Checking verifications of protocols and distributed systems by computer

Extended version of a tutorial at CONCUR'98 [32]

Jan Friso Groote[1,2] François Monin[2] Jaco van de Pol[2]

1: CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands.

2: Department of Mathematics and Computing Science, Eindhoven University of Technology,

P.O. Box 513, 5600 MB Eindhoven, The Netherlands.

E-mail: `JanFriso.Groote@cwi.nl` `monin@win.tue.nl` `jaco@win.tue.nl`

Abstract

We provide a treatise about checking proofs of distributed systems by computer using general purpose proof checkers. In particular, we present two approaches to verifying and checking the verification of the Sequential Line Interface Protocol (SLIP), one using rewriting techniques and one using the so-called cones and foci theorem. Both verifications are carried out in the setting of process algebra. Finally, we present an overview of literature containing checked proofs.

Note: The research of the second author is supported by Human Capital Mobility (HCM).

1 Proof checkers

Anyone trying to use a proof checker, e.g. Isabelle [67, 68], HOL [29], Coq [20], PVS [78], Boyer-Moore [14] or many others that exist today has experienced the same frustration. It is very difficult to prove even the simplest theorem. In the first place it is difficult to get acquainted to the logical language of the system. Most systems employ higher order logics that are extremely versatile and expressive. However, before we can use the system, we must learn the syntax to express definitions and theorems and we must also learn the language to construct proofs.

The second difficulty is to get used to strict logical rules that govern the reasoning allowed by the proof checker. Most of us have been educated in a mathematical style, which can be best described as intuitive reasoning with steps that are chosen to be sufficiently small to be acceptable by others. We all know examples of sound looking proofs of obviously wrong facts ('$1 = -1$', 'every triangle is isosceles', 'in every group of people all members have the same age'). In fact it is quite common that mathematical proofs contain flaws. Especially, the correctness of distributed programs and protocols is a delicate matter due to their nondeterministic and discrete character. Proof checkers are intended to ameliorate this situation.

One must get rid of the sloppiness of mathematical reasoning and get used to a more logical way of inferring facts. That is to say, one should not eliminate the mathematical intuition that helps guiding the proof, as the logical reasoning steps are so detailed that one easily looses track. And if this happens, even relatively short proofs, are impossible to find.

A typical exercise that was carried out using Coq during our first encounters with theorem checkers, gives an impression of the time required to provide a formal proof. We wanted to show that there does not exist a largest prime number. A well known mathematical proof of this fact goes like this.

Suppose there exists a largest prime n. So, as now the product of all prime numbers exists, let it be m. Now consider $m+1$. Clearly, dividing $m+1$ by any prime number yields remainder 1, and therefore $m+1$ is itself also a prime number, contradicting that n is the largest prime.

The formal proof requires that first a definition of natural numbers, the induction principle, multiplication, dividability and primality are given. Most theorem checkers contain nowadays libraries, where some of these notions, together with elementary lemmas are predefined and pre-proven. As a second step it is necessary to construct the product m of all prime numbers up to n (it is easier to construct the product of all numbers up to n) and prove that $m+1$ is not dividable by any number larger than 1. When doing this, it will turn out that the strict inductive proofs are not at all trivial, and need some thinking to find the appropriate induction hypotheses. It took more than a full month to provide the formalized proof, and we believe this to be typical for somebody with little experience in proof checking.

However, after having mastered a theorem checker, and after having proof checked the first theorems, the benefits from proof checking will become very obvious. In the first place one starts to appreciate the power of higher order logics and learns to see the difference between a proof, which can be transformed to be checked by a proof checker, and a 'proof' (or better 'intuitive story') for which the relation with a logical counterpart cannot be seen. On a more concrete level, one finds in almost any proof – and correctness proofs of distributed systems or protocols are no exception – flaws that even may have impact on the correctness of the protocol. A typical example is the equality between an implementation and specification stated on page 118 in [60] that was seen to be incorrect when a fully formalized proof was proof checked [47]. Using proof checkers can lead to a very strong emotion, which borders to addiction. As proof checkers makes one aware of ones own fallibility, which many people would not like to exhibit, the desire grows quickly to check every theorem using a proof checker. Unfortunately, proof checking is currently too time consuming to make this practical. However, the quality of proof checkers is steadily increasing meaning that from a certain point in the future proof checkers will be commonly used as they yield much more reliable proofs, and will most likely be more efficient than proving theorems by hand.

2 Proof checkers and concurrency

Concurrency and proof checkers are orthogonal fields. This means that proof checkers are not particularly aimed at any concurrency theory. Because we are most acquainted with proof checking within the context of process algebra, we provide a perspective from this field. However, most of our conclusions and guidelines carry over directly to any other perspective.

There are actually three requirements that need to be fulfilled for a theorem checker to be usable to check proofs of correctness of distributed systems.

1. The proof checker must be sufficiently expressive to encode the concepts occurring in the concurrency theory. Higher order provers such as Coq, PVS and Isabelle satisfy these requirements. For checkers that use restricted logics, such as Larch [37] and the Boyer-Moore prover [14], it is not immediately evident that they are suitable, as many concurrency theories use higher order concepts. However, in [84] a recent mechanical verification of the Oral Message Algorithm is provided in ACL2 (a successor of the Boyer-Moore prover [46]), and it is argued in that paper that most features of higher order logic can be easily translated in a first order framework.

2. The concurrency theory must have a sufficiently precise logical basis and reasoning in the theory must be in a sufficiently logical style. If this is not the case, one must expect to invest a lot of time providing a logical underpinning. An example from process algebra is found on page 35 of [5]. Here, the principle RSP (Recursive Specification Principle) is described rather sloppily by 'a guarded recursive specification has at most one solution'. In [7] a formulation of this principle is given in Coq, which fills almost an entire page of various definitions.

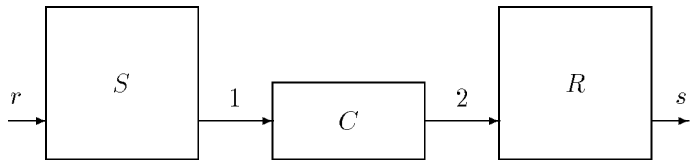

Figure 1: Architecture of the SLIP protocol

3. Finally, to really get a proof checker to work, the theory must be made effective. This means that either the formal proof cannot contain a too large number of steps, which can all be entered by hand, or the proof checker allows that large parts of the proof are constructed by the checker.

In one of our earliest encounters with a proof checker [8], we expanded the parallel operator into alternative and sequential composition using the standard axioms of ACP [5]. Given the large number of applications of axioms that were needed, we had to develop specific expansion theorems.

We have spent a lot of effort to make checking process algebraic proofs more tenable to be computer checked. This has boiled down in a method using cones and foci, which has been applied to a fully checked proof of the correctness of a distributed summing protocol [33]. Independently, an investigation into rewrite techniques has been carried out, which has been applied to the core of Philips' new Remote Control standard [36]. In the next sections we illustrate both techniques on the SLIP protocol.

3 The SLIP protocol

The Serial Line Interface Protocol (SLIP) is one of the protocols that is very commonly being used to connect individual computers via a modem and a phone line. It allows only one single stream of bidirectional information. This is a drawback, and therefore the SLIP protocol is gradually being replaced by the Point to Point Protocol (PPP) that allows multiple streams, such that several programs at one side can connect to several programs at the other side via one single line.

Basically, the SLIP protocol works by sending blocks of data. Each block is a sequence of bytes that ends with the special `end` byte. Confusion can occur when the `end` byte is also part of the ordinary data sequence. In this case, the `end` byte is 'escaped', by placing an `esc` byte in front of the `end` byte. Similarly, to distinguish an ordinary `esc` byte from the escape character `esc`, each `esc` in the data stream is replaced by two `esc` characters. In our modeling of the protocol, we ignore the process of dividing the data in blocks, but only look at the insertion and removal of `esc` characters in the data stream. We model the system by three components: a sender, inserting escape characters, a channel, modeling the medium along which data is transferred, and a receiver, removing the escape characters (see figure 1). We let the channel be a buffer of capacity one in this example.

We use four data types $\mathbb{N}$, **Bool**, *Byte* and *Queue* to describe the SLIP protocol and its external behaviour. The sort $\mathbb{N}$ contains the natural numbers. With 0 and S we denote the zero element and the successor function on $\mathbb{N}$. Numerals (e.g. 3) are used as abbreviations. The function $eq : \mathbb{N} \times \mathbb{N} \to \mathbf{Bool}$ is true when its arguments represent the same number. The sort **Bool** contains exactly two constants t (true) and f (false) and we assume that all required boolean connectives are defined.

The sort *Byte* contains the data elements to be transferred via the SLIP protocol. As the definition of a byte as a sequence of 8 bits is very detailed and actually irrelevant we only assume about *Byte* that it contains at least two not necessarily different constants `esc` and `end`, and a function $eq{:}Byte \times Byte \to Byte$ that represents equality. Using a proof checker, we can find out that we indeed did not need any other assumption on bytes.

Furthermore, to describe the external behaviour of the system, we introduce a sort *Queue* which we describe in slightly more detail to avoid the typical confusion that occurs with less standard data types. Queues are constructed using the empty queue $\emptyset$ and the constructor $in : Byte \times Queue \rightarrow Queue$. This means that we can apply induction over queues using these functions. Furthermore, we use the following auxiliary functions:

$$toe{:}Queue \rightarrow Byte,\ untoe{:}Queue \rightarrow Queue,$$
$$len{:}Queue \rightarrow \mathbb{N},\ empty, full{:}Queue \rightarrow \mathbf{Bool}$$

The function *toe* yields the element that was first inserted in the queue. The function *untoe* removes this element from the queue. We leave these functions undefined on the empty queue, as we do not require this information. The function *len* yields the length of the queue, *empty* says when the queue is empty and *full* yields a later to be explained criterion for what it means for a queue to be full. These functions are characterised by the following equations where d and d' range over *Byte* and q is a *Queue*.

$$toe(in(d,\emptyset)) = d,\ toe(in(d,in(d',q))) = toe(in(d',q))$$
$$untoe(in(d,\emptyset)) = \emptyset,\ untoe(in(d,in(d',q))) = in(d, untoe(in(d',q)))$$
$$empty(\emptyset) = \mathsf{t},\ empty(in(d,q)) = \mathsf{f}$$
$$len(\emptyset) = 0,\ len(in(d,q)) = S(len(q))$$
$$full(q) = eq(len(q),3) \vee (eq(len(q),2) \wedge (eq(toe(untoe(q)),\mathtt{esc}) \vee eq(toe(untoe(q)),\mathtt{end})))$$

We provide below the precise description of the SLIP protocol. For this we use process algebra with data in the form of μCRL ([5, 34]). The processes are defined by guarded recursive equations for the channel C, the sender S and the receiver R (cf. Figure 1). The equation for the channel C expresses that first a byte b is read using a read action via port 1, and subsequently this value is sent via port 2. After this the channel is back in its initial state, ready to receive another byte. The encircled numbers can be ignored for the moment. They serve to explicitly indicate the state of these processes and are used later.

Using the r action the sender reads a byte from a protocol user, who wants to use the service of the SLIP protocol to deliver this byte elsewhere. Using the two armed condition $p \lhd c \rhd q$, which must be read as if c then p else q, it is obvious that if b equals `esc` or `end` first an additional `esc` is sent to the channel (via action s_1) before b itself is sent. Otherwise, b is sent without prefix.

The receiver is equally straightforward. After receiving a byte b from the channel (via r_1) it checks whether it is an `esc`. If so, it removes it and delivers the trailing `end` or `esc`. Otherwise, it just delivers b. Both the sender and the receiver repeat themselves indefinitely, too.

In the fourth equation the SLIP protocol is defined by putting the sender, channel and receiver in parallel. We let the actions r_1 and s_1 communicate and the resulting action is called c_1. Similarly, r_2 and s_2 communicate into c_2. This is defined using the communication function γ by letting $\gamma(r_i, s_i) = c_i$ for $i = 1, 2$. The encapsulation operator $\partial_{\{r_1,s_1,r_2,s_2\}}$ forbids the actions r_1, s_1, r_2 and s_2 to occur on their own by renaming these actions to δ, which represents the process that cannot do anything. In this way the actions are forced to communicate. The hiding operation $\tau_{\{c_1,c_2\}}$ hides these communications by renaming them to the internal action τ. Using axioms $x\,\tau = x$ and $x{+}\tau\,x = \tau\,x$ in weak bisimulation [60], or $x\,(\tau\,(y+z)+z) = x\,(y+z)$ in branching bisimulation [5], the description of systems can be reduced, making obvious what the external behaviour of a system is. For the SLIP protocol the external actions are r and s that respectively read bytes to be transferred and delivers these bytes.

$$S = {}_{⓪}\textstyle\sum_{b:Byte} r(b)_{①}(s_1(\mathtt{esc})_{②}s_1(b)_{⓪}S \lhd eq(b,\mathtt{end}) \vee eq(b,\mathtt{esc}) \rhd s_1(b)_{⓪}S)$$

$$C = {}_{⓪}\textstyle\sum_{b:Byte} r_1(b)_{①}s_2(b)\,C$$

$$R = {}_{⓪}\textstyle\sum_{b:Byte} r_2(b)_{①}((\sum_{b:Byte} r_2(b)_{②}s(b)_{⓪}R) \lhd eq(b,\mathtt{esc}) \rhd s(b)_{⓪}R)$$

$$Slip = \tau_{\{c_1,c_2\}}\partial_{\{r_1,s_1,r_2,s_2\}}(S \parallel C \parallel R)$$

We want to obtain a better understanding of the protocol, because although rather simple, it is not straightforward to understand its external behaviour completely. Data that is read at r is of course delivered in sequence at s without loss or duplication of data. So, the protocol behaves like a kind of queue. The reader should now, before reading further, take a few minutes to determine the size of this queue[1]. Actually, the protocol behaves as a queue of size three, as long as there are no `esc` and `end` bytes being transferred. Simultaneously, one byte can be stored in the receiver, one in the channel and one in the sender. If an `esc` or `end` is in transfer, it matters whether it occurs at the first or second position in the queue. At the first position the `esc` or `end` is ultimately neatly stored in the receiver, taking up one byte position, allowing two other bytes to be simultaneously in transit. If this `esc` or `end` occurs at the second position, there must be a leading `esc` in the channel C, and the `esc` or `end` itself must be in the sender. Now, there is no place for a third byte. So, the conclusion is that the queue behaves itself as a queue of size three, except when an `esc` or `end` occurs at the second position in the queue, in which case the size is two. This explains the *full* predicate defined above, and yields the description of the external behaviour of the SLIP protocol below: If the queue is not full, an additional byte b can be read. If the queue is not empty an element can be delivered.

$$\begin{aligned} &Spec(q{:}Queue) = \\ &\quad \textstyle\sum_{b:Byte} r(b)\, Spec(in(b,q)) \lhd \neg full(q) \rhd \delta + \\ &\quad s(toe(q))\, Spec(untoe(q)) \lhd \neg empty(q) \rhd \delta \end{aligned}$$

The theorem that we are interested in proving and proof checking is:

Theorem 3.1.

$$\boxed{Slip = Spec(\emptyset)}$$

where '=' is interpreted as being branching or weakly bisimilar.

In Section 4 below we prove Theorem 3.1 directly using process algebraic axioms and rewriting techniques to make this approach tenable for proof checkers. In Section 5 we apply the cones and foci theorem and check the set of rather straightforward preconditions in PVS. The checked proofs can be obtained by contacting the authors.

4 Using rewrite systems in Isabelle/HOL

The direct proof method in process algebra consists of three steps:

1. Unfold the implementation by repeatedly calculating its first step expansion. This results in a system of guarded recursive equations.
2. Shrink this system by using the laws of weak (or branching) bisimulation.
3. Prove that the specification obeys the smaller set of equations.

[1] When trying to prove the correctness of the SLIP protocol, we erroneously took the size of the queue to be one. When proving equality between the SLIP protocol and such a queue, it became quickly obvious that this was a stupid thought. So, we took three for the size. But this is not correct, either.

The RSP-principle then guarantees that the specification and implementation are equal.

The bunch of work is in the first step expansion. Given a process $\tau_I\delta_H(S \parallel C \parallel R)$ this is of the form $\sum_i a_i\tau_I\delta_H(S_i \parallel C_i \parallel R_i)$, with a_i the possible first steps of the process. The process S_i denotes the sender after performance of a_i. The first step expansion must be repeated for the derivatives $\tau_I\delta_H(S_i \parallel C_i \parallel R_i)$. In this way, the computation tree of a process can be unfolded. To avoid an infinite unfolding of the process, names are introduced. These names can be used for sharing parts of the tree. The procedure of expansion is continued until a closed system of guarded equations is found. The introduction of new names and the criterion to terminate the unfolding remains the creative part of the proof.

The first step expansion is rather straightforwardly calculated using the axioms of process algebra. However, due to the large number of applications of axioms automation is desired. In Section 4.2 we will present a conditional higher-order rewrite system that given a parallel process computes its first-step expansion, without running into exceedingly large intermediary terms. But first we provide the laws of process algebra and its implementation in Isabelle/HOL. The method is applied to the SLIP protocol in Sections 4.3 and 4.4.

4.1 Formulation of Process Algebra in Isabelle

In Isabelle, terms have types, and the types are contained in classes. We introduce new classes `act` and `data`, and a communication function `gamma`. Here `act` is the class of action alphabets on which `gamma` is well-defined, and `data` is the class of types that may occur as data types in the processes. Given an alphabet `'a::act`, a type constructor `'a proc` is declared for the processes over the (polymorphic) alphabet `'a`.

After that, the process algebra operators are declared, and infix notation is introduced. We use `++, **, ||, !!, LL` for alternative, sequential, parallel composition, communication and left merge, respectively. Furthermore, `delta, tau, enc` and `hide` are used for δ, τ, encapsulation and hiding. `a<e>` denotes atomic action a with data element e, and `$ d::D. (p d)` denotes the process $\sum_{d:D} p(d)$. Finally, this approach uses the iterative construct `y @@ z` instead of the recursive definition $x = yx+z$. in traditional notation this is written y^*z, meaning that y is repeated zero or more times, and then z is executed. Recursive definitions would introduce new names (x), that must be manually folded and unfolded during proofs. As an example, the type of the summation operator is as follows:

```
$ :: ['d::data => 'a::act proc] => 'a proc
```

Here `'d` and `'a` are type variables, restricted to class `data` (for data types) and `act` (for action alphabets), respectively. Finally, the axioms of process algebra are turned into rules for Isabelle/HOL. Below we give an exhaustive list of the axioms we used. Note that we work with weak bisimulation which is slightly easier than branching bisimulation in the direct proof method. The conditions `gamdef a b c` and `gamundef a b` can be read as $\gamma(a,b) = c$ and $\gamma(a,b)$ is undefined, respectively.

```
A1 "x ++ y          = y ++ x"
A2 "(x ++ y) ++ z   = x ++ (y ++ z)"
A3 "x ++ x          = x"
A4 "(x ++ y) ** z   = x ** z  ++  y ** z"
A5 "(x ** y) ** z   = x ** y ** z"
A6 "x ++ delta      = x"
A7 "delta ** x      = delta"

D1  "(~ a mem H) --> enc H (a<d>) = a<d>"
D1d "enc H delta = delta"
D2  "a mem H --> enc H (a<d>) = delta"
D3  "enc H (x ++ y)  = enc H x  ++ enc H y "
```

```
D4  "enc H (x ** y)  = enc H x  ** enc H y "

CM1   "X||Y                    = X LL Y ++ Y LL X ++ X !! Y"
CM2   "a<d> LL X               = a<d> ** X"
CM2d  "delta LL X              = delta"
CM3   "a<d> ** X LL Y          = a<d> ** (X || Y)"
CM4   "(X++Y) LL Z             = X LL Z ++ Y LL Z"
CM5   "a<d> ** X !! b<e>       = (a<d> !! b<e>) ** X"
CM6   "a<d> !! b<e> ** X       = (a<d> !! b<e>) ** X"
CM7   "a<d> ** X !! b<e> ** Y = (a<d> !! b<e>) ** (X || Y)"
CM8   "(X ++ Y) !! Z           = X !! Z ++ Y !! Z"
CM9   "X !! (Y ++ Z)           = X !! Y ++ X !! Z"

CF1   "gamdef a b c --> a<d> !! b<d> = c<d>"
CF2   "gamundef a b --> a<d> !! b<e> = delta"
CF2'  "d ~= e       --> a<d> !! b<e> = delta"

SC1 "(x LL y) LL z  = x LL y || z"
SC2 "x LL delta     = x ** delta"
SC3 "x !! y         = y !! x"
SC4 "(x !! y) !! z  = x !! y !! z"
SC5 "x !!(y LL z)   = (x !! y) LL z"
SC6 "delta !! delta = delta"
HS  "x !! y !! z    = delta"

tau1 "x ** tau = x"
tau2 "x ++ tau ** x = tau ** x"

TI1  "~ a mem H --> hide H (a<e>) = a<e>"
TI1d "hide H delta = delta"
TI2  "a mem H --> hide H (a<e>)=tau"
TI3  "hide H (x ++ y) = hide H x ++ hide H y"
TI4  "hide H (x ** y) = hide H x ** hide H y"

S1 "$ d. x                = x"
S3 "$ d. (p d)            = ($ d. (p d)) ++ (p d)"
S4 "$ d. (p d) ++ (q d)  = ($ d. (p d)) ++ ($ d. (q d))"
S5 "($ d. (p d)) ** x    = $ d. (p d) ** x"
S6 "($ d. (p d)) LL x    = $ d. (p d) LL x"
S7 "($ d. (p d)) !! x    = $ d. (p d) !! x"
S8 "enc H ($ d. (p d))   = $ d. enc H (p d)"
S9 "hide H ($ d. (p d))  = $ d. hide H (p d)"

BKS1 "x @@ y = x ** (x @@ y) ++ y"
```

4.2 A rewrite system for the expansions

In order to find the first step expansion of a term, we have to apply the laws of process algebra with care. Many of these laws (regarded as rewrite rules) make copies of subterms leading to an unnecessary blow-up of intermediate terms (cf. CM1). Rather than programming a rewrite strategy in the theorem prover, we enlarge the usual rewrite rules with the context in which they may be applied. In this way we can control the application of the duplicating rewrite laws.

The essence of our strategy is to avoid the generation of many subterms that will eventually be encapsulated. We assume that the subterm to be expanded is of the shape `enc H (□++p)`. Here □ can be seen as the head and p as the tail of the list of summands to be processed. The rewrite rules are found by case analysis on the form of □. We will make sure that the duplication of subterms can only take place in the head of the term. The encapsulation is used to remove idle subterms as quickly as possible.

In order to start the system, a term `enc H (x || y || z)` first has to be transformed into `enc H (x || y || z ++ delta)`. The rewrite system then starts with the following rule:

```
enc H (x||y ++ p) = enc H (x LL y ++ x !! y ++ y LL x ++ p).
```

From now on the general shape will be `enc H (□ LL u ++ p)`, so we need an analogon of the previous rule:

```
enc H ((x || y) LL u ++ p)
 = enc H (x LL (y || u) ++ (x !! y) LL u ++ y LL (x || u) ++ p).
```

□ is either a single component or the communication between two components. These cases are dealt with by the following non-duplicating rules: CM2, CM3, CM5, CM6, CM7, CF1, CF2 and CF2' (and possibly their symmetric counterparts). Only the rules for alternative components (CM4, CM8 and CM9) are duplicating and have to be replaced by e.g.:

```
enc H ((x ++ y)LL u ++ p)  = enc H (x LL u ++ y LL u ++ p).
```

Eventually, the first summand is so small that it either can be discarded by the conditional rewrite rule

```
a mem H ==> enc H (a<d> ** x ++ p)   = enc H p,
```

or it contributes to the final result. In that case we apply

```
~ a mem H ==> enc H (a<d> ** x ++ p) = enc H p ++ a<d> ** enc H x,
```

in order to proceed with the next summand, which is the head of `p`. The summation symbols (`$`) are pulled to the front of the individual summands, using rules S4, S5, S6, S7 and its symmetric variant S7'. Eventually, a lot of summation signs can be eliminated after communication takes place, by adding rules like

```
 ($ d. (a<d> !! b<e>) ** (p d)) = (a<e>!!b<e>) ** p e.
```

As the latter rule is non-duplicating, we don't need the encapsulation context to steer its application. The iteration construct is only unfolded in certain contexts, such as

```
enc H (((x @@ y) !! z) LL u ++ p)
 = enc H ((x ** (x @@ y) !! z) LL u ++ (y !! z) LL u ++ p).
```

Finally, conditionals are pulled to the top of the terms by rules of the form:

```
(if b then p else q) !! x = (if b then (p !! x) else (q !! x)).
```

The complete set of rewrite rules can be found in the appendix. These rules have been proven in Isabelle using a much simpler rewrite system (basically the completion of the process algebra laws, cf. [1]). The rules have been gathered in a simplification set called `expand_ss`. Also tactics to automatically prove side conditions like $a \in H$ and `gamdef a b c` have been put into this simplification set. Finally, a tactic `choose` is defined, which (non-deterministically) applies the rule `enc H p = enc H (p ++ delta)`, in order to bring the term into the required shape. Using backtracking, the user can really choose which terms to expand.

4.3 Representation of the SLIP protocol

First, we have to define the alphabet of the SLIP protocol. We also define the communication-function `gamma` and state that `Act`, (in combination with `gamma`) is of class `act`. The latter yields some proof obligations that we now omit.

```
datatype Act = r | r1 | c1 | s1 | r2 | c2 | s2 | s
rule gamma_def
     "gamma == [(r1,s1,c1), (r2,s2,c2), (s1,r1,c1), (s2,r2,c2)]"
instance Act::act
```

Next we define the data types of the SLIP protocol. We deviate from the μCRL-specification, by using the lists from the Isabelle library, with hd, tl, @ (head, tail and append) instead of queues with toe and untoe.

```
types D
arities D:: data
consts ESC, END :: D
constdefs
  special :: "D=>bool"
  "special(d) == d=ESC | d=END"

  empty:: "D list=>bool"
  "empty(l) == l=[]"

  full :: "D list=>bool"
  "full(l) == length(l)=3 | (length(l)=2 & (special (hd (tl l))))"
```

Now we can introduce the specification. First we declare `Spec` and then we assert its recursive definition by an axiom

```
consts Spec :: "D list => Act proc"
rules Spec_def  "Spec(l) =
      (if (empty l) then delta else s<hd(l)> ** Spec(tl(l)))
       ++ (if (full l)  then delta else $ d. r<d> ** Spec(l @ [d]))"
```

We are now ready to define the protocol itself. Because we can now use iteration we don't need axioms but only definitions. For brevity we omit the types.

```
constdefs
  "HL == [r1,s1,r2,s2]"
  "TL == [c1,c2]"
  "S == ($d. r<d> ** (if (special d) then (s1<ESC> ** s1<d>)
                         else s1<d>)) @@ delta"
  "C == ($d::D. r1<d> ** s2<d>) @@ delta"
  "R == ($d. r2<d> ** (if (d=ESC) then ($d::D. r2<d> ** s<d>)
                         else s<d>)) @@ delta"
  "Slip == hide TL (enc HL (S || C || R))"
```

4.4 Verification of the SLIP protocol

With the machinery developed so far we can start the verification of the SLIP protocol. To this end we first define a number of auxiliary process terms.

```
constdefs
  "Slip1 d  == hide TL (enc HL (
    (if (special d) then (s1<ESC>**s1<d>) else s1<d>)**S||C||R))"
  "Slip2 d e == hide TL (enc HL (
    (if (special e) then (s1<ESC>**s1<e>) else s1<e>)**S||C||s<d>**R))"
  "Slip3 d  == hide TL (enc HL ( S || C || s<d> ** R))"
  "Slip4 d == hide TL (enc HL (s1<d> ** S || s2<ESC> ** C || R))"
  "Slip5 d == hide TL (enc HL (S || s2<d> ** C || R))"
  "Slip6 d e f == hide TL (enc HL (
    (if (special f) then (s1<ESC> ** s1<f>) else s1<f>) ** S
     || s2<e> ** C || s<d> ** R))"
```

We follow the three steps of the classical correctness proof. First the SLIP protocol is expanded.

```
Lemma1a: Slip = $ d. r<d> ** Slip1 d
Lemma1b: special(d) --> Slip1 d = tau ** Slip4 d
Lemma1c: ~special(d) --> Slip1 d = tau ** Slip5 d
Lemma1d: special(d) -->
   Slip4 d = tau ** (tau ** Slip3 d ++ ($ e. r<e> ** Slip2 d e))
Lemma1e: ~special(d) -->
   Slip5 d = tau ** Slip3 d ++ ($ e. r<e> ** Slip2 d e)
Lemma1f: Slip3 d = s<d> ** Slip ++ ($ e. r<e> ** Slip2 d e)
Lemma1g: special e -->
   Slip2 d e = tau ** s<d> ** Slip4 e ++ s<d> ** Slip1 e
Lemma1h: ~special e --> Slip2 d e =
 tau ** (s<d> ** Slip5 e ++ ($ f. r<f> ** Slip6 d e f)) ++ s<d> ** Slip1 e
Lemma1i: ~special e --> Slip6 d e f = s<d> ** Slip2 e f
```

To give an impression of the proof of this lemma the complete proof script for `Lemma1e` is printed below

```
by (rewrite_goals_tac [Slip5_def, S_def, C_def, R_def]);
br impI 1;
choose 1; by (asm_simp_tac expand_ss 1);
choose 1; back(); by (asm_simp_tac expand_ss 1);
by (rewrite_goals_tac [Slip2_def, Slip3_def, S_def, C_def, R_def]);
by (simp_tac tau_ss 1);
by (asm_full_simp_tac compare_ss 1);
qed "Lemma1e";
```

The first command unfolds the definitions in the left-hand side of the equation. The next command places the condition as an assumption in the context. Then one of the `enc`'s is chosen and expanded using the `expand_ss`-system. This is repeated for a second expansion. Note that the default choice of the system was wrong so we had to backtrack. After that we unfold the definitions in the right-hand side. Then we call the rewrite system for hiding, `tau_ss`. Finally the left- and right-hand side are compared. The latter step uses laws for commutativity of the alternative (A1) and parallel composition. Isabelle will not loop on such rules because it uses ordered rewriting.

By doing some subtle substitutions in the equations above and using the tau-laws (`tau1`, `tau2`) and the derived law $\tau(x+y)+x = \tau(x+y)$, we reduce the system to the following set of equations. These equations form a system of guarded recursive equations, of which `Slip` is a solution.

```
Lemma2a: Slip = $ d. r<d> ** Slip1 d
Lemma2b: Slip1 d = tau ** (s<d> ** Slip ++ ($ e. r<e> ** Slip2 d e))
```

```
Lemma2c: special(e) --> Slip2 d e = tau ** s<d> ** Slip1 e
Lemma2d: ~special(e) --> Slip2 d e =
         tau ** (s<d> ** Slip1 e ++ ($ f. r<f> ** s<d> ** Slip2 e f))
```

The next lemma indicates that `Spec[]` is another solution. For `Slip1 d` we substitute `tau ** Spec[d]` and for `Slip2 d e`, `tau ** Spec[d,e]` is substituted.

```
Lemma3a: Spec[] = $ d. r<d> ** tau ** Spec[d]
Lemma3b: tau ** Spec[d] =
           tau ** (s<d> ** Spec[] ++ ($ e. r<e> ** tau ** Spec[d,e]))
Lemma3c: special(e) --> tau ** Spec[d,e] = tau**s<d>**tau**Spec[e]
Lemma3d: ~special(e) --> tau ** Spec[d,e] =
           tau**(s<d>**tau**Spec[e] ++ ($ f. r<f>**s<d>**tau**Spec[e,f]))
```

Finally by RSP, `Slip = Spec[]`, but we didn't carry out this final step in Isabelle, as it would require quite a lot of extra formalization.

5 Using cones and foci in PVS

If protocols become more complex, it is not enough to resort to automating basic steps, but one must resort to effective meta theorems. As an example we present here the cones and foci theorem or general equality theorem [35, 33], and explain the formalisation of Theorem 3.1 and its proof in PVS (see [78]).

The basic observation underlying this method is that most verifications follow basically the same structure. The cones and foci theorem circumvents those verification steps that are similar and focuses on the parts that are different for each verification.

However, in order to be able to formulate such a general theorem, the format of processes as being used up till now is too general. Therefore, we introduce the so called linear process equation format to which large classes of processes can be automatically translated [13].

Definition 5.1. A *linear process equation* (*LPE*) over data type D is an expression of the form

$$X(d{:}D) = \sum_{i\in I} \sum_{e_i:E_i} c_i(f_i(d,e_i))\, X(g_i(d,e_i)) \lhd b_i(d,e_i) \rhd \delta$$

for some finite index set I, actions c_i, data types E_i, D_i, and functions $f_i : D \to E_i \to D_i$, $g_i : D \to E_i \to D$, $b_i : D \to E_i \to \mathbf{Bool}$. Here D represents the state space, c_i are the action labels, f_i represents the action parameters, g_i is the state transformation and b_i represent the condition determining whether an action is enabled.

Some remarks about this format are in order. First one should distinguish between the sum symbol with index $i \in I$ and the sum with index $e_i{:}E_i$. The first one is a shorthand for a finite number of alternative composition operators. The second one is a binder of the data variable e_i.

In [9] an LPE is defined as having also summands that allow termination. We have omitted these here, because they hardly occur in actual specifications and obscure the presentation of the theory.

LPEs are defined here having a single data parameter. The LPEs that we will consider generally have more than one parameter, but using cartesian products and projection functions, it is easily seen that this is an inessential extension.

Finally, we note that sometimes (and we actually do it below) it is useful to group summands per action such that $\Sigma_{i\in I}$ can be replaced by $\Sigma_{a\in Act}$ where Act is the set of action labels. Such LPEs are called clustered, and by introducing some auxiliary sorts and functions, any LPE can be transformed to a clustered LPE (provided actions have a unique type).

We call an LPE convergent if there are no infinite τ-sequences:

Definition 5.2. An LPE written as in Definition 5.1 is called *convergent* if there is a well-founded ordering $<$ on D such that for all $i \in I$ with $c_i = \tau$ and for all $e_i : E_i$, $d : D$ we have that $b_i(d, e_i)$ implies $g_i(d, e_i) < d$.

We describe the linear equation for *Slip*. We have numbered the different summands for easy reference. Note that the specification is already linear.

$$
\begin{array}{ll}
 & \mathit{LinImpl}(b_s{:}\mathit{Byte}, s_s{:}\mathbb{N}, b_c{:}\mathit{Byte}, s_c{:}\mathbb{N}, b_r{:}\mathit{Byte}, s_r{:}\mathbb{N}) = \\
\text{(a)} & \quad \sum_{b:\mathit{Byte}} r(b)\, \mathit{LinImpl}(b, 1, b_c, s_c, b_r, s_r) \\
 & \qquad \triangleleft eq(s_s, 0) \triangleright \delta + \\
\text{(b)} & \quad \tau\, \mathit{LinImpl}(b_s, 2, \mathsf{esc}, 1, b_r, s_r) \\
 & \qquad \triangleleft eq(s_c, 0) \wedge eq(s_s, 1) \wedge (eq(b_s, \mathsf{end}) \vee eq(b_s, \mathsf{esc})) \triangleright \delta + \\
\text{(c)} & \quad \tau\, \mathit{LinImpl}(b_s, 0, b_s, 1, b_r, s_r) \\
 & \qquad \triangleleft eq(s_c, 0) \wedge (eq(s_s, 2) \vee (eq(s_s, 1) \wedge \neg(eq(b_s, \mathsf{end}) \vee eq(b_s, \mathsf{esc})))) \triangleright \delta + \\
\text{(d)} & \quad \tau\, \mathit{LinImpl}(b_s, s_s, b_c, 0, b_c, 1) \\
 & \qquad \triangleleft eq(s_r, 0) \wedge eq(s_c, 1) \triangleright \delta + \\
\text{(e)} & \quad \tau\, \mathit{LinImpl}(b_s, s_s, b_c, 0, b_c, 2) \\
 & \qquad \triangleleft eq(s_r, 1) \wedge eq(b_r, \mathsf{esc}) \wedge eq(s_c, 1) \triangleright \delta + \\
\text{(f)} & \quad s(b_r)\, \mathit{LinImpl}(b_s, s_s, b_c, s_c, b_r, 0) \\
 & \qquad \triangleleft eq(s_r, 2) \vee (eq(s_r, 1) \wedge \neg eq(b_r, \mathsf{esc})) \triangleright \delta
\end{array}
$$

We obtained this form, by identifying three explicit states in the sender and receiver, and two in the channel. These have been indicated by encircled numbers in the defining equations of these processes. The states of these processes are indicated by the variables s_s, s_r and s_c respectively. Each of the three processes also stores a byte in certain states. The bytes for each process are indicated by b_s, b_r and b_c. The τ in summand (b) comes from hiding $c_1(\mathsf{esc})$, in summand (c) comes from $c_1(b_s)$, in (d) from $c_2(b_c)$ and in (e) from $c_2(b_c)$.

As we can obtain a linear equation for the SLIP protocol algorithmically, we do not think it useful to consider this aspect of the verification amenable for proof checking. Therefore, the following lemma has not been proof checked.

Lemma 5.3. For any $b_1, b_2, b_3{:}\mathit{Byte}$ it holds that

$$\mathit{LinImpl}(0, b_1, 0, b_2, 0, b_3) = \mathit{Slip}.$$

A very effective and commonly known notion is that of an invariant. Remarkably, invariants have hardly been used within process algebra. We use invariants without reference to an initial state.

Definition 5.4. An *invariant* of an LPE written as in Definition 5.1 is a function $I : D \to \mathbf{Bool}$ such that for all $i \in I$, $e_i : E_i$, and $d : D$ we have:

$$b_i(d, e_i) \wedge I(d) \to I(g_i(d, e_i)).$$

We list below a number of invariants of *LinImpl* that are sufficient to prove the results in the sequel. The proof of the invariants is straightforward, except that we need invariant 2 to prove invariant 3.

Lemma 5.5. The following expressions are invariants for *LinImpl*:

1. $s_s \leq 2 \wedge s_c \leq 1 \wedge s_r \leq 2$;
2. $eq(s_s, 2) \to (eq(b_s, \mathsf{esc}) \vee eq(b_s, \mathsf{end}))$;
3. $$\begin{array}{rl} \neg eq(s_s, 2) \to & ((eq(s_c, 0) \wedge \neg(eq(s_r, 1) \wedge eq(b_r, \mathsf{esc}))) \vee \\ & \ (eq(s_c, 1) \wedge ((eq(s_r, 1) \wedge eq(b_r, \mathsf{esc})) \leftrightarrow \\ & \qquad\qquad (eq(b_c, \mathsf{esc}) \vee eq(b_c, \mathsf{end}))))) \wedge \\ eq(s_s, 2) \to & ((eq(s_c, 1) \wedge eq(b_c, \mathsf{esc}) \wedge \neg(eq(s_r, 1) \wedge eq(b_r, \mathsf{esc}))) \vee \\ & \ (eq(s_c, 0) \wedge eq(s_r, 1) \wedge eq(b_r, \mathsf{esc}))). \end{array}$$

The next step is to relate the implementation and the specification. In order to do this abstractly, we first introduce a clustered linear process equation representing the implementation:

$$p(d{:}D_p) = \sum_{a\in Act}\sum_{e_a:E_a} a(f_a(d,e_a))\, p(g_a(d,e_a)) \lhd b_a(d,e_a) \rhd \delta$$

and a clustered linear process equation representing a specification:

$$q(d{:}D_q) = \sum_{a\in Act\setminus\{\tau\}}\sum_{e_a:E_a} a(f'_a(d,e_a))\, q(g'_a(d,e_a)) \lhd b'_a(d,e_a) \rhd \delta$$

Note that the specification does not have internal τ steps.

We relate the specification by means of a *state mapping* $h{:}D_p \to D_q$. The mapping h maps states of the implementation to states of the specification. In order to prove implementation and specification branching bisimilar, the state mapping should satisfy certain properties, which we call *matching criteria* because they serve to match states and transitions of implementation and specification. They are inspired by numerous case studies in protocol verification, and reduce complex calculations to a few straightforward checks.

In order to understand the matching criteria we first introduce an important concept, called a focus point. A focus point is a state in the implementation without outgoing τ-steps. Focus points are characterised by the *focus condition* $FC(d)$, which is true if d is a focus point, and false if not.

Definition 5.6. The *focus condition* $FC(d)$ of the implementation is the formula $\neg\exists e_\tau{:}E_\tau\,(b_\tau(d,e_\tau))$.

The set of states from which a focus point can be reached via internal actions is called the *cone* belonging to this focus point.

Now we formulate the criteria. We discuss each criterion directly after the definition. Here and below we assume that $\neg$ binds stronger than $\wedge$ and $\vee$, which in turn bind stronger than $\to$.

Definition 5.7. Let $h{:}D_p \to D_q$ be a state mapping. The following criteria are called the *matching criteria*. We refer to their conjunction by $C_{p,q,h}(d)$.

$$\text{The LPE for } p \text{ is convergent} \tag{1}$$

$$\forall e_\tau{:}E_\tau(b_\tau(d,e_\tau) \to h(d) = h(g_\tau(d,e_\tau))) \tag{2}$$

$$\forall a \in Act\setminus\{\tau\}\forall e_a{:}E_a\,(b_a(d,e_a) \to b'_a(h(d),e_a)) \tag{3}$$

$$\forall a \in Act\setminus\{\tau\}\forall e_a{:}E_a\,(FC_\Xi(d) \wedge b'_a(h(d),e_a) \to b_a(d,e_a)) \tag{4}$$

$$\forall a \in Act\setminus\{\tau\}\,\forall e_a{:}E_a\,(b_a(d,e_a) \to f_a(d,e_a) = f'_a(h(d),e_a)) \tag{5}$$

$$\forall a \in Act\setminus\{\tau\}\,\forall e_a{:}E_a\,(b_a(d,e_a) \to h(g_a(d,e_a)) = g'_a(h(d),e_a)) \tag{6}$$

Criterion (1) says that the implementation must be convergent. In effect this does not say anything else than that in a cone every internal action τ constitutes progress towards a focus point. In [35] also an extension of this method where convergence of the implementation is not necessary is presented.

Criterion (2) says that if in a state d in the implementation an internal step can be done (i.e. $b_\tau(d,e_\tau)$ is valid) then this internal step is not observable. This is described by saying that both states relate to the same state in the specification.

Criterion (3) says that when the implementation can perform an external step, then the corresponding point in the specification must also be able to perform this step. Note that in general, the converse need not hold. If the specification can perform an a-action in a certain state e, then it is only necessary that in every state d of the implementation such that $h(d) = e$ an a-step can be done *after some internal actions*.

This is guaranteed by criterion (4). It says that in a focus point of the implementation, an action a in the implementation can be performed if it is enabled in the specification.

Criteria (5) and (6) express that corresponding external actions carry the same data parameter (modulo h) and lead to corresponding states.

Using the matching criteria, we would like to prove that, for all $d{:}D_p$, $C_{p,q,h}(d)$ implies $p(d) = q(h(d))$. This can be done using the following theorem.

Theorem 5.8 *(General Equality Theorem [35]). Let p and q be defined as above. If I is an invariant of the defining LPE of p and $\forall d{:}D_p\ (I(d) \to C_{p,q,h}(d))$, then*

$$\forall d{:}D_p\ I(d) \to p(d) \triangleleft FC(d) \triangleright \tau\, p(d) = q(h(d)) \triangleleft FC(d) \triangleright \tau\, q(h(d)).$$

For the SLIP protocol we define the state mapping using the auxiliary function $cadd$. The expression $cadd(c, b, q)$ yields a queue with byte b added to q if boolean c equals true. If c is false, it yields q itself. Hence the conditional add is defined by the equations $cadd(\mathsf{f}, b, q) = q$ and $cadd(\mathsf{t}, b, q) = in(b, q)$.

The state mapping is in this case:

$$\begin{aligned}
&h(b_s, s_s, b_c, s_c, b_r, s_r) = \\
&\qquad cadd(\neg eq(s_s, 0), b_s, \\
&\qquad cadd(eq(s_c, 1) \wedge (\neg eq(b_c, \mathsf{esc}) \vee (eq(s_r, 1) \wedge eq(b_r, \mathsf{esc}))), b_c, \\
&\qquad cadd(eq(s_r, 2) \vee (eq(s_r, 1) \wedge \neg eq(b_r, \mathsf{esc}), b_r, \emptyset)))).
\end{aligned}$$

So, the state mapping constructs a queue out of the state of the implementation, containing at most b_s, b_c and b_r in that order. The byte b_s from the sender is in the queue if the sender is not about to read a new byte ($\neg eq(s_s, 0)$). The byte b_c from the channel is in the queue if the channel is actually transferring data ($eq(s_c, 1)$) and if it does not contain an escape character indicating that the next byte must be taken literally. Similarly, the byte b_r from the receiver must be in the queue if it is not empty and b_r is not an escape character.

The focus condition of the SLIP implementation can easily be extracted and is (slightly simplified using the invariant):

$$\begin{aligned}
&(eq(s_c, 0) \to eq(s_s, 0)) \wedge \\
&(eq(s_c, 1) \to (\neg eq(s_r, 0) \wedge (eq(s_r, 1) \to \neg eq(b_r, \mathsf{esc}))))
\end{aligned}$$

Lemma 5.9. For all $b_1, b_2, b_3{:}Byte$

$$Spec(\emptyset) = LinImpl(b_1, 0, b_2, 0, b_3, 0).$$

Proof. We apply Theorem 5.8 by taking $LinImpl$ for p, $Spec$ for q and the state mapping and invariant provided above. We simplify the conclusion by observing that the invariant and the focus condition are true for $s_s = 0$, $s_c = 0$ and $s_r = 0$. By moreover using that $h(b_1, 0, b_2, 0, b_3, 0) = \emptyset$, the lemma is a direct consequence of the generalized equation theorem. We are only left with checking the matching criteria:

1. The measure $13 - s_s - 3s_c - 4s_r$ decreases with each τ step.

2. (b) Distinction on s_r; use invariant. (c) Distinguish different values of s_s; use invariant. (d) Trivial. (e) Trivial.

3. (a) Lengthy. (f) Trivial.

4. (a) We must show that if the focus condition and $\neg full(h(b_s, s_s, b_c, s_c, b_r, s_r))$ hold, then $eq(s_s, 0)$. The proof proceeds by deriving a contradiction under the assumption $\neg eq(s_s, 0)$. If $eq(s_s, 1)$ it follows from the invariant and the focus condition that $len(h(b_s, s_s, b_c, s_c, b_r, s_r)) = 3$, contradicting that $\neg full(h(b_s, s_s, b_c, s_c, b_r, s_r))$. If $eq(s_s, 2)$, then $len(h(b_s, s_s, b_c, s_c, b_r, s_r)) = 2$,

$toe(untoe(h(b_s, s_s, b_c, s_c, b_r, s_r))) = b_s$ and $eq(b_s, \mathtt{esc}) \vee eq(b_s, \mathtt{end})$ in a similar way. This also contradicts $\neg full(h(b_s, s_s, b_c, s_c, b_r, s_r))$.

(f) We must show that the invariant, focus condition and $\neg empty(h(b_s, s_s, b_c, s_c, b_r, s_r))$ implies $eq(s_r, 2) \vee (eq(s_r, 1) \wedge \neg eq(b_r, \mathtt{esc}))$. Assume FC and, towards using contraposition, $\neg eq(s_r, 2) \wedge (\neg eq(s_r, 1) \vee eq(b_r, \mathtt{esc}))$. Using the invariant we deduce $eq(s_r, 0) \vee (eq(b_r, \mathtt{esc}) \wedge eq(s_r, 1)$. By the second conjunct of the FC (contraposition), we obtain $\neg eq(s_c, 1)$, so by the invariant, $eq(s_c, 0)$, and by the first conjunct of FC, $eq(s_s, 0)$ holds. By the definition of the state mapping h, we easily see that $h(b_s, s_s, b_c, s_c, b_r, s_r) = empty$.

5. (a) Trivial. (f) Use $toe(cadd(c_1, b_1, cadd(c_2, b_2, in(b_3, \emptyset)))) = b_3$.

6. (a) Trivial using definitions (f) Idem.

□

Using Lemmas 5.3 and 5.9 it is easy to see that Theorem 3.1 can be proven.

Only now we come to the actual checking of this protocol in PVS. We concentrate on proving the invariant and the matching criteria. We must choose a representation for all concepts used in the proof. As this would make the paper too long, we only provide some definitions and highlight some steps of the proof, giving a flavour of the input language of PVS.

We start off defining the data types.

```
Byte:TYPE+      Queue:TYPE=list[Byte]
endb:Byte       DX   :TYPE=[Byte,upto(2),Byte,upto(1),Byte,upto(2)]
esc :Byte       DY   :TYPE=[Queue]
```

We use as much of the built-in data types of PVS as possible. The advantage of this is that we can use all knowledge of PVS about these data types. A disadvantage is that the semantics of the data types in PVS may differ from the semantics of data types in the protocol, leading to mismatches between the computerized proof and the intended proof.

The types $\mathbb{N}$ and **Bool** are built in types of PVS and need not be defined. We declare `Byte` to be a nonempty type, with two elements `esc` and `endb` (`end` is a predefined symbol and can therefore not be used). For queues we take the built in type list and parameterize it with bytes. The type of the parameters of the linear implementation and the specification are now given by `DX` and `DY` respectively. The type `upto`(n) denotes a finite type with natural numbers up to and including n.

A function such as *untoe* can now be defined in the following way:

```
untoe(q:Queue):RECURSIVE Queue=if null?(q) then null else
                              if null?(cdr(q)) then null else
                                 (cons (car(q),untoe(cdr(q))))
                         endif endif
MEASURE(lambda(q:Queue) : length(q))
```

The function `car`, `cdr` and `null` are built in PVS. The `MEASURE` statement is added to help PVS finding criteria for the well-foundedness of the definition, which is in this case obtained via the length of the queue.

Below we show how a linear process equation is modeled. In essence the information contents of an LPE is the set D, the index set I, the sets E_i, the actions a_i and the function f_i, g_i and b_i.

We only provide the LPE representation for the linear implementation of the SLIP protocol. The set D is given as `DX` defined above. We group all τ-actions, which leaves us with three summands. We assume this a priori (and have even encoded this bound in all theorems) as making it more generic would make the presentation less clear. Using the knowledge that there are only three summands, we can define the sets E_i very explicitly: `E1:TYPE=Byte`, `E2:TYPE=upto(0)` and `E3:TYPE=upto(3)`.

Here, `upto(0)` is a set with exactly one element. Furthermore `E3` is taken to contain the numbers $0, \ldots, 3$ to refer to the different τ actions in the linear implementation.

The constituents of the different summands are given by the record fields `u1`, `u2` and `u3`. The notation `(#u1:=...,u2:=...,...#)` stands for a record with fields `u1`, etc. Each summand consists again of a record. The first field of this record gives the name of an action (in PVS we have defined `ra` for r, `sa` for s and `tau` for τ). The second, third and fourth components are the functions f_i, g_i and b_i.

```
L_Impl : LPE =
(#u1:=(#
    a:=ra,
    f:=(lambda (bs:Byte,ss:upto(2),bc:Byte,sc:upto(1),br:Byte,sr:upto(2)):
           (lambda (b:Byte) : b)),
    g:=(lambda (bs:Byte,ss:upto(2),bc:Byte,sc:upto(1),br:Byte,sr:upto(2)):
           (lambda (b:Byte) : (b,1,bc,sc,br,sr))),
    b:=(lambda (bs:Byte,ss:upto(2),bc:Byte,sc:upto(1),br:Byte,sr:upto(2)):
           (lambda (b:Byte) : (ss=0)))#),
  u2:=(#
    a:=sa,
    f:=(lambda (bs:Byte,ss:upto(2),bc:Byte,sc:upto(1),br:Byte,sr:upto(2)):
           (lambda (u:upto(0)):br)),
    g:=(lambda (bs:Byte,ss:upto(2),bc:Byte,sc:upto(1),br:Byte,sr:upto(2)):
           (lambda (u:upto(0)):(bs,ss,bc,sc,br,0))),
    b:=(lambda (bs:Byte,ss:upto(2),bc:Byte,sc:upto(1),br:Byte,sr:upto(2)):
           (lambda (u:upto(0)):((sr=2) or ((sr=1) and br/=esc))))#),
  u3:=(#
    a:=tau,
    f:=(lambda (bs:Byte,ss:upto(2),bc:Byte,sc:upto(1),br:Byte,sr:upto(2)):
           (lambda (st:upto(3)) : 0)),
    g:=(lambda (bs:Byte,ss:upto(2),bc:Byte,sc:upto(1),br:Byte,sr:upto(2)):
           (lambda (st:upto(3)) :
            if st=0 then (bs,2,esc,1,br,sr)
       else if st=1 then (bs,0,bs,1,br,sr)
       else if st=2 then (bs,ss,bc,0,bc,1)
       else (bs,ss,bc,0,bc,2) endif endif endif)),
    b:=(lambda (bs:Byte,ss:upto(2),bc:Byte,sc:upto(1),br:Byte,sr:upto(2)):
           (lambda (st:upto(3)) :
            if st=0 then (sc=0 and ss=1 and (bs=endb or bs=esc))
       else if st=1 then (sc=0 and (ss=2 or (ss=1 and not(bs=endb or bs=esc))))
       else if st=2 then (sr=0 and sc=1)
       else (sr=1 and br=esc and sc=1) endif endif endif))#)
#)
```

Below we provide a PVS description of what it means to be an invariant for a predicate I on a given LPE, and we formulate the general equation theorem. Here `Sol(lpox)` yields the solution of an LPO `lpox`. We turned the general equation theorem (Thm. 5.8) into an axiom, as our current aim is not to verify theoretical results.

```
Invlpox(lpox: LPE[DX],I: [DX -> bool]) : bool =
  (FORALL (e:E1,d:DX):(b(u1(lpox))(d)(e) and I(d))=>I(g(u1(lpox))(d)(e)))
  AND
  (FORALL (e:E2,d:DX):(b(u2(lpox))(d)(e) and I(d))=>I(g(u2(lpox))(d)(e)))
```

```
  AND
  (FORALL (e:E3,d:DX):(b(u3(lpox))(d)(e) and I(d))=>I(g(u3(lpox))(d)(e)))

GET : AXIOM FORALL (lpox: LPE[DX],lpoy: ALPE[DY],h: [DX -> DY],
                       I: [DX -> bool]) :
  Invlpox(lpox,I) and
 (FORALL (d: DX) : I(d) => Convx(lpox) and Crit2(lpox,d,h) and
                    Crit3(lpox,lpoy,d,h) and Crit4(lpox,lpoy,d,h) and
                    Crit5(lpox,lpoy,d,h) and Crit6(lpox,lpoy,d,h)) =>
  FORALL (d: DX) : I(d) =>
      condi(Sol(lpox)(d),FC(lpox,d),seq(tau,Sol(lpox)(d)))
                                 =
      condi(Sol(lpoy)(h(d)),FC(lpox,d),seq(tau,Sol(lpoy)(h(d))))
```

Here `seq, tau, condi` stand for the sequential operator $\cdot$, the τ process and the conditional construct, respectively. Moreover, `Convx,Crit2,...,Crit6` represent the six criteria of Definition 5.7 and `FC` the focus condition and `lpoy: ALPE[DY]` is an LPO without τ-actions.

The state mapping `stmapp` can be formalized in PVS in a very straightforward way (but we first define *cadd*):

```
cadd(x:bool,b:Byte,q:Queue):Queue =
  if x=false then q else cons(b,q) endif

stmapp(bs:Byte,ss:upto(2),bc:Byte,sc:upto(1),br:Byte,sr:upto(2)):Queue=
cadd(ss/=0,bs,cadd(sc=1 and (bc/=esc or (sr=1 and br=esc)),bc,
    cadd(sr=2 or (sr=1 and br/=esc),br,null)))
```

To assert the `MAINTHM` theorem described below in PVS corresponding to the main Lemma 5.9, `GET` is to be applied with the following instantation `L_IMPL, L_Spec, stmapp, Inv` where `Inv` is an encoded expression of the invariants defined in Lemma 5.5. After application of `GET` theorem one is confronted with a long list of proof obligations. They could be proved with several seperate lemmas. To get an impression of how they look like, we provide below the lemma that corresponds to the sixth matching criterion. It has been proven using the built in `grind` tactic.

```
Inv(bs,ss,bc,sc,br,sr) => (ss=0 =>
stmapp(b,1,bc,sc,br,sr)=cons(b,stmapp(bs,ss,bc,sc,br,sr))) AND ((sr=2
or (sr=1 and br/=esc)) =>
stmapp(bs,ss,bc,sc,br,0)=untoe(stmapp(bs,ss,bc,sc,br,sr)))
```

After having proved the whole list of obligations, we can conclude:

```
MAINTHM : THEOREM
forall (b1,b2,b3: Byte) :
     Sol(L_Impl)(b1,0,b2,0,b3,0)=Sol(L_Spec)(null)
```

6 Which proof checker to use?

This is an obvious question that is not easy to answer. We only have substantial experience with Coq, Isabelle and PVS, and only tried some others. The conclusion is that none of the checkers is perfect and all are suited for the verification of correctness proofs of protocols.

PVS has large built in libraries and has the largest amount of ad hoc knowledge and specialised decision procedures. This makes it an efficient theorem checker and relatively easy to use for beginners. However, it is not always obvious what the procedures do, hindering fundamental understanding of how the prover achieves its results. Moreover, these built-in procedures operate unchecked, and therefore may erroneously prove a lemma. There is no independent check in the system. Regularly, problems or bugs are reported, which are dealt with adequately.

Coq has by far the nicest underlying theory, which is not very easy to understand, however. Coq uses a strict separation between constructing a proof and checking it. Actually, using the Curry-Howard isomorphism, a term (=proof) of a certain type (=theorem) is constructed using the vernacular of Coq. After that the term and type are sent to a separate type checker, which double checks whether the term is indeed of that type, or equivalently the proof is indeed a proof of the theorem. In a few rare cases we indeed constructed proofs that were incorrect, and very nicely intercepted in this way. This gives Coq by far the highest reliability of the provers.

A disadvantage of Coq is that it is relatively hard to get going. This is due to the fact that the theory is difficult, and there are relatively few and underdeveloped libraries. Furthermore, automatic proof searching is less supported in Coq then in PVS and Isabelle.

Isabelle is the most difficult theorem prover to learn. This is due to the fact that the user must have knowledge of the object logic (HOL, but there are others) and the metalogic (Higher order minimal logic). An advantage of this two level approach is that proof search facilities have a nice underpinning in the meta logic. These facilities include backtracking, higher order unification and resolution. Although there are no proof objects that are separately checked such as in Coq, Isabelle operates through a kernel, making it much more reliable than PVS. Term rewriting is an exception, as it has been implemented outside this kernel for efficiency reasons, but it is very powerful as ordered conditional higher-order rewriting is implemented, and rather efficient.

7 Overview of the literature

Nowadays numerous proofs of protocols and distributed systems have been computer checked. The techniques that have been used for proving are mainly based on temporal logic and process algebra. The examples of computer checked verifications presented here do not cover the whole field. Moreover, a number of the authors published other verifications than the mentioned ones. Nevertheless, this overview gives a good impression of the state of the art.

In the context of process algebra [5] most such checks have been carried out using the language μCRL [34]. It has been encoded in the Coq system and applied to the verification of the alternating bit protocol [8, 7], Milner's scheduler [47], a bounded retransmission protocol [36] and parallel queues [48]. μCRL has also been encoded in PVS and a distributed summing protocol has been computer checked in [33] using the methodology presented in [35].

Temporal logic has been mainly used for proving safety (invariance) properties and liveness (eventuality) properties of concurrent systems. The temporal logic of actions (TLA), developed by Lamport [50], allows systems and properties to be described in the same language. The semantics of TLA has been formalized in the HOL theorem checker [29] in [81] and a mutual exclusion property for an increment example and the refinement of a specification were proven and the proof was checked.

In [22], a translator was devised to directly translate TLA into the language of Larch Prover [37]. Examples verified in this approach are an invariance property of a spanning tree algorithm [22], correctness of an N-bit multiplier [21]. TLA has also been applied for specifying and verifying an industrial retransmission protocol RLP1 (Radio Link Protocol) in [61] of which the proofs were checked with the theorem prover Eves [28].

A subset of the temporal formalism of Manna and Pnueli [59] has been encoded in the Boyer-Moore prover by Russinoff in [74] in order to mechanically verify safety and liveness properties of concurrent programs. He applied this system to check several concurrent algorithms of which the most difficult was the Ben-Ari's incremental garbage collection algorithm [75]. Furthermore, Goldschlag encoded the Unity formalism in the Boyer-Moore prover in [26, 27]. Unity, developed by Chandy and Misra [15], is a programming notation with a temporal logic for reasoning about the computations of the concurrent programs. To illustrate the suitability of the proof systems, Goldschlag respectively specified and proved the correctness of a solution to mutual exclusion algorithm, the solution of the dining philosopher's problem, a distributed algorithm computing the minimum node value in a tree and an n-bit delay insensitive FIFO queue. We can also mention that a distributed minimal spanning tree algorithm [23] was verified [41] using the Boyer-Moore theorem checker in the pre-post condition style; the last author also verified the PIF algorithm [42]. Finally, in the same manner a distributed memory management algorithm has been checked mechanically in [31].

The Unity community has also used the Larch Prover to study a communication protocol over faulty channels [16]. The informal proof of safety and liveness properties of the protocols given in [15] have been computer checked revealing some flaws. Unity has been implemented in other theorem checkers as in [17] where an industrial protocol is being studied.

Various protocols have been studied based on Input/Output automata proposed by Lynch and Tuttle [58]. A verification of a network transmission protocol has been checked in [66] using a model of I/O automata formalized in [66, 64]. In [18], a verification of a leader election protocol extracted from a serial multimedia bus protocol has been partially checked with PVS. Also an audio control protocol has been analysed in [12] in the context of the I/O automata model [57] of which some proofs were checked using the Coq system [39] and a similar protocol was studied with the Larch Prover in [30]. Still using the Larch Prover, a behavioral equivalence between two high-level specifications for a reliable communication protocol is proven in [79] and a proof of the bounded concurrent time stamp algorithm [19] made in [24] has been completely checked in [71]. In [56], the correctness of a simple timing-based counter and Fisher's mutual exclusion protocol were respectively formally proven with the Larch Prover.

Timed automata [57] have been modeled in PVS and applied in [2] to formally prove invariant properties of the generalized railroad crossing system based on the proof of [40]. The same authors [3] verified the Steam Boiler Controller problem leading to corrections of the manual proof in [53].

Other formal frameworks have been applied to the verification of previous examples. The alternating bit protocol was checked in Coq in [25]. We can mention [77] where the Fisher mutual exclusion protocol and the railroad crossing controller were verified in PVS. The former is also done with PVS in [54] and the latter is proved with the Boyer-Moore prover in [83]. In [80], the steam boiler was checked by Vitt and Hooman also using PVS. The last author also verified a processor-group membership protocol and the binary exponential backoff protocol [44, 45]; and a safety property, together with a real-time progress property of the ACCESS bus protocol in [43]. Also the biphase mark protocol, similar to the protocol in [12], was proved by Moore in [62].

As an interesting benchmark problem for specification and verification, the interactive convergence clock synchronization algorithm [51] has been mechanically checked respectively with the Boyer-Moore prover in [82] and with PVS in [73]. Also, several versions of the oral messages algorithm [52] have been proved correct in [84] with the new version ACL2 [46] of Nqthm and with PVS in [76, 72, 55]. Nqthm is also used by [65].

Since several years, numerous protocols have been checked in the field of security systems whith modal logic or general purpose formal methods. Among many checked cryptographic protocols, the protocols [6, 69, 70] were proved using Isabelle and the protocols [4, 11] were proved with Coq.

Examples of protocols or distributed systems have also been verified in a combination of theorem proving and model checking. An 8.2^m-bit multiplier was verified with LP for arbitrary values of m in [49]. In [63] the correctness of the alternating bit protocol was proved with Isabelle and the bounded retransmission protocol, previously checked in [39, 36], was proved with PVS in [38]. Also using Coq and a model checker, a broadcasting protocol has been verified in [10].

A The set of rewrite rules (appendix to Section 4)

We present the set of rewrite rules in four parts. First the rules for ACP and standard concurrency are presented. Then the extensions with the sum-operator, the star-operator and the conditionals is presented. All equations are to be read as rewrite rules from left to right.

A.1 ACP with standard concurrency

There are three kinds of rules here. Purely administrative rules, that just rearrange the terms. Then there are rules to compute left merges and communication merges. Finally, there are rules to contract terms containing deltas.
Administrative rules:

```
(x ++ y) ++ z  = x ++ (y ++ z)
(x ** y) ** z  = x ** (y ** z)
(x || y) || z  = x || (y || z)

enc H (x||y ++ p) =
   enc H (x LL y ++ x !! y ++ y LL x ++ p)

enc H ((x || y) LL u ++ p)  =
   enc H (x LL (y || u) ++ (x !! y) LL u ++ y LL (x || u) ++ p)

enc H (x !! (y || z) ++ p)  =
   enc H ((x !! y)LL z ++ (x !! z)LL y ++ p)

enc H ((x !! (y || z)) LL u ++ p)   =
   enc H ((x !! y) LL (z || u) ++ (x !! z) LL (y || u) ++ p)

enc H (((x ++ y) !! z) LL u ++ p)  =
   enc H ((x !! z) LL u ++ (y !! z) LL u ++ p)

enc H ((x !! (y ++ z)) LL u ++ p) =
   enc H ((x !! y)LL u ++ (x !! z)LL u ++ p)

enc H ((x ++ y)LL u ++ p)  =
   enc H (x LL u ++ y LL u ++ p)
```

These rules can be easily proved from the axioms, apart from associativity of `||`. The latter requires a lot of sophisticated applications of the laws of standard concurrency. In Isabelle, this proof heavily depends on ordered rewriting with the appropriate rules and backtracking.

Rules for left merge and communication merge. Rules four and five below need explicit typing, because we must ensure that `d` and `e` are of the same type `'a::data`. Without type annotations, Isabelle would choose the most general typing, giving `d` and `e` different types. `==>` denotes meta-implication, used in conditional rewrite rules.

```
delta LL X      = delta
a<d>  LL X      = a<d> ** X
a<d> ** X LL Y  = a<d> ** (X || Y)
(a<d::'a::data> !! b<e::'a>) ** x LL y = (a<d> !! b<e>) ** (x||y)
(a<d::'a::data> !! b<e::'a>)      LL x = (a<d> !! b<e>) ** x

a<d> ** X !! b<e>       = (a<d> !! b<e>) ** X
a<d>      !! b<e> ** Y  = (a<d> !! b<e>) ** Y
```

```
a<d> ** X !! b<e> ** Y  = (a<d> !! b<e>) ** (X || Y)

a mem H   ==> enc H (a<d> ** x ++ p) = enc H p
~ a mem H ==> enc H (a<d> ** x ++ p) = enc H p ++ a<d> ** enc H x

[|gamdef a b c  ; d=e |] ==> a<d> !! b<e> = c<d>
gamundef a b             ==> a<d> !! b<e> = delta
d ~= e                   ==> a<d> !! b<e> = delta
```

These rules follow easily from the axioms, apart from the fourth and fifth. The latter require a case distinction on whether $d = e$ and whether $\gamma(a, b)$ is defined or not. All cases are easy.
Delta contraction rules:

```
delta ** x  = delta
delta ++ x  = x
enc H delta = delta
delta !! x  = delta
x !! delta  = delta
```

The latter two follow from SC6 and the Handshaking axiom. The other rules are simply derivable.

A.2 Extension with the infinite sum operator

We now distinguish between rules that distribute the sum-operator, rules that eliminate it, and analogons of ACP-rules for the sum-operator.
Rules that distribute the sum:

```
($ d. (p d)) ** x    = $ d. (p d) ** x
($ d. (p d)) LL x    = $ d. (p d) LL xS6,
($ d. (p d)) !! x    = $ d. (p d) !! xS7,
x !! ($ d. (P d))    = $ d. x !! (P d)
$ d. (p d) ++ (q d)  = ($ d. (p d)) ++ ($ d. (q d))
```

These rules are such that the sum-operator is pulled to the front of the individual summands. These rules can easily be proved from the axioms.

Rules that eliminate the sum:

```
$ d. delta = delta
($ d. (a<d> !! b<e>) ** (p d)) = (a<e>!!b<e>) ** p e
($ d. (a<e> !! b<d>) ** (p d)) = (a<e>!!b<e>) ** p e
```

The proof of the last two rules is quite intricate. We sketch the proof of the first, as the second goes similarly. The derivation in traditional notation:

$$
\begin{aligned}
& \textstyle\sum_d((a(d)|b(e))p(d)) \\
\stackrel{S3}{=} \; & \textstyle\sum_d((a(d)|b(e))p(d)) + (a(e)|b(e))p(e) \\
\stackrel{S1}{=} \; & \textstyle\sum_d((a(d)|b(e))p(d)) + \sum_d((a(e)|b(e))p(e)) \\
\stackrel{S4}{=} \; & \textstyle\sum_d((a(d)|b(e))p(d) + (a(e)|b(e))p(e)) \\
\stackrel{*}{=} \; & \textstyle\sum_d((a(e)|b(e))p(e)) \\
\stackrel{S1}{=} \; & a(e)|b(e))p(e))
\end{aligned}
$$

For (*), distinguish cases, either $d = e$ (and use A3), or $d \neq e$ (and use CF2",A7,A6).

Sum-variants of ACP-rules.

```
enc H (($ d. (P d) !! (y || z)) ++ p)  =
   enc H (($ d. ((P d) !! y)LL z) ++ ($ d. ((P d) !! z)LL y) ++ p)

enc H (($ d. (P d) !! (y || z) LL u) ++ p)   =
   enc H (($ d. ((P d) !! y) LL (z || u)) ++ ($ d. ((P d) !! z) LL (y || u)) ++ p)

enc H (($ d. ((x++y)!!(P d)) LL u) ++ p)  =
   enc H (($ d. (x!!(P d)) LL u) ++ ($ d. (y!!(P d)) LL u) ++ p)

enc H (($ d. ((P d)!!(x++y)) LL u) ++ p)  =
   enc H (($ d. ((P d)!!x) LL u) ++ ($ d. ((P d)!!y) LL u) ++ p)

a mem H   ==> enc H ( ($d . a<d> ** (P d)) ++ p)  = (enc H p)

~ a mem H ==>  enc H ( ($d . a<d> ** (P d)) ++ p) =
                  (enc H p) ++ ($d . a<d> ** enc H (P d))
```

A.3 Extension with binary star

The following rules indicate when star-terms may be unfolded.

```
enc H (((x @@ y) !! z) LL u ++ p) =
   enc H ((x ** (x @@ y) !! z) LL u ++ (y !! z) LL u ++ p)

enc H ((z !! (x @@ y)) LL u ++ p) =
   enc H ((z !! x ** (x @@ y)) LL u ++ (z !! y) LL u ++ p)

enc H (x @@ y LL u ++ p) =
   enc H (x ** (x @@ y) LL u ++ y LL u ++ p)

enc H (($ d. ((x @@ y) !! (z d)) LL u) ++ p) =
   enc H (($d. (x ** (x @@ y) !! (z d)) LL u) ++ ($ d.(y !! (z d)) LL u) ++ p)

enc H (($ d. (z d !! (x @@ y)) LL u) ++ p) =
   enc H (($ d. (z d !! x ** (x @@ y)) LL u) ++ ($ d. (z d !! y) LL u) ++ p)
```

Due to unfolding, new terms of the form $(x + y)z$ can be introduced; therefore the following rule becomes necessary:

```
(x ++ y) ** z  = x ** z  ++  y ** z
```

A.4 Extension with conditionals

Conditionals are dealt with, by pulling them to the top of the terms. The effect is that the rewrite proof is by cases. The rules can themselves be proved by case distinction on b.

```
(if b then p else q) ** x = (if b then (p ** x) else (q ** x))

(if b then p else q) LL x = (if b then (p LL x) else (q LL x))

(if b then p else q) !! x = (if b then (p !! x) else (q !! x))

x !! (if b then p else q) = (if b then (x !! p) else (x !! q))

enc H ((if b then x else y) || z ++ p) =
   (if b then enc H (x || z ++ p) else enc H (y || z ++ p))
```

```
enc H ((if b then p else q)++z) =
   (if b then (enc H (p ++ z)) else (enc H (q ++ z )))

enc H ((if b then p else q)) = (if b then (enc H p) else (enc H q))
```

The rules found so far form one rewrite system, able to compute the first step expansion of a process.

A.5 Hiding

The following rules are used for hiding. These set of rules are not seen as extension to the previous rewrite systems, but as a separate rewrite system. In practice, first a number of expansions will be done, and only then the hiding tactic is invoked once.

```
x ** tau ** y = x ** y
a ** (if b then tau ** x else tau ** y) = a ** (if b then x else y)
hide H delta  = delta
~ a mem H ==> hide H (a<e>) = a<e>
a mem H   ==> hide H (a<e>) = tau
hide H (x ++ y)     = hide H x ++ hide H y
hide H (x ** y)     = hide H x ** hide H y
hide H ($ d. (p d)) = $ d. hide H (p d)
hide H (if b then p else q) = (if b then (hide H p) else (hide H q))
```

References

[1] G.J. Akkerman and J.C.M. Baeten. Term rewriting analysis in process algebra. *Technical report CS-R9130*, CWI, Amsterdam, 1991.

[2] M. Archer and C. Heitmeyer. Mechanical verification of timed automata: A case study. In *Proceedings 1996 IEEE Real-Time Technology and Applications Symposium* (RTAS'96). IEEE Computer Society Press, 1996.

[3] M. Archer and C. Heitmeyer. Verifying hybrid systems modeled as timed automata: a case study. In O. Maler, editor, *International Workshop, Hybrid and Real-Time Systems*, HART'97, LNCS 1201, pages 171-185, Springer-Verlag, 1997.

[4] M.M. Ayadi and D.D. Bolignagno. On the formal verification of delegation in SESAME. *IEEE COMPASS*, pages 23-34, 1997.

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

[6] G. Bella and L.C. Paulson. Using Isabelle to prove properties of the Kerberos authentication system. In H. Orman and C. Meadows, editors, *Workshop on Design and Formal Verification of Security Protocols*. DIMACS, 1997.

[7] M.A. Bezem, R. Bol and J.F. Groote. Formalizing process algebraic verifications in the calculus of constructions. *Formal Aspects of Computing*, 9(1):1-48, 1997.

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

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

[10] R. Bharadwaj, A. Felty and F. Stomp. Formalizing inductive proofs of network algorithms. In *Proceedings of the 1995 Asian Computing Science Conference*, 1995.

[11] D. Bolignagno and V. Menissier-Morain. Formal verification of cryptographic protocols using Coq. Technical Report, INRIA-Rocquencourt, 1996.

[12] D.J.B. Bosscher, I. Polak and F.W. Vaandrager. Verification of an audio control protocol. In H. Langmaack, W.P. de Roever and J. Vytopil, editors, *Proceedings of the third School and Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, LNCS 863, pages 170-192, Springer-Verlag, 1994.

[13] D.J.B. Bosscher and A. Ponse. Translating a process algebra with symbolic data values to linear format. In U.H. Engberg, K.G. Larsen, and A. Skou, editors, Proceedings of the Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Aarhus 1995, BRICS Notes Series, pages 119–130. University of Aarhus, 1995.

[14] R.S. Boyer, J S. Moore: *A Computational Logic Handbook*. Academic Press, Boston etc., 1988.

[15] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, Massachusetts, 1988. ISBN 0-201-05866-9.

[16] B. Chetali and P. Lescanne. Formal verification of a protocol for communications over faulty channels. In G. v. Bochmann, R. Dssouli and O. Rafiq, editors, *Proceedings of the IFIP TC6 Eighth International Conference on Formal Description Techniques*, pages 91-108, 1995.

[17] P. Crégut and B. Heyd. Coq-Unity. In *Actes des journées du GDR Programmation.*

[18] M.C.A. Devillers, W.O.D. Griffioen, J.M.T. Romijn and F.W. Vaandrager. Verification of a leader election protocol: formal methods applied to IEEE 1394. *Report CSI-R9728, Computing Science Institute*, Nijmegen, 1997.

[19] D. Dolev and N. Shavit. Bounded concurrent time-stamping. *SIAM Journal on Computing* 26(2):418-455, 1997.

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

[21] U. Engberg. *Reasoning in the temporal logic of actions. The design and implementation of an interactive computer system.* PhD thesis, Department of Computer Science, University of Aarhus, September 1995.

[22] U. Engberg, P. Grønning and L. Lamport. Mechanical verification of concurrent systems with TLA. In G. v. Bochmann and D.K. Probst, editors, *Proceedings of the Fourth International Conference on Computer Aided Verification* (CAV'92), LNCS 663, pages 44-55, Springer-Verlag, 1992.

[23] R.G. Gallager, P.A. Humblet, P.M. Spira: A distributed algorithm for minimal- weight spanning trees. *ACM Transactions on Programming languages* 5(1):66-77, 1983.

[24] R. Gawlick, N.A. Lynch and N. Shavit. Concurrent timestamping made simple. *Israel Symposium on Theory and Practice of Computing*, 1992.

[25] E. Gimenez. An application of co-inductive types in Coq: verification of the alternating bit protocol. In *Proceedings of the Workshop on Types for Proofs and Programs*, LNCS 1158, pages 135-152, Springer-Verlag, 1996.

[26] D.M. Goldschlag. Mechanically verifying concurrent programs with the Boyer-Moore prover. *IEEE Transactions on Software Engineering* SE-16(9): 1005-1022, September 1990.

[27] D.M. Goldschlag. Verifying safety and liveness properties of a delay insensitive fifo circuit on the Boyer-Moore prover. *International Workshop on Formal Methods in VSLI Design*, 1991.

[28] D. Gaigen, S. Kromodimoeljo, I. Meisels, W. Pase and M. Saaltink. EVES: An overview. In S. Prehn and H. Toetenel editors, *Proceedings of Formal Software Development Methods, VDM'91*, LNCS 552, pages 389-405, Springer-Verlag, 1991.

[29] M.J.C. Gordon and T.F. Melham. *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press, Cambridge, 1993.

[30] W.O.D. Griffioen. Proof-checking an audio control protocol with LP. *Report CS-R9570*, CWI, Amsterdam, 1995.

[31] J.F. Groote and W.H. Hesselink. Wait-free distributed memory management by Create and Read until Deletion (CRUD). *Technical Report SEN-R9811*, CWI, Amsterdam, 1998.

[32] J.F. Groote, F. Monin and J.C. van de Pol. Checking verifications of protocols and distributed systems by computer. In D. Sangiorgi and R. de Simone, *Proceedings of Concur'98*, Sophia Antipolis, LNCS 1466, pages 629–655, Springer-Verlag, 1998.

[33] J.F. Groote, F. Monin and J. Springintveld. A computer checked algebraic verification of a distributed summing protocol. *Computer Science Report 97-14*, Department of Mathematics and Computer Science, Eindhoven, 1997.

[34] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. In A. Ponse, C. Verhoef and S.F.M. van Vlijmen, editors, *Algebra of Communicating Processes*, Workshops in Computing, pp. 26-62, 1994.

[35] J.F. Groote and J. Springintveld. Focus points and convergent process operators. A proof strategy for protocol verification. *Technical Report 142, Logic Group Preprint Series*, Utrecht University, 1995. This report also appeared as *Technical Report CS-R9566*, CWI, Amsterdam, 1995

[36] J.F. Groote and J.C. van de Pol. A bounded retransmission protocol for large data packets. A case study in computer checked verification. In M. Wirsing and M. Nivat, editors, *Proceedings of AMAST'96*, LNCS 1101, pages 536-550, Springer-Verlag, 1996.

[37] J.V. Guttag, J.J. Horning, editors, with S.J. Garland, K.D. Jones, A. Modet and J.M. Wing. Larch: languages and tools for formal specifications. *Texts and Monographs in Computer Science*, Springer-Verlag, 1993.

[38] K. Havelund and N. Shankar. Experiments in theorem proving and model checking for protocol verification. In M.C. Gaudel and J. Woodcock, editors, *Third International Symposium of Formal Methods Europe* (FME'96), LNCS 1051, pages 662-681, Springer-Verlag, 1996.

[39] L. Helmink, M.P.A. Sellink and F.W. Vaandrager. Proof-checking a data link protocol. In H. Barendregt and T. Nipkow, editors, *Proceedings International Workshop TYPES'93*, LNCS 806, pages 127-165, Springer-Verlag, 1994.

[40] C. Heitmeyer and N.A. Lynch. The Generalized Railroad Crossing: A case study in formal verification of real-time systems. In *Proceedings of the 15th IEEE Real-Time Systems Symposium*, pages 120-131, 1994.

[41] W.H. Hesselink. The verified incremental design of a distributed spanning tree algorithm. *Computing Science Reports CS-R9602*, Groningen 1996.

[42] W.H. Hesselink. A mechanical proof of Segall's PIF algorithm. *Formal Aspects of Computing*, 9(2):208-226, 1997.

[43] J. Hooman. Verifying part of the ACCESS bus protocol using PVS. In P.S. Thiagarajan, editor, *15th Conference on the Foundations of Software Technology and Theoretical Computer Science*, LNCS 1026, pages 96-110, Springer-Verlag, 1995.

[44] J. Hooman. Verification of distributed real-time and fault-tolerant protocols. In M. Johnson, editor, *Sixth International Conference on Algebraic Methodology and Software Technology, AMSAT'97*, LNCS 1349, pages 261-275, Springer-Verlag, 1997.

[45] J. Hooman. Formal verification of the binary exponential backoff protocol. In M. Johnson, editor, *Proc. Estonian Academy of Sciences (special issue on the 9th Nordic Workshop on Programming Theory)* 4(2):89–105, 1998.

[46] M. Kaufmann and J S. Moore. An Industrial Strength Theorem Prover for a Logic Based on Common Lisp. *Transactions on Software Engineering*, 23(4):203-213, 1997.

[47] H.P. Korver and J. Springintveld. A computer-checked verification of Milner's scheduler. In M. Hagiya and J.C. Mitchell, editors, *Proceedings of the International Symposium on Theoretical Aspects of Computer Software (TACS'94)*, LNCS 789, pages 161-178, Springer-Verlag, 1994.

[48] H. Korver and A. Sellink. On automating process algebra proofs. *Technical Report 154*, Logic Group Preprint Series, Utrecht University, 1996.

[49] R.P. Kurshan and L. Lamport. Verification of multiplier: 64 bits and beyond. In C. Courcoubetis, editor, *Proceedings of the Fifth International Conference on Computer Aided Verification* (CAV'93), LNCS 697, pages 166-179, Springer-Verlag, 1993.

[50] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872-923, 1994.

[51] L. Lamport and P.M. Melliar-Smith. Synchronizing clocks in the presence of faults.*Journal of the ACM* 32(1):52-78, 1985.

[52] L. Lamport, R. Sostack and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems* 4(3):382-401, 1982.

[53] G. Leeb and N.A. Lynch. Proving safety properties of the Steam Boiler Controller: Formal methods for industrial applications: A case study. In J.-R. Abrial, et al., editors, *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*, LNCS 1165, Springer-Verlag, 1996.

[54] D. Lesens and H. Saïdi. Abstraction of Parameterized Networks. In F. Moller, editor, *Electronic Notes in Theoretical Computer Science* 9, (Special issue for Infinity'97, Bologna), 1997.

[55] P. Lincoln and J. Rushby. The formal verification of an algorithm for interactive consistency under a hybrid fault model. In C. Courcoubetis, editor, *Fifth International Conference on Computer-Aided Verification* (CAV'93), LNCS 697, pages 305-319, Springer-Verlag, 1993.

[56] V. Luchangco, E. Söylemez, S.J. Garland and N.A. Lynch. Verifying timing properties of concurrent algorithms. In D. Hogrefe and S. Leue, editors, *Proceedings of the Seventh International Conference on Formal Description Techniques for Distributed Systems* (FORTE'94), pages 259-273, IFIP WG6.1, Chapman&Hall, 1995.

[57] N.A. Lynch and F.W. Vaandrager. Forward and backward simulations for timing-based systems. In J.W. de Bakker, C. Huizing and G. Rozenberg, editors. *Proceedings of REX Workshop "Real-Time: Theory in Practice"*, LNCS 600, pages 397-446. Springer-Verlag, 1992.

[58] N.A. Lynch and M. Tuttle. An introduction to Input/Output automata. *CWI Quarterly* 2(3):219-246, 1989.

[59] Z. Manna and A. Pnueli. Verification of concurrent programs: the temporal framework. In R.S. Boyer and J S. Moore, editors, *The correctness Problem in Computer Science*, Academic Press, London, 1981.

[60] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[61] A. Mokkedem, M.J. Ferguson and R. deB. Johnston. A TLA solution to the specification and verification of the RLP1 retransmission protocol. In J. Fitzgerald, C.B. Jones and P. Lucas, editors, *Proceedings of the Fourth International Symposium of Formal Methods Europe* (FME'97), LNCS 1313. Springer-Verlag, 1997.

[62] J S. Moore. A formal model of asynchronous communication and its use in mechanically verifying a biphase mark protocol. *Journal of Formal Aspects of Computing Science* 6(1):60-91, 1994.

[63] O. Müller and T. Nipkow. Combining model checking and deduction for I/O-automata. In U.H. Engberg, K.G. Larsen and A. Skou, editors, *Proceedings of the First Workshop Tools and Algorithms for the Construction and Analysis of Systems*, (TACAS'95), LNCS 1019, pages 1-16, Springer-Verlag, 1995.

[64] O. Müller and T. Nipkow. Traces of I/O automata in Isabelle/HOLCF. In M. Bidoit and M. Dauchet, editors, *Proceedings of the Seventh International Joint on the Theory and Practice of Software Development* (TAPSOFT'97), LNCS 1214, pages 580-595, Springer-Verlag, 1997.

[65] M. Nagayama and C. Talcott. An NQTHM mechanization of an exercise in the verification of multi-process programs. *Technical Report STAN-CS-91-1370*, Stanford University, 1991.

[66] T. Nipkow and K. Slind. I/O automata in Isabelle/HOL. In P. Dybjer, B. Nordström and J.Smith, editors, *Proceedings of the International Workshop on Types for Proofs and Programs*, LNCS 996, pages 101-119, Springer-Verlag, 1994.

[67] L.C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.

[68] L.C. Paulson. *Isabelle: A Generic Theorem Prover.* LNCS 828, Springer-Verlag, 1994.

[69] L.C. Paulson. On two formal analyses of the Yahalom protocol. *Technical Report 432*, Computer Laboratory, University of Cambridge, 1997.

[70] L.C. Paulson. The inductive approach to verifying cryptographic protocols. *Computer Security Journal*, to appear 1998.

[71] T.P. Petrov, A. Pogosyants, S.J. Garland, V. Luchangco and N.A. Lynch. Computer-assisted verification of an algorithm for concurrent timestamps. In R. Gotzhein and J. Bredereke, editors, *Formal Description Techniques IX: Theory, Applications, and Tools*, (FORTE/PTSV'96: Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification), pages 29-44, Chapman & Hall, 1996.

[72] J. Rushby. Formal verification of an oral messages algorithm for interactive consistency. *Contractor Report 4NAS1-18969*, NASA, 1992.

[73] J. Rushby and F. von Henke. Formal verification of a fault-tolerant clock synchronization algorithm. *NASA Contractor Report 4239*, 1989.

[74] D.M. Russinoff. Verifying concurrent programs with the Boyer-Moore Prover. *Technical Report STP/ACT-218-90*, MCC, Austin, Texas, 1990.

[75] D.M. Russinoff. A Mechanically verified incremental garbage collector. *Technical Report STP/ACT–91*, MCC, Austin, Texas, 1991.

[76] N. Shankar. Mechanical verification of a generalized protocol for Byzantine fault-tolerant clock synchronization. In J. Vytopil, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, LNCS 571, pages 217-236, 1992.

[77] N. Shankar. Verification of real-time systems using PVS. In C. Courcoubetis, editor, *Fifth Conference on Computer-Aided Verification* (CAV'93), LNCS 697, pages 280-291, Springer-Verlag, 1993.

[78] N. Shankar, S. Owre and J.M. Rushby. *The PVS Proof Checker: A Reference Manual.* Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993.

[79] J.F. Søgaard-Andersen, S.J. Garland, J.V. Guttag, N.A. Lynch and A. Pogosyants. Computer-assisted simulation proofs. In C, Courcoubetis, editor, *Fifth Conference on Computer-Aided Verification* (CAV'93), LNCS 697, pages 305-319, Springer-Verlag, 1993.

[80] J. Vitt and J. Hooman. Assertional specification and verification using PVS of the Steam Boiler Control system. In J.-R. Abrial, et al., editors, *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*, LNCS 1165, Springer-Verlag, 1996.

[81] J. von Wright and T. Långbacka. Using a theorem prover for reasoning about concurrent algorithms. In G. v. Bochmann and D.K. Probst, editors, *Proceedings of the Fourth International Conference on Computer Aided Verification* (CAV'92), LNCS 663, pages 56-68, Springer-Verlag, 1992.

[82] W.D. Young. Verifying the interactive convergence clock synchronization algorithm using the Boyer-Moore theorem prover. *Contractor Report 189649*, NASA, 1992.

[83] W.D. Young. Modeling and verification of a simple real-time railroad gate controller. In M.G. Hinchey and J.P. Bowen, editors. *Applications of Formal Methods*, Prentice Hall International Series in Computer Science, pages 181-202, 1995.

[84] W.D. Young. Comparing verification systems: interactive consistency in ACL2. *IEEE COMPASS*, pages 35-55, 1996.