, b) = b$
	$andb(false, b) = false$
	$orb(true, b) = true$
	$orb(false, b) = b$
	$notb(true) = false$
	$notb(false) = true$

The following defines the data type *bit* with elements e_0 and e_1, which are added to the frames in the ABP.

sort	bit
func	$e_0, e_1 : bit$
	$eq_{bit} : bit \to bit \to \mathbf{Bool}$
	$toggle : bit \to bit$
var	$b : bit$
rew	$toggle(e_0) = e_1$
	$toggle(e_1) = e_0$
	$eq_{bit}(b, b) = true$
	$eq_{bit}(b, toggle(b)) = false$
	$eq_{bit}(toggle(b), b) = false$

The following definition describes the data type of elements to be transferred. We only describe a few properties necessary for the verification of the ABP. An equality function eq_D has been defined that is necessary to compare elements of data type D in the conditional operator. In order to have the rather desirable property that $eq_D(d, e) = true \leftrightarrow d = e$, we have introduced a selector function if_D and four axioms. This is formulated in Lemma A.1. The axioms are due to Jan Bergstra.

sort	D	
func	$d_0 : D$	
	$eq_D : D \to D \to \mathbf{Bool}$	
	$if_D : \mathbf{Bool} \to D \to D \to D$	
var	$d, e : D$	
rew	$if_D(true, d, e) = d$	*eqD5*
	$if_D(false, d, e) = e$	*eqD6*
	$eq_D(d, d) = true$	*eqD7*
	$if_D(eq_D(d, e), d, e) = e$	*eqD8*

Lemma A.1. *Let d, e be variables of sort D.*

- $d = e \to eq_D(d, e) = true.$ *eqD_elim*

- $eq_D(d,e) = true \to d = e.$ *eqD_intro*
- $eq_D(d,e) = false \to d \neq e.$ *eqD_intro'*

In the file ABP.v a number of axioms are given of the form *EQ__* saying that sorts D, *Frame*, *frame*, etc. are different. This is used in axiom *CF2''* in the file ABP.v.

Next we give the defining operators and equations for small frames (*frame*) containing only a bit, and large frames (*Frame*) containing both a bit and a data element. The functions *sce* and *lce* abbreviate respectively small checksum error and large checksum error. For readability we write *sce* and *lce* as $\perp$ and we omit *Tuple* and *tuple* in the main text.

sort	frame	
var	$b, b_1, b_2 : \mathit{bit}$	
	$d, e : \mathit{frame}$	
func	$\mathit{tuple} : \mathit{bit} \to \mathit{frame}$	
	$\mathit{sce} : \mathit{frame}$	
	$\mathit{eq_f} : \mathit{frame} \to \mathit{frame} \to \mathbf{Bool}$	
	$\mathit{if_f} : \mathbf{Bool} \to \mathit{frame} \to \mathit{frame} \to \mathit{frame}$	
rew	$\mathit{eq_f}(\mathit{sce}, \mathit{sce}) = true$	*eqf1*
	$\mathit{eq_f}(\mathit{sce}, \mathit{tuple}(b)) = false$	*eqf2*
	$\mathit{eq_f}(\mathit{tuple}(b), \mathit{sce}) = false$	*eqf3*
	$\mathit{eq_f}(\mathit{tuple}(b_1), \mathit{tuple}(b_2)) = \mathit{eq_{bit}}(b_1, b_2)$	*eqf4*
	$\mathit{if_f}(true, d, e) = d$	*eqf5*
	$\mathit{if_f}(false, d, e) = e$	*eqf6*
	$\mathit{eq_f}(d, d) = true$	*eqf7*
	$\mathit{if_f}(\mathit{eq_f}(d,e), d, e) = e$	*eqf8*

Lemma A.2. *Let d, e be variables of sort frame.*

- $d = e \to \mathit{eq_f}(d,e) = true.$ *eqf_elim*
- $\mathit{eq_f}(d,e) = true \to d = e.$ *eqf_intro*
- $\mathit{eq_f}(d,e) = false \to d \neq e.$ *eqf_intro'*

sort	Frame	
var	$b, b_1, b_2 : \mathit{bit}$	
	$d, d_1, d_2 : D$	
	$e, e' : \mathit{Frame}$	
func	$\mathit{Tuple} : \mathit{bit} \to D \to \mathit{Frame}$	
	$\mathit{lce} : \mathit{Frame}$	
	$\mathit{eq_F} : \mathit{Frame} \to \mathit{Frame} \to \mathbf{Bool}$	
	$\mathit{if_F} : \mathbf{Bool} \to \mathit{Frame} \to \mathit{Frame} \to \mathit{Frame}$	
rew	$\mathit{eq_F}(\mathit{lce}, \mathit{lce}) = true$	*eqF1*
	$\mathit{eq_F}(\mathit{lce}, \mathit{Tuple}(b,d)) = false$	*eqF2*
	$\mathit{eq_F}(\mathit{Tuple}(b,d), \mathit{lce}) = false$	*eqF3*
	$\mathit{eq_F}(\mathit{Tuple}(b_1,d_1), \mathit{Tuple}(b_2,d_2)) = \mathit{andb}(\mathit{eq_{bit}}(b_1,b_2), \mathit{eq_D}(d_1,d_2))$	*eqF4*
	$\mathit{if_F}(true, e', e) = e'$	*eqF5*
	$\mathit{if_F}(false, e', e) = e$	*eqF6*
	$\mathit{eq_F}(e, e) = true$	*eqF7*
	$\mathit{if_F}(\mathit{eq_F}(e,e'), e, e') = e'$	*eqF8*

Lemma A.3. *Let d, e be variables of sort Frame.*

- $d = e \rightarrow eq_F(d,e) = true.$ *eqF_elim*
- $eq_F(d,e) = true \rightarrow d = e.$ *eqF_intro*
- $eq_F(d,e) = false \rightarrow d \neq e.$ *eqF_intro'*

References

[1] J.C.M. Baeten, J.A. Bergstra, and J.W. Klop. Conditional axioms and α/β calculus in process algebra. In M. Wirsing, editor, *Formal Description of Programming Concepts – III, Proceedings of the* 3th *IFIP WG 2.2 working conference,* Ebberup 1986, pages 53–75, Amsterdam, 1987. North-Holland.

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

[3] H.P. Barendregt. Lambda calculi with types. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, pages 117–309. Oxford University Press, Oxford, 1992.

[4] K.A. Bartlett, R.A. Scantlebury, and P.T. Wilkinson. A note on reliable full–duplex transmission over half–duplex links. *Communications of the ACM*, 12:260–261, 1969.

[5] J.A. Bergstra and J.W. Klop. Process algebra: specification and verification in bisimulation semantics. In M. Hazewinkel, J.K. Lenstra, and L.G.L.T. Meertens, editors, *Mathematics and Computer Science II*, CWI Monograph 4, pages 61–94. North-Holland, Amsterdam, 1986.

[6] G. Birtwistle and P.A. Subrahmanyam, editors. *Current Trends in Hardware Verification and Automated Theorem Proving.* Springer-Verlag, 1989.

[7] N.G. de Bruijn. A survey of the project AUTOMATH. In J.R. Hindley and J.P. Seldin, editors, *Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 580–606. Academic Press, London, 1980.

[8] R. Cleaveland and P. Panangaden. Type theory and concurrency. *International Journal of Parallel Programming*, 17:153–206, 1988.

[9] R.L. Constable, S.F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Cremer, R.W. Harper, D.J. Howe, T.B. Knoblock, N.P. Mendler, P. Panangaden, J.T. Sasaki, and S.F. Smith. *Implementing Mathematics with the NuPrl Development System.* Prentice-Hall, inc., Englewood Cliffs, New Jersey, first edition, 1986.

[10] T. Coquand and G. Huet. The calculus of constructions. *Information and Control*, 76:95–120, 1988.

[11] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Paulin-Mohring, and B. Werner. The Coq proof assistant version 5.6 user's guide. Technical report, INRIA – Rocquencourt, 1991.

[12] U. Engberg, P. Grønning, and L. Lamport. Mechanical verification of concurrent systems with TLA. Technical report, Aarhus University, 1992.

[13] M.H. Gordon, R.M. Milner, and C. Wadsworth. *Edinburg LCF*, volume 78 of *Lecture Notes in Computer Science.* Springer-Verlag, Berlin, 1979.

[14] 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.

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

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

[17] Z. Luo, R. Pollack, and P. Taylor. How to use LEGO. Technical Report LFCS-TN-27, University of Edinburgh, Edinburgh, Scotland, October 1989.

[18] R. Milner. *A Calculus of Communicating Systems*, volume 92. Springer-Verlag, Berlin, 1980.

[19] C. Paulin-Mohring. Inductive definitions in the system Coq. Rules and properties. Technical report, ENS Lyon, 1992.

[20] M.P.A. Sellink. Verifying process algebra proofs in type theory. Technical report, Utrecht University, 1993.

[21] V. Stavridou, T.F. Melham, and R.T. Boute, editors. *Theorem Provers in Circuit Design.* North-Holland, 1992.