Formal Aspects of Computing (1997) 9: 1–48
© 1997 BCS

Formal Aspects of Computing

Formalizing Process Algebraic Verifications in the Calculus of Constructions

Marc Bezem[1], Roland Bol[2] and Jan Friso Groote[3]

[1]Dept. of Philosophy, Utrecht University, The Netherlands
[2]Dept. of Computing Science, Uppsala University, Uppsala, Sweden
[3]Dept. of Software Technology, CWI, Amsterdam, The Netherlands

Keywords: Formal verification; Process algebra; ACP; μCRL; Coq; Calculus of Constructions; Alternating Bit Protocol

Abstract. This paper reports on the first steps towards the formal verification of correctness proofs of real-life protocols in process algebra. We show that such proofs can be verified, and partly constructed, by a general purpose proof checker. The process algebra we use is μCRL, ACP$^\tau$ augmented with data, which is expressive enough for the specification of real-life protocols. The proof checker we use is Coq, which is based on the Calculus of Constructions, an extension of simply typed lambda calculus. The focus is on the translation of the proof theory of μCRL and μCRL-specifications to Coq. As a case study, we verified the Alternating Bit Protocol.

1. Introduction

This paper reports on the first steps towards the formal verification of correctness proofs of real-life protocols in process algebra. We show that such proofs can be verified, and partly constructed, by a general purpose proof checker. The focus is on the translation of process algebra (specifications and proof theory) to the language of the proof checker. As a case study, we verified the Alternating Bit Protocol (ABP) [BSW69]. We chose this protocol, not because there was any doubt about its correctness, but because it is small, well-known, and numerous correctness proofs are available in the literature [BaW90, BeK86b, BeG93, Dro94, Kam93].

The process algebra that we use is based on the Algebra of Communicating

Correspondence and offprint requests to: Marc Bezem, Department of Philosophy, Utrecht University, Heidelberglaan 8, 3584 CS Utrecht, The Netherlands. E-mail: bezem@phil.ruu.nl

Processes (ACP) of Bergstra and Klop [BeK86a]. More precisely, we use μCRL, ACP$^\tau$ augmented with data [GrP94, GrP93], which is expressive enough for the specification of real-life protocols. The proof checker we use is Coq [DFH93], which is based on the Calculus of Constructions, an extension of simply typed lambda calculus.

The word 'verification' usually refers to a mathematical proof in a combination of natural language and formal or informal mathematical notation. Consider for example the correctness proof of the ABP given in Sections 4.7 and 5.7 of [BaW90]. It consists of a series of steps so small that the reader is convinced of the correctness of each step. Indeed, the proof in [BaW90] is more detailed than most other verifications, because the intended reader is an undergraduate student.

For centuries, this form of verification was the best there was. But, as both the writer and the reader of the proof are human, what guarantee does it give that a proof is indeed correct? After all, to err is human. In some cases, especially now that computer programs and protocols are being incorporated in vital control systems, there is so much at stake that such a verification of a program is simply not enough. Especially in concurrent systems, where the number of situations can be exponential in the number of components, it is not at all unlikely that an unfortunate conjunction of circumstances is overseen during its design, testing, and verification-by-hand.

Recently it has become possible to let a computer program take over the role of the reader, or even that of the writer of proofs. In the first case such a program is called a proof checker, in the second case a theorem prover. The Coq-system, on which we focus in this paper, is a proof checker equipped with very limited theorem proving capabilities.

In contrast to a 'classical' verification, a *formal verification* is a proof formulated completely in a formal language; each step in it consists of the application of a formal proof rule. Theoretically, a formal verification could be done completely by hand, but on the basis of our experience (e.g. [Kam93]) we claim that, for real-life protocols, it can only be done using a computer. Such a verification is, by the nature of computers, a formal verification. To stress these observations, and also because a great deal of human input is still needed, we avoid the phrase 'automatic verification'.

If a proof checker is convinced of the correctness of a proof, should we be convinced too? One can never hope to achieve absolutely guaranteed correctness. But we claim that formal verification can provide a significant increase in the *level of confidence* in a protocol. In order to support this claim, we investigate which errors remain possible. We see the following types.

1. Errors of the computer system (hardware, operating system, etc.). These are relatively rare, and moreover usually result in error messages and/or sudden termination of the program, rather than in an erroneous proof being accepted by the proof checker.
2. Errors in the underlying theory of the proof checker. This theory should be stable and well-understood. For Coq, simply typed lambda calculus [Bar92] is basic and the Calculus of Constructions [CoH88] is well-understood. The theory of inductive types ([CoP90, PaM93], see Section 2.4) requires more study.
3. Programming errors in the proof checker. Indeed, the correctness of the proof checker must be checked thoroughly. As the program is much smaller (and

more modular) than the proofs we intend to verify, the level of confidence in large proofs is definitely raised, even if it is still not 100%.

4. The system we want to verify is usually formalized in a base theory different from the language of the proof checker. In this paper, the base theory is μCRL. This base theory might contain errors, or, less dramatically, axioms and proof rules that do not always apply (such as a fairness rule for a non-fair system). In this case the formal proof is correct, but it does not prove what we think it does.
5. The formalization of the system in the base theory might be incorrect. Again, the formal proof is correct. This error is more likely to occur than the previous one, because the base theory remains fixed, whereas we formalize a different system each time.
6. In order to use a proof checker, we translate the base theory and the theorem under consideration to the language of the proof checker. This translation can introduce errors.

The probability of the first three classes of errors can be reduced by verifying the same protocol on various different proof checkers (and platforms). The fourth and fifth class are orthogonal to the use of a proof checker. In this paper we concentrate on the translation of μCRL itself and μCRL-specifications to Coq. Special care must be taken when the translation of a specification deviates from its formalization 'because it is convenient in this particular proof checker'. Such errors can remain undiscovered much easier than the others, as the translation of a particular specification is used less often, and by less people, than the computer, the proof checker, and the translation of the base theory.

These considerations indicate that *the focus of the sceptical reader must shift from proofs to axioms*: a proof is the most likely place to find an error in an ordinary verification, but the proofs of a formal verification are most probably correct; for the axioms there is no such guarantee.

We hope that we have achieved a correct translation of μCRL to Coq, but the translation of a μCRL-specification into Coq is still done by hand. We choose to stay as close as possible to the definitions of μCRL and the ABP, even when this makes the proof somewhat clumsy. When we deviate from the original definitions, we do so explicitly and with motivation. If possible, we prove formally that the deviation is correct.

Formal verification is not limited to algebraic verification of protocols. In principle, it can be used for any formalism [Cou93], for example I/O-automata [LMW94, HSV94] and temporal logic [MaP82, OwL82, Hoo91]. Earlier attempts to automatic verification of propositions of process theory are from Cleaveland and Panangaden [ClP88], who gave an implementation of Milner's Calculus of Communicating Systems [Mil80] in the NuPrl system [CAB86] and from Engberg, Grønning and Lamport, who developed the Temporal Logic of Actions (TLA), which is a logic for specifying and reasoning about concurrent systems [EGL92]. A particularly impressive achievement is the assertional verification of wait-free linearization in [Hes94] and its formal elaboration [Hes95]. A recent approach to the ABP can be found in [Gim95], where the behaviour of processes is modelled by streams encoded as co-inductive types of Coq. In this stage of the development of the field it is very difficult to establish the relative merit of each of the results above, since their diversity makes comparison practically impossible. However, recent experience shows that the algebraic method discussed in this paper can handle larger protocols as well [BeG94a, KoS94, GrP96].

In the next section, we give an overview of μCRL and the ABP. Then we formalize the ABP in μCRL and sketch roughly the proof of its correctness. An introduction to Coq concludes this section. Section 3 is the core of the paper: it discusses how μCRL was translated to Coq, and which problems arose. Section 4 shows how the μCRL-specification of a protocol is translated into Coq, taking the ABP as an example. Section 5 describes in detail how a statement reflecting the correctness of the ABP can be proved from the axioms introduced in Section 3. The proof follows the sketch given in Section 2.3. The research on the topic of this paper is only just beginning; therefore we conclude the paper with a list of directions for future research.

2. Preliminaries

2.1. μCRL

μCRL is a specification formalism, combining the process algebra ACP$^\tau$ [BaW90] with data. We give a brief and informal introduction here; for a complete description of its syntax and semantics we refer to [GrP94], for its proof theory to [GrP93].

2.1.1. Syntax and Semantics

An algebra is usually a set, together with a number of operations on that set, in principle axiomatized by an equational theory. ACP$^\tau$ complies with this tradition. The set is a set of processes and the operations are

- constants (called atomic actions, the set of atomic actions *Act* is a parameter of ACP$^\tau$ that is often left implicit)
- the constants δ (deadlock) and τ (silent action)
- the unary operators ∂_L (encapsulation) and τ_L (abstraction or hiding), where L is a set of atomic actions
- the binary operators $+$, $\cdot$, $\|$, $|$, and $\lfloor\!\lfloor$, being alternative and sequential composition, merge, communication merge, and left merge. By convention, $\cdot$ binds strongest and $+$ weakest

We refer to [BaW90] for an explanation of these operators. The operator $|$ is an extension of another parameter of ACP$^\tau$, the communication function γ. This is a partial function which, given two atomic actions, returns an atomic action: their communication. γ must be associative and commutative. In this paper we assume *handshaking*, which means that no more than two processes can engage in a single communication. Technically, it means that $\gamma(\gamma(a,b),c)$ is undefined for all actions a,b,c.

Data is specified in μCRL by the declaration of sorts (types), functions (including constants) with their types and possibly rewrite rules (stating equalities between dataterms). The corresponding sections in a μCRL-specification are marked by the keywords **sort**, **func** and **rew**. The sort **Bool** containing the constants T and F is part of every μCRL-specification. Sorts may not be empty.

μCRL combines ACP$^\tau$ with data through the following mechanisms.

- An atomic action is composed of an action name and (zero or more) parameters; these parameters are dataterms. The section containing the declaration of action names (marked by the keyword **act**) also specifies the sorts of their parameters (overloading of action names is allowed).

- Communication is defined on action names (in a section marked **comm**). Two actions only communicate if their parameters are the same (w.r.t. the rewrite rules); the resulting action has the same parameters. Communication is used for both synchronization and transferring data in this way.
- The conditional operator $x \lhd b \rhd y$ takes processes x and y and a boolean b; it behaves as x if $b = T$ and as y if $b = F$.
- The sum operator $\sum_{d:D} x$ denotes the (possibly infinite) alternative composition of the processes $\sigma(x)$ for substitutions σ substituting an element of the sort D for d in x.
- Processes can be defined by (recursive) process specifications (keyword **proc**). Parameters are allowed in these definitions.

The conditional operator has a boolean as its middle argument. This is why the sort **Bool** is part of every μCRL-specification. The symbol '=' occurs in μCRL-specifications in rewrite rules, communication declarations, and process specifications. It is *not* a polymorphic function $D \rightarrow D \rightarrow$ **Bool**, thus it cannot be used for forming the middle argument of a conditional operator.[1] Moreover, it is not entirely trivial to define such a function $eq_D : D \rightarrow D \rightarrow$ **Bool** satisfying $eq_D(d,e) = T$ iff $d = e$. The following specification (by Jan Bergstra) does the trick.

Example 2.1.

$$\begin{array}{lll}
\textbf{sort} & \textbf{Bool}\ D & \\
\textbf{func} & T, F : & \rightarrow \textbf{Bool} \\
 & eq_D\ : D \rightarrow D & \rightarrow \textbf{Bool} \\
 & if_D\ : \textbf{Bool} \rightarrow D \rightarrow D \rightarrow D & \\
\textbf{var} & d, e : D & \\
\textbf{rew} & eq_D(d,d) & = T \\
 & if_D(T,d,e) & = d \\
 & if_D(F,d,e) & = e \\
 & if_D(eq_D(d,e),d,e) & = e
\end{array}$$

Claim 2.2. The equations in the previous example enforce

1. $eq_D(d,e) = T \leftrightarrow d = e$,
2. $eq_D(d,e) = F \leftrightarrow d \neq e$.

Proof of Claim 2.2. (Via the semantics of μCRL. A proof via the formal proof theory is given in the next subsection.)

1,→) $d = if_D(T,d,e) = if_D(eq_D(d,e),d,e) = e$.

1,←) $eq_D(d,e) = eq_D(d,d) = T$.

2,↔) From *1*, as the intended models are boolean preserving [GrP94], that is, $T \neq F$ and for all booleans b: $b = T \ \vee \ b = F$, thus in particular $eq_D(d,e) \neq T \rightarrow eq_D(d,e) = F$. □

[1] It is not without reason that an equation *between processes* cannot occur as the middle argument of a conditional operator: the guarded recursive process definition $P = (a \lhd P = \delta \rhd \delta)$ would lead to $a = \delta$.

Table 1. The axioms and rules for data.

REFL	$t = t$	reflexivity,
FACT	$t = u$	if $t = u$ is a rewrite rule,
REPL	$\frac{\phi[t/x] \quad t=u}{\phi[u/x]}$	replace t by u,
SUB	$\frac{\phi}{\phi[t/x]}$	substitute t for x,
IND	...	induction rules for sorts,
B1	$\neg(T = F)$	
B2	$b = T \vee b = F$	b is a boolean variable.

2.1.2. *Proof Theory*

The proof theory of μCRL is given in [GrP93] in a 'natural deduction' format. The formulae deduced ('μCRL property formulae') are mostly equations, and propositional logical combinations of those. The axioms and rules can be divided into four parts: data, ACP$^{\tau}$, process constructs relating processes with data and logical connectives. Some of these depend on the μCRL-specification under consideration, most notably its declarations of rewrite rules and process definitions.

For data, we have the axioms and rules listed in Table 1. μCRL has no explicit quantification; the rule SUB enforces that each variable is implicitly universally quantified. Its application is only allowed when x does not occur in any hypothesis needed for deriving ϕ. For the precise definitions of substitutions and induction rules we refer to [GrP93]. An induction rule for a sort is based on a set of constructors for that sort. Which functions form a constructor set of a sort is not part of the μCRL-specification (but see [GrW94]). Given a μCRL-specification, one can prove that a certain set is a constructor set only on the metalevel, using structural induction on closed terms. The axiom B1 is another reason for incorporating the booleans in every μCRL-specification: without this axiom one can never prove the inequality of two terms (the premiss of the rule CF2$'$ in Table 3).

For the logical connectives, μCRL has a large number of inference rules. For those, we refer to [GrP93] (see also the proof below), except that we mention the rule RAA (reductio ad absurdum), stating that if falsum ($\bot$) is derivable from $\neg\phi$, then ϕ can be derived. As usual $\neg\phi$ abbreviates $\phi \rightarrow \bot$, thus negation and implication behave classically. But in proofs it turns out that we do not need RAA, which means that our results also hold from an intuitionistic viewpoint.

Proof of Claim 2.2. We can now prove Claim 2.2 formally in the proof theory of μCRL. For reasons of space, we do not write the names of derivation rules to the left of the line, but below it (above it for rules without premises). $\rightarrow$I, [n] denotes the rule for the introduction of an implication, where n is a pointer to the cancelled hypothesis(-es). $\rightarrow$E denotes implication elimination, i.e., modus ponens. $\phi \vee \psi$ is introduced in μCRL as an abbreviation of $\neg\phi \rightarrow \psi$.

1,$\rightarrow$)

$$\dfrac{\dfrac{\dfrac{\dfrac{\text{FACT}}{if(eq(d,e),d,e)=e} \qquad \dfrac{(1)}{eq(d,e)=T}}{\underset{\text{REPL}}{} \quad if(T,d,e)=e} \qquad \dfrac{\text{FACT}}{if(T,d,e)=d}}{\underset{\text{REPL}}{} \quad d=e}}{\underset{\rightarrow\text{I, [1]}}{} \quad eq(d,e)=T \rightarrow d=e}$$

Table 2. The axioms of ACP$^\tau$ in μCRL. $a, b \in \text{Act} \cup \{\delta, \tau\}$.

A1	$x + y = y + x$	CM1	$x \| y = x ⌊\!⌊ y + y ⌊\!⌊ x + x \mid y$
A2	$x + (y + z) = (x + y) + z$	CM2	$a ⌊\!⌊ x = a \cdot x$
A3	$x + x = x$	CM3	$a \cdot x ⌊\!⌊ y = a \cdot (x \| y)$
A4	$(x + y) \cdot z = x \cdot z + y \cdot z$	CM4	$(x + y) ⌊\!⌊ z = x ⌊\!⌊ z + y ⌊\!⌊ z$
A5	$(x \cdot y) \cdot z = x \cdot (y \cdot z)$	CM5	$a \cdot x \mid b = (a \mid b) \cdot x$
A6	$x + \delta = x$	CM6	$a \mid b \cdot x = (a \mid b) \cdot x$
A7	$\delta \cdot x = \delta$	CM7	$a \cdot x \mid b \cdot y = (a \mid b) \cdot (x \| y)$
		CM8	$(x + y) \mid z = x \mid z + y \mid z$
T1	$x \cdot \tau = x$	CM9	$x \mid (y + z) = x \mid y + x \mid z$
D1	$\partial_L(a) = a \quad \text{if } a \notin L$	TI1	$\tau_L(a) = a \quad \text{if } a \notin L$
D2	$\partial_L(a) = \delta \quad \text{if } a \in L$	TI2	$\tau_L(a) = \tau \quad \text{if } a \in L$
D3	$\partial_L(x + y) = \partial_L(x) + \partial_L(y)$	TI3	$\tau_L(x + y) = \tau_L(x) + \tau_L(y)$
D4	$\partial_L(x \cdot y) = \partial_L(x) \cdot \partial_L(y)$	TI4	$\tau_L(x \cdot y) = \tau_L(x) \cdot \tau_L(y)$
SC1	$(x ⌊\!⌊ y) ⌊\!⌊ z = x ⌊\!⌊ (y \| z)$	DC1	$\delta \mid x = \delta$
SC2	$x ⌊\!⌊ \delta = x \cdot \delta$	TC1	$\tau \mid x = \delta$
SC3	$x \mid y = y \mid x$	Handshaking	$x \mid (y \mid z) = \delta$
SC4	$(x \mid y) \mid z = x \mid (y \mid z)$		
SC5	$x \mid (y ⌊\!⌊ z) = (x \mid y) ⌊\!⌊ z$		

1,←)

$$\text{→I, [1]}\ \dfrac{\text{REPL}\ \dfrac{\dfrac{\text{FACT}}{eq(d,d)=T} \qquad \dfrac{(1)}{d=e}}{eq(d,e)=T}}{d=e \rightarrow eq(d,e)=T}$$

2,→)

$$\text{→I, [1]}\ \dfrac{\text{→I, [2]}\ \dfrac{\text{→E}\ \dfrac{\text{REPL}\ \dfrac{\dfrac{(1)}{eq(d,e)=F} \qquad \text{REPL}\ \dfrac{\dfrac{\text{FACT}}{eq(d,d)=T} \qquad \dfrac{(2)}{d=e}}{eq(d,e)=T}}{T=F} \qquad \dfrac{\text{B1}}{\neg T=F}}{\bot}}{\neg d=e}}{eq(d,e)=F \rightarrow \neg d=e}$$

2,←)

$$\text{→I, [1]}\ \dfrac{\text{→E}\ \dfrac{\text{→I, [2]}\ \dfrac{\text{→E}\ \dfrac{\text{→E}\ \dfrac{\dfrac{(2)}{eq(d,e)=T} \qquad \dfrac{1,\rightarrow}{eq(d,e)=T \rightarrow d=e}}{d=e} \qquad \dfrac{(1)}{\neg d=e}}{\bot}}{\neg eq(d,e)=T} \qquad \text{SUB}\ \dfrac{\dfrac{\text{B2}}{b=T \vee b=F}}{eq(d,e)=T \vee eq(d,e)=F}}{eq(d,e)=F}}{\neg d=e \rightarrow eq(d,e)=F}$$

Proofs are usually not given in such detail, for obvious reasons. For the same reasons, it is preferable that such details need not be provided to the proof checker explicitly. □

For processes, μCRL inherited the axioms A1–A7, CM1–CM9, D1–D4, T1 (called B1 in [BaW90]) and TI1–TI4 from ACP$^\tau$, listed in Table 2. All closed instances without process variables of the axioms SC1–SC5, DC1, TC1, and Handshaking are derivable. SC3 and SC4 directly reflect the properties of the communication function γ (corresponding axioms for $\|$ are mentioned also in [BaW90], but these are derivable). The handshaking assumption similarly results in the axiom Handshaking. SC4, CM5, CM6, and CM9 are derivable.

The axioms for the communication merge are more complicated than those of ACP$^\tau$, because of the presence of data. The presentation here differs slightly from

Table 3. Axioms relating processes and data. $a, b, c \in \mathrm{Act} \cup \{\delta, \tau\}$.

CF1	$a(t_1,\ldots,t_m) \mid b(t_1,\ldots,t_m) = c(t_1,\ldots,t_m)$	if $\gamma(a,b) = c$, $m \geq 0$,
CF2	$a(t_1,\ldots,t_m) \mid b(t'_1,\ldots,t'_m) = \delta$	if $\gamma(a,b)$ is undefined, in particular, if a or b is δ or τ,
CF2′	$\dfrac{\neg(t_i = t'_i)}{a(t_1,\ldots,t_m) \mid b(t'_1,\ldots,t'_m) = \delta}$	$1 \leq i \leq m$,
CF2″	$a(t_1,\ldots,t_m) \mid b(t'_1,\ldots,t'_{m'}) = \delta$	if a and b have different sorts, in particular, if $m \neq m'$.
COND1	$x \lhd T \rhd y = x$	
COND2	$x \lhd F \rhd y = y$	
SUM1	$\sum_{d:D} p = p$	if d not free in p,
SUM2	$\sum_{d:D} p = \sum_{e:D}(p[e/d])$	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	$\sum_{d:D}(p_1 \mathbin{\lfloor\!\lfloor} p_2) = \sum_{d:D} p_1 \mathbin{\lfloor\!\lfloor} p_2$	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	$\sum_{d:D} \partial_L(p) = \partial_L(\sum_{d:D} p)$	
SUM9	$\sum_{d:D} \tau_L(p) = \tau_L(\sum_{d:D} p)$	
SUM11	$\dfrac{p_1 = p_2}{\sum_{d:D} p_1 = \sum_{d:D} p_2}$	if d not free in the assumptions of the proof of $p_1 = p_2$.

[GrP93], where actions without parameters are treated as a special case. See also Section 3. The axioms for the conditional and sum operators are mostly obvious. For SUM8 and SUM9, recall that encapsulation and hiding are carried out at the level of action *names*. In [GrP93], SUM10 states that renaming distributes over summation; we have omitted renaming here.

The rules REFL, REPL, and SUB also apply to processes. The counterpart of FACT is called REC: $p = q$ if $p = q$ is a process equation. Finally, there are some more complicated inference rules inherited from ACP$^\tau$: RDP, RSP, and fair abstraction. These rules refer to the (recursive) specifications of processes. RDP, the Recursive Definition Principle, states that such a specification has at least one solution. RSP, the Recursive Specification Principle, states that two processes are equal, if they are both solutions of the same guarded recursive specification. The Cluster Fair Abstraction Rule CFAR [BaW90] can be paraphrased informally as: 'Any process will eventually leave a τ-cluster'. The details are discussed in Sections 3.5, 3.6, and 3.7.

2.2. 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 Fig. 1.

The numbered connection lines in Fig. 1 represent gates, through which the components can communicate. The sender S reads data from the input at gate 1,

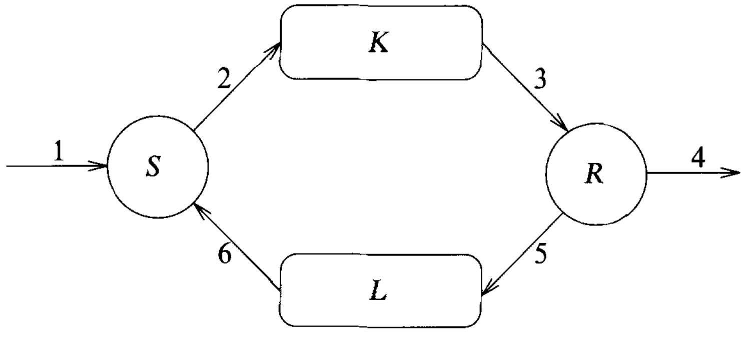

Fig. 1. Alternating Bit Protocol.

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 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: $s_3(\bot)$ for K and $s_6(\bot)$ for L. Moreover, the channels are assumed not to corrupt data *ad infinitum* (in that case it is obviously impossible to ensure reliable transmission). This fairness assumption justifies the use of the proof rule CFAR later on.

The ABP roughly works as follows. S, K, R, and L run strictly synchronized, i.e., K sends a message if and only if it receives one from S, R sends a message if and only if it receives one from K, etc. (except that S sends the very first message without receiving something from L).

S reads a datum d from the input and sends a frame (e_0, d) via K to R. As long as K corrupts the data, R receives frames $\bot$ and reacts by sending bits e_1 via L to S, so that S sends the frame again. Once R receives a frame (e_0, d), it writes d to the output and acknowledges this receipt by sending the bit e_0 via L to S. From then on, R sends a bit e_0 via L to S, each time it receives an incoming frame (e_0, d) or $\bot$. Process S sends a frame (e_0, d) each time it receives something from L, until that something is an acknowledging bit e_0. In that case S reads a new datum d' from the input and starts sending frames (e_1, d') to R. So now the cycle starts all over, with e_0 and e_1 exchanged. That is, R reacts to incoming frames $\bot$ by sending e_0, and after it receives a frame (e_1, d'), it writes d' to the output and starts acknowledging the receipt of frame (e_1, d') by sending bits e_1 to S. 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.

2.3. Specification and Verification of the ABP in μCRL

We now present a formalization of the ABP in μCRL. It follows closely the definition of the ABP in [BaW90], except that now data is treated more formally (which also involved some renamings). We make no difference between a bit and a boolean. Therefore we have no separate sort *bit*, but use **Bool** instead. The sort *bool_Err* (*Frame_Err*) is the disjoint sum of the sort **Bool** ($D \times$ **Bool**) and a singleton sort containing an error element, with an injection *ibool* :**Bool**$\to$*bool_Err* (*iFrame* :$D \times$ **Bool** $\to$*Frame_Err*). We assume D to be a given, nonempty sort; we do not specify its elements. The correctness of the ABP follows from the derivability in μCRL of $ABP = Buffer$.

sort **Bool**
bool_Err
Frame_Err

func	T, F	:	$\to$ **Bool**
	neg	: **Bool**	$\to$ **Bool**
	$ibool$	: **Bool**	$\to$ $bool\_Err$
	$errorbit$	:	$\to$ $bool\_Err$
	$iFrame$	: $D \times$ **Bool**	$\to$ $Frame\_Err$
	$errorframe$	:	$\to$ $Frame\_Err$

var b_1, b_2 : **Bool**
$d_1, d_2 : D$

rew eq_S and if_S for all sorts, see Example 2.1
$neg(b_1) = eq_{\mathbf{Bool}}(b_1, F)$
$eq_{bool\_Err}(ibool(b_1), ibool(b_2)) = eq_{\mathbf{Bool}}(b_1, b_2)$
$eq_{bool\_Err}(ibool(b_1), errorbit) = F$
$eq_{Frame\_Err}(iFrame(d_1, b_1), iFrame(d_2, b_2)) =$
$if_{\mathbf{Bool}}(eq_{\mathbf{Bool}}(b_1, b_2), eq_D(d_1, d_2), F)$
$eq_{Frame\_Err}(iFrame(d_1, b_1), errorframe) = F$

act	r_1, s_4	$: D$
	r_2, s_2, c_2	$: D \times$ **Bool**
	r_3, s_3, c_3	$: Frame\_Err$
	r_5, s_5, c_5	: **Bool**
	r_6, s_6, c_6	$: bool\_Err$
	i	

comm $r_2 \mid s_2 = c_2$
$r_3 \mid s_3 = c_3$
$r_5 \mid s_5 = c_5$
$r_6 \mid s_6 = c_6$

proc $Buffer = \sum_{d:D}(r_1(d) \cdot s_4(d)) \cdot Buffer$
$ABP = \tau_{\{c_2,c_3,c_5,c_6,i\}}(\partial_{\{r_2,s_2,r_3,s_3,r_5,s_5,r_6,s_6\}}(Sd \parallel Rc \parallel K \parallel L))$
$K = \sum_{f:D\times\mathbf{Bool}}(r_2(f) \cdot (i \cdot s_3(iFrame(f)) + i \cdot s_3(errorframe))) \cdot K$
$L = \sum_{b:\mathbf{Bool}}(r_5(b) \cdot (i \cdot s_6(ibool(b)) + i \cdot s_6(errorbit))) \cdot L$

$$\begin{aligned}
&Sd = Sb(T) \cdot Sb(F) \cdot Sd\\
&Rc = Rb(F) \cdot Rb(T) \cdot Rc\\
&Sb(b : \mathbf{Bool}) = \textstyle\sum_{d:D} r_1(d) \cdot Sf(d,b)\\
&Sf(d : D, b : \mathbf{Bool}) = s_2(d,b) \cdot Tf(d,b)\\
&Tf(d : D, b : \mathbf{Bool}) =\\
&\qquad (r_6(ibool(neg(b))) + r_6(errorbit)) \cdot Sf(d,b) + r_6(ibool(b))\\
&Rb(b : \mathbf{Bool}) =\\
&\qquad (\textstyle\sum_{d:D} r_3(iFrame(d,b)) + r_3(errorframe)) \cdot s_5(b) \cdot Rb(b) +\\
&\qquad \textstyle\sum_{d:D} r_3(iFrame(d,neg(b))) \cdot s_4(d) \cdot s_5(neg(b))
\end{aligned}$$

We now outline the correctness proof of the ABP as formalized in Section 5. For additional details we refer to Sections 4.7 and 5.7 of [BaW90]. We use H to abbreviate $\{r_2, s_2, r_3, s_3, r_5, s_5, r_6, s_6\}$ and I to abbreviate $\{c_2, c_3, c_5, c_6, i\}$.

In order to exploit the symmetry in the protocol, we abstract from the state of the alternating bit in the sender and the receiver. That is, we define

$$\begin{aligned}
&Sd(b : \mathbf{Bool}) = Sb(b) \cdot Sb(neg(b)) \cdot Sd(b)\\
&Rc(b : \mathbf{Bool}) = Rb(neg(b)) \cdot Rb(b) \cdot Rc(b)
\end{aligned}$$

It is obvious, and easy to prove by RSP, that $Sd = Sd(T)$ and $Rc = Rc(T)$. We also need the equally obvious equations $Sd(b) = Sb(b) \cdot Sd(neg(b))$ and $Rc(b) = Rb(neg(b)) \cdot Rc(neg(b))$.

We introduce some more auxiliary definitions. The aim of these is to give a *linear* description of the protocol before hiding. That is, the equations are of the form $X() = \sum a() \cdot Y()$, where $\sum$ denotes a mixture of alternative compositions and summations, X and Y are process variables and a an action. If we fill in all parameters of X, we obtain a *state* of the protocol, and the equation then gives all possible actions with their resulting states. This linearization is depicted in Fig. 22 of [BaW90]; Figures 3 and 4 constitute the same figure somewhat simplified.

In these definitions, we use the syntax $\langle X \mid E \rangle$ from [BaW90] to denote the process defined by the process variable X in the recursive specification E. The advantage of this notation over μCRL is that we can distinguish various (sub)systems of equations. This is particularly useful when it comes to applying RSP and CFAR formally on systems of equations, as is done in Section 5.2, respectively 5.4.

$$\begin{aligned}
ABP\_nohide(b) &= \partial_H(Sd(b) \parallel Rc(b) \parallel K \parallel L)\\
First(d,b) &= r_1(d) \cdot \langle X_1 \mid E_1(d,b) \rangle\\
Exit1(d,b) &= c_3(iFrame(d,b)) \cdot s_4(d) \cdot \langle X_1 \mid E_2(d,b) \rangle\\
Exit2(b) &= c_6(ibool(b)) \cdot ABP\_nohide(neg(b))
\end{aligned}$$

$$E_1(d,b) \triangleq \left\{ \begin{aligned}
X_1 &= c_2(d,b) \cdot X_2\\
X_2 &= i \cdot Exit1(d,b) + i \cdot X_3\\
X_3 &= c_3(errorframe) \cdot X_4\\
X_4 &= c_5(neg(b)) \cdot X_5\\
X_5 &= i \cdot X_6 + i \cdot X_7\\
X_6 &= c_6(errorbit) \cdot X_1\\
X_7 &= c_6(ibool(neg(b))) \cdot X_1
\end{aligned} \right\}$$

$$E_2(d,b) \triangleq \{ \begin{array}{rl} X_1 = & c_5(b) \cdot X_2 \\ X_2 = & i \cdot \mathit{Exit2}(b) + i \cdot X_3 \\ X_3 = & c_6(\mathit{errorbit}) \cdot X_4 \\ X_4 = & c_2(d,b) \cdot X_5 \\ X_5 = & i \cdot X_6 + i \cdot X_7 \\ X_6 = & c_3(\mathit{errorframe}) \cdot X_1 \\ X_7 = & c_3(\mathit{iFrame}(d,b)) \cdot X_1 \end{array} \}$$

The major task of the verification is to prove the following lemma.

Lemma 2.3. $\mathit{ABP\_nohide}(b) = \sum_{d:D} \mathit{First}(d,b)$.

Proof. By numerous applications of the axioms, we can infer the possible first actions of $\mathit{ABP\_nohide}(b)$ and their resulting states. It turns out that

$$\mathit{ABP\_nohide}(b) = \sum_{d:D} (r_1(d) \cdot \partial_H(\mathit{Sf}(d,b) \cdot \mathit{Sb}(\mathit{neg}(b)) \cdot \mathit{Sd}(b) \| \mathit{Rc}(b) \| K \| L))$$

Unfolding the definition of *First* in the lemma, and stripping the first action on both sides, we arrive at the proof obligation

$$\partial_H(\mathit{Sf}(d,b) \cdot \mathit{Sb}(\mathit{neg}(b)) \cdot \mathit{Sd}(b) \| \mathit{Rc}(b) \| K \| L) = \langle X_1 \mid E_1(d,b) \rangle$$

The lefthandside of this equation describes the next state of the protocol. We continue by determining the possible first actions of this next state, and the state after that, and so on. After lots of steps, we derive

$$\begin{aligned} &\partial_H(\mathit{Sf}(d,b) \cdot \mathit{Sb}(\mathit{neg}(b)) \cdot \mathit{Sd}(b) \| \mathit{Rc}(b) \| K \| L) = \\ &\quad c_2(d,b) \cdot (\, i \cdot \mathit{SomeState} + \\ &\qquad\qquad i \cdot c_3(\mathit{errorframe}) \cdot \ldots \cdot \\ &\qquad\qquad\qquad \partial_H(\mathit{Sf}(d,b) \cdot \mathit{Sb}(\mathit{neg}(b)) \cdot \mathit{Sd}(b) \| \mathit{Rc}(b) \| K \| L)) \end{aligned}$$

where *SomeState* is of the form $\partial_H(\mathit{SenderState} \| \mathit{ReceiverState} \| \mathit{KState} \| \mathit{LState})$. The righthandside of this equation corresponds to the structure of E_1, therefore we can conclude by RSP that the aforementioned proof obligation follows from $\mathit{SomeState} = \mathit{Exit1}(d,b)$. Extracting first actions twice more, and unfolding the definition of *Exit1*, we arrive at the proof obligation $\mathit{SomeState}' = \langle X_1 \mid E_2(d,b) \rangle$. This one is tackled again by RSP, and results in $\mathit{SomeState}'' = \mathit{Exit2}(b)$. Finally, we extract the first action $c_6(\mathit{ibool}(b))$ of $\mathit{SomeState}''$, and arrive at

$$\partial_H(\mathit{Sb}(\mathit{neg}(b)) \cdot \mathit{Sd}(b) \| \mathit{Rb}(b) \cdot \mathit{Rc}(b) \| K \| L) = \mathit{ABP\_nohide}(\mathit{neg}(b))$$

This equation follows immediately from our observations upon the introduction of $\mathit{Sd}(b)$ and $\mathit{Rc}(b)$. □

Theorem 2.4. $\mathit{ABP} = \mathit{Buffer}$.

Proof. By unfolding *First*, axiom TI4, applying CFAR on the clusters E_1 and E_2, and axiom T1, we derive

$$\tau_I(\mathit{First}(d,b)) = r_1(d) \cdot s_4(d) \cdot \tau_I(\mathit{ABP\_nohide}(\mathit{neg}(b))).$$

Combined with Lemma 2.3, we conclude

$$\tau_I(\mathit{ABP\_nohide}(b)) = (\sum_{d:D} r_1(d) \cdot s_4(d)) \cdot \tau_I(\mathit{ABP\_nohide}(\mathit{neg}(b))).$$

It is now straightforward to show that ABP, being $\tau_I(ABP\_nohide(T))$, and *Buffer* both satisfy the equation

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

So, a final application of RSP concludes the proof. □

2.4. The Coq Proof Checker

For a complete overview of the Coq proof checker, we refer to [DFH93]. It is based on the Calculus of Constructions, an extension of simply typed lambda calculus, but a deep understanding of that formalism, in particular of the identification of propositions and types, is not necessary for understanding the use we make of Coq (propositions are of type `Prop` and types of type `Set`). One can declare types, and state the existence of (constructor) functions with their types, including constants. One can express quantification and higher order logic. The implication and negation behave constructively.

Coq extends the Calculus of Constructions by *inductive definitions* of sorts and propositions. A sort is defined inductively by listing its constructors. Such a definition of an `Inductive Set` yields an induction principle and a `Match`-function, which enables the definition of (primitive recursive) functions by induction on the constructors. Together, they imply that every term of that sort is equal to a constructor term, and that all constructor terms are different. For example, the sort **Bool** can be translated to Coq as

```
Inductive Set bool = true : bool | false : bool.
```

Equality in Coq is a ternary polymorphic function `<_>_=_` (see below). It has a so-called *dependent type*: `(D:Set)D->D->Prop`. That is, for each `D`, `<D>_=_` is a function of type `D->D->Prop`. A simpler example of a dependent type is the type of the function `[D:Set][d:D]d`, the polymorphic identity function (square brackets denote lambda-abstraction in Coq). Its type is `(D:Set)D->D`. In fact, the notation `P->Q` is an abbreviation of `(x:P)Q` when `x` does not occur in `Q`.

From the inductive definition of `bool`, one can prove `~(<bool>true=false)` (`true` and `false` are not equal) and `(b:bool)<bool>b=true\/<bool>b=false` (for all `b` of type `bool`, `b` is either `true` or `false`). These statements correspond to the axioms B1 and B2 in μCRL. One must realize that inductive definitions come with a powerful elimination principle (see below). Otherwise, one easily writes a seemingly reasonable specification which is nevertheless incorrect, perhaps even inconsistent. For this reason and others, explained later, we shall not use this translation. It would certainly not be a good idea to define processes inductively, as there is no assumption in the semantics of μCRL that all processes can be built from the given actions and operators.

By the propositions-as-types paradigm, propositions can also be defined inductively. An inductively defined type is the least set that is closed under the constructors (such that all constructor terms differ); an inductively defined proposition is the least proposition that is closed under the rules given for it. Rather than giving a formal definition, we give an example.

Example 2.5. We consider the transitive closure function, which, given a relation R on $D \times D$, returns the transitive closure of R. The relation R is represented in Coq by its characteristic function of type D->D->Prop. ([R:D->D->Prop]x denotes $\lambda R.x$)

```
Inductive Definition TC [R:D->D->Prop] : D->D->Prop =
Base : (x,y  :D) (R x y) ->            (TC R x y)|
Trans: (x,y,z:D) (R x y) -> (TC R y z) -> (TC R x z)
```

This definition says that $TC(R)$ is the *least* relation closed under the above rules; therefore an elimination principle comes with this definition: in order to prove a proposition $P(x,y)$ under the assumption $TC(R)(x,y)$, it is sufficient to prove

$$R(x,y) \rightarrow P(x,y) \text{ and } R(x,y) \wedge TC(R)(y,z) \wedge P(y,z) \rightarrow P(x,z)$$

This seems somewhat stronger than the usual induction scheme without the conjunct $TC(R)(y,z)$, but it is actually equivalent.

Also basic notions in Coq, such as truth, falsity, and equality, are inductively defined.

```
Inductive Definition True  : Prop = I: True
Inductive Definition False : Prop =          .
Syntax eq "<_>_=_".
Inductive Definition eq [A:Set;x:A] : A->Prop = refl_equal: <A>x=x
```

I is by definition the proof of the nullary relation True; the elimination principle for True is a tautology. False is the empty nullary relation; with this definition comes the axiom False_ind: (P:Prop)False->P, the ex-falso rule, which reflects the minimality property (or the elimination principle) for False. Finally, equality on a set A is defined through the statement 'for x:A, the unary relation "being equal to x" contains only x'. This definition gives the induction principle (A:Set)(x:A)(P:A->Prop)(P x)->(a:A)(<A>x=a)->(P a). Thus the effect of eliminating[2] <A>b=a is that (usually all) occurrences of a are replaced by b. Equations can be used as term rewrite rules from right to left in this way.[3] Conjunction and disjunction are also inductively defined. Eliminating a conjunctive hypothesis A/\B yields two hypotheses A and B; eliminating A\/B yields two new proof obligations, one with hypothesis A and one with B.

A proof in Coq starts from the statement that one wants to prove, which is then transformed by applying *tactics*. A tactic replaces a *proof obligation* by zero or more new ones. A proof obligation consists of two parts: the *goal* (initially the statement that one wants to prove) and the *context*, a set of declarations of variables and premisses that can be used in the proof[4]. A proof is completed if there are no more proof obligations. Some typical tactics are:

Intro	moves a universal quantifier or the premiss of an implication from the goal to the context.
Apply H	applies resolution on the goal and H, a hypothesis from the context, global axiom, or theorem. If H is an implication, each premiss yields a new proof obligation.

[2] By eliminating H, we mean applying the induction principle for the main constructor of H.

[3] The fact that some of our axioms are written 'backwards' is a relic of a Coq version that could only rewrite in this direction. Version 5.8 has also a tactic Rewrite for rewriting from left to right.

[4] According to the propositions-as-types paradigm, there is no fundamental distinction between a declaration d:D with D:Set and a hypothesis H:P with P:Prop.

`Elim H`	For a declaration `H:D`, where `D` is an inductive set, this amounts to structural induction. For a hypothesis `H:P`, where the main predicate of `P` is inductively defined, it applies the elimination principle.
`Contradiction`	looks for a hypothesis `False`.
`Assumption`	looks for a hypothesis equal to the current goal.
`Exact H`	succeeds if the goal is exactly the hypothesis, axiom, or theorem `H`.
`Unfold name`	unfolds the definition of `name`.
`Pattern` *position*	allows the selection of redexes for term rewriting.
`Auto`	tries to complete the proof by applying hypotheses and designated theorems.
`Idtac`	does not change the proof obligation (sometimes useful in complicated tactics).

Complicated tactics can be constructed from the basic ones. They can succeed, fail, or run out of space. A basic tactic fails if it is not applicable.

tactic_1 ; tactic_2	applies tactic_1 and then tactic_2 on all proof obligations generated by tactic_1.
tactic_0 ; $[\text{tactic}_1 \mid \ldots \mid \text{tactic}_n]$	applies tactic_0 and then $\text{tactic}_1, \ldots, \text{tactic}_n$ to the n proof obligations generated by tactic_0.
tactic_1 `Orelse` tactic_2	tries to apply tactic_1, if it fails tactic_2 is applied.
`Try` tactic_1	tries to apply tactic_1, but it does not fail even if tactic_1 does.
`Repeat` tactic_1	repeats tactic_1 until it fails. This tactic never fails.

`Auto` never fails: if it cannot complete the proof, it leaves the goal unchanged. `Auto;Exact I` gives a version of `Auto` that can fail. (`Exact I` cannot be applicable after `Auto`, because `Auto` tries it.)

3. The Translation of μCRL into Coq

3.1. μCRL versus Coq

μCRL and its proof theory share a significant number of concepts with Coq; we name (data)types, equality, implication, axioms, and derivability. The most formal way to proceed is to ignore these similarities, and to encode each μCRL-concept in Coq. That is, to define a sort `muCRL_Prop` of μCRL property formulae and to encode μCRL-derivability inductively as the least relation `Dv : muCRL_Prop->Prop` that contains all axioms and is closed under all inference rules of μCRL:

```
Inductive Definition Dv:muCRL_Prop->Prop=

REFL: (D:sorts) (has_sort t D)  -> (Dv (equal D t t)) |

REPL: (Phi:muCRL_Prop)(D:sorts)
      (Dv (subst D t x Phi))    ->
      (Dv (equal D t u))        -> (Dv (subst D u x Phi)) |
A1:    (p,q:proc)                (Dv (equal proc (alt p q) (alt q p))) |

ArrowI: (Phi,Psi:muCRL_Prop)
        ((Dv Phi)->(Dv Psi))     -> (Dv (implies Phi Psi)) |
...
```

In this example, `equal` encodes the equality predicate of μCRL, `subst` encodes substitution, `sorts` the declaration of sorts, `has_sort` the declaration of variables, `alt` the $+$ on processes, `implies` implication between μCRL property formulae, and so on.

Translating μCRL to Coq in this way might be possible (the above formulation of `ArrowI` is problematic), but it is cumbersome: it gives rise to unreadable Coq texts and makes it impossible to automate the bulk of the proof (in the version 5.8.3 of Coq we used). Namely, proofs in process algebra typically use a subset of the axioms (and derived equations) as a term rewriting system, computing normal forms for process terms (modulo associativity and commutativity of $+$). Hand-written, such a part of the proof appears as *term* $=$ *term* $= \ldots =$ *term*; formally each step is an application of REPL. In the above translation, the intermediate terms cannot be found by Coq; the user must provide them. This makes it effectively impossible to find even the most trivial proof automatically. In other words, with this translation we cannot hope to achieve a granularity of Coq proofs that comes anywhere near the granularity of hand-written proofs. Consequently, this approach is not (yet) scalable to real-life protocols.

Therefore we take another approach: rather than encoding μCRL in Coq, we embed μCRL in Coq, that is, we map μCRL-concepts to the 'same' concepts in Coq as much as possible. Such a translation renders Coq texts that are relatively easy to read, and intuitive proofs. The obvious problem with this approach is of course its soundness (and completeness). However, the soundness of the encoding approach is also not immediate, as it is not even proved yet that Coq is consistent [CoP90, PaM93], i.e., `False` might be derivable. In fact, the problem lies in the inductive sets and definitions, on which the encoding relies much more than our embedding approach. Clearly, any such soundness result cannot be obtained as long as this consistency of Coq is not proved.[5]

So the axioms of μCRL are translated to axioms in Coq; inference rules (e.g. SUM11) become implications (see Section 3.4 for the details). Also the rewrite rules of a μCRL-specification are translated to axioms, which is justified by FACT. Is the consistency of Coq in the empty state already unproven, adding axioms makes it even harder to prove consistency. One might therefore argue that a better way to proceed would be to define the proposition `muCRL` as the conjunction of its axioms and rules (which can be done conveniently by an inductive definition), and to use that as a premise to all lemmas and theorems. We feel that this approach does not add any confidence in the results: the question remains if this proposition `muCRL` entails `False` in Coq. From a practical point of view, the approach makes proofs much harder to read because the names of the axioms are lost.

There are some obvious mismatches between Coq and μCRL to take care of. The most obvious mismatch occurs between the classical implication of μCRL and the constructive implication of Coq. In this case the rules of μCRL are

[5] We have been informed recently that the required result was obtained in [Wer94] for a subset of Coq that includes the techniques used in this paper.

stronger than those of Coq, so soundness is not at stake. We could have added the axiom `(P:Prop)~~P->P`, but it turned out that we did not need it.

Another potential source of problems is equality. The equality `<_>_=_` of Coq has the Leibniz property, i.e., two terms are equal if and only if they can be substituted for each other in every context of type `Prop`. This is a strong requirement, as these contexts are built from the expressive language of Coq. Whether $=$ in μCRL can be interpreted conservatively as Leibniz equality in Coq is a subject for specialized study, see [Sel96] for a partial answer.

Finally, μCRL has no explicit quantification, but instead the substitution rule. This rule entails that all variables are implicitly universally quantified. These quantifiers are made explicit in our translation. Yet not all variables in μCRL are bound in this way: the sum operator $\sum_{d:D}(x)$ binds the variable d of datatype D in x. We translate this binding to lambda abstraction, see Section 3.4 for the details.

3.2. Data

A significant part of the proof theory of μCRL can be translated to Coq independently of a particular μCRL-specification. Only the set of action names, the communication function γ, and the set of sorts parameterize this translation. The two sets are finite; therefore we define them as `Inductive Sets`[6], simply enumerating the members. These are the only `Inductive Sets` we use. From these definitions it is easy to prove that all actions, respectively sorts, are different (we need inequality of sorts to verify the side-condition of axiom CF2$''$).

For simplicity, we allow actions to have precisely one data argument. For actions that have more than one parameter in the specification, pairing can be used. Actions without parameter get the dummy argument `i`, which is the only element of the trivial sort one. Thus a translation of a μCRL-specification begins with the following definition, where the ... must be replaced by sorts specific for the specification. Why the sort *nat* of naturals is needed is explained in Section 3.7.

```
Inductive Set types = onetype:types | booltype:types | nattype:types | ...
```

In fact, this declaration gives us sort *names*. The sorts themselves are created through the declaration of a function `type : types->Set`.

```
Parameter type : types->Set.

Definition one  = (type onetype).
Definition bool = (type booltype).
Definition nat  = (type nattype).
```

A consequence of this approach is that we cannot define these sorts inductively. Thus we must declare the constructors and induction principles for these sorts explicitly. We can also not use the `Match`-function, therefore we must axiomatize the functions `zero` and `pred`, which allow us to prove that naturals of the form $S^n(0)$ differ for different n.[7]

[6] In Section 3.4 we explain why we cannot identify sorts from μCRL with sorts in Coq.
[7] Alternatively, we could postulate a bijection between the sort `nat` as defined here and inductively defined naturals. Section 5.4 might be simplified by the resulting ability to use the `Match`-function.

```
Parameter i          : one.
Parameter true,false : bool.
Parameter 0          : nat.
Parameter S          : nat->nat.

Axiom I1     : (j:one)  <one> j=i.
Axiom B1     :           ~<bool>true=false.
Axiom B2     : (b:bool) <bool>b=true \/ <bool>b=false.
Axiom nat_ind: (P:nat->Prop)(n:nat) (P 0)->((y:nat)(P y)->(P (S y)))->(P n).

Parameter zero : nat->bool.
Parameter pred : nat->nat.

Axiom zero0:          <bool>(zero 0    )=true.
Axiom zeroS: (n:nat) <bool>(zero (S n))=false.
Axiom pred0:          <nat> (pred 0    )=0.
Axiom predS: (n:nat) <nat> (pred (S n))=n.
```

As we noted, μCRL has two equalities: the 'built-in' = for both data (**rew**) and processes (**proc**), and the user-defined $eq_D : D \to D \to \mathbf{Bool}$ for each sort D. We chose not to translate eq_D into Coq by literally translating the rewrite rules of Example 2.1, but by defining it by its intended meaning, namely part 1 of Claim 2.2.

```
Parameter eql: (T:types)(type T)->(type T)->bool.
Axiom def_eql: (T:types)(d,e:(type T)) <bool>(eql T d e)=true<-><(type T)>d=e.
```

3.3. Actions and Communication

Actions in μCRL are declared with their respective sorts, but overloading of action names is allowed: one may declare an action r with sort D and another action r with a different sort E. In the translation into Coq, actions are declared without their sorts (in other words: action *names* are declared). Thus there can be actions in the translation that are not present in the original specification. As these actions will not occur in the processes, this mismatch is harmless.

The **comm** section of a μCRL specification, defining the communication function γ of ACP$^\tau$, is translated to the function gamma in Coq. Recall that communication in μCRL is defined on action names only, that is, if two actions (of different sort) have the same name, then they must communicate in the same way. This facilitates a correct translation into Coq: gamma is specified only for the action name r, not for 'r:D' and 'r:E' separately. It is not easy to specify partial functions in Coq, therefore when $\gamma(a, b)$ is undefined, its translation (gamma a b) returns the special action name delta. The process τ in μCRL behaves similarly to an atomic action, so a second special action name tau is introduced.

Thus, we expect the translation of a μCRL-specification to provide definitions

```
Inductive Set act = ... | delta:act | tau:act.

Definition gamma: act->act->act = ...
```

gamma must have certain properties, which are stated as five proof obligations (goals) in Coq. We must prove these goals in order to show that gamma satisfies the desired properties. These properties can be used as lemmas in the correctness proof of the protocol as well. The first two properties are that delta and tau do not communicate. The third is that the communication of two actions is not τ

(allowing this would complicate defining guardedness, see Section 3.6). The fourth is that gamma is commutative, as is required in [BaW90]. It is also required there that gamma is associative, but we assumed handshaking, the fifth property, which is stronger.

```
Goal (a    :act) <act>(gamma delta a        )=delta.
Goal (a    :act) <act>(gamma tau   a        )=delta.
Goal (a,b  :act)~<act>(gamma a     b        )=tau.
Goal (a,b  :act) <act>(gamma a     b        )=(gamma b a).
Goal (a,b,c:act) <act>(gamma a    (gamma b c))=delta.
```

In general, the proof of these goals depends on the definition of gamma. However, thanks to the fact that actions are defined inductively, we can use the Match-function for this definition (see Section 4 for an example). With such a definition, proving these goals becomes automatic: the literal text of the proofs need not depend on gamma; it is always a straightforward case analysis.

3.4. Processes and Axioms

The distinction between the action *a* and the process *a* is not always obvious in process algebra. In the current setting, it is obvious that a process is formed from an action name, its sort, and an element of that sort. However, there is only one process δ and one process τ. Thus we declare

```
Parameter proc   : Set.
Parameter ia     : (T:types) act->(type T)->proc.

Definition Delta  = (ia onetype delta i).
Definition Tau    = (ia onetype tau   i).

Axiom Delta_Data : (T:types)(t:(type T)) <proc>Delta=(ia T delta t).
Axiom Tau_Data   : (T:types)(t:(type T)) <proc>Tau  =(ia T tau   t).
```

It remains to model sets of actions (for hiding and encapsulation), before we declare the operators on processes. Similar to the relation *R* in Example 2.5, we model such sets by their characteristic function act->Prop[8]. A small complication is that we have added delta and tau to the set of actions, and that these cannot be encapsulated, nor hidden. Thus we define the function goodset, which, given a set of actions, returns the same set without delta and tau.

```
Definition ehset   = act->Prop.
Definition goodset : ehset->ehset = [L:ehset]
                     [a:act] (~(<act>a=delta))/\(~(<act>a=tau))/\(L a).

Parameter alt,seq,mer,Lmer,comm :                  proc->proc->proc.
Parameter cond                  :            proc->bool->proc->proc.
Parameter sum                   : (T:types) ((type T)->proc)->proc.
Parameter enc,hide              :                 ehset->proc->proc.
```

[8] Sellink [Sel93] suggests to represent the sets for hiding and encapsulation as lists. This turns out to be unnecessary cumbersome, but raises an interesting question. Suppose that we have sets as a sort in the specification of the protocol. Then the μCRL-specification contains an algebraic specification of sets based on lists, such as the one given by Groote and Van Wamel [GrW94] (a function $D \to$ **Bool** can be declared in μCRL, but not used as a sort). Is it allowed in this case to use the characteristic function representation, or should we translate the algebraic list-based specification dutifully into Coq? The latter is more formal, but further away from the informal specification, which requires sets. Notice that this problem does not occur for the sets of actions for encapsulation and hiding, as these sets are not sorts, but built-in syntactic objects in μCRL.

Now it is clear why we cannot identify μCRL-sorts with sorts in Coq: proc could then be used as a μCRL-sort. This would again allow the process definition $P = (a \triangleleft P = \delta \triangleright \delta)$, which implies $a = \delta$, and also $\sum_{x:\texttt{proc}} x$, the sum of all processes.

The μCRL term $\sum_{d:T}(x)$ is translated to (sum T [d:(type T)]x). This means that sum has type (T:types)((type T)->proc)->proc. The axiom SUM2 of μCRL is now recognised as α-conversion, and can therefore be omitted in the translation. The freeness requirements of the variables in the other SUM-axioms are verified automatically: if they are not satisfied, then an unbound variable would occur. The premiss of SUM11 refers to the equality of two processes with a free variable $d : D$; it is translated to $\forall d \in D : p_1(d) = p_2(d)$.

Most of the axioms of μCRL translate directly into Coq, as they are simply equations between processes; variables are universally quantified. For example, A1 translates to

```
Axiom A1:(x,y:proc)<proc>(alt x y)=(alt y x).
```

The derivable axioms SC4, CM5, CM6, and CM9 are not translated to axioms, but to lemmas. Some axioms have side-conditions, most notably the CF-axioms, D1, D2, TI1 and TI2. The CF-axioms have been simplified in comparison with Table 3.

```
Axiom CF1 : <proc> (cond (ia T (gamma a b) t) (eql T t t') Delta)=
                                        (comm (ia T a t) (ia T b t')).
Axiom CF2 : ~<types>T=U -> <proc> Delta=(comm (ia T a t) (ia U b u )).
```

CF1 covers not only the case of actual communication (CF1 in Table 3), but also the case where communication fails because the actions do not communicate or the data is not the same (CF2 and CF2′). Claim 2.2 or the axiom def_eql justifies this formulation, which effectively replaces the premiss $\neg(t_i = t_i')$ of CF2′ by $eq_T(t_i, t_i') = F$. The only remaining case is that of CF2″: actions with different sorts (and hence incomparable data), which is covered by CF2.

Apart from the axioms listed, there are many 'derived axioms' or lemmas. These are discussed in Section 3.8.

3.5. Recursive Specifications and RDP

Informally, RDP states that a recursive specification has at least one solution. Thus we need to translate what is a recursive specification, and what is a solution of it. First, we consider the case of a single recursive equation. Such an equation, written as $X(d) = G(X,d)$, can be seen as the definition of the *process operator* G of type (D->proc)->D->proc. (This is a generalization of the *linear* process operators of [BeG94b], where G must be in a particular normal form.) A solution of the recursive equation is then a fixed point of G, and has type D->proc.

In the general case, we have a set of process variables ProcVar and a function Typ from ProcVar to types giving their associated sorts (similar to actions, we let process variables have exactly one data parameter). A solution of a system of recursive equations is now a function that interprets each process variable as a function from its data parameter to a process, thus the type of a solution (in fact, of any such interpretation) is Inttype = (X:ProcVar)(type (Typ X))->proc. The system of recursive equations DefEq itself is then a process operator Inttype->Inttype (similar to G above). The solution is its fixed point.

For example, the system $\{X = a \cdot Y(T), \quad Y(b : \mathbf{Bool}) = X + a \cdot Y(not(b))\}$ is defined as follows (note that DefEq needs the old interpretation of process variables iPV to interpret the occurrence of a process variable in the body of an equation as a process).

```
Inductive Definition ProcVar = X:ProcVar | Y:ProcVar.
Definition Typ    = [P:ProcVar] (<types>Match P with (* X *) onetype
                                                     (* Y *) booltype).
Definition Inttype = (P:ProcVar)(type (Typ P))->proc.
Definition DefEq = [iPV:Inttype][P:ProcVar]
                   (<[P:ProcVar](type (Typ P))->proc>Match P with
                   (* X *) [j:one ](seq (ia onetype a i)
                                        (iPV Y true)    )
                   (* Y *) [b:bool](alt (iPV X i)
                                        (seq (ia onetype a i)
                                             (iPV Y (neg b)) ))).
```

RDP states that a system of recursive equations has a solution, i.e., that a process operator has a fixed point. Thus we declare the solution function Sol:(Inttype->Inttype)->Inttype giving a solution for each system of equations (think of it as the μ-operator). That (Sol DefEq) is indeed a solution for DefEq is stated in axiom RDP. (A Variable declaration is local within a Section; it is translated to a universal quantification outside.)

```
Section RDP.
Variable   ProcVar : Set.
Variable   Typ     : ProcVar->types.
Local      Inttype = (X:ProcVar)(type (Typ X))->proc.
Variable   DefEq   : Inttype->Inttype.

Parameter Sol : (Inttype->Inttype)->Inttype.
Axiom RDP     : <Inttype>(Sol DefEq)=(DefEq (Sol DefEq)).
End RDP.
```

3.6. RSP

RSP states that guarded systems of equations have unique solutions. So we must define guardedness in Coq. A single recursive equation is guarded if we can determine for all n the first n visible actions of its solution by repeatedly unfolding the equation. For example, if we have $X(b : \mathbf{Bool}) = (\tau \lhd b \rhd a) \cdot X(not(b))$, then $X(T) = \tau \cdot X(F) = \tau \cdot a \cdot X(T)$, so we can determine the first visible action (a) of $X(T)$ by unfolding the equation twice. Further applications of the equation give us further visible actions: the equation is guarded.

In contrast, if we have $Y = a \cdot \tau_{\{a\}}(Y)$, then this equation gives us the first visible action, but a second unfolding yields $Y = a \cdot \tau_{\{a\}}(a \cdot \tau_{\{a\}}(Y)) = a \cdot \tau \cdot \tau_{\{a\}}(\tau_{\{a\}}(Y)) = a \cdot \tau_{\{a\}}(Y)$. Clearly, further unfoldings do not yield further visible actions for Y, so this equation is unguarded. Indeed, both a and $a \cdot \delta$ are solutions for this equation, thus RSP should not be applicable. In view of this second example, we will simply consider every recursive equation in which the hiding operator[9] occurs as unguarded (unless of course we can remove the hiding operator by rewriting the system using the axioms).

Now we return to the first example. We note that when we unfold $X(T)$, we

[9] Allowing $\gamma(a, b) = \tau$ would give similar problems for $\|$, $|$ and $\lfloor\!\lfloor$, consider e.g. $Z = a \cdot (b \mid Z)$.

obtain $X(F)$ without a visible action (*guard*) in front. We say that $X(T)$ *depends unguarded* on $X(F)$. On the other hand, unfolding $X(F)$ yields $X(T)$ only behind a guard, so $X(F)$ does not depend unguarded on $X(T)$. We can have the same notion in a system of equations: if we replace $X(T)$ by Y and $X(F)$ by Z then we obtain the system $\{Z = \tau \cdot Y, Y = a \cdot Z\}$ in which Z depends unguarded on Y, but Y does not depend unguarded on Z.

We conclude that 'depends unguarded on' is a binary relation R on pairs of the form (X, e), where X is a process variable and e is data of the correct type for X. R must be well-founded for the system to be guarded.[10] Rather than writing an axiomatization that tries to compute R, we let the user provide R. Then we check that R is well-founded (see also [BeG94c]) and that for all process variables X and data e of the type for X, the body of the equation for $X(e)$ is *safe* w.r.t. X, e, and R, that is, if $Y(f)$ occurs in this body, either it occurs behind a guard, or $R(X, e, Y, f)$ holds. What follows is the translation of this into Coq; the details are explained thereafter.

```
Parameter Safe : (ProcVar:Set)
                  (Typ:ProcVar->types)
                   (iPV:(X:ProcVar)(type (Typ X))->proc)
                    (X:ProcVar)
                     (e:(type (Typ X)))->
                      ((X:ProcVar)(type (Typ X))->
                       (Y:ProcVar)(type (Typ Y))->Prop)->proc->Prop.

Section RSP.
Variable ProcVar  : Set.
Variable Typ      : ProcVar -> types.
Local    typ      = [X:ProcVar](type (Typ X)).
Local    Inttype  = (X:ProcVar)(typ X)->proc.
Local    RT       = (X:ProcVar)(typ X) -> (Y:ProcVar)(typ Y) -> Prop.
Variable iPV      : Inttype.
Variable DefEq    : Inttype->Inttype.
Variable X        : ProcVar.
Variable e        : (typ X).
Variable R        : RT.

Local RSafe : proc->Prop = (Safe ProcVar Typ iPV X e R).
Local TSafe : proc->Prop = (Safe ProcVar Typ iPV X e
                                 [X:ProcVar][e:(typ X)]
                                 [Y:ProcVar][f:(typ Y)]True).

Variable x,y : proc.
Variable T   : types.

Axiom S0:(Y:ProcVar)(f:(typ Y))     (R X e Y f) -> (RSafe (iPV Y f) ).
Axiom S1:(a:act)(t:(type T))                       (RSafe (ia T a t)).
Axiom S2:(a:act)(t:(type T))
                      ~(<act>a=tau) -> (TSafe y) -> (RSafe (seq (ia T a t) y).
Axiom S3:                 (RSafe x) -> (RSafe y) -> (RSafe (seq  x y)).
Axiom S4:                 (RSafe x) -> (RSafe y) -> (RSafe (alt  x y)).
Axiom S5:                 (RSafe x) -> (RSafe y) -> (RSafe (mer  x y)).
Axiom S6:                 (RSafe x) -> (RSafe y) -> (RSafe (Lmer x y)).
Axiom S7:                 (RSafe x) -> (RSafe y) -> (RSafe (comm x y)).
Axiom S8:(p:(type T)->proc)((d:(type T)) (RSafe (p d)))
                                            -> (RSafe (sum  T p)).
Axiom S9:(L:ehset)                (RSafe x) -> (RSafe (enc  L x)).
```

[10] Apart from cyclic ones, this also excludes unfounded specifications like $X(n : nat) = X(S(n))$.

```
Variable ProcVar' : Set.
Variable Typ'     : ProcVar' -> types.
Local    typ'     = [X':ProcVar'](type (Typ' X')).
Local    Inttype' = (X':ProcVar')(typ' X')->proc.

Local TSafe' = [iPV':Inttype']
               (Safe ProcVar' Typ' iPV' X' e'
                     [X':ProcVar'][e':(typ' X')]
                     [Y':ProcVar'][f':(typ' Y')]True).

Axiom S10:(DefEq':Inttype'->Inttype')(X':ProcVar')(e':(typ' X'))
         ( (iPV':Inttype')(X':ProcVar')(d':(typ' X'))
                     (TSafe' iPV' (DefEq' iPV' X' d')) )->
         (RSafe (Sol ProcVar' Typ' DefEq' X' e')).
```

S0 states that $Y(f)$ can occur unguarded in the defining equation of $X(e)$, provided $R(X,e,Y,f)$ holds. S2 states that all process variables may occur after a guard; the effect is obtained by replacing R by the relation that is always true. The premiss (TSafe y) serves to check that no hiding operator occurs in y.

S10 states that the system may refer to another system of equations. This other system must be proved safe[11] w.r.t. the relation that is always true, i.e. it must not contain hiding and, more importantly, it must not contain variables of the current system (technically: the defining equations DefEq' of this new system must not depend on the iPV of the current one). For example, following the notation of [BW90], we could have $E = \{X = a \cdot \langle X' \mid E'_X \rangle\}$, with $E'_X = \{X' = X + b \cdot X'\}$. Notice that in μCRL we cannot distinguish this combination from the flattened system $\{X = a \cdot X', \quad X' = X + b \cdot X'\}$, but that we need the distinction to modularize proofs.

One can observe that the above combination of E and E'_X is in fact safe, because the flattened system is. Indeed, we can allow the stronger variant of axiom S10 below, which allows the occurrence of those variables $Y(f)$ that were allowed to occur unguarded anyway in the equation for $X(e)$, because $R(X,e,Y,f)$ holds. It does however not change R to the relation that is always true after encountering a guard. Anyway, we do not need this stronger version of S10 if we only build one system on top of the other, instead of mutually recursive ones.

```
Axiom S10:(DefEq':Inttype'->Inttype')(X':ProcVar')(e':(typ' X'))
     ( (iPV':Inttype')(X':ProcVar')(d':(typ' X'))
       ( (Y:ProcVar)(f:(typ Y)) (R X e Y f)->(TSafe' iPV' (iPV Y f)) )->
                                     (TSafe' iPV' (DefEq' iPV' X' d')) )->
     (RSafe (Sol ProcVar' Typ' DefEq' X' e')).
```

Finally, we can state the axiom RSP. Given are an interpretation of process variables iPV, the system of equations DefEq and the relation R. The system is guarded if R is well-founded and all bodies are safe (for no X and d, there is an infinite descending chain from X and d, and the body of the equation for X and d

[11] We need not prove that this other system is guarded! If it is not, then it will not have a unique solution, but the unique solution of the current system will contain the (not uniquely determined) term (Sol ProcVar' Typ' DefEq' X' e').

is safe). If the system is guarded and `iPV` is indeed a solution[12], then `iPV` equals the canonical solution `(Sol ProcVar Typ DefEq)` of the system.

```
Inductive Definition WF : (X:ProcVar)(typ X)->Prop =
WF1: (X:ProcVar)(d:(typ X))
        ((Y:ProcVar)(e:(typ Y))(R X d Y e)->(WF Y e))
     -> (WF X d).

Definition Guarded = (X:ProcVar)(d:(typ X))(iPV:Inttype)
(WF X d) /\ (Safe ProcVar Typ iPV X d R (DefEq iPV X d)).

Axiom RSP:
Guarded ->
((X:ProcVar)(d:(typ X))<proc> (iPV X d) = (DefEq iPV  X d)) ->
<Inttype> iPV = (Sol ProcVar Typ DefEq).

End RSP.
```

3.7. Fair Abstraction

As we noted before, the ABP can function correctly only if the channels do not corrupt data ad infinitum. This assumption was translated into process algebra in various ways, most notably in the form of fair abstraction rules. For an overview we refer to Section 5.6 of [BaW90]. We chose to translate CFAR^b into Coq (Cluster Fair Abstraction Rule for branching bisimulation, we omit the superscript b further on). Informally, a cluster is a (maximal) set of states of a process such that each state in it can reach each other in it by taking only hidden steps. CFAR deals with all possible clusters, as opposed to KFAR$_n$, which only deals with cycles of n states[13].

We have adapted CFAR to the presence of data as follows. Instead of a single cluster, we like to collaps a number of clusters at the same time. For example, if we have a process definition

$$X(n : nat) = b(n) + i \cdot (X(n+9) \lhd (n \bmod 10) = 0 \rhd X(n-1)),$$

then we want to infer

$$\forall n : nat\ \tau \cdot \tau_{\{i\}}(X(n)) = \tau \cdot (b(10\,(n \operatorname{div} 10)) + \ldots + b(10\,(n \operatorname{div} 10) + 9)).$$

There are infinitely many clusters, therefore we cannot collaps each cluster separately. One way to proceed would be to fix a k : *nat* and to define

$$Y_k(m : [0..9]) = b(10k + m) + i \cdot (Y_k(9) \lhd m = 0 \rhd Y_k(m-1)).$$

Then we prove by CFAR

$$\text{for all } m : [0..9]\colon \tau \cdot \tau_{\{i\}}(Y_k(m)) = \tau \cdot (b(k) + \ldots + b(k+9)).$$

[12] We must put this premiss as `((X:ProcVar)(d:(typ X))<proc>(iPV X d)=(DefEq iPV X d))`, rather than `<Inttype>iPV=(DefEq iPV)`, because the latter equality does not follow from the former in Coq.

[13] As the structure of c and i actions in the ABP turns out not to be a cycle, we need CFAR in our proof. Alternatively, we could hide the c actions first. Then applying T1 yields a cycle of i actions of length 2. Hiding the i actions and applying KFAR$_2$, yields the desired result, provided that we add the axiom $\tau_I(\tau_J(x)) = \tau_{I \cup J}(x)$.

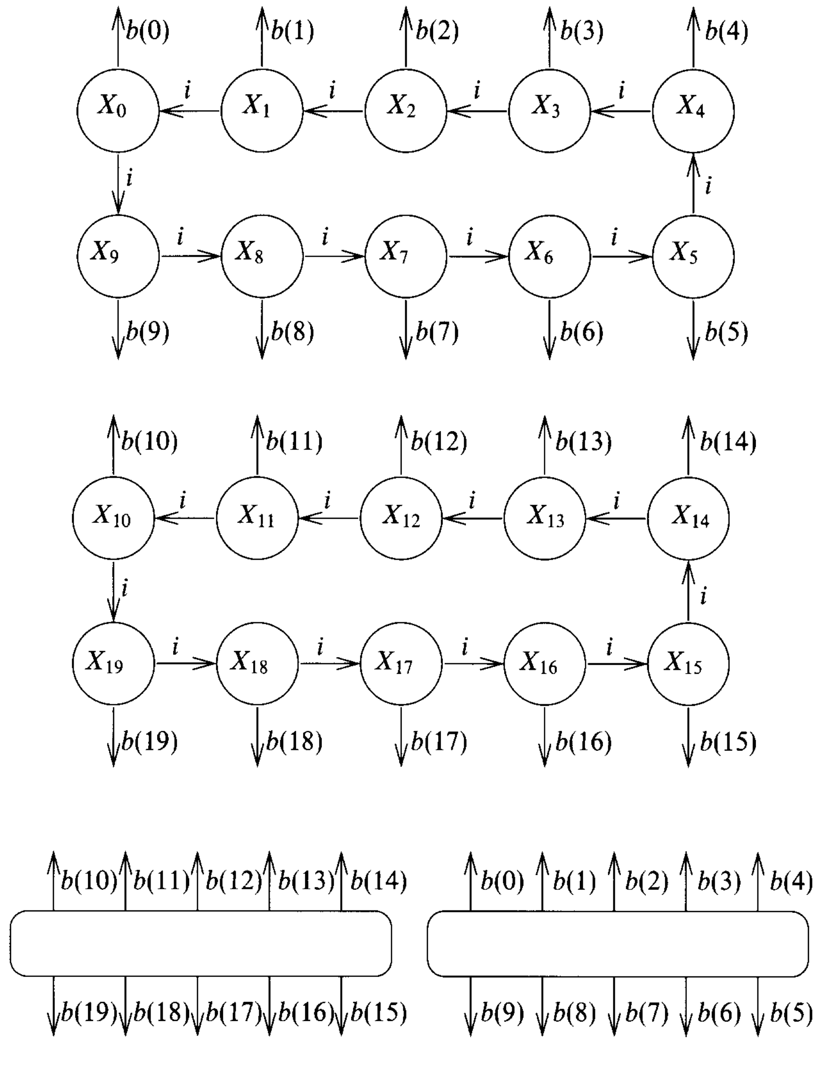

Fig. 2. Collapsing two clusters.

Finally we prove by RSP $X(n) = Y_{n\,\mathrm{div}\,10}(n \bmod 10)$. We cannot formalize this approach in μCRL, because there k should be a formal parameter of Y, leaving us with many clusters again. However, our translation of recursive specifications into Coq does not prevent parameterized specifications such as the one of Y_k: we can encode this approach in Coq, albeit clumsily (we must add a new datatype with ten elements and a function interpreting them as 0..9).

Therefore we chose a formulation of CFAR that collapses multiple clusters explicitly. First we number the different clusters. Then we number the different pairs (X, d) within each cluster, where X is a process variable and d a data parameter of the type of X. That is, we assume having the following functions.

- *cluster*(X, d) gives the number of the cluster to which the pair (X, d) belongs.

- *element*(X,d) gives the order number of (X,d) within its cluster.
- *process*(n,m) $(n,m \in nat)$ returns $X(d)$ such that *cluster*$(X,d) = n$ and *element*$(X,d) = m$. It returns δ if $n \geq$ the number of clusters or $m \geq$ the number of processes in the cluster.
- *Exit*(n,m) $(n,m \in nat)$ returns the exit process of the mth item in the nth cluster. Again it is δ if n or m are too large.
- $a(X,d,m)$ is the action (including data) that leads from $X(d)$ to the mth item in the cluster of $X(d)$. It is δ if there is no such action.

In our translation into Coq, the user must provide these functions for each application of CFAR, and show that they have the following properties (let L be the set of actions going to be hidden).

1. For all X and d: $X(d) = process(cluster(X,d), element(X,d))$.
2. For all n and m: if for no X and d: $(n,m) = (cluster(X,d), element(X,d))$, then $Exit(n,m) = process(n,m) = \delta$.
3. The system of equations can be written in the form[14]
$$X(d) = \begin{array}{l}\sum_{m:nat} a(X,d,m) \cdot process(cluster(X,d),m) + \\ Exit(cluster(X,d), element(X,d)).\end{array}$$
4. Each $a(X,d,m)$ is either δ, τ, or its action name is in L.
5. All clusters are connected: we can step from $X(d)$ to $Y(e)$ exactly if the action $a(X,d,element(Y,e)) \neq \delta$; a cluster is connected if for all $X(d)$ and $Y(e)$ in it, we can go from $X(d)$ to $Y(e)$ in one or more steps.
6. The system is guarded.

Given definitions satisfying these properties, CFAR concludes for all X and d:

$$\tau \cdot \tau_L(X(d)) = \tau \cdot \tau_L(\sum_{m:nat} Exit(cluster(X,d),m)).$$

In our example, we could use the following functions.

$$\begin{array}{ll} cluster(X,n) & = n \operatorname{div} 10 \\ element(X,n) & = n \bmod 10 \\ process(k,m) & = X(10k+m) \text{ if } m \leq 9, \quad \delta \text{ otherwise} \\ Exit(k,m) & = b(10k+m) \text{ if } m \leq 9, \quad \delta \text{ otherwise} \\ a(X,n,m) & = i \text{ if } m = (n-1) \bmod 10, \quad \delta \text{ otherwise.} \end{array}$$

We now provide the representation of CFAR in Coq. Notice that `process` needs an interpretation of process variables, and that the definition of $a(X,d,m)$ is split in three parts: sort, action name, and data.

```
Section CFAR.
Variable ProcVar : Set.
Variable Typ     : ProcVar -> types.
Local    typ     = [X:ProcVar](type (Typ X)).
Local    Inttype = (X:ProcVar)(typ X)->proc.
Variable DefEq   : Inttype->Inttype.
Variable R       : (X:ProcVar)(typ X)->(Y:ProcVar)(typ Y)->Prop.
Variable L       : ehset.
```

[14] Here we see a summation over the natural numbers. Since we have only summation over sorts, we need *nat* as a built-in sort.

```
Variable cluster : (X:ProcVar)(typ X) -> nat.
Variable element : (X:ProcVar)(typ X) -> nat.
Variable process : Inttype -> nat -> nat -> proc.
Variable Exit    : nat -> nat -> proc.
Variable D'      : (X:ProcVar)(typ X) -> nat -> types.
Variable a       : (X:ProcVar)(typ X) -> nat -> act.
Variable d'      : (X:ProcVar)(d:(typ X))(n:nat) (type (D' X d n)).

Definition CheckInside = (X:ProcVar)(d:(typ X))(iPV:Inttype)
(*1*) <proc>(process iPV (cluster X d) (element X d)) = (iPV X d).

Definition CheckOutside = (n,m:nat)(iPV:Inttype)
(*2*) ((X:ProcVar)(d:(typ X)) ~(<nat>n=(cluster X d) /\
                                <nat>m=(element X d)   )) ->
      <proc>(process iPV n m)=Delta /\ <proc>(Exit n m)=Delta.

Definition CheckDef = (X:ProcVar)(d:(typ X))(iPV:Inttype)
(*3*) <proc>(DefEq iPV X d)=
            (alt (sum nattype [n:nat](seq (ia (D' X d n) (a X d n) (d' X d n))
                                          (process iPV (cluster X d) n)))
                 (Exit (cluster X d) (element X d))).

Definition Checka = (X:ProcVar)(d:(typ X))(n:nat)
(*4*) <act>(a X d n)=delta \/ <act>(a X d n)=tau \/ (goodset L (a X d n)).

Inductive Definition Conn : (X,Y:ProcVar)(typ X)->(typ Y)->Prop
= conn1: (X,Y:ProcVar)(d:(typ X))(e:(typ Y))
          ~<act>(a X d (element Y e))=delta -> (Conn X Y d e)
| connt: (Z:ProcVar)(f:(typ Z))
         (X,Y:ProcVar)(d:(typ X))(e:(typ Y))
          (Conn X Z d f) -> (Conn Z Y f e)  -> (Conn X Y d e).

Definition CheckConn = (X,Y:ProcVar)(d:(typ X))(e:(typ Y))
(*5*) <nat>(cluster X d)=(cluster Y e) -> (Conn X Y d e).

Axiom CFAR: (X:ProcVar)(d:(typ X))
      CheckInside -> CheckOutside -> CheckDef -> Checka -> CheckConn ->
(*6*) (Guarded ProcVar Typ DefEq R) ->
      <proc>(seq Tau (hide L (Sol ProcVar Typ DefEq X d)))=
            (seq Tau (hide L (sum nattype [n:nat](Exit (cluster X d) n)))).
End CFAR.
```

How we use this formulation of CFAR in proving the correctness of the ABP is outlined in Section 5.4.

3.8. A Library of Lemmas

Although the axioms and rules are the most important part of the translation of μCRL into Coq, it would be incomplete without a library of lemmas that are useful regardless of the protocol being verified. The current library is listed in Tables 4, 5, and 6; this library will grow further when more protocols are verified. We distinguish the following parts of our library.

- Lemmas about standard data: the sorts *nat* and **Bool**, and equality. These lemmas are typically trivial, requiring only a few lines of proof. Nevertheless they are necessary to automate parts of the further proof. See Table 4.
- A few short lemmas about actions. See Table 4.

Table 4. Booleans, equality, naturals and actions.

negfalse	$neg(F) = T$	refl_eql	$eq_D(t,t) = T$
negtrue	$neg(T) = F$	sym_eql	$eq_D(t,u) = eq_D(u,t)$
negneg	$neg(neg(b)) = b$	make_equal	$t = u \to eq_D(t,u) = T$
not_eql_true_false	$eq_{\mathbf{Bool}}(F,T) = F$	make_eql	$t \neq u \to eq_D(t,u) = F$
not_eql_b_negb	$eq_{\mathbf{Bool}}(b, neg(b)) = F$	make_uneql	$eq_D(t,u) = F \to t \neq u$
O_S	$S(n) \neq 0$	not_goodset	$a \notin L \to a \notin goodset(L)$
unequal_S	$n \neq m \to S(n) \neq S(m)$	comm_action	$\exists c : a(t) \mid a'(u) = c(t)$

$b \in \mathbf{Bool}$, D a sort, $t, u \in D$, $m, n \in nat$, $a, a', c \in Act \cup \{\delta, \tau\}$

Table 5. Derived axioms.

A6′	$\delta + x = x$	SC6	$x \| y = y \| x$
D1_Delta	$\partial_L(\delta) = \delta$	SC7	$(x \| y) \| z = x \| (y \| z)$
TI1_Delta	$\tau_L(\delta) = \delta$	DC2	$x \mid \delta = \delta$
CM2′	$\delta \mathbin{\|\!\lfloor} x = \delta$	Handshaking′	$(x \mid y) \mid z = \delta$

SUM7′	$\sum_{e:E}(x \mid y) = x \mid \sum_{e:E} y$	if e not free in x
SUM7″	$\sum_{d:D} \sum_{e:E}(x \mid y) = \sum_{d:D} x \mid \sum_{e:E} y$	if e not free in x and d not free in y
DLCSS	$\sum_{d:D} \sum_{e:E} \partial_L((x \mid y) \mathbin{\|\!\lfloor} z) = \partial_L((\sum_{d:D} x \mid \sum_{e:E} y) \mathbin{\|\!\lfloor} z)$	if e not free in x and z and d not free in y and z
SUMmand	$\sum_{d:D} x = x[d'/d] + \sum_{d:D}(\delta \lhd eq_D(d,d') \rhd x)$	
EXP_bool	$x[b/c] + x[neg(b)/c] = \sum_{c:\mathbf{Bool}} x$	
EXP3	$x \| (y \| z) = x \mathbin{\|\!\lfloor} (y \| z) + y \mathbin{\|\!\lfloor} (x \| z) + z \mathbin{\|\!\lfloor} (x \| y) + (y \mid z) \mathbin{\|\!\lfloor} x + (x \mid y) \mathbin{\|\!\lfloor} z + (x \mid z) \mathbin{\|\!\lfloor} y$	
EXP4	$x \| (y \| (z \| u)) = x \mathbin{\|\!\lfloor} (y \| (z \| u)) + y \mathbin{\|\!\lfloor} (x \| (z \| u)) + z \mathbin{\|\!\lfloor} (x \| (y \| u)) + u \mathbin{\|\!\lfloor} (x \| (y \| z))$ $+ (z \mid u) \mathbin{\|\!\lfloor} (x \| y) + (y \mid z) \mathbin{\|\!\lfloor} (x \| u) + (y \mid u) \mathbin{\|\!\lfloor} (x \| z)$ $+ (x \mid y) \mathbin{\|\!\lfloor} (z \| u) + (x \mid z) \mathbin{\|\!\lfloor} (y \| u) + (x \mid u) \mathbin{\|\!\lfloor} (y \| z)$	
COND3	$x = x \lhd b \rhd x$	
COND4	$x \lhd b \rhd y = y \lhd neg(b) \rhd x$	
COND5	$(x \otimes z) \lhd b \rhd (y \otimes z) = (x \lhd b \rhd y) \otimes z$	
COND5′	$(x \otimes y) \lhd b \rhd (x \otimes z) = x \otimes (y \lhd b \rhd z)$	
COND6	$(x \lhd b \rhd z) + (y \lhd b \rhd z) = (x + y) \lhd b \rhd z$	
COND6′	$(z \lhd b \rhd x) + (z \lhd b \rhd y) = z \lhd b \rhd (x + y)$	
COND7	$b = c \to x \lhd b \rhd z = x \lhd b \rhd (y \lhd c \rhd z)$	
COND7′	$b = c \to y \lhd b \rhd x = (y \lhd c \rhd z) \lhd b \rhd x$	
COND8	$b = neg(c) \to x \lhd b \rhd y = x \lhd b \rhd (y \lhd c \rhd z)$	
COND8′	$b = neg(c) \to z \lhd b \rhd x = (y \lhd c \rhd z) \lhd b \rhd x$	
COND9	$\sum_{d:D}(x \lhd b \rhd y) = (\sum_{d:D} x) \lhd b \rhd y$	if d not free in y
COND9′	$\sum_{d:D}(x \lhd b \rhd y) = x \lhd b \rhd (\sum_{d:D} y)$	if d not free in x
COND9″	$\sum_{d:D}(x \lhd b \rhd y) = (\sum_{d:D} x) \lhd b \rhd (\sum_{d:D} y)$	
COND10	$\partial_L(x) \lhd b \rhd \partial_L(y) = \partial_L(x \lhd b \rhd y)$	
COND11	$\tau_L(x) \lhd b \rhd \tau_L(y) = \tau_L(x \lhd b \rhd y)$	

$b, c \in \mathbf{Bool}$, D and E sorts, $d, d' \in D$, $e \in E$, $\otimes$ any binary process operator.

- Derived axioms. For example symmetric versions of axioms, like A6′: $\delta + x = x$. A large number of lemmas about the conditional operator can also be derived by a case analysis on the condition being true or false. See Table 5. Proofs are still only a few lines. SUMmand occurs as Lemma 4.3.2 in [GrP93]. EXP_bool is an instance of the final remark of the same lemma.

Table 6. Rules.

Split_alt	$z = x \rightarrow$	$w = y \rightarrow$	$z + w = x + +y$	
RuleA3	$z = x \rightarrow$	$z = y \rightarrow$	$z = y + x$	
RuleA6	$\delta = x \rightarrow$	$z = y \rightarrow$	$z = y + x$	
RuleA6′	$\delta = x \rightarrow$	$z = y \rightarrow$	$z = x + y$	
RuleA7		$\delta = x \rightarrow$	$\delta = x \cdot y$	
ID_enc		$x = y \rightarrow$	$\partial_L(x) = \partial_L(y)$	
RuleD1_delta		$\delta = x \rightarrow$	$\delta = \partial_L(x)$	
RuleTI1_delta		$\delta = x \rightarrow$	$\delta = \tau_L(x)$	
RuleCM2′		$\delta = x \rightarrow$	$\delta = x \mathbin{\|\!\!\lfloor} y$	
RuleSUM1		$x = y \rightarrow$	$x = \sum_{d:D} y$	if d not free in x
RuleSUMrep		$\dfrac{x \lhd eq_D(d,d') \rhd \delta = y}{x = \sum_{d:D} y}$		if d not free in x and the assumptions of the premiss
RuleCOND1		$T = b \rightarrow$	$x = x \lhd b \rhd y$	
RuleCOND2		$F = b \rightarrow$	$y = x \lhd b \rhd y$	
Split_COND	$(eq_D(d,d')$	$= T \rightarrow$	$x = y) \rightarrow$	
		$z = w \rightarrow$	$x \lhd eq_D(d,d') \rhd z = y \lhd eq_D(d,d') \rhd w$	

$b \in$ **Bool**, D a sort, $d, d' \in D$.

- Expansions of the merge, which are a special kind of derived axioms. They are used to determine the first actions of a process defined as the parallel composition of several components. For all n, EXPn is an instance of the *expansion theorem* ([BaW90], Theorem 4.3.5)

$$\begin{aligned} &x_1 \| \dots \| x_n = \\ &\quad \textstyle\sum_{i=1\dots n} x_i \mathbin{\|\!\!\lfloor} (x_1 \| \dots \| x_{i-1} \| x_{i+1} \| \dots \| x_n) + \\ &\quad \textstyle\sum_{i=1\dots n} \sum_{j=i+1\dots n} (x_i \mid x_j) \mathbin{\|\!\!\lfloor} (x_1 \| \dots \| x_{i-1} \| x_{i+1} \| \dots \| x_{j-1} \| x_{j+1} \| \dots \| x_n). \end{aligned}$$

  Note that the summations are actually shorthand for a sequence of alternative compositions. The expansion theorem cannot conveniently be translated in its full generality, i.e., with the number of components n as a parameter. Thus each version must be proved separately, with larger proofs for larger values of n. Another disadvantage is that an expansion makes many copies of the constituing components $x_1 \dots x_n$. A different proof technique avoiding both disadvantages is being developed by Van de Pol [PoS93].
- Axioms restated as rules. The axioms as they are support simplification 'inside out': for proving $y \cdot x = \delta$, we first rewrite y to δ and then apply A7: $\delta \cdot x = \delta$. Often (see Section 5.3) we would like the opposite: first apply `RuleA7`: $y = \delta \rightarrow y \cdot x = \delta$ and then proceed proving the premiss $y = \delta$. Proving these rules is of course trivial. See Table 6.

4. The Translation of the ABP

• sort

Apart from D, *bool_Err*, and *Frame_Err*, we must also declare a sort for $D \times$ **Bool**, which we obviously name *Frame*. Together with the built-in sorts, we get the following definitions.

```
Inductive Set types = onetype:types | booltype:types    | nattype:types |
      Dtype:types | Frametype:types | bool_Errtype:types | Frame_Errtype:types.

Definition D        = (type Dtype).
Definition Frame     = (type Frametype).
Definition bool_Err  = (type bool_Errtype).
Definition Frame_Err = (type Frame_Errtype).
```

• **func** and **rew**

It remains to translate the ABP-specific function declarations and rewrite rules, including those needed because of the introduction of type `Frame` (which also allows a more intuitive formulation of the axiom `same_err_frame`). Note that the defining equation of *neg* in the specification is simple enough to translate it to a `Definition` in Coq, whereas the remaining functions are declared and their defining equations turned into axioms. For constructors (here `pair`, `iFrame`, `errorframe`, `ibool`, and `errorbit`) and projections (`data_of` and `bit_of`) this appears to be the only way.

```
Section ABP_DATA.
Variable b,c:bool.
Variable d  :D.
Variable f,g:Frame.

Parameter pair       :D->bool ->Frame.
Parameter data_of    :   Frame->D.
Parameter bit_of     :   Frame->bool.

Axiom pair_inj: <bool>(eql Frametype f (pair (data_of f) (bit_of f)))=true.
Axiom bit_inj : <bool>(eql booltype  b (bit_of (pair d b)))          =true.
Axiom data_inj: <bool>(eql Dtype     d (data_of (pair d b)))         =true.

Definition neg = [b:bool](eql booltype b false).

Parameter iFrame     :   Frame->Frame_Err.
Parameter errorframe :          Frame_Err.
Parameter ibool      :   bool ->bool_Err.
Parameter errorbit   :          bool_Err.

Axiom same_err_bit   : <bool>(eql booltype           b        c )=
                             (eql bool_Errtype   (ibool b)  (ibool c)).
Axiom find_errorbit  : <bool>(eql bool_Errtype   (ibool b)  errorbit )=false.
Axiom same_err_frame : <bool>(eql Frametype          f          g )=
                             (eql Frame_Errtype (iFrame f) (iFrame g)).
Axiom find_errorframe: <bool>(eql Frame_Errtype (iFrame f) errorframe)=false.
End ABP_DATA.
```

• **act**

When we consider the actions of the ABP, the actions r_1 and s_4 stand out, as there are no communicating s_1 and r_4 actions. Therefore we renamed them to `ain` (input action) and `aout` (output action). We can now drop the indices of the remaining r, s, and c actions, as their sorts differ. The only communication is now $\gamma(r,s) = \gamma(s,r) = c$. Finally we renamed i to `int`, because `i` is already used as the inhabitant of one. Thus we have the following definitions.

```
Inductive Set act =
    ain:act | aout:act | int:act | r:act | s:act | c:act | delta:act | tau:act.

Definition gamma = [e,f:act] (<act>Match e with
    delta delta delta
    (<act>Match f with delta delta delta delta c    delta delta delta)
    (<act>Match f with delta delta delta c    delta delta delta delta)
    delta delta delta).
```

This definition of `gamma` is by case analysis. First, if `e` is `ain`, `aout`, `int`, `c`, delta, or tau, then (gamma e f) is delta. Second, if e is r or s, then (gamma e f) is delta unless f is s respectively r.

• **proc**
As we did earlier in Section 2.3, we add structure to the μCRL-specification by distinguishing four (sub)systems of equations.

1. The buffer, containing only the equation for *Buffer*,
2. the sender, containing the equations for *Sb*, *Sf*, and *Tf*,
3. the receiver, containing only the equation for *Rb*, and
4. the equations for *Sd*, *Rc*, *K*, and *L*.

The equations for *ABP_nohide* and *ABP* are not recursive. Therefore we translated them to Definitions.

```
(* Buffer *)
Inductive Set PVBuf  = Buf : PVBuf.
Definition    TypBuf = [X:PVBuf]onetype.
Definition    BufEq  = [iPV:PVBuf->one->proc][X:PVBuf][j:one]
                         (sum Dtype [d:D](seq (ia Dtype ain d)
                                             (seq (ia Dtype aout d)
                                                (iPV Buf i)      ))).
Definition    Buffer = (Sol PVBuf TypBuf BufEq Buf i).

Section ABPdef.

(* The Sender *)
Inductive Set SendSubState = Sb:SendSubState | Sf:SendSubState | Tf:SendSubState.

Definition SSSTyp = [X:SendSubState](<types>Match X with booltype
                                                         Frametype
                                                         Frametype).

Definition SSSDef = [iPV:(X:SendSubState)(type (SSSTyp X))->proc]
                    [X:SendSubState]
(<[X:SendSubState](type (SSSTyp X))->proc>Match X with
(*Sb *)[b:bool]  (sum Dtype [d:D](seq (ia Dtype ain d) (iPV Sf (pair d b))))
(*Sf *)[f:Frame] (seq (ia Frametype s f) (iPV Tf f))

(*Tf *)[f:Frame] (alt (seq (alt (ia bool_Errtype r errorbit)
                                (ia bool_Errtype r (ibool (neg (bit_of f)))))
                           (iPV Sf f))
                      (ia bool_Errtype r (ibool (bit_of f))))).
(* The Receiver *)
Inductive Set RecSubState = Rb:RecSubState.
Definition RSSTyp = [X:RecSubState]booltype.

Definition RSSDef = [iPV:RecSubState->bool->proc][X:RecSubState]
(*Rb *)[b:bool] (alt (seq (alt (ia Frame_Errtype r errorframe)
                               (sum Dtype [d:D]
                                  (ia Frame_Errtype r (iFrame (pair d b)))))
                          (seq (ia booltype s b) (iPV Rb b)))
                     (sum Dtype [d:D]
                        (seq (ia Frame_Errtype r (iFrame (pair d (neg b))))
                             (seq (ia Dtype aout d) (ia booltype s (neg b)))))).
(* The ABP *)
Inductive Set Components = Sd : Components | Rc : Components |
                           CK : Components | CL : Components.
Definition CompTyp = [X:Components]onetype.
```

```
Variable phase : bool.

Definition CompDef = [iPV:Components->one->proc][X:Components]
(<one->proc>Match X with
(*Sd *)[j:one]  (seq (Sol SendSubState SSSTyp SSSDef Sb phase)
                   (seq (Sol SendSubState SSSTyp SSSDef Sb (neg phase))
                      (iPV Sd i)))
(*Rc *)[j:one]  (seq (Sol RecSubState RSSTyp RSSDef Rb (neg phase))
                   (seq (Sol RecSubState RSSTyp RSSDef Rb phase)
                      (iPV Rc i)))
(*CK *)[j:one]  (sum Frametype [f:Frame]
                   (seq (ia Frametype r f)
                      (alt (seq (ia onetype int i)
                              (seq (ia Frame_Errtype s (iFrame f))
                                 (iPV CK i)))
                           (seq (ia onetype int i)
                              (seq (ia Frame_Errtype s errorframe)
                                 (iPV CK i))))))
(*CL *)[j:one]  (sum booltype [b:bool]
                   (seq (ia booltype r b)
                      (alt (seq (ia onetype int i)
                              (seq (ia bool_Errtype s (ibool b))
                                 (iPV CL i)))
                           (seq (ia onetype int i)
                              (seq (ia bool_Errtype s errorbit)
                                 (iPV CL i)))))) ).

Definition Encaps = [a:act](<Prop>Match a with False False False True
                                               True  False False False).
Definition ABP_nohide = (enc Encaps
                           (mer (Sol Components CompTyp CompDef Sd i)
                           (mer (Sol Components CompTyp CompDef Rc i)
                           (mer (Sol Components CompTyp CompDef CK i)
                                (Sol Components CompTyp CompDef CL i) ))) ).

Definition Hiding = [a:act](<Prop>Match a with False False True  False
                                               False True  False False).
Definition ABP = (hide Hiding ABP_nohide).
End ABPdef.
```

The role of the boolean phase in the equations for Sd and Rc deserves some explanation. Clearly, these equations resemble the equations for $Sd(b : \mathbf{Bool})$ and $Rc(b : \mathbf{Bool})$, with phase in the role of b, more than the parameterless equations for Sd and Rc. However, the type of Sd and Rc is not bool, but one. Thus phase is not the formal translation of the formal parameter b. In fact, we have here the translation of the equation $Sd_b = Sb(b) \cdot Sb(neg(b)) \cdot Sd_b$. In this equation, b is an *informal* parameter in the process algebraic sense; the equation can be seen as shorthand for the two equations $Sd_T = Sb(T) \cdot Sb(neg(T)) \cdot Sd_T$ and $Sd_F = Sb(F) \cdot Sb(neg(F)) \cdot Sd_F$. ABP_nohide and ABP inherit the parameter phase.

5. Proving the Correctness of the ABP in Coq

This section discusses in detail the correctness proof of the ABP in Coq. Significant parts of it become more clear by running Coq (version 5.8.3, which can be obtained by ftp from nuri.inria.fr = 128.93.1.26) on the complete verification, which can be obtained from the authors. The structure of this section is as follows. Section 5.1 gives a few basic lemmas about data and actions in the ABP.

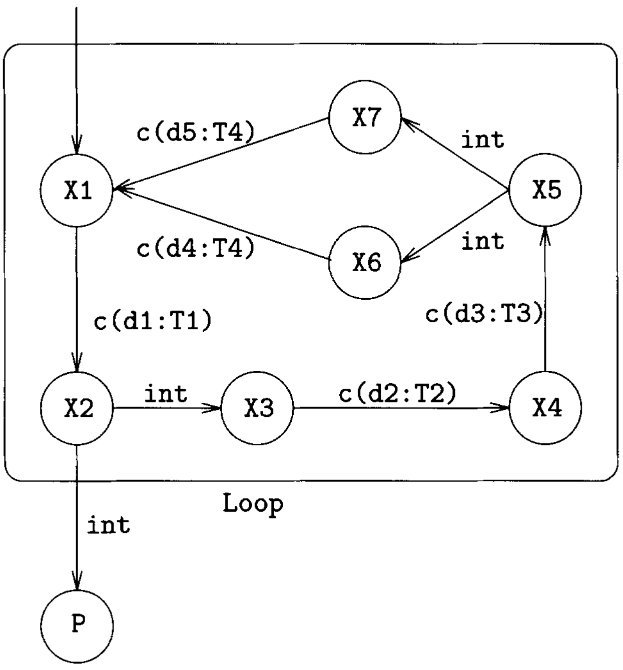

Fig. 3. The generic inner loop.

Section 5.2 corresponds to the definitions preceding Lemma 2.3, and contains preparations for the applications of RSP in its proof. Section 5.3 discusses how we extract the first possible action(s) from a state of the protocol, as is done repeatedly in the proof of Lemma 2.3. Section 5.4 discusses the application of CFAR, which corresponds to the first line of the proof of Theorem 2.4. Finally, Section 5.5 corresponds to the remainder of the proof of Theorem 2.4.

5.1. Data and Actions in the ABP

We proved the following lemmas about the data in the ABP.

```
Section ABP_data.
Variable b,c:bool.
Variable d,e:D.
Variable f  :Frame.

Lemma pair_inj_equal: <Frame>f=(pair (data_of f) (bit_of f)).
Lemma  bit_inj_equal:  <bool>b=(bit_of (pair d b)).
Lemma data_inj_equal:     <D>d=(data_of (pair d b)).

Lemma differ_frame: <bool>(eql    Dtype d e)=false \/
                    <bool>(eql booltype b c)=false ->
                    <bool>(eql Frametype (pair d b) (pair e c))=false.
Lemma same_bool:    <bool>(eql Frametype (pair d b) (pair e b))=(eql Dtype d e).
Lemma nack:         <bool>(eql Frametype f (pair d (neg (bit_of f))))=false.
Lemma ack:          <bool>(eql Frametype f (pair d (bit_of f)))
                         =(eql Dtype (data_of f) d).
```

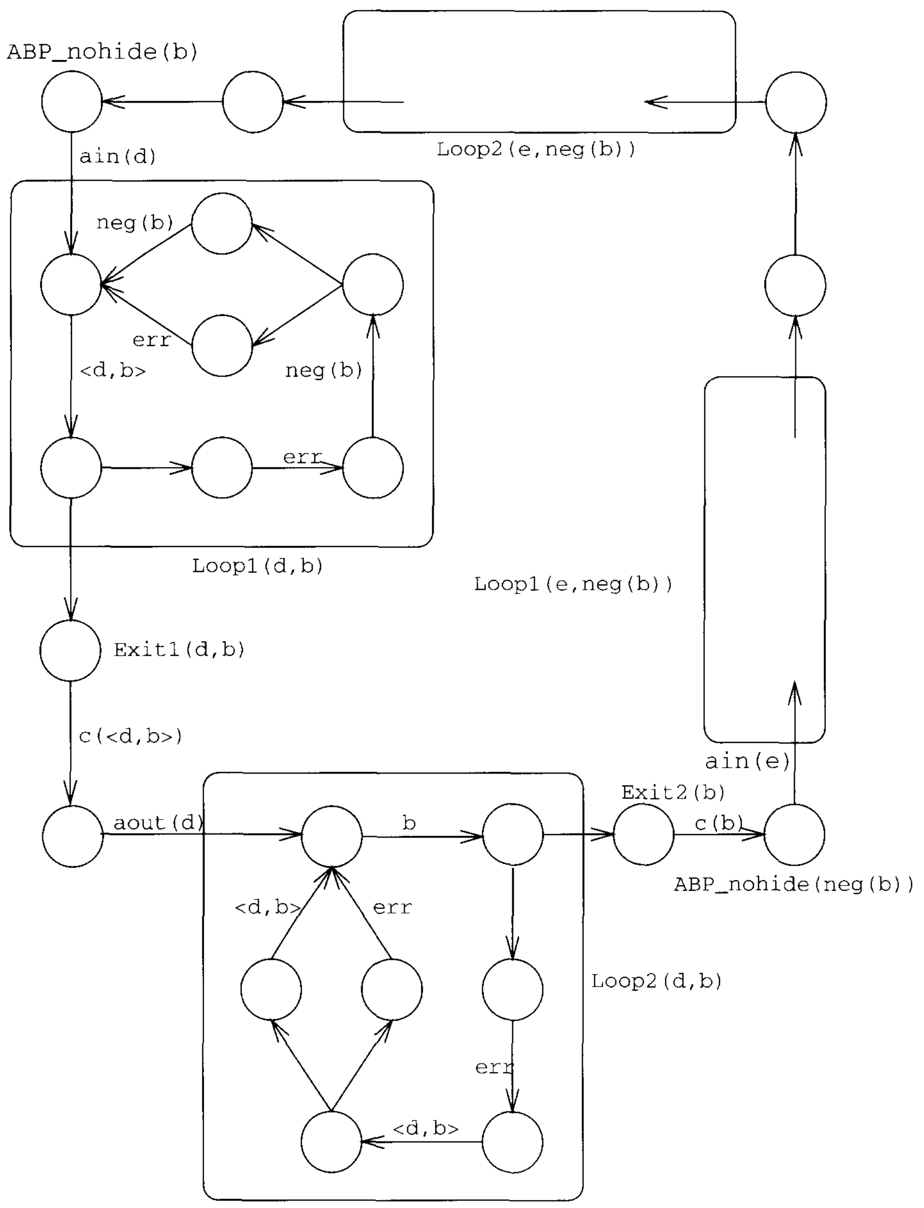

Fig. 4. Putting the loop definitions in place.

```
End ABP_data.

Definition Differtypes = [T,U:types](<Prop>Match T with
(<Prop>Match U with False True True True True True True)
(<Prop>Match U with True False True True True True True)
(<Prop>Match U with True True False True True True True)
(<Prop>Match U with True True True False True True True)
(<Prop>Match U with True True True True False True True)
(<Prop>Match U with True True True True True False True)
(<Prop>Match U with True True True True True True False)).

Lemma differtypes: (T,U:types)(Differtypes T U)->~<types>T=U.
```

The aim of these lemmas is the following. After applying EXP4, we obtain terms containing the communication merge. After some more rewriting (see Section 5.3), we can rewrite with `CF1` or `CF2`. The result of `CF1` is a conditional, the condition being `(eql T t t')`. With the above lemmas, we built a tactical that rewrites this condition to `true` (by `same_bool` and `ack`) or `false` (by `differ_frame` and `nack`). The first three lemmas are used to put the data in a form matching the left sides of the other four. For rewriting with `CF2`, the premiss `~<types>T=U` must be proved. As we have enumerated the datatypes by an `Inductive Set`, this can be done automatically by applying `differtypes`: when `T` and `U` are filled in, `(Differtypes T U)` beta-reduces to `True` (or to `False`, but then `CF1` should be applied instead).

Apart from the lemmas mentioned in Section 3.3, which establish the necessary properties of `gamma`, we proved the following lemmas about actions. The aim of the first three lemmas is to prove that certain actions are not `tau` (for guardedness, see `S2`) and not `delta` (for connectedness of a cluster, see `conn1`). The last two lemmas state that the encapsulation and hiding sets are 'good' in the sense that they do not contain `tau` and `delta`.

```
Section ABP_actions.
Variable a,b:act.

Lemma not_tau_action:
 (<Prop>Match a with True True True True True True True False)->~(<act>tau=a).
Lemma not_delta_action:
 (<Prop>Match a with True True True True True True False True)->~(<act>delta=a).
Lemma not_action_action: ~(<act>b=a)->~(<act>a=b).

Lemma goodHiding: (Hiding a)->(goodset Hiding a).
Lemma goodEncaps: (Encaps a)->(goodset Encaps a).
End ABP_actions.
```

5.2. Auxiliary Definitions and RSP

In this section, we translate the definitions preceding Lemma 2.3 into Coq. Then we add two more definitions necessary for the application of RSP. Finally, we show how RSP is applied by a typical example.

In Section 2.3, we defined the 'inner loops' E_1 and E_2 of the ABP: the loops that occur when a message is corrupted in a channel. The following definitions represent the common structure of E_1 and E_2, depicted in Fig. 3. They are parameterized by the data sent (d1,...,d5), the types of this data, and the exit process `P`. In this way, we need to apply CFAR only once, on this common structure, instead of twice.

```
Section CFARLoop.
Variable T1,T2,T3,T4 : types.
Variable d1          : (type T1).
Variable d2          : (type T2).
Variable d3          : (type T3).
Variable d4,d5       : (type T4).
Variable P           : proc.
```

```
Inductive Set PVLoop = X1 : PVLoop | X2 : PVLoop | X3 : PVLoop | X4 : PVLoop
                     | X5 : PVLoop | X6 : PVLoop | X7 : PVLoop.
Definition TypLoop   = [X:PVLoop]onetype.
Definition RLoop     = [X:PVLoop][d:one][Y:PVLoop][e:one]False.
Definition DefEqLoop = [iPV:PVLoop->one->proc][X:PVLoop][d:one]
(<proc>Match X with
        (*X1*)          (seq (ia T1      c   d1) (iPV X2 i))
        (*X2*)    (alt  (seq (ia onetype int i ) P)
                        (seq (ia onetype int i ) (iPV X3 i)))
        (*X3*)          (seq (ia T2      c   d2) (iPV X4 i))
        (*X4*)          (seq (ia T3      c   d3) (iPV X5 i))
        (*X5*)    (alt  (seq (ia onetype int i ) (iPV X6 i))
                        (seq (ia onetype int i ) (iPV X7 i)))
        (*X6*)          (seq (ia T4      c   d4) (iPV X1 i))
        (*X7*)          (seq (ia T4      c   d5) (iPV X1 i))   ).
End CFARLoop.
```

Next, we use the above definition to define the first half of the main loop of the ABP, exactly as in Section 2.3, see Fig. 4; the second half is treated by symmetry.

```
Section StepDefs.
Variable b:bool.
Variable d:D.

Definition Exit2 = (seq (ia bool_Errtype c (ibool b)) (ABP_nohide (neg b))).

Definition DefEqLoop2 =
   (DefEqLoop booltype bool_Errtype Frametype  Frame_Errtype
              b        errorbit     (pair d b) errorframe (iFrame (pair d b))
    Exit2).

Definition Exit1 =
   (seq (ia Frame_Errtype c (iFrame (pair d b)))
      (seq (ia Dtype aout d)
         (Sol PVLoop TypLoop DefEqLoop2 X1 i))).

Definition DefEqLoop1 =
   (DefEqLoop Frametype  Frame_Errtype booltype bool_Errtype
              (pair d b) errorframe    (neg b)  errorbit (ibool (neg b))
    Exit1).

Definition First = (seq (ia Dtype ain d) (Sol PVLoop TypLoop DefEqLoop1 X1 i)).
```

According to the proof sketch of Lemma 2.3, we must apply RSP to show that (Sol PVLoop TypLoop DefEqLoop1 X1 i) (that is, $\langle X_1 \mid E_1\rangle(d,b)$) is equal to the encapsulated merge of the four components in certain states. But our formulation of RSP does not conclude the equality of two processes, but of two solution functions for a system of equations. Thus we need a function which returns this encapsulated merge for X1, and (Sol PVLoop TypLoop DefEqLoop1 Xk i) for Xk, $2 \leq k \leq 7$. Similarly for DefEqLoop2.

```
Definition DefEqLoop1' = [iPV:PVLoop->one->proc][X:PVLoop][j:one]
        (<proc>Match X with
        (*X1*)   (enc Encaps
                    (mer (seq (Sol SendSubState SSSTyp SSSDef Sf (pair d b))
                              (seq (Sol SendSubState SSSTyp SSSDef Sb (neg b))
                                   (Sol Components CompTyp (CompDef b) Sd i)))
                    (mer (Sol Components CompTyp (CompDef b) Rc i)
                    (mer (Sol Components CompTyp (CompDef b) CK i)
                    (Sol Components CompTyp (CompDef b) CL i) ))) )
        (*X2*)   (DefEqLoop1 iPV X2 i)
                 ...
        (*X7*)   (DefEqLoop1 iPV X7 i) ).

Definition DefEqLoop2' = [iPV:PVLoop->one->proc][X:PVLoop][j:one]
        (<proc>Match X with
        (*X1*)   (enc Encaps
                    (mer (seq (Sol SendSubState SSSTyp SSSDef Tf (pair d b))
                         (seq (Sol SendSubState SSSTyp SSSDef Sb (neg b))
                              (Sol Components CompTyp (CompDef b) Sd i)))
                    (mer (seq (ia booltype s b)
                         (seq (Sol RecSubState RSSTyp RSSDef  Rb b)
                              (Sol Components CompTyp (CompDef b) Rc i)))
                    (mer (Sol Components CompTyp (CompDef b) CK i)
                         (Sol Components CompTyp (CompDef b) CL i) ))) )
        (*X2*)   (DefEqLoop2 iPV X2 i)
                 ...
        (*X7*)   (DefEqLoop2 iPV X7 i) ).
End StepDefs.
```

As an example, we consider the application of RSP in the first inner loop, starting from

```
<proc>(Sol PVLoop TypLoop (DefEqLoop1 b d) X1 i)
     =(enc Encaps (mer (seq (Sol SendSubState SSSTyp SSSDef Sf (pair d b))
                       (seq (Sol SendSubState SSSTyp SSSDef Sb (neg b))
                            (Sol Components CompTyp (CompDef' b) Sd i)))
                  (mer (Sol Components CompTyp (CompDef' b) Rc i)
                  (mer (Sol Components CompTyp (CompDef' b) CK i)
                       (Sol Components CompTyp (CompDef' b) CL i)))))
  ============================
    b : bool
    d : D
```

Our first step is to execute the command

```
Elim (RSP PVLoop TypLoop (Sol PVLoop TypLoop
(DefEqLoop1' b d)) (DefEqLoop1 b d) RLoop).
```

This instance of RSP says:

```
(b:bool) (d:D) (X:PVLoop) (d0:(type (TypLoop X)))
  (Guarded PVLoop TypLoop (DefEqLoop1 b d) RLoop)->
  (  (X0:PVLoop) (d1:(type (TypLoop X0)))
     (<proc>(Sol PVLoop TypLoop (DefEqLoop1' b d) X0 d1)
           =(DefEqLoop1 b d (Sol PVLoop TypLoop (DefEqLoop1' b d)) X0 d1))) ->
  (<proc>(Sol PVLoop TypLoop (DefEqLoop1' b d) X d0)
       =(Sol PVLoop TypLoop (DefEqLoop1  b d) X d0))
```

Thus the effect is that two subgoals are added, and `DefEqLoop1` is replaced by `DefEqLoop1'` in the first subgoal. This goal is now solved by `Rewrite (RDP PVLoop); Unfold DefEqLoop1'; Apply refl_equal`. That is, we prove that the definition of the process variable `X1` in the loop `DefEqLoop1'` is exactly the desired encapsulated merge.

The second subgoal is that the loop is guarded. This is proved by

```
Unfold Guarded;
Induction X;
Split;[ Apply WF1; Intros; Contradiction
      | Unfold DefEqLoop1; Unfold DefEqLoop; Unfold Exit1; Auto 10].
```

That is, we unfold the definition of guarded, and then continue by a case distinction on `X:PVLoop`. Thus we perform the remaining tactic seven times: for `X1` to `X7`. Guardedness is defined as the conjunction of well-foundedness and safeness. As the relation `RLoop` is always `False`, well-foundedness is easily proved. Safeness is proved automatically after unfolding some definitions. Typically, Coq finds the tactical

```
Intros; Apply S2;
[Apply not_action_action; Apply not_tau_action; Exact I | Apply S0; Exact I],
```

but the cases for `X2` and `X5` are a little harder because they have two exits. For `X2`, Coq finds

```
Apply S4; [Apply S3; [Apply S1 |
                      Apply S3; [Apply S1 |
                                 Apply S2; [Apply not_action_action;
                                                Apply not_tau_action; Exact I |
                                             Apply S10; Intros;
       Apply SafeLoop2]]] |
           Apply S2; [Apply not_action_action; Apply not_tau_action; Exact I |
                      Apply S0; Exact I]]
```

`SafeLoop2` is one of a series of lemmas that the recursive specifications of the sender and receiver, the components, `Loop2`, and finally `Loop1` are `Safe` w.r.t. the relation that is always `True`. In other words, these lemmas prove that we have a sequence of recursive systems, one depending on the other (in the above order), but without mutual dependencies. These proofs are straightforward.

After rewriting by RDP once, the third subgoal is

```
(X:PVLoop) (j:(type (TypLoop X)))
  (<proc>(DefEqLoop1' b d (Sol PVLoop TypLoop (DefEqLoop1' b d)) X j)
      =(DefEqLoop1  b d (Sol PVLoop TypLoop (DefEqLoop1' b d)) X j))
```

This is proved again by case distinction. For `X2` to `X7` it is trivial, because `DefEqLoop1` and `DefEqLoop1'` coincide. For `X1`, we unfold some definitions and obtain

```
<proc>(seq (ia Frametype c (pair d b))
        (Sol PVLoop TypLoop (DefEqLoop1' b d) X2 i))
     =(enc Encaps (mer (seq (Sol SendSubState SSSTyp SSSDef Sf (pair d b))
                            (seq (Sol SendSubState SSSTyp SSSDef Sb (neg b))
                               (Sol Components CompTyp (CompDef b) Sd i)))
                       (mer (Sol Components CompTyp (CompDef b) Rc i)
                          (mer (Sol Components CompTyp (CompDef b) CK i)
                             (Sol Components CompTyp (CompDef b) CL i)))))
```

This goal is almost the same as our starting point. The fact that in the lefthandside `X1` is unfolded to $c\cdot$`X2` is not important. The important change is that we have `DefEqLoop1'` on the lefthandside: after unfolding `X2` to $i\cdot Exit1+i\cdot$`X3`, `X3` to $c\cdot$`X4`, and so on, we do not return to `X1` but to the encapsulated merge that is currently the righthandside. This means that we can prove the goal by linearizing the righthandside several times. This is the topic of the next section.

5.3. Linearization

This section corresponds to Lemma 2.3. We outline how we prove in Coq

```
(b:bool)<proc>(ABP_nohide b)=(sum Dtype (First b)).
```

As we noted in the proof of Lemma 2.3, the bulk of the verification consists of proving this lemma. We must linearize (determine the possible first actions of) a process of the form ∂_H(*SenderState* $\|$ *ReceiverState* $\|$ *KState* $\|$ *LState*) for all 18 states in the first half of the ABP. This is by far the most time and space consuming part of the proof. In this section, we discuss in detail the tactical that performs this task without any user guidance. The tactical is specialized for the ABP, and will have to be adapted for other protocols.

It is clear that future research must concentrate on improving the linearization technique, in order to verify larger protocols. It must become much more efficient, and (almost) completely independent of the protocol. This seems ambitious at first, but for effective μCRL-specifications [GrP94], all that is needed is an efficient encoding of term-rewriting in Coq. On the other hand, it must be investigated whether proof checkers based on term-rewriting are capable of also handling the other parts of the verification. If so, they might be better candidates than Coq for formal protocol verification. We now return to our current linearization tactical.

The possible first actions of a state of the ABP are determined by the possible first actions of the substates of the four constituing components. It turns out that the term describing such a substate can have four syntactical forms: `(Sol Components ...)`, `(seq (Sol SendSubState ...) x)`, `(seq (Sol RecSubState ...) x)` and `(seq` *action* `x)`.

Expanding the merge yields the alternative composition of four terms `(Lmer` *Substate1 Substates*`)` and six terms `(Lmer (comm` *Substate1 Substate2*`)` *Substates*`)`. Our first step is to apply RDP on *Substate1* and *Substate2* unless they are in the fourth syntactical form. That is, we replace a process variable `(Sol ...)` by its definition `(DefEq (Sol ...))` only if it plays a role in determining the first possible actions. Then we unfold `DefEq`. `DefEq` occurs also as an argument of `Sol`, and that occurrence should not be unfolded. Therefore we replace it by a renamed copy `DefEq'` before (respectively during) this tactical.

For example, `Unfold_Lmer_comm_Sol1` is the lemma

```
(ProcVar:Set) (Typ:ProcVar->types)
  (DefEq,DefEq':((X:ProcVar)(type (Typ X))->proc)->
                (X:ProcVar)(type (Typ X))->proc)
    (X:ProcVar) (d:(type (Typ X))) (x,z:proc)
      (<(((X0:ProcVar)(type (Typ X0))->proc)->
          (X0:ProcVar)(type (Typ X0))->proc) >DefEq=DefEq') ->
        (<proc>(Lmer (comm       (Sol ProcVar Typ DefEq'  X d) z) x)
             =(Lmer (comm (DefEq (Sol ProcVar Typ DefEq') X d) z) x))
```

The first part of the linearization tactical is the following.

```
Elim EXP4;
Repeat
  (Rewrite (Unfold_Lmer_Sol Components CompTyp (CompDef b) (CompDef' b));
   [Idtac|Apply refl_equal]);
Repeat
  (Rewrite (Unfold_Lmer_comm_Sol1 Components CompTyp (CompDef b) (CompDef' b));
   [Idtac|Apply refl_equal]);
Repeat
  (Rewrite (Unfold_Lmer_comm_Sol2 Components CompTyp (CompDef b) (CompDef' b));
   [Idtac|Apply refl_equal]);
Unfold CompDef;
Try (Replace  SSSDef with  SSSDef';[Idtac|Apply refl_equal]);
Try (Replace  RSSDef with  RSSDef';[Idtac|Apply refl_equal]);
Repeat (Rewrite (Unfold_Lmer_seq_Sol SendSubState SSSTyp SSSDef SSSDef');
        [Idtac|Apply refl_equal]);
```

```
Repeat (Rewrite (Unfold_Lmer_seq_Sol RecSubState RSSTyp RSSDef RSSDef');
       [Idtac|Apply refl_equal]);
Repeat (Rewrite (Unfold_Lmer_comm_seq_Sol1 SendSubState SSSTyp SSSDef SSSDef');
       [Idtac|Apply refl_equal]);
Repeat (Rewrite (Unfold_Lmer_comm_seq_Sol1 RecSubState RSSTyp RSSDef RSSDef');
       [Idtac|Apply refl_equal]);
Repeat (Rewrite (Unfold_Lmer_comm_seq_Sol2 SendSubState SSSTyp SSSDef SSSDef');
       [Idtac|Apply refl_equal]);
Repeat (Rewrite (Unfold_Lmer_comm_seq_Sol2 RecSubState RSSTyp RSSDef RSSDef');
       [Idtac|Apply refl_equal]);
Unfold SSSDef RSSDef;
```

We are now faced with terms having the following structure (in the worst case).

```
(enc H (alt (Lmer (comm (alt (seq (alt (action)
                                       (sum T [t:(type T)]action))
                                  (... unimportant ...))
                             (sum T [t:(type T)](seq action x)))
                        (... similar ...))
                  (... unimportant ...))
            (... similar ...)))
```

We continue by bringing out the alts, and then by bringing out the sums. We use several distributivity axioms, and need only the special lemma DLCSS (see Table 5). We need this lemma because we cannot rewrite terms that occur inside a sum, for these terms do not denote processes, but functions of type `(type T)->proc`. We cannot conclude in Coq that two such functions `f` and `g` are equal, even if `(t:(type T))<proc>(f t)=(g t)`.

```
Repeat Elim A4; (* over seq  *)
Repeat Elim CM8;(* left  over comm *)
Repeat Elim CM9;(* right over comm *)
Repeat Elim CM4;(* over Lmer *)
Repeat Elim D3; (* over enc  *)
Repeat Elim A2; (* over alt  *)

Repeat Rewrite SUM5; (*       over seq  *)
Repeat Elim DLCSS;   (* two   over comm, Lmer, and enc  *)
Repeat Rewrite SUM7; (* left  over comm *)
Repeat Elim SUM7';   (* right over comm *)
Repeat Rewrite SUM6; (* one   over Lmer *)
Repeat Rewrite SUM9; (* one   over enc  *)
```

Now we have a long list of alternatives. Most of these will turn out to be equal to `Delta`. Therefore we continue by trying to rewrite each alternative to `Delta`. We cannot rewrite the term as a whole, because we cannot rewrite inside sums. This is the main reason for using 'axioms restated as rules'. The tactical has the following structure.

```
Repeat (
   Repeat ( (Apply RuleA6' Orelse Apply True_ind);
            [tactical for rewriting one alternative to Delta | Try Exact I]);
   Apply Split_alt Orelse Apply RuleA6);
tactical for an alternative that is not Delta
```

This tactical is applied on a goal of the form `<proc>target=alternatives`. `target` is the linearized form (which we do not compute, but is defined beforehand, as in Lemma 2.3), which consists of one or two alternatives. `alternatives` is the long list. We can pick the first alternative off the list by applying

```
RuleA6':~(<proc>Delta=x) -> (<proc>z=y) -> (<proc>z=(alt x y)).
```

The first subgoal is now attempted; the second one is treated in the next iteration. The application of RuleA6' fails when we have only one alternative left. In that case, we do not need to do anything, except that the remaining tactical expects two subgoals. Thus in that case we apply True_ind: (P:Prop)P->True->P. In this case the second subgoal True is solved by Try Exact I, which has otherwise no effect.

If the tactical for rewriting one alternative to Delta fails, then the inner loop terminates: this alternative is not Delta, but (one of) the alternative(s) in target. If the target contains more than one alternative, then we apply

```
Split_alt:~(<proc>z=x) -> (<proc>w=y) -> (<proc>(alt z w)=(alt x y))}.
```

We must ensure before starting the linearization that we encounter the alternatives from the list in the correct order. If the target is (reduced to) one alternative, then we apply

```
RuleA6:~(<proc>Delta=x) -> (<proc>z=y) -> (<proc>z=(alt y x))}.
```

Next we consider the tactical for rewriting an alternative to Delta. First, we remove the sums, which are already on top. Then we take the first actions of both sides (which are by now sequences of actions) and make them into a communication (comm action action), which we try to prove equal to Delta. (Recall that the tacticals Try ... and Repeat ... never fail: if we have an alternative without communication, nothing happens.) It can be Delta for three reasons: the actions have different types, the actions do not communicate (their gamma is delta), or the data are incompatible. Finally, we push the Delta outward. Recall that Auto;Exact I serves as the version of Auto that can fail.

```
Repeat (Apply RuleSUM1;Intro); (* remove sums *)

Repeat Elim A5;  (*       over seq  *)
Repeat Elim CM7; (* two   over comm *)
Repeat Elim CM6; (* right over comm *)
Repeat Elim CM5; (* left  over comm *)

Try (Replace (bit_of (pair d b)) with b;
     [Idtac|Apply (make_eql booltype);Apply bit_inj]);

           (Elim CF2;[Idtac|Auto;Exact I]) (* types *)
Orelse Try (Elim CF1;Unfold gamma;
            (Elim Delta_Data;Elim COND3)  (* actions *)
            Orelse                        (* data *)
            (tactical for incompatible data Orelse
              (Elim sym_eql; tactical for incompatible data));Elim COND2);

Try Elim A7;
Try Elim CM2';
Try Elim CM2;
Try Elim CM3;
Try Elim D4;
Try Rewrite D2;
Auto;Exact I
```

In this, the tactical for incompatible data reads

```
( Try Elim same_err_frame;
  Rewrite differ_frame;
  [Idtac|Right;Apply not_eql_b_negb])
```

```
Orelse Rewrite find_errorframe
Orelse (Try Elim same_err_bit;Rewrite not_eql_b_negb)
Orelse Rewrite find_errorbit
Orelse Rewrite not_eql_b_negb
```

This concludes the tactical for rewriting an alternative to `Delta`. We continue by linearizing further the remaining alternatives. First, we remove the summation, if any. If the target is a summation too, then it is of the same type, and we must apply SUM11. Otherwise, we have a goal of the form $c(t) \cdot P = \sum_{d:D} \partial_H((s(t) \mid (r(d) \cdot Q(d))) \| \ldots)$ (omitting other components and actions). That is, one component sends data t of type D, while another component is willing to receive any item of type D. In this case, we must apply RuleSUMrep, except if D is **Bool**, in which case we apply EXP_bool.

```
(Apply (SUM11 Dtype);                        Intro d      ) Orelse
(Apply (RuleSUMrep Frametype (pair d b));Intro NewVar ) Orelse
(Apply (RuleSUMrep Dtype     d);         Intro NewVar ) Orelse
Try Elim (EXP_bool b);
```

What follows is similar to the tactical rewriting a communication to `Delta`, except that we now expect matching types, communicating actions, and compatible data (except for booleans: due to the use of EXP_bool).

```
Try (Replace (bit_of (pair d b)) with b;
     [Idtac|Apply (make_eql booltype);Apply bit_inj]);
Repeat Elim A5;
Repeat Elim CM7;
Try (
Elim CF1;Unfold gamma;
(  (* If EXP_bool is used, we have two communications; one succeeds, *)
      Elim CF1;Unfold gamma;Rewrite refl_eql;Elim COND1;
   (* and one is Delta. *)
      (Rewrite not_eql_b_negb Orelse (Rewrite sym_eql;Rewrite not_eql_b_negb));
      Elim COND2;Elim A7;Elim CM2';Elim D1_Delta;
   (* The Delta goes. *)
      (Elim A6 Orelse Elim A6'))
Orelse
   (Rewrite refl_eql;Elim COND1)
Orelse ...
```

If RuleSUMrep is used as mentioned above, it changes the proof obligation to $(c(t) \cdot P) \lhd eq_D(t,d) \rhd \delta = \partial_H((s(t) \mid (r(d) \cdot Q(d))) \| \ldots)$. CF1 replaces the communication by a second conditional, with the same condition (after simplification and modulo symmetry). This second conditional is taken outside, and then cancelled against the one on the lefthandside by the rule Split_COND. This rule gives two subgoals. One is $c(t) \cdot P = \partial_H((c(t) \cdot Q(d)) \| \ldots)$ given the hypothesis $eq_D(d,t)$, the other is $\delta = \partial_H((\delta \cdot Q(d)) \| \ldots)$. The hypothesis in the first is necessary for replacing $Q(d)$ by $Q(t)$. (The tactic `Clear` removes the hypothesis and the new variable d from the context, in order to avoid name clashes when the tactical is applied again.)

```
Orelse ...
  (Unfold Delta;
   Elim (COND5 seq); Elim (COND5 Lmer); Elim COND10;
   Try Elim same_err_frame;
   Try Elim same_err_bit;
   Try Rewrite negneg;
   Try Rewrite same_bool;
   Apply Split_COND Orelse (Elim sym_eql;Apply Split_COND);
   [Intro H;
```

```
    (Replace NewVar with (pair d b); [Idtac|Apply (make_eql Frametype);Auto])
   Orelse (Replace NewVar with d;     [Idtac|Apply (make_eql Dtype);Auto]);
   Clear H NewVar
  | Elim A7;Elim CM2';Elim D1_Delta;Apply refl_equal]));
```

Finally, we can get the first action on top by taking it outside the left-merge (which returns to a merge) and the encapsulation. We remove the first actions on both sides by an instance of the trivial rule f_equal, namely

```
(f:proc->proc)(x,y:proc)(<proc>x=y)->(<proc>(f x)=(f y)),
```

where f is (seq *action*). SC7 restores the expected association of the merges.

```
Try Elim CM3;
Try Elim D4;
Try (Rewrite D1;[Idtac|Auto]);
Repeat Apply (f_equal proc proc);
Repeat Elim SC7.
```

5.4. Applying CFAR

We apply CFAR on the general loop depicted in Fig. 3, and assume declarations of T1,...,T4 and d1,...,d5 accordingly. This loop consists of one cluster of seven elements, X1,..., X7, all of type one. Thus we must define the following functions.

$$\begin{array}{lll l}
\mathit{cluster}(\mathtt{X}n,\mathtt{i}) & = 0 & & \\
\mathit{element}(\mathtt{X}n,\mathtt{i}) & = n-1 & & \\
\mathit{process}(k,m) & = \mathtt{X}(m+1) & \text{if } k=0 \text{ and } m<7, & \delta \text{ otherwise} \\
\mathit{Exit}(k,m) & = i\cdot P & \text{if } k=0 \text{ and } m=1, & \delta \text{ otherwise} \\
a(\mathtt{X1},\mathtt{i},m) & = c(d_1) & \text{if } m=1, & \delta \text{ otherwise} \\
a(\mathtt{X2},\mathtt{i},m) & = i & \text{if } m=2, & \delta \text{ otherwise} \\
a(\mathtt{X3},\mathtt{i},m) & = c(d_2) & \text{if } m=3, & \delta \text{ otherwise} \\
a(\mathtt{X4},\mathtt{i},m) & = c(d_3) & \text{if } m=4, & \delta \text{ otherwise} \\
a(\mathtt{X5},\mathtt{i},m) & = i & \text{if } m=5 \text{ or } m=6, & \delta \text{ otherwise} \\
a(\mathtt{X6},\mathtt{i},m) & = c(d_4) & \text{if } m=0, & \delta \text{ otherwise} \\
a(\mathtt{X7},\mathtt{i},m) & = c(d_5) & \text{if } m=0, & \delta \text{ otherwise.}
\end{array}$$

In Coq, we define *element* through the Match-function. We cannot do that for *process* and *Exit*, because nat is not inductively defined. The problem is circumvented by making extensive use of the conditional construct. For example, *Exit* is defined as

$$\lambda\ k,m : nat \quad (i\cdot P \lhd eq_{nat}(n,1) \rhd \delta) \lhd eq_{nat}(k,0) \rhd \delta.$$

The definition of *process* contains eight conditionals!

As we noted in Section 3.7, the function *a* must be split in three parts in Coq: sort, action name, and data. Because <proc>(ia D delta d)=Delta for all sorts D and data d, we can define sort and data independent of *m*:

```
Definition D' = [X:PVLoop][j:one][m:nat]
(<types>Match X with T1 onetype T2 T3 onetype T4 T4).

Definition d' = [X:PVLoop][j:one][m:nat]
(<[X:PVLoop](type (D' X j m))>Match X with d1 i d2 d3 i d4 d5).
```

In contrast, the function a giving the action name depends on both the process variable and *m*. Here it is really a problem that nat is not inductively defined. If it were, we could define a by two nested Matches. As it is, we found no other way than writing an axiom am for each m ($0 \leq m < 7$) and one axiom a7 for $m \geq 7$.

```
Parameter a : PVLoop->one->nat->act.

Axiom a0: (X:PVLoop)
<act>(<act>Match X with delta delta delta delta delta c c)=(a X i 0).
Axiom a1: (X:PVLoop)
<act>(<act>Match X with c delta delta delta delta delta delta)=(a X i (S 0)).
...
Axiom a7: (n:nat)(X:PVLoop) <act>delta=(a X i (S (S (S (S (S (S (S n)))))))).
```

Our aim is to prove the following goal.

```
( (iPV:PVLoop->one->proc)(X:PVLoop)(d:one)
  (Safe PVLoop TypLoop iPV X d [X:PVLoop][e:one][Y:PVLoop][f:one]True P))->
<proc>(seq Tau (hide Hiding
                (Sol PVLoop TypLoop
                     (DefEqLoop T1 T2 T3 T4 d1 d2 d3 d4 d5 P) X1 i)))
    =(seq Tau (hide Hiding P)).
```

The assumption that P is safe is necessary for proving that the cluster is guarded. It will be trivial to verify it for *Exit1* and *Exit2* later.

Before we can apply CFAR, we must bring the exit process in the correct form, that is, we must prove $\tau \cdot \tau_I(P) = \tau \cdot \tau_I(\sum_{n:nat} Exit(0,n))$. This is rather easy: because there is only one exit $i \cdot P$ for $n = 1$, we can apply SUMmand with $d' = 1$ and manipulate the conditionals to prove that the remaining sum is δ. Then we take the hiding inside to hide the action i.

We can now apply CFAR:

```
Apply (CFAR PVLoop TypLoop (DefEqLoop T1 T2 T3 T4 d1 d2 d3 d4 d5 P) RLoop
          Hiding cluster element process Exit D' a d' X1 i).
```

The prerequisites CheckInside and CheckOutside are relatively easy to verify, although the large number of conditionals in process makes the proofs somewhat cumbersome. Verifying CheckDef is even more cumbersome: for each i, we must simplify $\sum_{n:nat} a(\mathrm{X}i, \mathrm{i}, n) \cdot process(0,n)$. For most values of n, $a(\mathrm{X}i, \mathrm{i}, n)$ is δ. We use SUMmand to isolate the useful value(s) of n, and rewrite the remaining sum to δ. Instead of induction on n, we apply the lemma

```
(n:nat)<nat>n=0 \/
       <nat>n=(S 0) \/
       ... \/
       <nat>n=(S (S (S (S (S (S 0)))))) \/
       <nat>Ex([m:nat] <nat>n=(S (S (S (S (S (S (S m)))))))).
```

The same lemma is used to prove Checka, which is otherwise trivial. CheckConn states that each state must be reachable from each other state within the cluster. In order to avoid double induction, we apply transitivity first, and prove that each state is reachable from X1, and vice versa. This part of the proof is implemented by 'walking forward' through the loop. Finally, proving guardedness was already discussed in Section 5.2.

In the ABP, we need CFAR only once, and on a loop of only seven states. We conclude that the current definitions are good enough in this situation. But it is clear that for larger loops, and for protocols that require multiple applications of CFAR, more sophisticated proof techniques are necessary, in particular for CheckDef and CheckConn. Improved techniques for linearization will probably apply to CheckDef also. For CheckConn, an existing efficient algorithm for checking that a graph is strongly connected must be translated to Coq. Here we see a reversal of the programs-as-proofs paradigm: instead of extracting a program from a proof, we want to translate an existing program (and its verification) to a proof generator.

5.5. Completing the Proof

We define the process `BufferTwice` as the process that satisfies the final equation in the proof of Theorem 2.4, namely the defining equation of a buffer unfolded twice.

```
Definition BufferTwice =
              (Sol PVBuf TypBuf [V:PVBuf->one->proc](BufEq (BufEq V)) Buf i).
```

We prove that this equation is guarded (trivial) and then by RSP that `<proc>BufferTwice = Buffer`. And, we prove `<proc>Buffer=(ABP true)` by replacing `Buffer` by `BufferTwice`, `(ABP true)` by `(hide Hiding (sum Dtype (First true)))` and applying RSP again. The goal is now

```
<proc>(hide Hiding
        (sum Dtype [d:D](seq (ia Dtype ain d)
                         (Sol PVLoop TypLoop (DefEqLoop1 true d) X1 i))))
    =(sum Dtype [d:D](seq (ia Dtype ain d)
                     (seq (ia Dtype aout d)
                        (sum Dtype [d0:D](seq (ia Dtype ain d0)
                                        (seq (ia Dtype aout d0)
                                           (hide Hiding
                                              (sum Dtype (First true))))))))
```

We continue by moving the hiding inside the sum and removing the summation on both sides. Then we add a `tau`-action after the `ain`-action (using TAU1). Then we move the hiding further, inside these actions. Now we can apply the instance of CFAR discussed in the previous section on the first loop. Again we add a `tau`-action, this time after the `aout`-action, move the hiding further, and apply CFAR on the second loop. Stripping the `ain`- and `aout`-actions on both sides, we arrive at the goal

```
<proc>(hide Hiding (ABP_nohide (neg true)))
    =(sum Dtype [d:D](seq (ia Dtype ain d)
                        (seq (ia Dtype aout d)
                           (hide Hiding (sum Dtype (First true))))))
```

Now we replace `(ABP_nohide (neg true))` by `(sum Dtype (First (neg true)))`, and repeat the proof steps of the previous paragraph. The resulting goal is

```
<proc>(hide Hiding (ABP_nohide (neg (neg true))))
    =(hide Hiding (sum Dtype (First true)))
```

Replacing `(neg (neg true))` by `true` and then `(ABP_nohide true)` by `(sum Dtype (First true))` concludes the proof.

6. Future Work

A number of directions for future research is immediately obvious:

- Improving the proof theory of μCRL, see e.g. [BeG94b].

- Improving the proof techniques of this paper, in particular linearization and the verification of the premisses of CFAR.
- Proving the soundness of the translation w.r.t. μCRL. This is a moving target, as changes to Coq are still made, and changes to μCRL are proposed, e.g. in [GrW94].
- Verification of other protocols, probably developing new proof techniques at the same time, see e.g. [BeG94a, KoS94, GrP96].
- Extending μCRL with (discrete) real time [BaB92] and translating the resulting formalism to Coq in order to verify timed protocols [KaP93, Klu91].
- Investigate if other proof checkers, or perhaps even theorem provers, are more suitable than Coq for the verification of protocols. It appears that the proofs consist for a significant part of term rewriting, which is not easy to do in Coq.

Acknowledgments

We thank Jaco van de Pol, Jan Springintveld, Alex Sellink, Erik Poll, Jos Baeten, and Jan Bergstra for some valuable discussions.

References

[Bar92] Barendregt, H. P.: 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, 1992.

[BaB92] Baeten, J. C. M. and Bergstra, J. A.: Discrete time process algebra. In W. R. Cleaveland, editor, *Proceedings Concur'92*, LNCS 630, pages 401–420. Springer Verlag, 1992.

[BeG93] Bezem, M. and Groote, J. F.: A formal verification of the alternation bit protocol in the calculus of constructions. Technical Report 88, Logic Group Preprint Series, Utrecht University, March 1993.

[BeG94a] Bezem, M. and Groote, J. F.: A correctness proof of a one-bit sliding window protocol in μCRL. *The Computer Journal*, 37(4):289–307, 1994.

[BeG94b] Bezem, M. and Groote, J. F.: Invariants in process algebra with data. In B. Jonsson and J. Parrow, editors, *Proceedings Concur'94*, LNCS 836, pages 401–416. Springer Verlag, 1994.

[BeG94c] Bezem, M. and Groote, J. F.: Proving a graph well founded using resolution. Technical Report 113, Logic Group Preprint Series, Utrecht University, May 1994.

[BeK86a] Bergstra, J. A. and Klop, J. W.: 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.

[BeK86b] Bergstra, J. A. and Klop, J. W.: Verification of an alternating bit protocol by means of process algebra. In W. Bibel and K. P Jantke, editors, *Math. Methods of Spec. and Synthesis of Software Systems 1985*, LNCS 215, pages 9–23. Springer Verlag, 1986.

[BSW69] Bartlett, K. A., Scantlebury, R. A. and Wilkinson, P. T.: A note on reliable full-duplex transmission over half-duplex links. *Communications of the ACM*, 12:260–261, 1969.

[BaW90] Baeten, J. C. M. and Weijland, W. P.: *Process Algebra*, volume 18 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1990.

[CAB86] Constable, R. L., Allen, S. F., Bromley, H. M., Cleaveland, W. R., Cremer, J. F., Harper, R. W., Howe, D. J., Knoblock, T. B., Mendler, N. P., Panangaden, P., Sasaki, J. T. and Smith, S. F.: *Implementing Mathematics with the NuPrl Development System*. Prentice-Hall, inc., Englewood Cliffs, New Jersey, first edition, 1986.

[CoH88] Coquand, T. and Huet, G.: The calculus of constructions. *Information and Control*, 76:95–120, 1988.

[Cou93] Courcoubetis, C.: editor. *Proceedings of the 5th International Conference on Computer Aided Verification*, Elounda, Greece, June/July 1993. Springer-Verlag, 1993.

[ClP88] Cleaveland, R. and Panangaden, P.: Type theory and concurrency. *International Journal of Parallel Programming*, 17:153–206, 1988.

[CoP90] Coquand, T. and Paulin, C.: Inductively Defined Types. In P. Martin-Löf and G. Mints, editors, *COLOG-88*, LNCS 417, pages 50–66. Springer-Verlag, 1990.

[DFH93] Dowek, G., Felty, A., Herbelin, H., Huet, G., Murthy, C., Parent, C., Paulin-Mohring, C. and Werner, B.: The Coq Proof Assistant User's Guide, version 5.8. Technical report, INRIA-Rocquencourt and CNRS - ENS Lyon, 1993.

[Dro94] Drost, N. J.: *Process Theory and Equation Solving*. PhD thesis, University of Amsterdam, February 1994. (Section 2.5.1).

[EGL92] Engberg, U., Grønning, P. and Lamport, L.: Mechanical verification of concurrent systems with TLA. In G. v. Bochmann and D. K. Probst, editors, *Proceedings of the 4th International Workshop on Computer Aided Verification*, Montreal, Canada, volume 663 of *Lecture Notes in Computer Science*, pages 44–55. Springer-Verlag, 1992.

[Gim95] Giménez, E.: Co-Inductive Types in Coq : An Experiment with the Alternating Bit Protocol. Submitted for the proceedings of the BRA Workshop on Types for Proofs and Programs. Also available by ftp at `ftp.ens-lyon.fr/pub/users/LIP/ABP.ps.Z`, June 1995.

[GrP93] Groote, J. F. and Ponse, A.: Proof theory for μCRL: a language for processes with data. In D. J. Andrews, J. F. Groote, and C. A. Middelburg, editors, *Proceedings of the International Workshop on Semantics of Specification Languages*, Utrecht, The Netherlands, pages 231–250. Workshops in Computer Science, Springer-Verlag, 1993.

[GrP94] Groote, J. F. and Ponse, A.: The syntax and semantics of μCRL. In A. Ponse, C. Verhoef, and S. F. M van Vlijmen, editors, *Algebra of Communicating Processes (Proceedings ACP'94)*, pages 26–62, 1994.

[GrP96] Groote, J. F. and van de Pol, J. C.: A bounded retransmission protocol for large data packets. A case study in computer checked verification. In M. Wissing and M. Nivat, editors, *Proceedings of AMAST'96, Munich, Lecture Notes in Computer Science 1101*, Springer Verlag, pages 536–550, 1996.

[GrW94] Groote, J. F. and van Wamel, J. J.: Algebraic data types and induction in μCRL. Technical Report P9409, University of Amsterdam, April 1994.

[Hes94] Hesselink, W. H.: Wait-free linearization with an assertional proof. *Distributed Computing*, 8:65–80, 1994.

[Hes95] Hesselink, W. H.: Wait-free linearization with a mechanical proof. *Distributed Computing*, 9:(to appear), 1995.

[Hoo91] Hooman, J.: *Specification and Compositional Verification of Real-Time Systems*, LNCS 558. PhD thesis, Eindhoven University of Technology, 1991.

[HSV94] Helmink, L., Sellink, M. P. A. and Vaandrager, F. W.: Proof-checking a data link protocol. In *Proceedings Workshop Esprit BRA Types for Proofs and Programs*, Nijmegen, The Netherlands, May 1993, LNCS 806. Springer-Verlag, 1994.

[Kam93] Kamsteeg, G.: A formal verification of the Alternating Bit Protocol in μCRL. Technical Report 93–37, Dept. of Comp. Sci., Leiden University, Netherlands, 1993.

[Klu91] Klusener, A. S.: Abstraction in real time process algebra. In J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, editors, *Proceedings of the REX workshop "Real-Time: Theory in Practice"*, LNCS 600. Springer-Verlag, 1991.

[KaP93] Kaart, M. and Polak, I.: Het alternating bit protocol met time-out in discrete tijd. Technical Report P9323, Programming Research Group, University of Amsterdam, September 1993. (in Dutch).

[KoS94] Korver, H. and Springintveld, J.: A computer-checked verification of Milner's Scheduler. In: M. Hagiya, J. C. Mitchell, editors, *Proceedings TACS'94, Sendai Japan, Lecture Notes in Computer Science 789*, pages 161–178. Springer-Verlag 1994. Full version: Technical Report 101, Logic Group Preprint Series, Utrecht University, November 1993.

[LMW94] Lynch, N., Merritt, M., Weihl, W. and Fekete, A.: *Atomic Transactions*. Morgan Kaufmann Publishers, 1994.

[Mil80] Milner, R.: *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[MaP82] Manna, Z. and Pnueli, A.: Verification of concurrent programs, a temporal proof system. In *Foundations of Computer Science IV, Distributed Systems: Part 2* Mathematical Centre Tracts 159, pages 163–255, 1982.

[OwL82] Owicki, S. and Lamport, L.: Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, 1982.

[PaM93] Paulin-Mohring, C.: Inductive definitions in the system Coq. In *Typed Lambda Calculi and Applications*, LNCS 664, pages 328–345, 1993.

[PoS93] van de Pol, J. and Sellink, M. P. A.: Personal communication, 1993.

[Sel93] Sellink, M. P. A.: Verifying process algebra proofs in type theory. In D. J. Andrews, J. F. Groote, and C. A. Middelburg, editors, *Proceedings of the International Workshop on Semantics of Specification Languages*, Utrecht, The Netherlands, pages 315–339. Workshops in Computer Science, Springer-Verlag, 1993.

[Sel96] Sellink, M. P. A.: On the conservativity of Liebniz equality. Technical Report P9611, Programming Research Group, University of Amsterdam, 1996.

[Wer94] Werner, B.: *Une théorie des Constructions Inductives.* PhD thesis, Université de Paris 7, May 1994.

Received December 1994
Accepted in revised form November 1995 by A. J. R. G. Milner