A Computer-Checked Verification of Milner's Scheduler

Henri Korver[1] and Jan Springintveld[2]

[1] CWI, The Netherlands, henri@cwi.nl.
[2] Utrecht University, The Netherlands, jans@phil.ruu.nl.

Abstract. We present an equational verification of Milner's scheduler, which we checked by computer. To our knowledge, this is the first time that the scheduler is proof-checked for a general number n of scheduled processes.

Addresses: The first author can be reached at CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands. The second author can be reached at Utrecht University, Dept. of Philosophy, P.O. Box 80126, 3508 TC Utrecht, The Netherlands.

Support: The work of the first author took place in the context of EC Basic Research Action 7166 CONCUR 2. The work of the second author is supported by the Netherlands Computer Science Research Foundation (SION) with financial support of the Netherlands Organisation for Scientific Research (NWO).

1 Introduction

The correctness of many protocols crucially depends on the characteristics of data; one can think of the use of natural numbers, modulo calculations, lists, etc. Illustrative examples of such protocols are Milner's Scheduler [16], the Bakery Protocol [9] and the Sliding Window Protocol [19, 20].

However, traditionally process theories do not concentrate on data. For instance, Milner's correctness proof of the scheduler [16] relies for a considerable part on meta-reasoning about data. The presence of informal meta-reasoning obstructs the computer-checked verification of correctness proofs for such protocols. Hence the need arises for a process theory which comprises a formal treatment of data types. μCRL (*micro* CRL) [11, 12, 13], which is process algebra [1] combined with data [5], is such a theory. In addition to the usual process algebra operations, μCRL contains two important constructs relating processes and data: the $(\_ \triangleleft \_ \triangleright \_)$-operator (*then if else*) and the Σ-operator for summation over data. Moreover processes and the corresponding axioms and rules are parametrised with data and an induction principle for data is added.

As a case study, we formalise the correctness proof of Milner's scheduler in the proof theory of μCRL. The result of this exercise is twofold. First, a bug was detected in Milner's proof, which led to a reformulation of his result: Milner's scheduler only works correct if at least two processes are scheduled. (Milner claims that his scheduler also works correctly if only one process is scheduled, however this is not true in his particular set-up. This may seem a small error, but still!) Second, a completely formal and computer-checked proof was obtained. As far as we know this is the first (computer-checked) verification of Milner's scheduler for *every* number $n \geq 2$ of scheduled processes. This is to be contrasted with existing verifications of Milner's scheduler for various *instances* of n by the so-called 'bisimulation tools' (see e.g. [6], where the scheduler is treated for 80 cyclers).

The actual proof checking is done using the system Coq (see [4]), a proof assistant based on type theory. This case study (consisting of giving a formal proof *and* checking it in Coq) is part of a series of such case studies. Protocols that have been verified in this way are the Alternating Bit Protocol [2], a Bounded Retransmission Protocol [10], both in the setting of ACP and μCRL, and the same Bounded Retransmission Protocol in the setting of I/O automata [14].

Among these exercises, the verification of Milner's scheduler stands out, because this protocol has quite a complicated interaction between processes and data. This is reflected in the correctness proof; most proofs in this paper consist of a combination of induction over data types, ordinary process algebra expansions and calculations with sums and conditionals. Hence these proofs are rather intricate; and initially some mistakes were made in the proof that were not easy to repair, all of which were detected while checking the proofs with Coq. This process lasted approximately three months. The complete proof development can be found in the file `Scheduler.v`, which can be obtained by contacting the authors. The size of this file is about 140 Kbyte. Of this, 20% is taken up by the proofs in section 5, which constitute the core of Milner's proof. Of the remaining 80%, roughly 30% consists of lemmas concerning the data types. The remaining 50% is divided equally over the other sections.

Coq is a proof-assistant based on the *formulas as types, proofs as terms* paradigm (see [7]). In this paradigm, a formula is translated as a type in a typed lambda calculus and proofs of this formula are translated as lambda-terms of the corresponding type. Coq is an assistant in the sense that the proof is built up step by step by the user, while the computer checks the correctness of each step. Small proof steps can be done automatically by Coq. The actual construction of the lambda-term (the proof) is hidden from the user: the user just enters commands which are close to expressions in traditional proofs. Therefore the reasoning in Coq is quite close to reasoning in 'every day' mathematics. The type theory underlying Coq is an extension of the Calculus of Constructions (see [3]) with Inductive Types (see [17]). The presence of inductive types enables the user to reason with induction over datatypes.

Rather than treating the details of the implementation in Coq, this paper concentrates on formalising specifications and proofs in such a way that their correctness can be verified automatically. The details about implementing process algebra in Coq are well-covered in [2, 18]. Even, not all the details of the μCRL proof itself are given in this paper. For example, in order to formalise Milner's proof of Theorem 5.2.2, we had to refine the renaming mechanism of

μCRL and add the so-called *alphabet axioms*. These subtleties are worked out in the full version of this paper [15].

The paper is organised as follows. In section 2, we present Milner's scheduler and specify it in μCRL. A revised correctness criterion (see above) for Milner's scheduler is formulated in section 3. In section 4, we formalise in μCRL the meta-syntax (the Π-notation) which is the basis of Milner's proof. In section 5, we prove Milner's scheduler correct in μCRL. The proof of Milner is followed as close as possible such that readers who are familiar with it, can concentrate fully on how the proof is made precise in μCRL. A summary of the proof system is given in appendix A. The datatypes that are used in the paper are specified in appendix B.

As a final remark we note that, although the results in this paper are all proof-checked, we do not claim that there are no misprints in this paper. Translating formulas from the Coq notation to the usual notation is still a *human* business.

2 Specifying Milner's scheduler

The scheduler as described by Milner [16] schedules n processes $P(i), 1 \leq i \leq n$, in succession modulo n, i.e. after process $P(n)$ process $P(1)$ is activated again. Furthermore, a process may never be reactivated before it has terminated. The process $P(i)$ consists of a request for task initiation $\overline{a}(i)$ followed by a (here unspecified) task $Task(i)$ of which termination is indicated by $\overline{b}(i)$.

The scheduler is built from n cyclers which are positioned in a ring as depicted in Figure 1. Cycler $A(1,n)$ takes care of process $P(1)$ and cycler $D(i,n), 2 \leq i \leq n$, takes care of process $P(i)$. The first cycler $A(1,n)$ plays a special role as it starts up the system. Cycler $A(i,n)$ initiates process $P(i)$ by performing an action $a(i)$, signaling that $Task(i)$ can start. Then, by performing an action $s(i)$, it informs the next cycler $D(i +_n 1, n)$ that it is $P(i +_n 1)$'s turn to be initiated. Next, it waits for termination of process $P(i)$, indicated by $b(i)$, and in parallel it waits for a signal $s(i -_n 1)$ indicating that it is again $P(i)$'s turn to be initiated.

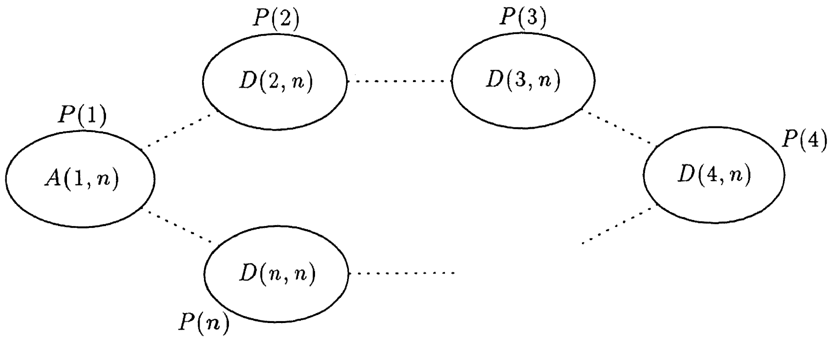

Fig. 1. The scheduler.

Finally, the cycler returns to its initial state. Cycler $D(i,n)$ first receives a signal indicating that it may start. Then it immediately evolves into the initial state of $A(i,n)$. The formal specification is as follows.

$$
\begin{array}{ll}
\textbf{act} & a, b, \overline{a}, \overline{b}, \hat{a}, \hat{b}, r, s : nat \\
\textbf{comm} & a \mid \overline{a} = \hat{a},\ \ b \mid \overline{b} = \hat{b} \\
\textbf{proc} & A(i : nat, n : nat) = a(i)s(i)(b(i) \parallel r(i -_n 1))A(i,n) \\
 & D(i : nat, n : nat) = r(i -_n 1)A(i,n) \\
 & P(i : nat) = \overline{a}(i)\,Task(i)\overline{b}(i)P(i) \\
 & Task(i : nat) = \ \ldots
\end{array}
$$

Here we take the existence of the data type *nat* (natural numbers) for granted; its specification can be found in appendix B. We also use modulo calculations, e.g. above we have introduced the operator $-_n$ which is subtraction modulo n. Below we shall also use the operator $+_n$ which is addition modulo n. The specification of $-_n$ and $+_n$ can be found in appendix B.

For convenience of reference the following processes are defined.

$$
\begin{array}{ll}
\textbf{proc} & B(i : nat, n : nat) = b(i)A(i,n) \\
 & E(i : nat, n : nat) = b(i)D(i,n) + r(i -_n 1)B(i,n) \\
 & C(i : nat, n : nat) = s(i)E(i,n)
\end{array}
$$

The whole system is obtained by putting the n cyclers in parallel.

$$
\begin{array}{ll}
\textbf{act} & c : nat \\
\textbf{comm} & r \mid s = c \\
\textbf{proc} & \Pi_2(m : nat, n : nat) = (\Pi_2(m-1,n) \parallel D(m,n)) \lhd m \geq 2 \rhd \delta \\
 & Sched(n : nat) = \tau_{\{c\}}(\partial_{\{r,s\}}(A(1,n) \parallel \Pi_2(n,n)))
\end{array}
$$

Our specification of the scheduler is completely given within the syntax of μCRL. This is in contrast with Milner's CCS specification:

$$Sched \stackrel{\mathrm{Def}}{=} (A_1 \mid D_2 \mid \ldots \mid D_n) \setminus \{c_1, \ldots, c_n\},$$

where the dots $(\ldots)$ and the variable n (which plays an important role) are informal notation.

3 A correctness criterion for the scheduler

The system of n cyclers as given above is called Milner's scheduler as the system is supposed to work as a scheduler. Below the notion of a scheduler, which is taken from [16], is specified in μCRL.

$$
\begin{array}{l}
\textbf{proc}\ Schedspec(i : nat, X : list, n : nat) = \\
\qquad\qquad \Sigma_{j:nat}(b(j)Schedspec(i, rem(j,X), n) \lhd test(j,X) \rhd \delta) + \\
\qquad\qquad \delta \lhd test(i,X) \rhd a(i)Schedspec(i +_n 1, in(i,X), n)
\end{array}
$$

The process $Schedspec(i, X, n)$ describes a scheduler in the state when any $P(j)$, $j \in X$, may terminate, and also $P(i)$ may be initiated provided that $i \notin X$.

In the specification above we use the function *in* for inserting an element in a list and the function *rem* for removing an element from a list. The function *test* checks whether or not a number is in the list. The specification of *in, rem* and *test* can be found in appendix B. Note that we used lists as parameters instead of sets because we found it easier to mechanise the reasoning with lists.

Now, we can formulate the correctness of Milner's scheduler as follows:

$$\boxed{\mathrm{n} \geq 2 \rightarrow Sched(n) = Schedspec(1, \varnothing, n)}$$

One can easily check that the restriction $n \geq 2$ is essential. However, Milner's correctness criterion does not refer to such a restriction, which unavoidably leads to the existence of an incorrect step in the corresponding proof.[3] And this is the only bug we found in Milner's proof; apart from this small oversight his verification is very accurate.

4 Formalising Milner's Π notation

In his proof Milner often uses the meta-notation $\Pi_{i \in X} P_i$ standing for the parallel composition of all processes P_i with $i \in X \subseteq \{1, \ldots, n\}$. In this notation one can rewrite the CCS-scheduler given in section 2 as

$$Sched = (A_1 | \Pi_{j \in \{2,\ldots,n\}} D_j) \setminus \{c_1, \ldots, c_n\}.$$

By using this notation many crucial steps in Milner's proof are lifted to meta-level. For instance the two following meta-identities (given in CCS notation)

1. $i \notin X \rightarrow (\Pi_{j \in X} D_j) | D_i = \Pi_{j \in X \cup \{i\}} D_j$
2. $i \in X \rightarrow \Pi_{j \in X} D_j = D_i | (\Pi_{j \in X - \{i\}} D_j)$

are often used in Milner's proof. Below we formalise Milner's Π-notation in μCRL and prove identities such as given above completely within the proof theory (see Lemma 4.1).

It is straightforward to simulate the set-theoretic operations which are used by Milner by operations on lists. Beside the functions already mentioned, we use the well-known functions 'empty' (*empty*), 'head' (*hd*) and 'tail' (*tl*). Now we define the processes Π_D and Π_E as follows.

$$\begin{aligned}
\textbf{proc}\quad & \Pi_D(X : list, n : nat) = \\
& \qquad \delta \lhd empty(X) \rhd (D(hd(X), n) \parallel \Pi_D(tl(X), n)) \\
& \Pi_E(X : list, n : nat) = \\
& \qquad \delta \lhd empty(X) \rhd (E(hd(X), n) \parallel \Pi_E(tl(X), n))
\end{aligned}$$

[3] In the first step in Subcase $i+1 \notin X$ (see [16], page 120) the identity $\Pi_{j \notin X \cup \{i\}} D_j = D_{i+1} | \Pi_{j \notin X \cup \{i,i+1\}} D_j$ (where $i, i+1, j \in \{1, \ldots n\}$, $X \subseteq \{1, \ldots n\}$) is used. However this identity is false in case $n < 2$.

The analogues of the meta-identities mentioned above are given in the following lemma. The same can be proved for Π_E instead of Π_D.

Lemma 4.1.

1. $\Pi_D(in(i,X),n) = D(i,n) \parallel \Pi_D(X,n)$
2. $test(i,X) \to \Pi_D(X,n) = D(i,n) \parallel \Pi_D(rem(i,X),n)$

Proof.

1. Immediate by definition.
2. This case is shown with induction on X. The induction follows $\varnothing$ and in.
   - $X = \varnothing : test(i,\varnothing) = F$ and the implication follows.
   - $X = in(j,Y)$:

$$
\begin{aligned}
& D(i,n) \parallel \Pi_D(rem(i,in(j,Y)),n) \\
\overset{\text{B.1.1}}{=}\quad & (D(i,n) \parallel \Pi_D(rem(i,in(j,Y)),n)) \\
& \lhd eq(i,j) \rhd (D(i,n) \parallel \Pi_D(rem(i,in(j,Y)),n)) \\
\overset{\text{B.4.3,B.4.4,B.2}}{=}\quad & (D(j,n) \parallel \Pi_D(Y,n)) \\
& \lhd eq(i,j) \rhd (D(i,n) \parallel \Pi_D(in(j,rem(i,Y)),n)) \\
\overset{4.1.1}{=}\quad & \Pi_D(in(j,Y),n) \\
& \lhd eq(i,j) \rhd (D(i,n) \parallel D(j,n) \parallel \Pi_D(rem(i,Y),n)) \\
\overset{\text{SC}}{=}\quad & \Pi_D(in(j,Y),n) \\
& \lhd eq(i,j) \rhd (D(j,n) \parallel D(i,n) \parallel \Pi_D(rem(i,Y),n)) \\
\overset{\text{B.4.6,I.H.}}{=}\quad & \Pi_D(in(j,Y),n) \lhd eq(i,j) \rhd (D(j,n) \parallel \Pi_D(Y,n)) \\
\overset{4.1.1}{=}\quad & \Pi_D(in(j,Y),n) \lhd eq(i,j) \rhd \Pi_D(in(j,Y),n) \\
\overset{\text{B.1.1}}{=}\quad & \Pi_D(in(j,Y),n)
\end{aligned}
$$

$\square$

As a further example of Milner's Π-notation, consider the expression $\Pi_{j \notin X} D_j$, which should be read as $\Pi_{j \in \{1,\ldots,n\} \setminus X} D_j$. We write this as $\Pi_D(X^n, n)$. Here, X^n means $fill(1,n) - X$, where $fill(1,n)$ is the list of natural numbers from 1 up to and including n. For technical convenience, lists are always 'filled' in decreasing order, e.g. $fill(1,4) = in(4, in(3, in(2, in(1,\varnothing))))$. $X - Y$ is the analogue of set difference and is defined using the function rem. The predicate $X \subseteq Y$ states that every number which occurs in X also occurs in Y.

Furthermore, we adopt the convention that we often omit the left hand side of boolean equations for easy notation, i.e. we may write $test(i,X)$ as a short hand for $test(i,X) = T$.

Some care has to be taken to ensure that the representation of sets by lists is well-defined. For instance, $\Pi_{j\in\{1,1\}}D_j = \Pi_{j\in\{1\}}D_j$ but $\Pi_D(in(1, in(1,\varnothing)), n) = D(1,n) \parallel D(1,n) \neq D(1,n) = \Pi_D(in(1,\varnothing), n)$. For ruling this out we only use lists where every element occurs at most once in X. The predicate $unique(X)$ states that X has this property. Another point is the identity $\Pi_{j\in\{1,2\}}D_j = \Pi_{j\in\{2,1\}}D_j$. To deal with this, we define the predicate $perm(X,Y)$ as $X \subseteq Y$ and $Y \subseteq X$. The following lemma shows how the constructions on lists are used for manipulating with the Π_D construct.

Lemma 4.2. *(Π-permutation).*

1. $\Pi_D(in(i, in(j,X)), n) = \Pi_D(in(j, in(i,X)), n)$
2. $unique(X) \wedge unique(Y) \wedge perm(X,Y) \rightarrow$
$$\Pi_D(X,n) = \Pi_D(Y,n)$$
3. $test(j,X) \wedge X \subseteq fill(1,n) \wedge unique(X) \rightarrow$
$$\Pi_D(rem(j,X)^n, n) = \Pi_D(in(j,X^n), n)$$

Proof.

1. By 4.1.1 and standard concurrency.
2. The key step in the proof is 4.2.1.
3. By 4.2.2 and the fact that $perm(rem(j,X)^n, in(j,X^n))$, $unique(rem(j,X)^n)$ and $unique(in(j,X^n))$.

$\square$

Lemma 4.2.2 states that lists behave like sets when they appear as parameter in Π. In the next lemma it is shown how we can expand the Π-construct to a summation. This is one of the key steps in the main proof.

Lemma 4.3. *(Π-Expansion).*

1. $unique(X) \rightarrow$
$$\Pi_D(X,n) = \Sigma_{j:nat}(r(j -_n 1)(A(j,n) \parallel \Pi_D(rem(j,X),n)) \triangleleft test(j,X) \triangleright \delta)$$
2. $unique(X) \rightarrow$
$$\begin{aligned}\Pi_E(X,n) = {} & \Sigma_{j:nat}(b(j)(D(j,n) \parallel \Pi_E(rem(j,X),n)) \triangleleft test(j,X) \triangleright \delta) + \\ & \Sigma_{j:nat}(r(j -_n 1)(B(j,n) \parallel \Pi_E(rem(j,X),n)) \triangleleft test(j,X) \triangleright \delta)\end{aligned}$$

Proof.

1. Without proving it here (see [15]), we claim that
(I) $\Sigma_{j:nat}(r(j -_n 1)(A(j,n) \parallel \Pi_D(rem(j, in(i,X)),n)) \triangleleft test(j, in(i,X)) \triangleright \delta)$
$= \Sigma_{j:nat}(r(j -_n 1)(A(j,n) \parallel \Pi_D(rem(j, in(i,X)),n)) \triangleleft test(j,X) \triangleright \delta)$
$+ \, r(i -_n 1)(A(i,n) \parallel \Pi_D(X,n))$.
We proceed the proof by induction on X. The basis step ($X = \varnothing$) is trivial. The induction step ($X = in(i,Y)$) goes as follows:

$$
\begin{aligned}
& \Pi_D(in(i,Y),n) \\
& \stackrel{4.1.1}{=} \quad D(i,n) \parallel \Pi_D(Y,n) \\
& \stackrel{\text{CM1}}{=} \quad \Pi_D(Y,n) \,⌊\!⌊\, D(i,n) + D(i,n) \,⌊\!⌊\, \Pi_D(Y,n) + D(i,n) \mid \Pi_D(Y,n) \\
& \stackrel{\text{I.H. (twice)}}{=} \Sigma_{j:nat}(r(j -_n 1)(A(j,n) \parallel \Pi_D(rem(j,Y),n)) \lhd test(j,Y) \rhd \delta) \\
& \qquad ⌊\!⌊\, D(i,n) \\
& + \quad D(i,n) \,⌊\!⌊\, \Pi_D(Y,n) \\
& + \quad \Sigma_{j:nat}(r(j -_n 1)(A(j,n) \parallel \Pi_D(rem(j,Y),n)) \lhd test(j,Y) \rhd \delta) \\
& \qquad \mid D(i,n) \\
& \stackrel{\text{A.3}}{=} \quad \Sigma_{j:nat}(r(j -_n 1)(A(j,n) \parallel \Pi_D(rem(j,Y),n) \parallel D(i,n)) \\
& \qquad \lhd test(j,Y) \rhd \delta) \\
& + \quad D(i,n) \,⌊\!⌊\, \Pi_D(Y,n) \\
& \stackrel{4.1.1}{=} \quad \Sigma_{j:nat}(r(j -_n 1)(A(j,n) \parallel \Pi_D(in(i,rem(j,Y)),n)) \\
& \qquad \lhd test(j,Y) \rhd \delta) \\
& + \quad D(i,n) \,⌊\!⌊\, \Pi_D(Y,n) \\
& \stackrel{\text{B.4.3}}{=} \quad \Sigma_{j:nat}(r(j -_n 1)(A(j,n) \parallel \Pi_D(rem(j,in(i,Y)),n)) \\
& \qquad \lhd test(j,Y) \rhd \delta) \\
& + \quad D(i,n) \,⌊\!⌊\, \Pi_D(Y,n)
\end{aligned}
$$

the application of B.4.3 hangs on $unique(in(i,Y)) \wedge test(j,Y) \rightarrow \neg eq(i,j)$

$$
\begin{aligned}
& \stackrel{\text{CM3}}{=} \quad \Sigma_{j:nat}(r(j -_n 1)(A(j,n) \parallel \Pi_D(rem(j,in(i,Y)),n)) \\
& \qquad \lhd test(j,Y) \rhd \delta) \\
& + \quad r(i -_n 1)(A(i,n) \parallel \Pi_D(Y,n)) \\
& \stackrel{(1)}{=} \quad \Sigma_{j:nat}(r(j -_n 1)(A(j,n) \parallel \Pi_D(rem(j,in(i,Y)),n)) \\
& \qquad \lhd test(j,in(i,Y)) \rhd \delta)
\end{aligned}
$$

2. Similar to (1).

□

5 The correctness proof

In this section we verify that Milner's scheduler indeed satisfies the criterion stated in section 3. This is proved as Theorem 5.2.5. The essential step in Milner's proof is the introduction of the process

$$\begin{array}{ll}\textbf{proc} & Sched(i:nat, X:list, n:nat) = \\ & \quad \tau_{\{c\}}(\partial_{\{r,s\}}(\\ & \qquad\qquad (B(i,n) \parallel \Pi_D(X^n,n) \parallel \Pi_E(rem(i,X),n)) \\ & \qquad \lhd test(i,X) \rhd \\ & \qquad\qquad (A(i,n) \parallel \Pi_D(in(i,X)^n,n) \parallel \Pi_E(X,n))))\end{array}$$

which forms the bridge between the processes $Sched(n)$ and $Schedspec(i, X, n)$. We follow Milner's proof very closely. First we show that $Sched(i, X, n)$ satisfies the (guarded) defining equation of $Schedspec(i, X, n)$. This is done by distinguishing two cases: the case where X contains number i and the case where X does not. Then by using RSP we have $Sched(i, X, n) = Schedspec(i, X, n)$. Finally, a simple calculation shows that $Sched(n)$ is an instance of $Sched(i, X, n)$, i.e. $Sched(n) = Schedspec(1, \varnothing, n)$, and we are done. All these calculations can be found in Theorem 5.2, the main proof.

Before embarking on the main proof we need to verify that we can simulate the process Π_2 from the specification by the Π_D and the *fill*-construct.

Lemma 5.1. $m \geq 2 \rightarrow \Pi_D(fill(2,m),n) = \Pi_2(m,n)$

Proof. Omitted. □

Now we have reached the point where we can prove the main theorem.

Theorem 5.2. *We write 'Cond' for* '$n \geq 2 \wedge i \geq 1 \wedge i \leq n \wedge X \subseteq fill(1,n) \wedge unique(X)$'.

1. $test(i,X) \wedge Cond \rightarrow$
   $Sched(i,X,n) = \Sigma_{j:nat}(b(j)Sched(i, rem(j,X),n) \lhd test(j,X) \rhd \delta)$
2. $\neg test(i,X) \wedge Cond \rightarrow$
   $Sched(i,X,n) = \Sigma_{j:nat}(b(j)Sched(i, rem(j,X),n) \lhd test(j,X) \rhd \delta)$
   $\qquad + a(i)Sched(i +_n 1, in(i,X),n)$
3. $Cond \rightarrow Sched(i,X,n) = Schedspec(i,X,n)$
4. $n \geq 2 \rightarrow Sched(n) = Sched(1,\varnothing,n)$
5. $n \geq 2 \rightarrow Sched(n) = Schedspec(1,\varnothing,n)$.

In (1) we may replace $n \geq 2$ in *Cond* by $n \geq 1$.

Proof.

1.
$$\begin{array}{ll} & Sched(i,X,n) \\ = & \tau_{\{c\}}(\partial_{\{r,s\}}(B(i,n) \parallel \Pi_D(X^n,n) \parallel \Pi_E(rem(i,X),n))) \\ = & b(i)\tau_{\{c\}}(\partial_{\{r,s\}}(A(i,n) \parallel \Pi_D(X^n,n) \parallel \Pi_E(rem(i,X),n))) \\ + & \Sigma_{j:nat}(b(j) \\ & \quad \tau_{\{c\}}(\partial_{\{r,s\}}(B(i,n) \parallel \Pi_D(X^n,n) \parallel D(j,n) \parallel \\ & \qquad \Pi_E(rem(j,rem(i,X)),n))) \lhd test(j,rem(i,X)) \rhd \delta)\end{array}$$

by expansion, using Π-Expansion (4.3) and Sum Expansion (A.3)

$$\begin{array}{ll}
\stackrel{B.4.5}{=} & b(i)Sched(i, rem(i,X), n) \\
+ & \Sigma_{j:nat}(b(j) \\
& \quad \tau_{\{c\}}(\partial_{\{r,s\}}(B(i,n) \parallel \Pi_D(X^n, n) \parallel D(j,n) \parallel \\
& \qquad\qquad \Pi_E(rem(j, rem(i,X)), n))) \\
& \triangleleft test(j, rem(i,X)) \triangleright \delta)
\end{array}$$

$$\begin{array}{ll}
\stackrel{4.1.1}{=} & b(i)Sched(i, rem(i,X), n) \\
+ & \Sigma_{j:nat}(b(j) \\
& \quad \tau_{\{c\}}(\partial_{\{r,s\}}(B(i,n) \parallel \Pi_D(in(j, X^n), n) \parallel \\
& \qquad\qquad \Pi_E(rem(j, rem(i,X)), n))) \\
& \triangleleft test(j, rem(i,X)) \triangleright \delta)
\end{array}$$

$$\begin{array}{ll}
\stackrel{4.2.3,B.4.7}{=} & b(i)Sched(i, rem(i,X), n) \\
+ & \Sigma_{j:nat}(b(j)Sched(i, rem(j,X), n) \triangleleft test(j, rem(i,X)) \triangleright \delta)
\end{array}$$

$$\begin{array}{ll}
\stackrel{A.4}{=} & \Sigma_{j:nat}(b(j)Sched(i, rem(j,X), n) \triangleleft test(j,X) \triangleright \delta)
\end{array}$$

2. The same idea as in (1) although a bit more complicated.
3. In 5.2.1 we have shown that $Sched(i, X, n)$ is a solution for the (guarded) defining equation of $Schedspec(i, X, n)$ when $test(i, X)$. In 5.2.2 we have shown that $Sched(i, X, n)$ is a solution of $Schedspec(i, X, n)$ when $\neg test(i, X)$. So by the excluded middle principle we know that $Sched(i, X, n)$ is a solution for $Schedspec(i, X, n)$. Then by RSP we may conclude that $Sched(i, X, n)$ and $Schedspec(i, X, n)$ are equal. Note that we each time assume that $Cond$ holds.
4. Without proving it here, we claim that

   (I) $A(i,n) \parallel \delta = A(i,n)$.

   We proceed as follows:

$$\begin{array}{ll}
\multicolumn{2}{l}{Sched(1, \varnothing, n)} \\
= & \tau_{\{c\}}(\partial_{\{r,s\}}(A(1,n) \parallel \Pi_D(in(1,\varnothing)^n, n) \parallel \Pi_E(\varnothing, n))) \\
\stackrel{B.4.2}{=} & \tau_{\{c\}}(\partial_{\{r,s\}}(A(1,n) \parallel \Pi_D(fill(2,n), n) \parallel \delta)) \\
\stackrel{5.1}{=} & \tau_{\{c\}}(\partial_{\{r,s\}}(A(1,n) \parallel \Pi_2(n,n) \parallel \delta)) \\
\stackrel{SC}{=} & \tau_{\{c\}}(\partial_{\{r,s\}}((A(1,n) \parallel \delta) \parallel \Pi_2(n,n))) \\
\stackrel{(I)}{=} & \tau_{\{c\}}(\partial_{\{r,s\}}(A(1,n) \parallel \Pi_2(n,n))) \\
= & Sched(n).
\end{array}$$

5. $Sched(n) \stackrel{5.2.4}{=} Sched(1, \varnothing, n) \stackrel{5.2.3}{=} Schedspec(1, \varnothing, n)$.

$\square$

6 Concluding remarks

The experiment can be considered successful: we have brought down Milner's proof to a completely formal level and checked it by computer. Yet we also have to admit that formalising and checking Milner's proof was not a bed of roses.

First, identities that are simple at meta-level are not easy to prove in a formalised setting, e.g. the Π-Expansion lemma. Generally speaking, the identities that were most difficult to prove were those that involve processes which heavily interact with data.

Second, we had to write out and check a large amount of small proof steps. This is not only hard work, but, again, identities that are trivial at meta-level (and therefore mostly omitted) can sometimes be quite tedious at formal level.

Although the verification was not an easy task, we are confident that by doing more of such protocol verifications we obtain more skill and experience in doing calculations such as given in the paper. Moreover, we believe that proof-checkers can be improved in generating more proof steps by themselves, e.g. by using more advanced *tactics.* This will lead to a situation where proof-checked verification of distributed systems becomes feasible.

7 Acknowledgements

We are very grateful to Jos Baeten, Jan Friso Groote, Tonny Hurkens, Alban Ponse and Freek Wiedijk for their comments on previous versions of this paper. Furthermore, we are indebted to Willem Jan Fokkink for teaching us modulo arithmetic. At last, we are very thankful to Marc Bezem, Jan Friso Groote (again), Jaco van de Pol and Alex Sellink for supporting us with the Coq system.

A An overview of the proof theory for μCRL

A.1 The proof system

In [12] a proof system has been given which allows to prove identities about processes with data. Table 1 lists the axioms of ACP in μCRL, followed by the axioms for hiding TI, standard concurrency SC and branching bisimulation B. For an explanation of the axioms we refer to [12], except for the following points. We distinguish between *actions* (e.g. $r(i)$ is an action) and *gates*, which are 'incomplete' actions (e.g. r is a gate). The function *label* extracts the gate from an action. The communication axioms, denoted by CF, make use of the function γ. It is defined as follows: $\gamma(a,b) = c$ if $label(a) \mid label(b) = label(c)$ is declared in **comm** and otherwise $\gamma(a,b)$ is undefined.

Table 2 lists the typical μCRL axioms and rules for interaction between data and processes. The axioms for summation are denoted by SUM, the axioms for the conditional by COND and the rules for the booleans by BOOL.

Beside the axioms and rules mentioned above, μCRL incorporates two other important proof principles. First, it supports an principle for induction not only on data but also on data in processes. The second principle is RSP (Recursive Specification Principle) taken from [1] extended to processes with data. Informally, it says that each guarded recursive specification has at most one solution.

A1 $x + y = y + x$ A2 $x + (y + z) = (x + y) + z$ A3 $x + x = x$ A4 $(x + y) \cdot z = x \cdot z + y \cdot z$ A5 $(x \cdot y) \cdot z = x \cdot (y \cdot z)$ A6 $x + \delta = x$ A7 $\delta \cdot x = \delta$ CM1 $x \parallel y = x \mathbin{\|\!\!\lfloor} y + y \mathbin{\|\!\!\lfloor} x + x \mid y$ CM2 $a \mathbin{\|\!\!\lfloor} x = a \cdot x$ CM3 $a \cdot x \mathbin{\|\!\!\lfloor} y = a \cdot (x \parallel y)$ 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 $a \mid b \cdot x = (a \mid b) \cdot x$ CM7 $a \cdot x \mid b \cdot y = (a \mid b) \cdot (x \parallel y)$ CM8 $(x + y) \mid z = x \mid z + y \mid z$ CM9 $x \mid (y + z) = x \mid y + x \mid z$	CF1 $n_1 \mid n_2 = n_3$ if $\gamma(n_1, n_2) = n_3$ CF1′ $n_1(t_1, \ldots, t_m) \mid n_2(t_1, \ldots, t_m) =$ $n_3(t_1, \ldots, t_m)$ if $\gamma(n_1, n_2) = n_3$ CF2 $a \mid b = \delta$ if $\gamma(\mathit{label}(a), \mathit{label}(b))$ is undefined 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 $1 \leq i \leq m$ CF2″ $n_1(t_1, \ldots, t_m) \mid n_2(t_1', \ldots, t_{m'}') = \delta$ if $m \neq m'$ D1 $\partial_H(a) = a$ if $\mathit{label}(a) \notin H$ D2 $\partial_H(a) = \delta$ if $\mathit{label}(a) \in H$ D3 $\partial_H(x + y) = \partial_H(x) + \partial_H(y)$ D4 $\partial_H(x \cdot y) = \partial_H(x) \cdot \partial_H(y)$
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 $\tau_I(x \cdot y) = \tau_I(x) \cdot \tau_I(y)$
SC1 $(x \mathbin{\|\!\!\lfloor} y) \mathbin{\|\!\!\lfloor} z = x \mathbin{\|\!\!\lfloor} (y \parallel z)$ SC6 $x \parallel \delta = x\delta$ SC3 $x \mid y = y \mid x$ SC7 $x \mid \delta = \delta$	SC4 $(x \mid y) \mid z = x \mid (y \mid z)$ SC5 $x \mid (y \mathbin{\|\!\!\lfloor} z) = (x \mid y) \mathbin{\|\!\!\lfloor} z$ SC8 $x \mid (y \mid z) = \delta$
B1 $x\tau = x$	B2 $z(\tau(x + y) + x) = z(x + y)$

Table 1. ACP-like axioms and rules in μCRL.

SUM1	$\Sigma_{d:D}(p) = p$	if d not free in p
SUM2	$\Sigma_{d:D}(p) = \Sigma_{e:D}(p[e/d])$	if e not free in p
SUM3	$\Sigma_{d:D}(p) = \Sigma_{d:D}(p) + p$	
SUM4	$\Sigma_{d:D}(p_1 + p_2) = \Sigma_{d:D}(p_1) + \Sigma_{d:D}(p_2)$	
SUM5	$\Sigma_{d:D}(p_1 \cdot p_2) = \Sigma_{d:D}(p_1) \cdot p_2$	if d not free in p_2
SUM6	$\Sigma_{d:D}(p_1 \mathbin{\lfloor\!\lfloor} p_2) = \Sigma_{d:D}(p_1) \mathbin{\lfloor\!\lfloor} p_2$	if d not free in p_2
SUM7	$\Sigma_{d:D}(p_1 \mid p_2) = \Sigma_{d:D}(p_1) \mid p_2$	if d not free in p_2
SUM8	$\Sigma_{d:D}(\partial_H(p)) = \partial_H(\Sigma_{d:D}(p))$	
SUM9	$\Sigma_{d:D}(\tau_I(p)) = \tau_I(\Sigma_{d:D}(p))$	
SUM11	$\dfrac{\begin{array}{c}\mathcal{D}\\ p_1 = p_2\end{array}}{\Sigma_{d:D}(p_1) = \Sigma_{d:D}(p_2)}$	provided d not free in the assumptions of $\mathcal{D}$
COND1	$x \triangleleft T \triangleright y = x$	
COND2	$x \triangleleft F \triangleright y = y$	
BOOL1	$\neg(T = F)$	
BOOL2	$\neg(b = T) \rightarrow b = F$	

Table 2. Axioms for summation and conditionals.

A.2 Basic lemmas for μCRL

In this section, we present a number of elementary lemmas (see [9]) which are derived from the proof system given above. These lemmas are used in the verification of the scheduler, but are also interesting in their own right as it is very likely that they are needed in every μCRL verification. The first lemma shows that for applying an induction on a boolean variable b (see appendix B), one only has to check the cases $b = T$ and $b = F$.

Lemma A.1. *(Specialised induction rule for* **Bool***).*

$$(p = q)[T/b] \wedge (p = q)[F/b] \rightarrow p = q.$$

The following lemma presents a rule which is derived from the SUM axioms. This rule appears to be a powerful tool to eliminate sum expressions in μCRL calculations.

Lemma A.2. *(Sum Elimination). Let D be a given sort that is equipped with an equality function $eq : D \times D \rightarrow$* **Bool** *with the obvious property $eq(d, d) = T$. Then, we have*

$$\Sigma_{d:D}(p \triangleleft eq(d,t) \triangleright \delta) = p[t/d].$$

The next lemma is used for expanding sums in parallel compositions.

Lemma A.3. *(Sum Expansion). If the variable $d : D$ does not occur free in term q, then we have*

1. $\Sigma_{d:D}(a \cdot p \lhd c \rhd \delta) \mathbin{\lfloor\!\lfloor} q = \Sigma_{d:D}(a \cdot (p \parallel q) \lhd c \rhd \delta)$.
2. $\Sigma_{d:D}(a(d) \cdot p \lhd c \rhd \delta) \mid b(e) \cdot q = \Sigma_{d:D}((a(d) \mid b(e)) \cdot (p \parallel q) \lhd c \rhd \delta)$

The last proposition is used in Theorem 5.2.1.

Proposition A.4. *Let p be a process.*

$$test(i, X) \rightarrow$$
$$b(i)p + \Sigma_{j:nat}(b(j)p \lhd test(j, rem(i, X)) \rhd \delta) = \Sigma_{j:nat}(b(j)p \lhd test(j, X) \rhd \delta)$$

Proof.

$$\begin{aligned}
&\Sigma_{j:nat}(b(j)p \lhd test(j, X) \rhd \delta)\\
&\overset{\text{B.4.1}}{=} \quad \Sigma_{j:nat}(b(j)p \lhd eq(j, i) \text{ or } test(j, rem(i, X)) \rhd \delta)\\
&\overset{\text{B.1.2,SUM4}}{=} \quad \Sigma_{j:nat}(b(j)p \lhd eq(j, i) \rhd \delta)\\
&\quad + \quad \Sigma_{j:nat}(b(j)p \lhd test(j, rem(i, X)) \rhd \delta)\\
&\overset{\text{A.2}}{=} \quad b(i)p + \Sigma_{j:nat}(b(j)p \lhd test(j, rem(i, X)) \rhd \delta)
\end{aligned}$$

□

B Elementary data types

Below, we present the data identities we needed in the scheduler verification. Although all these results have been proof-checked we do not present the proofs here, since they are standard.

B.1 About booleans

sort	**Bool**
func	$T, F :\rightarrow$ **Bool**
	not : **Bool** $\rightarrow$ **Bool**
	and : **Bool** $\times$ **Bool** $\rightarrow$ **Bool**
	or : **Bool** $\times$ **Bool** $\rightarrow$ **Bool**
var	b, b_1, b_2 : **Bool**
rew	$\text{not}(T) = F$
	$\text{not}(F) = T$
	$T \text{ and } b = b$
	$F \text{ and } b = F$
	$T \text{ or } b = T$
	$F \text{ or } b = b$

Lemma B.1.

1. $x \lhd b \rhd x = x$,
2. $x \lhd b_1$ or $b_2 \rhd \delta = x \lhd b_1 \rhd \delta + x \lhd b_2 \rhd \delta$.

Proof. Easy via Lemma A.1. $\square$

B.2 About natural numbers

$$
\begin{array}{ll}
\textbf{sort} & nat \\
\textbf{func} & 0 :\rightarrow nat \\
 & S, P : nat \rightarrow nat \\
 & +, -, : nat \times nat \rightarrow nat \\
 & eq, \geq, \leq, <, >: nat \times nat \rightarrow \textbf{Bool} \\
 & if : \textbf{Bool} \times nat \times nat \rightarrow nat \\
\textbf{var} & n, m, z : nat \\
\textbf{rew} & P(0) = 0 \\
 & P(S(n)) = n \\
 & n + 0 = n \\
 & n + S(m) = S(n + m) \\
 & n - 0 = n \\
 & n - S(m) = P(n - m) \\
 & eq(0, 0) = T \\
 & eq(0, S(n)) = F \\
 & eq(S(n), 0) = F \\
 & eq(S(n), S(m)) = eq(n, m) \\
 & n \geq 0 = T \\
 & 0 \geq S(n) = F \\
 & S(n) \geq S(m) = n \geq m \\
 & n \leq m = m \geq n \\
 & n > m = n \geq S(m) \\
 & n < m = S(n) \leq m \\
 & if(T, n, m) = n \\
 & if(F, n, m) = m
\end{array}
$$

We write $n \leq m$ for $n \leq m = T$. Idem for $\geq$, $>$ and $<$. We write $eq(n, m)$ for $eq(n, m) = T$. We write 1 for $S(0)$ and 2 for $S(S(0))$. We write $i - 1$ for $P(i)$ and $i - 2$ for $P(P(i))$. We write $n \leq m$ for $m \geq n$ and $n > m$ for $n \geq S(m)$ and $n < m$ for $S(n) \leq m$.

Lemma B.2. $eq(n, m) = T \leftrightarrow n = m$

B.3 About modulo arithmetic

The following definition is due to Willem Jan Fokkink.

func $\mathrm{mod} : nat \times nat \to nat$
$+ : nat \times nat \times nat \to nat$
var $i, j, n : nat$
rew $i \bmod 0 = i$
$i \bmod n = if(eq(i,0), n, if(i > n, (i-n) \bmod n, i))$
$i +_n j = (i+j) \bmod n$
$i -_n j = (i-j) \bmod n$

Note that we defined a slightly non-standard modulo function to follow Milner's proof as close as possible. In particular, we need our functions to have values in the positive natural numbers. The usual definition of the modulo function yields for instance $2 \bmod 2 = 0$, but our (and Milner's) definition yields $2 \bmod 2 = 2$.

Lemma B.3.

1. $i \bmod 1 = 1$
2. $n \geq 2 \wedge i \leq n \wedge i \geq 1 \to (i +_n 1) -_n 1 = i$
3. $n \geq 2 \wedge i \leq n \wedge i \geq 1 \to (i -_n 1) +_n 1 = i$

B.4 About lists of naturals

sort $list$
func $\varnothing : \to list$
$in, rem, {}^n : nat \times list \to list$
$test : nat \times list \to \mathbf{Bool}$
$hd : list \to nat$
$tl : list \to list$
$if : \mathbf{Bool} \times list \times list \to list$
$empty, unique : list \to \mathbf{Bool}$
$fill : nat \times nat \to list$
$- : list \times list \to list$
$\subseteq, perm : list \times list \to \mathbf{Bool}$
var $i, j, k, n, m : nat$
$X, Y : list$
rew $test(j, \varnothing) = F$
$test(j, in(k, X)) = if(eq(j,k), T, test(j, X))$
$rem(j, \varnothing) = \varnothing$
$rem(j, in(k, X)) = if(eq(j,k), X, in(k, rem(j, X)))$
$hd(\varnothing) = 0$
$hd(in(j, X)) = j$
$tl(\varnothing) = \varnothing$
$tl(in(j, X)) = X$
$empty(\varnothing) = T$
$empty(in(j, X)) = F$
$X - \varnothing = X$
$X - in(j, Y) = rem(j, X - Y)$

$$\varnothing \subseteq X = T$$
$$in(j, X) \subseteq Y = test(j, Y) \text{ and } X \subseteq Y$$
$$unique(\varnothing) = T$$
$$unique(in(j, X)) = if(test(j, X), F, unique(X))$$
$$perm(X, Y) = X \subseteq Y \text{ and } Y \subseteq X$$
$$fill(m, n) = if(n < m, \varnothing, if(eq(n, 0), in(0, \varnothing), in(n, fill(m, P(n)))))$$
$$X^n = fill(1, n) - X$$

Lemma B.4.

1. $test(i, X) \rightarrow (test(j, X) = eq(i, j) \text{ or } test(j, rem(i, X)))$,
2. $in(1, \varnothing)^n = fill(2, n)$,
3. $\neg eq(i, j) \rightarrow rem(j, in(i, Y)) = in(i, rem(j, Y))$,
4. $eq(i, j) \rightarrow rem(i, in(j, Y)) = Y$,
5. $test(i, X) \rightarrow in(i, rem(i, X))^n = X^n$,
6. $(test(i, X) \wedge X = in(j, Y) \wedge \neg eq(i, j)) \rightarrow test(i, Y)$.
7. $rem(i, rem(j, X)) = rem(j, rem(i, X))$,

References

1. J.C.M. Baeten and W.P. Weijland. *Process Algebra*. Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.
2. M. Bezem 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.
3. T. Coquand and G. Huet. The calculus of constructions. *Information and Control*, 76:95–120, 1988.
4. G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Murthy, C. Parent, C. Paulin-Mohring, and B. Werner. The Coq proof assistant user's guide. Version 5.8. Technical report, INRIA - Rocquencourt, May 1993.
5. H. Ehrig and B. Mahr. *Fundamentals of algebraic specifications I*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1985.
6. J.-C. Fernandez, A. Kerbrat and L. Mounier. Symbolic Equivalence Checking. In C. Courcoubetis, editor, *Proceedings of the* 5^{th} *International Conference*, CAV '93, Elounda, Greece, volume 697 of *Lecture Notes in Computer Science*, pages 85–97. Springer-Verlag, 1993.
7. J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge tracts in theoretical computer science*. Cambridge University Press, Cambridge, 1989.
8. R.J. van Glabbeek and W.P. Weijland. Branching time and abstraction in bisimulation semantics (extended abstract). In G.X. Ritter, editor, *Information Processing 89*, pages 613–618. North-Holland, 1989.
9. 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.
10. J.F. Groote and J.C. van de Pol. A bounded retransmission protocol for large data packets. A case study in computer checked verification. Technical Report 100, Logic Group Preprint Series, Utrecht University, October 1993.

11. J.F. Groote and A. Ponse. The syntax and semantics of μCRL. Technical Report CS-R9076, CWI, Amsterdam, 1990.
12. J.F. Groote and A. Ponse. Proof theory for μCRL. Technical Report CS-R9138, CWI, Amsterdam, 1991.
13. J.F. Groote and A. Ponse. μCRL: A base for analysing processes with data. In E. Best and G. Rozenberg, editors, *Proceedings* 3rd *Workshop on Concurrency and Compositionality, Goslar, GMD-Studien Nr. 191*, pages 125–130. Universität Hildesheim, 1991.
14. L. Helmink, M.P.A. Sellink, and F. Vaandrager. Proof-checking a data link protocol. 1993. To appear.
15. H. Korver and J. Springintveld. A Computer-Checked Verification of Milner's Scheduler. Technical Report Logic Group Preprint Series No. 101, Utrecht University, November, 1993. Full version.
16. R. Milner. *Communication and Concurrency.* Prentice-Hall International, Englewood Cliffs, 1989.
17. C. Paulin-Mohring. Inductive definitions in the system Coq. Rules and properties. In M. Bezem and J.F. Groote, editors, *Proceedings of the* 1st *International Conference on Typed Lambda Calculi and Applications, TLCA '93,* Utrecht, The Netherlands, volume 664 of *Lecture Notes in Computer Science*, pages 328–345. Springer-Verlag, 1993.
18. M.P.A. Sellink. Verifying process algebra proofs in type theory. Technical Report Logic Group Preprint Series No. 87, Utrecht University, 1993.
19. N.V. Stenning. A data transfer protocol. *Computer Networks*. 1:99–110, 1976.
20. A.S. Tanenbaum. *Computer networks.* Prentice-Hall International, Englewood Cliffs, 1989.